

# **Java Generics**

## **Frequently Asked Questions**

**written and maintained by**  
**Angelika Langer**

**January 2015**

This is a snapshot of the online Java Generics FAQ in printable form. The most up-to-date version of the original document can be found at

**[www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html](http://www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html)**.

All text and content found in this document and online at URLs starting with <http://www.AngelikaLanger.com/GenericsFAQ/> (collectively, "the Java Generics FAQ") are the sole property of Angelika Langer. Copyright © 2004-2022 by Angelika Langer. All rights reserved.

# About This FAQ

© Copyright 2004-2022. by Angelika Langer. All Rights Reserved.

[How is this FAQ organized at large?](#)

[How are the individual answered questions organized?](#)

[Whom is the FAQ for?](#)

[How do I best read the FAQ - top-to-bottom or zig-zag?](#)

[Whom do I contact when I have questions, comments or suggestions related to this FAQ?](#)

[Table Of Contents](#)

[Acknowledgements](#)

[Change Log](#)

[Copyright Notice](#)

---

## How is this FAQ organized at large?

The FAQ has 4 major sections:

- [Fundamentals of Java Generics](#)
- [Language Features of Java Generics](#)
- [Practicalities - Programming With Java Generics](#)
- [Technicalities - Under the Hood of the Compiler](#)

"Fundamentals of Java Generics" explains the very basics of Java generics, what generics are and what their purpose and benefit is.

"Language Features of Java Generics" is a huge section that explains all language features of Java Generics:

- [Generic Types](#)
- [Generic Methods](#)
- [Type Parameters](#)
- [Type Arguments](#)

This section can be seen as the theoretical foundation and lays the ground for understanding more practical and more advanced issues.

"Practicalities - Programming With Java Generics" aims to address issues that arise when you use Java Generics in practice. It discusses topics such as "How do I design a generics API?", "Should my class be generic or not?", "What happens when I mix generic and non-generic Java?", "Why is class `Class` generic and what is this good for?". The focus is on programming techniques on the one hand and pitfalls and surprises on the other hand. It is likely that this section will grow over time as programming idioms and techniques using Java Generics will be discovered.

"Technicalities - Under the Hood of the Compiler" addresses more advanced and more esoteric aspects of Java Generics, which will become the more interesting the more you learn about Java Generics. Some of these topics are typically of little interest when you begin using generics. For instance, the section explains what the notorious "unchecked" warnings are, what translation by type erasure is, what wildcard capture and type inference are, what the type system looks like now that it includes generic types, and how overload resolution works in presence of generic methods.

In addition to these four main sections the FAQ has a [Table Of Contents](#), a reference to [More Information on Java Generics](#) such as tutorials, books, articles, and websites, a [Glossary](#), and an alphabetic [Index](#).

## How are the individual answered questions organized?

There is a question, a short answer and a sometimes fairly long answer, plus a cross reference section.

I've found that many readers underestimate the reference section - to their own detriment. The cross references refer to related items and point in particular to entries that explain terms or facts mentioned in the text, but not explained there. In other words, if there is something in the explanation that has not been explained yet, go to the reference section and see whether you can find a reference to a corresponding entry that explains it. If not, let me know.

LINK TO THIS [Preface.FAQ002](#)

REFERENCES

---

## Whom is the FAQ for?

*For practicing Java programmers who have to do with Java Generics.*

First a disclaimer: the FAQ does not aim to be mainly a tutorial. Tutorials can be found elsewhere. It has tutorial style answers, but it goes beyond what a casual or first-time generics user might want to know about Java Generics. On the other hand, it is still written for humans, as opposed to compiler builders or computer science theorists whose pet issue is type theory. While type theory is an interesting topic, the main target audience is programmers who use or want to use generics in practice.

The FAQ is supposed to grow and mature over time. In particular the section on programming techniques using generics will hopefully gain weight as the body of experience with Java generics grows. If you have a technique or observation made in practice that you want to share with other programmers, feel free to send email and suggest it for inclusion in the FAQ.

LINK TO THIS [Preface.FAQ003](#)

REFERENCES [Where can I find a generics tutorial?](#)  
[Whom do I contact when I have questions, comments or suggestions related to this FAQ?](#)

---

## How do I best read the FAQ - top-to-bottom or zig-zag?

*Both should be possible.*

The FAQ is kind of organized in a way that you can read it sequentially, starting with the first question until the bitter end. However, you will find that occasional jumps back and forth will be inevitable.

For instance, when the language features are explained the term *type erasure* will be mentioned. Type erasure is part of what the compiler does during compilation and it is discussed in detail in the Technicalities section. So you will probably jump forward to the explanation of type erasure to see what it is in principle, spare yourself the details, and return to the language feature from which you started your excursion to the Technicalities section.

Beyond that, the FAQ is mainly intended as a reference and can be consulted any order you feel like. Related topics are grouped together so that it might be convenient to read certain sections that are of immediate interest and ignore the rest for the time being.

You might find that the same fact is explained repeatedly in different entries. This is intended. A certain amount of redundancy is inevitable in order to make it possible to use the FAQ as a reference. If you feel you already know this, skip it and move on.

Also, if you use the FAQ as a reference, the table of contents and the index should help finding what you are looking for. If not, let me know.

LINK TO THIS [Preface.FAQ004](#)

REFERENCES [Table Of Contents](#)  
[Index](#)  
[Whom do I contact when I have questions, comments or suggestions related to this FAQ?](#)

---

## Whom do I contact when I have questions, comments or suggestions related to this FAQ?

If you want to provide feedback or have questions regarding Java generics, to which you cannot find an answer in this document, feel free to send me [EMAIL](#) or use the [GENERICS FAQ](#) form. I might not be capable of answering in a timely manner or perhaps not at all, but I will try.

LINK TO THIS [Preface.FAQ005](#)

REFERENCES

---

# Table Of Contents

## [About This FAQ](#)

## [Fundamentals of Java Generics](#)

## [Language Features of Java Generics](#)

### [Generic Types](#)

#### [Fundamentals](#)

#### [Concrete Instantiations](#)

#### [Raw Types](#)

#### [Wildcard Instantiations](#)

### [Generic Methods](#)

#### [Fundamentals](#)

### [Type Parameters](#)

#### [Fundamentals](#)

#### [Type Parameter Bounds](#)

#### [Usage](#)

#### [Scope](#)

#### [Static Context](#)

### [Type Arguments](#)

[Fundamentals](#)  
[Wildcards](#)  
[Wildcard Bounds](#)

## **Practicalities - Programming With Java Generics**

[Using Generic Types](#)  
[Using Generic Methods](#)  
[Coping With Legacy](#)  
[Defining Generic Types and Methods](#)  
[Designing Generic Methods](#)  
[Working With Generic Interfaces](#)  
[Implementing Infrastructure Methods](#)  
[Using Runtime Type Information](#)  
[Reflection](#)

## **Technicalities - Under the Hood of the Compiler**

[Compiler Messages](#)  
[Heap Pollution](#)  
[Type Erasure](#)  
[Type System](#)  
[Exception Handling](#)  
[Static Context](#)  
[Type Argument Inference](#)  
[Wildcard Capture](#)  
[Wildcard Instantiations](#)  
[Cast and instanceof](#)

## **More Information on Java Generics**

### **Glossary**

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

### **Index**

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

---

## **About This FAQ**

[How is this FAQ organized at large?](#)  
[How are the individual answered questions organized?](#)  
[Whom is the FAQ for?](#)  
[How do I best read the FAQ - top-to-bottom or zig-zag?](#)  
[Whom do I contact when I have questions, comments or suggestions related to this FAQ?](#)

---

## **Fundamentals of Java Generics**

[What are Java generics?](#)  
[What is the primary purpose of Java generics?](#)  
[What is the benefit of using Java generics?](#)

What does type-safety mean?

---

## **Language Features of Java Generics**

Which language features are related to Java generics?

---

### Generic Types

#### Fundamentals

- What is a generic or parameterized type?
- How do I define a generic type?
- Are there any types that cannot have type parameters?
- How is a generic type instantiated (to form a parameterized type)?
- Why do instantiations of a generic type share the same runtime type?
- Can I cast to a parameterized type?
- Can I use parameterized types in exception handling?
- Can generic types have static members?

#### Concrete Instantiations

- What is a concrete instantiation of a generic type?
- Are different concrete instantiations of the same generic type compatible?
- Can I use a concrete parameterized type like any other type?
- Can I create an array whose component type is a concrete parameterized type?
- Can I declare a reference variable of an array type whose component type is a concrete parameterized type?
- How can I work around the restriction that there are no arrays whose component type is a concrete parameterized type?
- Why is there no class literal for the concrete parameterized type?

#### Raw Types

- What is the raw type?
- Can I use a raw type like any other type?

#### Wildcard Instantiations

- What is a wildcard instantiation of a generic type?
- What is an unbounded wildcard instantiation of a generic type?
- Which methods and fields are accessible/inaccessible through a reference variable of a wildcard parameterized type?
- What is the difference between the unbounded wildcard parameterized type and the raw type?
- Can I use a wildcard parameterized type like any other type?
- Can I create an object whose type is a wildcard parameterized type?
- Can I create an array whose component type is a wildcard parameterized type?
- Can I declare a reference variable of an array type whose component type is a bounded wildcard parameterized type?
- Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?
- Can I declare a reference variable of an array type whose component type is an unbounded wildcard parameterized type?
- Can I derive from a wildcard instantiation of a parameterized type?
- Why is there no class literal for wildcard parameterized types?

---

### Generic Methods

#### Fundamentals

- What is a generic method?
- How do I invoke a generic method?

---

## Type Parameters

### Fundamentals

- [What is a type parameter?](#)
- [Where is a type parameter visible \(or invisible\)?](#)

### Type Parameter Bounds

- [What is a bounded type parameter?](#)
- [Which types are permitted as type parameter bounds?](#)
- [Can I use a type parameter as a type parameter bound?](#)
- [Can I use different instantiations of a same generic type as bounds of a type parameter?](#)
- [How can I work around the restriction that a type parameter cannot have different instantiations of a same generic type as its bounds?](#)
- [Does a bound that is a class type give access to all its public members?](#)
- [How do I decrypt "Enum<E extends Enum<E>>"?](#)

### Usage

- [Can I use a type parameter like a type?](#)
- [Can I create an object whose type is a type parameter?](#)
- [Can I create an array whose component type is a type parameter?](#)
- [Can I cast to the type that the type parameter stands for?](#)
- [Can I use a type parameter in exception handling?](#)
- [Can I derive from a type parameter?](#)
- [Why is there no class literal for a type parameter?](#)

### Scope

- [Where is a type parameter visible \(or invisible\)?](#)
- [Can I use a type parameter as part of its own bounds or in the declaration of other type parameters?](#)

### Static Context

- [Is there one instances of a static field per instantiation of a generic type?](#)
- [Why can't I use a type parameter in any static context of the generic class?](#)

---

## Type Arguments

### Fundamentals

- [What is a type argument?](#)
- [Which types are permitted as type arguments?](#)
- [Are primitive types permitted as type arguments?](#)
- [Are wildcards permitted as type arguments?](#)
- [Are type parameters permitted as type arguments?](#)
- [Do I have to specify a type argument when I want to use a generic type?](#)
- [Do I have to specify a type argument when I want to invoke a generic method?](#)

### Wildcards

- [What is a wildcard?](#)
- [What is an unbounded wildcard?](#)
- [What is a bounded wildcard?](#)
- [What do multi-level wildcards mean?](#)
- [If a wildcard appears repeatedly in a type argument section, does it stand for the same type?](#)

### Wildcard Bounds

- [What is a wildcard bound?](#)
  - [Which types are permitted as wildcard bounds?](#)
  - [What is the difference between a wildcard bound and a type parameter bound?](#)
-

## Practicalities - Programming With Java Generics

### Using Generic Types

- Should I prefer parameterized types over raw types?
- Why shouldn't I mix parameterized and raw types, if I feel like it?
- Should I use the generic collections or better stick to the old non-generic collections?
- What is a checked collection?
- What is the difference between a `Collection<?>` and a `Collection<Object>`?
- How do I express that a collection is a mix of objects of different types?
- What is the difference between a `Collection<Pair<String, Object>>`, a `Collection<Pair<String, ?>>` and a `Collection<? extends Pair<String, ?>>`?
- How can I make sure that the same wildcard stand for the same type?

### Using Generic Methods

- Why doesn't method overloading work as I expect it?
- Why doesn't method overriding work as I expect it?

### Coping With Legacy

- What happens when I mix generic and non-generic code?
- Should I re-engineer all my existing classes and generify them?
- How do I generify an existing non-generic type or method?
- Can I safely generify a supertype, or does it affect all subtypes?
- How do I avoid breaking binary compatibility when I generify an existing type or method?

### Defining Generic Types and Methods

- Which types should I design as generic types instead of defining them as regular non-generic types?
- Do generics help designing parallel class hierarchies?
- When would I use an unbounded wildcard instantiation instead of a bounded or concrete instantiation?
- When would I use a wildcard instantiation instead of a concrete instantiation?
- When would I use a wildcard instantiation with an lower bound?
- How do I recover the actual type of the `this` object in a class hierarchy?
- What is the "getThis" trick?
- How do I recover the element type of a container?
- What is the "getTypeArgument" trick?

### Designing Generic Methods

- Why does the compiler sometimes issue an unchecked warning when I invoke a "varargs" method?
- What is a "varargs" warning?
- How can I suppress a "varargs" warning?
- When should I refrain from suppressing a "varargs" warning?
- Which role do wildcards play in method signatures?
- Which one is better: a generic method with type parameters or a non-generic method with wildcards?
- Under which circumstances are the generic version and the wildcard version of a method equivalent?
- Under which circumstances do the generic version and the wildcard version of a method mean different things?
- Under which circumstances is there no transformation to the wildcard version of a method possible?
- Should I use wildcards in the return type of a method?
- How do I implement a method that takes a wildcard argument?
- How do I implement a method that takes a multi-level wildcard argument?
- I want to pass a `U` and a `X<U>` to a method. How do I correctly declare that method?

### Working With Generic Interfaces

- Can a class implement different instantiations of the same generic interface?



- [Can a subclass implement another parameterized interface than any of its superclasses does?](#)
- [What happens if a class implements two generic interfaces that define the same method?](#)
- [Can an interface type nested into a generic type use the enclosing type's type parameters?](#)

### Implementing Infrastructure Methods

- [How do I best implement the equals method of a generic type?](#)
- [How do I best implement the clone method of a generic type?](#)

### Using Runtime Type Information

- [What does the type parameter of class java.lang.Class mean?](#)
- [How do I pass type information to a method so that it can be used at runtime?](#)
- [How do I generically create objects and arrays?](#)
- [How do I perform a runtime type check whose target type is a type parameter?](#)

### Reflection

- [Which information related to generics can I access reflectively?](#)
  - [How do I retrieve an object's actual \(dynamic\) type?](#)
  - [How do I retrieve an object's declared \(static\) type?](#)
  - [What is the difference between a generic type and a parameterized type in reflection?](#)
  - [How do I figure out whether a type is a generic type?](#)
  - [Which information is available about a generic type?](#)
  - [How do I figure out whether a type is a parameterized type?](#)
  - [Which information is available about a parameterized type?](#)
  - [How do I retrieve the representation of a generic method?](#)
  - [How do I figure out whether a method is a generic method?](#)
  - [Which information is available about a generic method?](#)
  - [Which information is available about a type parameter?](#)
  - [What is a generic declaration?](#)
  - [What is a wildcard type?](#)
  - [Which information is available about a wildcard?](#)
- 

## **Technicalities - Under the Hood of the Compiler**

### Compiler Messages

- [What is an "unchecked" warning?](#)
- [How can I disable or enable unchecked warnings?](#)
- [What is the -Xlint:unchecked compiler option?](#)
- [What is the SuppressWarnings annotation?](#)
- [How can I avoid "unchecked cast" warnings?](#)
- [Is it possible to eliminate all "unchecked" warnings?](#)
- [Why do I get an "unchecked" warning although there is no type information missing?](#)

### Heap Pollution

- [What is heap pollution?](#)
- [When does heap pollution occur?](#)

### Type Erasure

- [How does the compiler translate Java generics?](#)
- [What is type erasure?](#)
- [What is reification?](#)
- [What is a bridge method?](#)
- [Under which circumstances is a bridge method generated?](#)
- [Why does the compiler add casts when it translates generics?](#)
- [How does type erasure work when a type parameter has several bounds?](#)
- [What is a reifiable type?](#)
- [What is the type erasure of a parameterized type?](#)
- [What is the type erasure of a type parameter?](#)
- [What is the type erasure of a generic method?](#)

- [Is generic code faster or slower than non-generic code?](#)
- [How do I compile generics for use with JDK <= 1.4?](#)

### Type System

- [How do parameterized types fit into the Java type system?](#)
- [How does the raw type relate to instantiations of the corresponding generic type?](#)
- [How do instantiations of a generic type relate to instantiations of other generic types that have the same type argument?](#)
- [How do unbounded wildcard instantiations of a generic type relate to other instantiations of the same type?](#)
- [How do wildcard instantiations with an upper bound relate to other instantiations of the same type?](#)
- [How do wildcard instantiations with a lower bound relate to other instantiations of the same type?](#)
- [Which super-subtype relationships exist among instantiations of generic types?](#)
- [Which super-subset relationships exist among wildcards?](#)
- [Does "extends" always mean "inheritance"?](#)

### Exception Handling

- [Can I use parameterized types in exception handling?](#)
- [Why are generic exception and error types illegal?](#)
- [Can I use a type parameter in exception handling?](#)
- [Can I use a type parameter in a catch clause?](#)
- [Can I use a type parameter in in a throws clause?](#)
- [Can I throw an object whose type is a type parameter?](#)

### Static Context

- [How do I refer to static members of a parameterized type?](#)
- [How do I refer to a \(non-static\) inner class of a parameterized type?](#)
- [How do I refer to an interface type nested into a parameterized type?](#)
- [How do I refer to an enum type nested into a parameterized type?](#)
- [Can I import a particular instantiation of generic type?](#)
- [Why are generic enum types illegal?](#)

### Type Argument Inference

- [What is type argument inference?](#)
- [Is there a correspondence between type inference for method invocation and type inference for instance creation?](#)
- [What is the "diamond" operator?](#)
- [What is type argument inference for instance creation expressions?](#)
- [Why does the type inference for an instance creation expression fail?](#)
- [What is type argument inference for generic methods?](#)
- [What explicit type argument specification?](#)
- [Can I use a wildcard as the explicit type argument of a generic method?](#)
- [What happens if a type parameter does not appear in the method parameter list?](#)
- [Why doesn't type argument inference fail when I provide inconsistent method arguments?](#)
- [Why do temporary variables matter in case of invocation of generic methods?](#)

### Wildcard Capture

- [What is the capture of a wildcard?](#)
- [What is the capture of an unbounded wildcard compatible to?](#)
- [Is the capture of a bounded wildcard compatible to the bound?](#)

### Wildcard Instantiations

- [Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard instantiation?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard instantiation?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard instantiation?](#)
- [Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard instantiation?](#)
- [Which methods that use the type parameter as upper wildcard bound in a parameterized argument](#)

- [or return type are accessible in a wildcard instantiation?](#)
- [Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard instantiation?](#)
- [In a wildcard instantiation, can I read and write fields whose type is the type parameter?](#)
- [Is it really impossible to create an object whose type is a wildcard parameterized type?](#)

#### Cast and instanceof

- [Which types can or must not appear as target type in an instanceof expression?](#)

#### Overloading and Overriding

- [What is method overriding?](#)
- [What is method overloading?](#)
- [What is the @Override annotation?](#)
- [What is a method signature?](#)
- [What is a subsignature?](#)
- [What are override-equivalent signatures?](#)
- [When does a method override its supertype's method?](#)
- [What are covariant-return types?](#)
- [What are substitutable return types?](#)
- [Can a method of a non-generic subtype override a method of a generic supertype?](#)
- [Can a method of a generic subtype override a method of a generic supertype?](#)
- [Can a generic method override a generic one?](#)
- [Can a non-generic method override a generic one?](#)
- [Can a generic method override a non-generic one?](#)
- [What is overload resolution?](#)

---

## **More Information on Java Generics**

[Where can I find a generics tutorial?](#)

[Where can I find a specification of the Java generics language features?](#)

[Which books cover Java generics?](#)

[What webpages are devoted to Java generics?](#)

---

## **Glossary**

[A](#)·[B](#)·[C](#)·[D](#)·[E](#)·[F](#)·[G](#)·[H](#)·[I](#)·[J](#)·[K](#)·[L](#)·[M](#)·[N](#)·[O](#)·[P](#)·[Q](#)·[R](#)·[S](#)·[T](#)·[U](#)·[V](#)·[W](#)·[X](#)·[Y](#)·[Z](#)

---

## **Index**

[A](#)·[B](#)·[C](#)·[D](#)·[E](#)·[F](#)·[G](#)·[H](#)·[I](#)·[J](#)·[K](#)·[L](#)·[M](#)·[N](#)·[O](#)·[P](#)·[Q](#)·[R](#)·[S](#)·[T](#)·[U](#)·[V](#)·[W](#)·[X](#)·[Y](#)·[Z](#)

---

# **Acknowledgements**

The FAQ is written and maintained by [Angelika Langer](#).

Klaus Kreft, Bruce Eckel and Cay Horstmann reviewed this FAQ and provided invaluable feedback.

A number of individuals at (or formerly at) Sun Microsystems (now Oracle) patiently answered countless questions I posed regarding Java generics. My thanks to Gilad Bracha, Neal Gafter, and Peter von der Ahé.

Suggestions and corrections, that inspired changes in this FAQ, have been provided by the following people (in alphabetic order): Eric Armstrong, Richard Grin, Markus Keller, Richard Kennard, Sascha Kratky, Keith Lea, Mike Lehmann, Maurice Naftalin, Joe Soroka, Frank Tip, Phil Wadler.

---

# Change Log

Below is a list of minor and major modifications of the FAQ.

October 2004      First release.

January 2005      Terminology change.

A reader pointed out that the [draft of JLS 3](#) uses terms different from what I use in this FAQ document. This is because I started writing the FAQ before even a draft of JLS 3 was available. There still is no final specification of the language features of Java Generics. Nonetheless I decided to reduce the mismatch somewhat by changing some of the terms used throughout this document.

The most significant deviation from the JLS 3 terminology is the use of the term "parameterized type". The JLS uses the term "parameterized type" to refer to types with a type argument list, such as `List<String>` or `List<?>`. I used the term "parameterized type" to distinguish between types with type parameters, such as `List`, which is a type that declares a type parameter `E` that stands for the type of the elements in the list, and plain old types without type parameters, such as `String`, `Date`, `Point`, etc. Hence I made the following change:

<i>formerly used in this FAQ</i>	<i>=&gt; now used in this FAQ (and in JLS 3)</i>
<code>parameterized</code> type/method	<i>=&gt;</i> <code>generic</code> type/method
instantiation of a parameterized type	<i>=&gt;</i> <code>parameterized</code> type

There remains an issue regarding use of the term "instantiation". The term "instantiation" is used in the JLS in conjunction with object creation: a type is instantiated to form an instance of the type, that is, an object of that type. I reuse the term "instantiation" to refer to the "creation" of a parameterized type from a generic type, that is, for the manifestation of a generic type such as `List` with its formal type parameter `E` as a parameterized type with an actual type argument as in `List<String>` or `List<?>`. This can be regarded as misleading, because nothing is created, not even a new type, different from what happens in C++ for instance when a template is instantiated. Yet I could not think of a more precise or adequate term that would denote the fact that formal type parameters are replaced by actual type arguments in a generic type or method. In other words, I "instantiate" generic types to form parameterized types, and I talk of "instantiations" of a generic type, which are the parameterized type, as opposed to "instances" of a type, which would be objects, like in the JLS.

May 2005      Updated the reference section; since April 19, 2005 the Java Language Specification, 3rd edition is available on the web at: <http://java.sun.com/docs/books/jls/>

August 2005      Corrected a couple of bugs. Affected are [TypeArguments.FAQ103](#), [TypeArguments.FAQ104](#), [TechnicalDetails.FAQ501](#), [TechnicalDetails.FAQ502](#), [ProgrammingIdioms.FAQ201](#), [ProgrammingIdioms.FAQ303](#). Added some new items: [ProgrammingIdioms.FAQ006A](#), [ProgrammingIdioms.FAQ050](#), [ProgrammingIdioms.FAQ302A](#) thru [ProgrammingIdioms.FAQ302C](#), [TechnicalDetails.FAQ050](#), [TechnicalDetails.FAQ051](#).

Added a [glossary](#) in order to get the terminology straight. I noticed that documents originating from Sun Microsystems, such as the language specification ([JLS 3](#)) and the [tutorial](#), use slightly different terms than you will find in books and articles from other sources. The glossary aims to list commonly used terms related to Java Generics and to explain what they denote and which ones are synonyms. The controversial terms are

invocation/instantiation of generic/parameterized types/methods. The JLS talks of an *invocation* of a generic type or method, where most other authors talk of an *instantiation* of a generic type or method. Some authors equate generic type and parameterized type, while the JLS equates invocation and parameterized type.

- October 2005 Added some stuff on overloading and overriding in the context of Java generics. The new items are: [ProgrammingIdioms.FAQ051](#) and [TechnicalDetails.FAQ801](#) thru [TechnicalDetails.FAQ830](#). Further additions: [ProgrammingIdioms.FAQ103A](#), [ProgrammingIdioms.FAQ201A](#), [ProgrammingIdioms.FAQ300](#), [TechnicalDetails.FAQ007](#), [TechnicalDetails.FAQ402A](#), [TechnicalDetails.FAQ609](#).
- December 2005 Added new item: [TypeParameters.FAQ107](#). Made available a [PDF version](#) of this FAQ document.
- May 2006 Added new item: [ProgrammingIdioms.FAQ205](#).
- November 2006 Added an additional example to item [ProgrammingIdioms.FAQ300](#). My thanks to Peter von der Ahé and Bob Lee for supplying the example.
- December 2006 Added whole new section on Java Generics and Reflection: [ProgrammingIdioms.Reflection](#). Corrected a mistake in entry [TechnicalDetails.FAQ821](#). My thanks to Jingyi Wang for spotting the mistake.
- May 2007 Corrected some typos. My thanks to Oleksii Shurubura and Zhang Xu for spotting the bugs. Added new item describing the "getThis" trick: [ProgrammingIdioms.FAQ206](#).
- October 2007 Corrected some typos. My thanks to Steven Adams and Seb Rose for spotting the bugs.
- February 2008 Corrected some typos. My thanks to Liu Yuqing for spotting the bugs. German speaking readers might be interested in a series of articles that I made available at [EffectiveJava](#). The articles are drafts of contributions to our "Effective Java" column published in a German print magazine in 2007.
- March 2008 Corrected some typos. My thanks to Marcel Rüedi for spotting the bugs.
- April 2008 Replaced an incorrect example in [TypeParameters.FAQ402](#) by a more useful one and corrected a bug in [TypeParameters.FAQ105](#). My thanks to Jürgen Jäschke and Martin Goerg for spotting the bugs.
- July 2008 Corrected some typos. My thanks to Rob Ross for spotting the bugs. Added some links and bookmarks to the [PDF version](#) of this FAQ document for better navigation. It is not perfect, but I hope it helps.
- August 2008 Added a link and an additional example to [Practicalities.FAQ603](#). My thanks to Dan Frankowski for suggesting the addition. Rewrote [Technicalities.FAQ402A](#) because the compilers nowadays apply smarter type inference strategies than they did when the entry was written; as a result some of the source code examples were no longer meaningful. My thanks to Steven Busack for bringing the changes to my attention. Added two new entries [Practicalities.FAQ207](#) and [Practicalities.FAQ208](#) that describe a programming technique that Jesse Glick brought to my attention. Kudos to Jesse.
- October 2008 Corrected some typos. My thanks to Gary Gregory for spotting the bugs.
- August 2009 Corrected some typos. My thanks to Ganesh Hegde and Barry Soroka for spotting the bugs.
- June 2010 Corrected bugs in [Technicalities.FAQ103](#) and [Technicalities.FAQ608](#). My thanks to Clive Scott for spotting the bugs.
- August 2010 Made an update to acknowledge additions and corrections intended for release with Java 7. Specifically, I added a couple of entries regarding the problem with varargs and generics, an explanation of the new varargs warning, and whether or not one can safely suppress the varargs warnings (see [Practicalities.FAQ300A](#)). In addition, I added a section on the diamond operator, the improved type inference for instance creation expressions and its limitations (see [Technicalities.FAQ400](#)). My thanks to Maurizio Cimadamore of Oracle for a

couple of clarifications.  
Corrected a typo. My thanks to Prasanth Jalasutram for spotting the typo.

- May 2011 Corrected minor omission in [Technicalities.FAQ824](#). My thanks to Gene Carpenter for spotting the omission.
- August 2011 Made a couple of minor adjustments in [Practicalities.FAQ300A](#), [Practicalities.FAQ300B](#), and [Practicalities.FAQ300C](#) due to differences between the prototype implementation and the final release of Java 7. Specifically, the annotation `@SuppressWarnings("varargs")` has been replaced by the `@SafeVarargs` annotation and the "varargs" warnings became plain "unchecked" warnings.
- February 2012 Corrected some formatting and grammar issues. My thanks to Chris Dailey and Alan Frankel for spotting the deficiencies.
- November 2012 Updated the section on related information such as books and websites (see [Information.FAQ002](#) and [Information.FAQ005](#)).
- February 2013 Corrected an error in [Technicalities.FAQ400D](#) regarding use of the diamond operator on anonymous inner classes. My thanks to Rabea Gransberger for spotting the bug.
- April 2013 Updated the sections related to type parameter inference for invocation of generic methods and the diamond operator. Type inference has been changed and improved substantially for Java 8. Numerous situations that resulted in error messages in Java 5, 6, and 7 compile just fine in Java 8. The changes concern [Technicalities.FAQ400C](#), [Technicalities.FAQ400D](#), [Technicalities.FAQ403](#), [Technicalities.FAQ405](#).  
Updated entry [TypeParameters.FAQ107](#) for clarification. The entry discusses the lack of lower bounds on type parameters, i.e., a feature that does not (yet) exist, but is occasionally considered useful for certain situations. I added examples in order to illustrate when the feature would be harmful or useful.
- February 2014 Fixed an incorrect example in [TypeParameters.FAQ102](#). My thanks to Dheeru Mundluru for spotting the bug.
- August 2014 Edited by Moez AbdelGawad, who offered to suggest corrections and to fix typos and grammatical mistakes.
- January 2015 Type inference was slightly modified in Java 8 which affected the example in [Technicalities.FAQ404](#). My thanks to Chad Berchek for bringing it to my attention and to Dan Smith of Oracle for explaining the type inference modification to me.
- 

## Copyright Notice

All text and content found at URLs starting with <http://www.AngelikaLanger.com/GenericsFAQ/> (collectively, "the Java Generics FAQ") are the sole property of Angelika Langer. Copyright @ 2004-2022 by [Angelika Langer](#). All rights reserved.

Except as specifically granted below, you may not modify, copy, publish, sell, display, transmit (in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise), adapt, distribute, store in a retrieval system, create derivative works, or in any other way use or exploit the contents of the Java Generics FAQ, without the prior consent of the author.

All rights, titles and interest, including copyrights and other applicable intellectual property rights, in any of the material belongs to the provider of the material. You do not acquire proprietary interest in such materials by accessing them on my web site. You must abide by all copyright notices and restrictions on use of the material accessed.

In particular, I do NOT grant permission to copy the Java Generics FAQ or any part of it to a public Web



server. Link to the original pages instead. (The problem with you putting a page on your server is that the search engines will find it and send my readers to your server instead. Thus they will be deprived of the most up-to-date version of the document.)

### **Non-commercial Use**

As a reader of the Java Generics FAQ, you are granted a nonexclusive, nontransferable, limited license to access and use for non-commercial (research and information) purposes the materials that are made available to you as the Java Generics FAQ. This license includes the right to download and print out one copy of the Java Generics FAQ so long as you neither change nor delete any author attribution, legend or copyright notice. I request that you do not break up the document and attribute the source in such a way that someone can find the most up-to-date version on the Web, e.g. by including a link to <http://www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html>. If you redistribute the hardcopy of the Java Generics FAQ, please send me [email](#) informing me of the usage.

Other forms of non-commercial use may be allowed with prior written permission. If permission is granted for non-commercial use, credit to the copyright owner must be displayed as prescribed by copyright owner in every exposure of text. To request non-commercial use of copyrighted materials, please [email](#) your request and provide the following information: your name, organization, and title, your request (the text or texts you wish to quote) and the non-commercial purpose of the use.

### **Commercial Use**

Commercial use on paper, electronic storage devices of all types, online via the Internet or any part of the World Wide Web or via any other electronic distribution is strictly prohibited without prior payment to the copyright owner, as well as attribution of ownership of copyright and URL in every exposure of text. To negotiate legal use of text owned and copyrighted by Angelika Langer, please send me [email](#).

### **Disclaimers**

The author has taken care in the preparation of this material, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the information or programs contained herein.

# Fundamentals of Java Generics

© Copyright 2004-2022 by Angelika Langer. All Rights Reserved.

[What are Java generics?](#)

[What is the primary purpose of Java generics?](#)

[What is the benefit of using Java generics?](#)

[What does type-safety mean?](#)

---

## What are Java generics?

*Java Generics, sometimes used as a plural noun (generics) and sometimes as a singular noun (Generics), is a language feature of Java that allows for the definition and use of generic types and methods.*

'Java Generics' is a technical term denoting a set of language features related to the definition and use of *generic types and methods*. Generic types or methods differ from regular types and methods in that they have type parameters.

Examples of generic types can be found in the Collections framework of the J2SE 5.0 platform libraries. A class like `LinkedList<E>` is a generic type. It has a type parameter `E` that represents the type of the elements stored in the list. Instead of just using a `LinkedList`, not saying anything about the type of elements the list contains, we can use a `LinkedList<String>` or a `LinkedList<Integer>`, thereby specifying that we mean a list of strings or integral values respectively.

Generic types are instantiated to form *parameterized types* by providing actual type arguments that replace the formal type parameters. A class like `LinkedList<E>` is a generic type, that has a type parameter `E`. Instantiations, such as `LinkedList<String>` or a `LinkedList<Integer>`, are called parameterized types, and `String` and `Integer` are the respective actual type arguments.

LINK TO THIS [Fundamentals.FAQ001](#)

REFERENCES [Language Features of Java Generics](#)

---

## What is the primary purpose of Java generics?

*Java generics were invented primarily for implementation of generic collections.*

The need for generic types stems mainly from the implementation and use of collections, like the ones in the Java Collections framework. Programmers often want to specify that a collection contains elements of a certain type, such as a list of integral values or a list of strings. The Collections framework in non-generic Java did not offer any homogenous collections of elements of the same type. Instead, all collections were holding `Object` references and for that reason they were potentially heterogenous, that is, a mix of objects of different types. This was also visible in the collection APIs: the non-generic collections accepted objects of arbitrary type for insertion into the collection and returned an `Object` reference when an element was retrieved from the collection (see package `java.util` in Java 1.4).

In non-generic Java, homogenous collections of elements had required implementation of different classes, such as a class `IntegerList` and a class `StringList` for holding integral values and strings respectively. Naturally, implementing a separate collection class for every conceivable element type is neither feasible nor desirable. A more reasonable goal is to have a single implementation of the collection class and use it to hold elements of different types. In other words, rather than implementing a class `IntegerList` and `StringList`,



we want to have one generic implementation `List` that can be easily used in either case, as well as in other unforeseen cases.

This is what generics are for: the implementation of one generic class can be instantiated for a variety of types. There is one generic class `List` in the generic collections framework (see package [java.util](#) in Java 5.0). It permits specification of `List<Integer>`, `List<String>`, etc. each of which is a homogenous collection of integral values, strings, etc. In generic Java, the generic class `List` is a so-called *generic class* that has a type parameter. Uses such as `List<Integer>` and `List<String>` are so-called *parameterized types*. They are *instantiations* of the generic class, where the type parameter is replaced by the concrete type arguments `Integer` and `String`.

The use of the Java generics language features were initially motivated by the need to have a mechanism for efficient implementation of homogenous collections, but the language feature is not restricted to collections. The J2SE 5.0 platform libraries contains numerous examples of generic types and methods that have nothing to do with collections. Examples are the weak and soft references in package [java.lang.ref](#), which are special purpose references to objects of a particular type represented by a type parameter. Or the interface `Callable` in package [java.util.concurrent](#), which represents a task and has a `call` method that returns a result of a particular type represented by a type parameter. Even class `Class` in package [java.lang](#) is a generic class since Java 5.0, whose type parameter denotes the type that the `Class` object represents.

LINK TO THIS [Fundamentals.FAQ002](#)

REFERENCES [What is the benefit of using Java generics?](#)  
[What is a parameterized or generic type?](#)  
[How is a generic type instantiated?](#)

---

## What is the benefit of using Java generics?

### *Early error detection at compile time.*

Using a parameterized type such as `LinkedList<String>`, instead of `LinkedList`, enables the compiler to perform more type checks and requires fewer dynamic casts. This way errors are detected earlier, in the sense that they are reported at compile-time by means of a compiler error message rather than at runtime by means of an exception.

Consider the example of a `LinkedList<String>`. The type `LinkedList<String>` expresses that the list is a homogenous list of elements of type `String`. Based on the stronger information the compiler performs type checks in order to ensure that a `LinkedList<String>` contains only strings as elements. Any attempt to add an alien element is rejected with a compiler error message.

Example (using a parameterized type):

```
LinkedList<String> list = new LinkedList<String>();
list.add("abc");           // fine
list.add(new Date());     // error
```

Using a plain `LinkedList`, the compiler had not issued any message and both elements would have been added to the list. This is because the non-parameterized `LinkedList` does not mandate that all elements must be of the same or any particular type. A non-parameterized list is a sequence of elements of type `Object` and hence arbitrary.

Same example (using a non-parameterized type):

```
LinkedList list = new LinkedList();
list.add("abc");           // fine
list.add(new Date());     // fine as well
```

Since it is ensured that a `LinkedList<String>` contains strings it is not necessary to cast an element retrieved from the list to type `String`.

Example (using a parameterized type):

```
LinkedList<String> list = new LinkedList<String>();
list.add("abc");
String s = list.get(0); // no cast needed
```

Using a plain `LinkedList`, there is no knowledge and no guarantee regarding the type of the element retrieved. All retrieval methods return an `Object` reference, which must be cast down to the actual type of the element retrieved.

Same example (using a non-parameterized type):

```
LinkedList list = new LinkedList();
list.add("abc");
String s = (String)list.get(0); // cast required
```

The cast would fail at runtime with a `ClassCastException` in case the element retrieved is not of type `String`. This type of runtime failure cannot happen with a parameterized list because the compiler already prevents insertion of an alien element into the sequence.

LINK TO THIS [Fundamentals.FAQ003](#)

REFERENCES [What is a parameterized or generic type?](#)  
[How is a generic type instantiated?](#)

---

## What does type-safety mean?

*In Java, a program is considered type-safe if it compiles without errors and warnings and does not raise any unexpected `ClassCastException`s at runtime.*

The idea is that a well-formed program enables the compiler to perform enough type checks based on static type information that no unexpected type error can occur at runtime. An unexpected type error in this sense is a `ClassCastException` being raised without any visible cast expression in the source code.

LINK TO THIS [Fundamentals.FAQ004](#)

REFERENCES [How does the compiler translate Java generics?](#)  
[Why does the compiler add casts when it translates generics?](#)  
[What is an "unchecked" warning?](#)

---

# Language Features of Java Generics

© Copyright 2004-2022 by Angelika Langer. All Rights Reserved.

## Which language features are related to Java generics?

---

### Generic Types

#### Fundamentals

- What is a parameterized or generic type?
- How do I define a generic type?
- Are there any types that cannot have type parameters?
- How is a generic type instantiated?
- Why do instantiations of a generic type share the same runtime type?
- Can I cast to a parameterized type?
- Can I use parameterized types in exception handling?
- Can generic types have static members?

#### Concrete Instantiations

- What is a concrete instantiation?
- Are different concrete instantiations of the same generic type compatible?
- Can I use a concrete parameterized type like any other type?
- Can I create an array whose component type is a concrete parameterized type?
- Can I declare a reference variable of an array type whose component type is a concrete parameterized type?
- How can I work around the restriction that there are no arrays whose component type is a concrete parameterized type?
- Why is there no class literal for the concrete parameterized type?

#### Raw Types

- What is the raw type?
- Why are raw types permitted?
- Can I use a raw type like any other type?

#### Wildcard Instantiations

- What is a wildcard instantiation?
- What is the unbounded wildcard instantiation?
- What is the difference between the unbounded wildcard parameterized type and the raw type?
- Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?
- Can I use a wildcard parameterized type like any other type?
- Can I create an object whose type is a wildcard parameterized type?
- Can I create an array whose component type is a wildcard parameterized type?
- Can I declare a reference variable of an array type whose component type is a bounded wildcard parameterized type?
- Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?
- Can I declare a reference variable of an array type whose component type is an unbounded wildcard parameterized type?
- Can I derive from a wildcard parameterized type?
- Why is there no class literal for wildcard parameterized type?

---

## Generic Methods

### Fundamentals

- [What is a generic method?](#)
- [How do I invoke a generic method?](#)

---

## Type Parameters

### Fundamentals

- [What is a type parameter?](#)
- [Where is a type parameter visible \(or invisible\)?](#)

### Type Parameter Bounds

- [What is a type parameter bound?](#)
- [Which types are permitted as type parameter bounds?](#)
- [Can I use a type parameter as a type parameter bound?](#)
- [Can I use different instantiations of a same generic type as bounds of a type parameter?](#)
- [How can work around the restriction that a type parameter cannot have different instantiations of a same generic type as its bounds?](#)
- [Does a bound that is a class type give access to all its public members?](#)
- [How do I decrypt "Enum<E extends Enum<E>>"?](#)

### Usage

- [Can I use a type parameter like a type?](#)
- [Can I create an object whose type is a type parameter?](#)
- [Can I create an array whose component type is a type parameter?](#)
- [Can I cast to the type that the type parameter stands for?](#)
- [Can I use a type parameter in exception handling?](#)
- [Can I derive from a type parameter?](#)
- [Why is there no class literal for a type parameter?](#)

### Scope

- [Where is a type parameter visible \(or invisible\)?](#)
- [Can I use a type parameter as part of its own bounds or in the declaration of other type parameters?](#)
- [Can I use the type parameter of an outer type as part of the bounds of the type parameter of an inner type or a method?](#)

### Static Context

- [Is there one instances of a static field per instantiation of a generic type?](#)
- [Why can't I use a type parameter in any static context of the generic class?](#)

---

## Type Arguments

### Fundamentals

- [What is a type argument?](#)

- [Which types are permitted as type arguments?](#)
- [Are primitive types permitted as type arguments?](#)
- [Are wildcards permitted as type arguments?](#)
- [Are type parameters permitted as type arguments?](#)
- [Do type parameter bounds restrict the set of types that can be used as type arguments?](#)
- [Do I have to specify a type argument when I want to use a generic type?](#)
- [Do I have to specify a type argument when I want to invoke a generic method?](#)

### Wildcards

- [What is a wildcard?](#)
- [What is an unbounded wildcard?](#)
- [What is a bounded wildcard?](#)
- [What do multi-level wildcards mean?](#)
- [If a wildcard appears repeatedly in a type argument section, does it stand for the same type?](#)

### Wildcard Bounds

- [What is a wildcard bound?](#)
- [Which types are permitted as wildcard bounds?](#)
- [What is the difference between a wildcard bound and a type parameter bound?](#)

---

## **Which language features are related to Java generics?**

### *Features for definition and use of generic types and methods.*

Java Generics support definition and use of generic types and methods. It provides language features for the following purposes:

- definition of a generic type
- definition of a generic method
  
- type parameters
  - type parameter bounds
- type arguments
  - wildcards
  - wildcard bounds
  - wildcard capture
- instantiation of a generic type
  - raw type
  - concrete instantiation
  - wildcard instantiation
- instantiation of a generic method
  - automatic type inference
  - explicit type argument specification

**LINK TO THIS**      [LanguageFeatures.FAQ001](#)

**REFERENCES**      [What is a parameterized or generic type?](#)  
[What is a generic method?](#)  
[What is a type parameter?](#)  
[What is a bounded type parameter?](#)

[What is a type argument?](#)

[What is a wildcard?](#)

[What is a bounded wildcard?](#)

[What is the capture of a wildcard?](#)

[How is a generic type instantiated?](#)

[What is the raw type?](#)

[What is a concrete instantiation?](#)

[What is a wildcard instantiation?](#)

[How is a generic method instantiated?](#)

[What is type argument inference?](#)

[What is explicit type argument specification?](#)

# Generic And Parameterized Types

© Copyright 2004-2022 by Angelika Langer. All Rights Reserved.

## Fundamentals

- [What is a parameterized or generic type?](#)
- [How do I define a generic type?](#)
- [Are there any types that cannot have type parameters?](#)
- [How is a generic type instantiated?](#)
- [Why do instantiations of a generic type share the same runtime type?](#)
- [Can I cast to a parameterized type?](#)
- [Can I use parameterized types in exception handling?](#)
- [Can generic types have static members?](#)

## Concrete Instantiations

- [What is a concrete instantiation of a generic type?](#)
- [Are different concrete instantiations of the same generic type compatible?](#)
- [Can I use a concrete parameterized type like any other type?](#)
- [Can I create an array whose component type is a concrete parameterized type?](#)
- [Can I declare a reference variable of an array type whose component type is a concrete parameterized type?](#)
- [How can I work around the restriction that there are no arrays whose component type is a concrete parameterized type?](#)
- [Why is there no class literal for the concrete parameterized type?](#)

## Raw Types

- [What is the raw type?](#)
- [Why are raw types permitted?](#)
- [Can I use a raw type like any other type?](#)

## Wildcard Instantiations

- [What is a wildcard instantiation?](#)
- [What is the unbounded wildcard instantiation?](#)
- [What is the difference between the unbounded wildcard parameterized type and the raw type?](#)
- [Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)
- [Can I use a wildcard instantiation like any other type?](#)
- [Can I create an object whose type is a wildcard parameterized type?](#)
- [Can I create an array whose component type is a wildcard parameterized type?](#)
- [Can I declare a reference variable of an array type whose component type is a bounded wildcard parameterized type?](#)
- [Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)
- [Can I declare a reference variable of an array type whose component type is an unbounded wildcard parameterized type?](#)
- [Can I derive from a wildcard parameterized type?](#)
- [Why is there no class literal for wildcard parameterized type?](#)

---

## Generic And Parameterized Types

---

---

## Fundamentals

---

### What is a parameterized or generic type?

*A generic type is a type with formal type parameters. A parameterized type is an instantiation of a generic type with actual type arguments.*

A *generic type* is a reference type that has one or more type parameters. These type parameters are later replaced by type arguments when the generic type is instantiated (or *declared*).

Example (of a generic type):

```
interface Collection<E> {
    public void add (E x);
    public Iterator<E> iterator();
}
```

The interface `Collection` has one type parameter `E`. The type parameter `E` is a place holder that will later be replaced by a type argument when the generic type is instantiated and used. The instantiation of a generic type with actual type arguments is called a *parameterized type*.

Example (of a parameterized type):

```
Collection<String> coll = new LinkedList<String>();
```

The declaration `Collection<String>` denotes a parameterized type, which is an instantiation of the generic type `Collection`, where the place holder `E` has been replaced by the concrete type `String`.

LINK TO THIS [GenericTypes.FAQ001](#)

REFERENCES [What is a type parameter?](#)

---

### How do I define a generic type?

*Like a regular type, but with a type parameter declaration attached.*

A generic type is a reference type that has one or more type parameters. In the definition of the generic type, the type parameter section follows the type name. It is a comma separated list of identifiers and is delimited by angle brackets.

Example (of a generic type):

```
class Pair<X,Y> {
    private X first;
    private Y second;

    public Pair(X a1, Y a2) {
        first = a1;
        second = a2;
    }
    public X getFirst() { return first; }
    public Y getSecond() { return second; }
```



```
public void setFirst(X arg) { first = arg; }
public void setSecond(Y arg) { second = arg; }
}
```

The class `Pair` has two type parameters `X` and `Y`. They are replaced by type arguments when the generic type `Pair` is instantiated. For instance, in the declaration `Pair<String, Date>` the type parameter `X` is replaced by the type argument `String` and `Y` is replaced by `Date`.

The scope of the identifiers `X` and `Y` is the entire definition of the class. In this scope the two type parameters `X` and `Y` are used like they were types (with some restrictions). In the example above, the type parameters are used as the argument and return type of instance methods and the types of instance fields.

Type parameters can be declared with bounds. Bounds give access to methods of the unknown type that the type parameter stands for. In our example, we do not invoke any methods of the unknown types `X` and `Y`. For this reason, the two type parameters are unbounded.

LINK TO THIS [GenericTypes.FAQ002](#)

REFERENCES [What is a type parameter?](#)  
[What is a bounded type parameter?](#)  
[What is a type parameter bound?](#)

---

## Are there any types that cannot have type parameters?

*All types, except enum types, anonymous inner classes and exception classes, can be generic..*

Almost all reference types can be generic. This includes classes, interfaces, nested (static) classes, nested interfaces, inner (non-static) classes, and local classes.

The following types cannot be generic:

*Anonymous inner classes.* They can implement a parameterized interface or extend a parameterized class, but they cannot themselves be generic classes. A generic anonymous class would be nonsensical. Anonymous classes do not have a name, but the name of a generic class is needed for declaring an instantiation of the class and providing the type arguments. Hence, generic anonymous classes would be pointless.

*Exception types.* A generic class must not directly or indirectly be derived from class `Throwable`. Generic exception or error types are disallowed because the exception handling mechanism is a runtime mechanism and the Java virtual machine does not know anything about Java generics. The JVM would not be capable of distinguishing between different instantiations of a generic exception type. Hence, generic exception types would be pointless.

*Enum types.* Enum types cannot have type parameters. Conceptually, an enum type and its enum values are static. Since type parameters cannot be used in any static context, the parameterization of an enum type would be pointless.

LINK TO THIS [GenericTypes.FAQ003](#)

REFERENCES [Can I use generic / parameterized types in exception handling?](#)  
[Why are generic exception and error types illegal?](#)  
[Why are generic enum types illegal?](#)

---

## How is a generic type instantiated?

*By providing a type argument per type parameter.*

In order to use a generic type we must provide one type argument per type parameter that was declared for the

generic type. The type argument list is a comma separated list that is delimited by angle brackets and follows the type name. The result is a so-called parameterized type.

Example (of a generic type):

```
class Pair<X,Y> {
    private X first;
    private Y second;

    public Pair(X a1, Y a2) {
        first = a1;
        second = a2;
    }
    public X getFirst() { return first; }
    public Y getSecond() { return second; }
    public void setFirst(X arg) { first = arg; }
    public void setSecond(Y arg) { second = arg; }
}
```

If we want to use the generic type `Pair` we must specify the type arguments that shall replace the place holders `x` and `y`. A type argument can be a concrete reference type, such as `String`, `Long`, `Date`, etc.

Example (of a concrete parameterized type):

```
public void printPair(Pair<String,Long> pair) {
    System.out.println("(" + pair.getFirst() + ", " + pair.getSecond() + ")");
}

Pair<String,Long> limit = new Pair<String,Long>("maximum",1024L);
printPair(limit);
```

The instantiation `Pair<String,Long>` is a concrete parameterized type and it can be used like a regular reference type (with a couple of restrictions that are discussed later). In the example, we have been using the concrete parameterized type as argument type of a method, as type of a reference variable, and in a new expression for creation of an object.

In addition to concrete instantiation there so-called *wildcard instantiations*. They do not have concrete types as type arguments, but so-called *wildcards*. A wildcard is a syntactic construct with a "?" that denotes not just one type, but a family of types. In its simplest form a wildcard is just a question mark and stands for "all types".

Example (of a wildcard parameterized type):

```
public void printPair(Pair<?,?> pair) {
    System.out.println("(" + pair.getFirst() + ", " + pair.getSecond() + ")");
}

Pair<?,?> limit = new Pair<String,Long>("maximum",1024L);
printPair(limit);
```

The declaration `Pair<?,?>` is an example of a wildcard parameterized type, where both type arguments are wildcards. Each question mark stands for a *separate* representative from the family of "all types". The resulting family of instantiations comprises *all* instantiations of the generic type `Pair`. (Note: the concrete type arguments of the family members need *not* be identical; each "?" stands for a separate type.) A reference variable or method parameter whose type is a wildcard parameterized type, such as `limit` and `pair` in the example, can refer to any member of the family of types that the wildcard denotes.

It is permitted to leave out the type arguments altogether and not specify type arguments at all. A generic type without type arguments is called *raw type* and is only allowed for reasons of compatibility with non-generic Java code. Use of raw types is discouraged. The Java Language Specification even states that it is possible that future versions of the Java programming language will disallow the use of raw types.

**LINK TO THIS**[GenericTypes.FAQ004](#)**REFERENCES**

[What is a type argument?](#)  
[Which types are permitted as type arguments?](#)  
[What is a wildcard?](#)  
[What is a concrete parameterized type?](#)  
[What is a wildcard parameterized type?](#)  
[Can I use a concrete parameterized type like any other type?](#)  
[Can I use a wildcard parameterized like any other type?](#)  
[What is the raw type?](#)

---

## Why do instantiations of a generic type share the same runtime type?

*Because of type erasure.*

The compiler translates generic and parameterized types by a technique called *type erasure*. Basically, it elides all information related to type parameters and type arguments. For instance, the parameterized type `List<String>` is translated to type `List`, which is the so-called *raw type*. The same happens for the parameterized type `List<Long>`; it also appears as `List` in the bytecode.

After translation by type erasure, all information regarding type parameters and type arguments has disappeared. As a result, all instantiations of the same generic type share the same runtime type, namely the raw type.

Example (printing the runtime type of two parameterized types):

```
System.out.println("runtime type of ArrayList<String>: "+new  
ArrayList<String>().getClass());  
System.out.println("runtime type of ArrayList<Long> : "+new  
ArrayList<Long>().getClass());
```

---

```
prints: runtime type of ArrayList<String>: class java.util.ArrayList  
runtime type of ArrayList<Long> : class java.util.ArrayList
```

The example illustrates that `ArrayList<String>` and `ArrayList<Long>` share the runtime type `ArrayList`.

**LINK TO THIS**[GenericTypes.FAQ005](#)**REFERENCES**

[How does the compiler translate Java generics?](#)  
[What is type erasure?](#)  
[What is the raw type?](#)

---

## Can I cast to a parameterized type?

*Yes, you can, but under certain circumstances it is not type-safe and the compiler issues an "unchecked" warning.*

All instantiations of a generic type share the same runtime type representation, namely the representation of the raw type. For instance, the instantiations of a generic type `List`, such as `List<Date>`, `List<String>`, `List<Long>`, etc. have different static types at compile time, but the same dynamic type `List` at runtime.

A cast consists of two parts:

- a static type check performed by the compiler at compile time and
- a dynamic type check performed by the virtual machine at runtime.

The static part sorts out nonsensical casts, that cannot succeed, such as the cast from `String` to `Date` or from `List<String>` to `List<Date>`.

The dynamic part uses the runtime type information and performs a type check at runtime. It raises a `ClassCastException` if the dynamic type of the object is not the target type (or a subtype of the target type) of the cast. Examples of casts with a dynamic part are the cast from `Object` to `String` or from `Object` to `List<String>`. These are the so-called downcasts, from a supertype down to a subtype.

Not all casts have a dynamic part. Some casts are just static casts and require no type check at runtime. Examples are the casts between primitive types, such as the cast from `long` to `int` or `byte` to `char`. Another example of static casts are the so-called upcasts, from a subtype up to a supertype, such as the casts from `String` to `Object` or from `LinkedList<String>` to `List<String>`. Upcasts are casts that are permitted, but not required. They are automatic conversions that the compiler performs implicitly, even without an explicit cast expression in the source code, which means, the cast is not required and usually omitted. However, if an upcast appears somewhere in the source code then it is a purely static cast that does not have a dynamic part.

Type casts with a dynamic part are potentially unsafe, when the target type of the cast is a parameterized type. The runtime type information of a parameterized type is non-exact, because all instantiations of the same generic type share the same runtime type representation. The virtual machine cannot distinguish between different instantiations of the same generic type. Under these circumstances the dynamic part of a cast can succeed although it should not.

Example (of unchecked cast):

```
void m1() {
    List<Date> list = new ArrayList<Date>();
    ...
    m2(list);
}
void m2(Object arg) {
    ...
    List<String> list = (List<String>) arg;    // unchecked warning
    ...
    m3(list);
    ...
}
void m3(List<String> list) {
    ...
    String s = list.get(0);    // ClassCastException
    ...
}
```

The cast from `Object` to `List<String>` in method `m2` looks like a cast to `List<String>`, but actually is a cast from `Object` to the raw type `List`. It would succeed even if the object referred to were a `List<Date>` instead of a `List<String>`.

After this successful cast we have a reference variable of type `List<String>` which refers to an object of type `List<Date>`. When we retrieve elements from that list we would expect `Strings`, but in fact we receive `Dates` - and a `ClassCastException` will occur in a place where nobody had expected it.

We are prepared to cope with `ClassCastExceptions` when there is a cast expression in the source code, but we do not expect `ClassCastExceptions` when we extract an element from a list of strings. This sort of unexpected `ClassCastException` is considered a violation of the type-safety principle. In order to draw attention to the potentially unsafe cast the compiler issues an "unchecked" warning when it translates the dubious cast expression.

As a result, the compiler emits "unchecked" warnings for every dynamic cast whose target type is a parameterized type. Note that an upcast whose target type is a parameterized type does *not* lead to an

"unchecked" warning, because the upcast has no dynamic part.

LINK TO THIS [GenericTypes.FAQ006](#)

REFERENCES [Why do instantiations of the same generic type share the same runtime type?](#)  
[What does type-safety mean?](#)  
[What is the type erasure of a parameterized type?](#)

---

## Can I use parameterized types in exception handling?

*No. Exception and error types must not be generic.*

It is illegal to define generic type that are directly or indirectly derived from class `Throwable`. Consequently, no parameterized types appear anywhere in exception handling.

LINK TO THIS [GenericTypes.FAQ007](#)

REFERENCES [Why are generic exception and error types illegal?](#)

---

## Can generic types have static members?

*Yes.*

Generic types can have static members, including static fields, static methods and static nested types. Each of these static members exists once per enclosing type, that is, independently of the number of objects of the enclosing type and regardless of the number of instantiations of the generic type that may be used somewhere in the program. The name of the static member consists - as is usual for static members - of the scope (packages and enclosing type) and the member's name. If the enclosing type is generic, then the type in the scope qualification must be the raw type, not a parameterized type.

LINK TO THIS [GenericTypes.FAQ008](#)

REFERENCES [How do I refer to static members of a generic or parameterized type?](#)  
[How do I refer to a \(non-static\) inner class of a generic or parameterized type?](#)  
[How do I refer to an interface type nested into a generic or parameterized type?](#)  
[How do I refer to an enum type nested into a generic or parameterized type?](#)  
[Can I import a particular parameterized type?](#)

---

---

## Concrete Instantiations

---

### What is a concrete parameterized type?

*An instantiation of a generic type where all type arguments are concrete types rather than wildcards.*

Examples of concrete parameterized types are `List<String>`, `Map<String,Date>`, but not `List<? extends Number>` OR `Map<String,?>`.

LINK TO THIS [GenericTypes.FAQ101](#)

REFERENCES [What is a wildcard?](#)

## Is List<Object> a supertype of List<String>?

*No, different instantiations of the same generic type for different concrete type arguments have no type relationship.*

It is sometimes expected that a List<Object> would be a supertype of a List<String>, because Object is a supertype of String. This expectation stems from the fact that such a type relationship exists for arrays: Object[] is a supertype of String[], because Object is a supertype of String. (This type relationship is known as *covariance*.) The super-subtype-relationship of the component types extends into the corresponding array types. No such a type relationship exists for instantiations of generic types. (Parameterized types are *not* covariant.)

The lack of a super-subtype-relationship among instantiations of the same generic type has various consequences. Here is an example.

Example:

```
void printAll(ArrayList<Object> c) {
    for (Object o : c)
        System.out.println(o);
}

ArrayList<String> list = new ArrayList<String>();
... fill list ...
printAll(list);    // error
```

A ArrayList<String> object cannot be passed as argument to a method that asks for a ArrayList<Object> because the two types are instantiations of the same generic type, but for *different* type arguments, and for this reason they are not compatible with each other.

On the other hand, instantiations of different generic types for *the same* type argument can be compatible.

Example:

```
void printAll(Collection<Object> c) {
    for (Object o : c)
        System.out.println(o);
}

List<Object> list = new ArrayList<Object>();
... fill list ...
printAll(list);    // fine
```

A List<Object> is compatible to a Collection<Object> because the two types are instantiations of a generic supertype and its generic subtype and the instantiations are for the same type argument Object.

Compatibility between instantiations of the same generic type exist only among wildcard instantiations and concrete instantiations that belong to the family of instantiations that the wildcard instantiation denotes.

LINK TO THIS [GenericTypes.FAQ102](#)

### REFERENCES

[What is a concrete parameterized type?](#)  
[What is a wildcard parameterized type?](#)  
[How do parameterized types fit into the Java type system?](#)  
[How does the raw type relate to instantiations of the corresponding generic type?](#)  
[How do instantiations of a generic type relate to instantiations of other generic types?](#)

## Can I use a concrete parameterized type like any other type?

*Almost.*

Concrete parameterized types are concrete instantiations of a generic type. They are almost like types; there are only a few restrictions. They can NOT be used for the following purposes:

- for creation of arrays
- in exception handling
- in a class literal
- in an instanceof expression

LINK TO THIS [GenericTypes.FAQ103](#)

REFERENCES [Can I create an array whose component type is a concrete parameterized type?](#)  
[Can I use parameterized types in exception handling?](#)  
[Why is there no class literal for the concrete parameterized type?](#)

---

## Can I create an array whose component type is a concrete parameterized type?

*No, because it is not type-safe.*

Arrays are covariant, which means that an array of supertype references is a supertype of an array of subtype references. That is, `Object[]` is a supertype of `String[]` and a string array can be accessed through a reference variable of type `Object[]`.

Example (of covariant arrays):

```
Object[] objArr = new String[10]; // fine
objArr[0] = new String();
```

In addition, arrays carry runtime type information about their component type, that is, about the type of the elements contained. The runtime type information regarding the component type is used when elements are stored in an array in order to ensure that no "alien" elements can be inserted.

Example (of array store check):

```
Object[] objArr = new String[10];
objArr[0] = new Long(0L); // compiles; fails at runtime with ArrayStoreException
```

The reference variable of type `Object[]` refers to a `String[]`, which means that only strings are permitted as elements of the array. When an element is inserted into the array, the information about the array's component type is used to perform a type check - the so-called *array store check*. In our example the array store check will fail because we are trying to add a `Long` to an array of `Strings`. Failure of the array store check is reported by means of a `ArrayStoreException`.

Problems arise when an array holds elements whose type is a concrete parameterized type. Because of type erasure, parameterized types do not have exact runtime type information. As a consequence, the array store check does not work because it uses the dynamic type information regarding the array's (non-exact) component type for the array store check.



Example (of array store check in case of parameterized component type):

```
Pair<Integer,Integer>[] intPairArr = new Pair<Integer,Integer>[10]; // illegal
Object[] objArr = intPairArr;
objArr[0] = new Pair<String,String>("", ""); // should fail, but would succeed
```

If arrays of concrete parameterized types were allowed, then a reference variable of type `Object[]` could refer to a `Pair<Integer,Integer>[]`, as shown in the example. At runtime an array store check must be performed when an array element is added to the array. Since we are trying to add a `Pair<String,String>` to a `Pair<Integer,Integer>[]` we would expect that the type check fails. However, the JVM cannot detect any type mismatch here: at runtime, after type erasure, `objArr` would have the dynamic type `Pair[]` and the element to be stored has the matching dynamic type `Pair`. Hence the store check succeeds, although it should not.

If it were permitted to declare arrays that holds elements whose type is a concrete parameterized type we would end up in an unacceptable situation. The array in our example would contain different types of pairs instead of pairs of the same type. This is in contradiction to the expectation that arrays hold elements of the same type (or subtypes thereof). This undesired situation would most likely lead to program failure some time later, perhaps when a method is invoked on the array elements.

Example (of subsequent failure):

```
Pair<Integer,Integer>[] intPairArr = new Pair<Integer,Integer>[10]; // illegal
Object[] objArr = intPairArr;
objArr[0] = new Pair<String,String>("", ""); // should fail, but would succeed

Integer i = intPairArr[0].getFirst(); // fails at runtime with ClassCastException
```

The method `getFirst` is applied to the first element of the array and it returns a `String` instead of an `Integer` because the first element in the array `intPairArr` is a pair of strings, and not a pair of integers as one would expect. The innocent-looking assignment to the `Integer` variable `i` will fail with a `ClassCastException`, although no cast expression is present in the source code. Such an unexpected `ClassCastException` is considered a violation of type-safety.

In order to prevent programs that are not type-safe all arrays holding elements whose type is a concrete parameterized type are illegal. For the same reason, arrays holding elements whose type is a wildcard parameterized type are banned, too. Only arrays with an unbounded wildcard parameterized type as the component type are permitted. More generally, reifiable types are permitted as component type of arrays, while arrays with a non-reifiable component type are illegal.

LINK TO THIS [GenericTypes.FAQ104](#)

#### REFERENCES

[What does type-safety mean?](#)  
[Can I declare a reference variable of an array type whose component type is a concrete parameterized type?](#)  
[Can I create an array whose component type is a wildcard parameterized type?](#)  
[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)  
[What is a reifiable type?](#)

---

## Can I declare a reference variable of an array type whose component type is a concrete parameterized type?

*Yes, you can, but you should not, because it is neither helpful nor type-safe.*

You can declare a reference variable of an array type whose component type is a concrete parameterized type. Arrays of such a type must not be created. Hence, this reference variable cannot refer to an array of its type. All that it can refer to is `null`, an array whose component type is a non-parameterized subtype of the concrete parameterized type, or an array whose component type is the corresponding raw type. Neither of these cases is overly useful, yet they are permitted.



Example (of an array reference variable with parameterized component type):

```
Pair<String,String>[] arr = null; // fine
arr = new Pair<String,String>[2]; // error: generic array creation
```

The code snippet shows that a reference variable of type `Pair<String,String>[]` can be declared, but the creation of such an array is rejected. But we can have the reference variable of type `Pair<String,String>[]` refer to an array of a non-parameterized subtype.

Example (of another array reference variable with parameterized component type):

```
class Name extends Pair<String,String> { ... }

Pair<String,String>[] arr = new Name[2]; // fine
```

Which raises the question: how useful is such an array variable if it never refers to an array of its type? Let us consider an example.

Example (of an array reference variable referring to array of subtypes; not recommended):

```
void printArrayOfStringPairs(Pair<String,String>[] pa) {
    for (Pair<String,String> p : pa)
        if (p != null)
            System.out.println(p.getFirst()+" "+p.getSecond());
}

Pair<String,String>[] createArrayOfStringPairs() {
    Pair<String,String>[] arr = new Name[2];
    arr[0] = new Name("Angelika","Langer"); // fine
    arr[1] = new Pair<String,String>("a","b"); // fine (causes ArrayStoreException)

    return arr;
}

void extractStringPairsFromArray(Pair<String,String>[] arr) {
    Name name = (Name)arr[0]; // fine
    Pair<String,String> p1 = arr[1]; // fine
}

void test() {
    Pair<String,String>[] arr = createArrayOfStringPairs();
    printArrayOfStringPairs(arr);
    extractStringPairsFromArray(arr);
}
```

The example shows that a reference variable of type `Pair<String,String>[]` can refer to an array of type `Name[]`, where `Name` is a non-parameterized subtype of `Pair<String,String>[]`. However, using a reference variable of type `Pair<String,String>[]` offers no advantage over using a variable of the actual type `Name[]`. Quite the converse; it is an invitation for making mistakes.

For instance, in the `createArrayOfStringPairs` method the compiler would permit code for insertion of elements of type `Pair<String,String>` into the array though the reference variable of type `Pair<String,String>[]`. Yet, at runtime, this insertion will always fail with an `ArrayStoreException` because we are trying to insert a `Pair` into a `Name[]`. The same would happen if we tried to insert a raw type `Pair` into the array; it would compile with an "unchecked" warning and would fail at runtime with an `ArrayStoreException`. If we used `Name[]` instead of `Pair<String,String>[]` the debatable insertions would not compile in the first place.

Also, remember that a variable of type `Pair<String,String>[]` can never refer to an array that contains elements of type `Pair<String,String>`. When we want to recover the actual type of the array elements, which is the subtype `Name` in our example, we must cast down from `Pair<String,String>` to `Name`, as is demonstrated in the `extractStringPairsFromArray` method. Here again, using a variable of type `Name[]`

would be much clearer.

Example (improved):

```
void printArrayOfStringPairs(Pair<String,String>[] pa) {
    for (Pair<String,String> p : pa)
        if (p != null)
            System.out.println(p.getFirst()+" "+p.getSecond());
}
Name[] createArrayOfStringPairs() {
    Name[] arr = new Name[2];
    arr[0] = new Name("Angelika","Langer"); // fine
    arr[1] = new Pair<String,String>("a","b"); // error
    return arr;
}
void extractStringPairsFromArray(Name[] arr) {
    Name name = arr[0]; // fine (needs no cast)
    Pair<String,String> p1 = arr[1]; // fine
}
void test() {
    Name[] arr = createArrayOfStringPairs();
    printArrayOfStringPairs(arr);
    extractStringPairsFromArray(arr);
}
```

Since an array reference variable whose component type is a concrete parameterized type can never refer to an array of its type, such a reference variable does not really make sense. Matters are even worse than in the example discussed above, when we try to have the variable refer to an array of the raw type instead of a subtype. First, it leads to numerous "unchecked" warnings because we are mixing use of raw and parameterized type. Secondly, and more importantly, this approach is not type-safe and suffers from all the deficiencies that lead to the ban of arrays of concrete instantiation in the first place.

No matter how you put it, you should better refrain from using array reference variable whose component type is a concrete parameterized type. Note, that the same holds for array reference variable whose component type is a wildcard parameterized type. Only array reference variable whose component type is an unbounded wildcard parameterized type make sense. This is because an unbounded wildcard parameterized type is a reifiable type and arrays with a reifiable component type can be created; the array reference variable can refer to an array of its type and the deficiencies discussed above simply do not exist for unbounded wildcard arrays.

LINK TO THIS [GenericTypes.FAQ104A](#)

REFERENCES

[What does type-safety mean?](#)

[Can I create an array whose component type is a concrete parameterized type?](#)

[Can I declare a reference variable of an array type whose component type is a bounded wildcard parameterized type?](#)

[Can I create an array whose component type is a wildcard parameterized type?](#)

[Can I declare a reference variable of an array type whose component type is an unbounded wildcard parameterized type?](#)

[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)

[What is a reifiable type?](#)

---

## How can I work around the restriction that there are no arrays whose component type is a concrete parameterized type?

*You can use arrays of raw types, arrays of unbounded wildcard parameterized types, or collections of concrete parameterized types as a workaround.*

Arrays holding elements whose type is a concrete parameterized type are illegal.

Example (of illegal array type):

```
static void test() {
    Pair<Integer,Integer>[] intPairArr = new Pair<Integer,Integer>[10]; // error
    addElements(intPairArr);
    Pair<Integer,Integer> pair = intPairArr[1];
    Integer i = pair.getFirst();
    pair.setSecond(i);
}
static void addElements(Object[] objArr) {
    objArr[0] = new Pair<Integer,Integer>(0,0);
    objArr[1] = new Pair<String,String>("", ""); // should fail with
    ArrayStoreException
}
```

The compiler prohibits creation of arrays whose component type is a concrete parameterized type, like `Pair<Integer,Integer>` in our example. We discussed in the preceding entry why is it reasonable that the compiler qualifies a `Pair<Integer,Integer>[]` as illegal. The key problem is that compiler and runtime system must ensure that an array is a *homogenous* sequence of elements of the same type. One of the type checks, namely the array-store-check performed by the virtual machine at runtime, fails to detect the offending insertion of an alien element. In the example the second insertion in the `addElements` method should fail, because we are adding a pair of strings to an array of integral values, but it does not fail as expected. The reasons were discussed in the preceding entry.

If we cannot use arrays holding elements whose type is a concrete parameterized type, what do we use as a workaround?

Let us consider 3 conceivable workarounds:

- array of raw type
- array of unbounded wildcard parameterized type
- collection instead of array

---

Raw types and unbounded wildcard parameterized type are permitted as component type of arrays. Hence they would be alternatives.

Example (of array of raw type):

```
static void test() {
    Pair[] intPairArr = new Pair[10];
    addElements(intPairArr);
    Pair<Integer,Integer> pair = intPairArr[1]; // unchecked warning
    Integer i = pair.getFirst(); // fails with
    ClassCastException
    pair.setSecond(i);
}
static void addElements(Object[] objArr) {
    objArr[0] = new Pair<Integer,Integer>(0,0);
    objArr[1] = new Pair<String,String>("", ""); // should fail, but succeeds
}
```

Use of the raw type, instead of a parameterized type, as the component type of an array, is permitted. The downside is that we can stuff any type of pair into the raw type array. There is no guarantee that a `Pair[]` is homogenous in the sense that it contains only pairs of the same type. Instead the `Pair[]` can contain a mix of arbitrary pair types.

This has numerous side effects. When elements are fetched from the `Pair[]` only raw type `Pair` references are received. Using raw type `Pairs` leads to unchecked warnings in various situations, for instance, when we try to

access the pair member or, like in the example, when we assign the `Pair` to the more specific `Pair<Integer, Integer>`, that we really wanted to use.

Let us see whether an array of an unbounded wildcard parameterized type would be a better choice.

Example (of array of unbounded wildcard parameterized type):

```
static void test() {
    Pair<?,?>[] intPairArr = new Pair<?,?>[10];
    addElements(intPairArr);
    Pair<Integer,Integer> pair = intPairArr[1]; // error
    Integer i = pair.getFirst();
    pair.setSecond(i);
}
static void addElements(Object[] objArr) {
    objArr[0] = new Pair<Integer,Integer>(0,0);
    objArr[1] = new Pair<String,String>("", ""); // should fail, but succeeds
}
```

---

```
error: incompatible types
found   : Pair<?,?>
required: Pair<java.lang.Integer, java.lang.Integer>
    Pair<Integer,Integer> pair = intPairArr[1];
                                             ^
```

A `Pair<?,?>[]` contains a mix of arbitrary pair types; it is not homogenous and semantically similar to the raw type array `Pair[]`. When we retrieve elements from the array we receive references of type `Pair<?,?>`, instead of type `Pair` in the raw type case. The key difference is that the compiler issues an error for the wildcard pair where it issues "unchecked" warnings for the raw type pair. In our example, we cannot assign the `Pair<?,?>` to the more specific `Pair<Integer, Integer>`, that we really wanted to use. Also, various operations on the `Pair<?,?>` would be rejected as errors.

As we can see, arrays of raw types and unbounded wildcard parameterized types are very different from the illegal arrays of a concrete parameterized type. An array of a concrete wildcard parameterized type would be a *homogenous* sequence of elements of the exact same type. In contrast, arrays of raw types and unbounded wildcard parameterized type are *heterogenous* sequences of elements of different types. The compiler cannot prevent that they contain different instantiations of the generic type.

By using arrays of raw types or unbounded wildcard parameterized types we give away the static type checks that a homogenous sequence would come with. As a result we must use explicit casts or we risk unexpected `ClassCastException`s. In the case of the unbounded wildcard parameterized type we are additionally restricted in how we can use the array elements, because the compiler prevents certain operations on the unbounded wildcard parameterized type. In essence, arrays of raw types and unbounded wildcard parameterized types are semantically very different from what we would express with an array of a concrete wildcard parameterized type. For this reason they are not a good workaround and only acceptable when the superior efficiency of arrays (as compared to collections) is of paramount importance.

---

While arrays of concrete parameterized types are illegal, collections of concrete parameterized types are permitted.

Example (using collections):

```
static void test() {
    ArrayList<Pair<Integer,Integer>> intPairArr = new
    ArrayList<Pair<Integer,Integer>>(10);
    addElements(intPairArr);
    Pair<Integer,Integer> pair = intPairArr.get(1);
}
```

```

Integer i = pair.getFirst();
pair.setSecond(i);
}
static void addElements(List<?> objArr) {
    objArr.add(0, new Pair<Integer, Integer>(0, 0)); // error
    objArr.add(1, new Pair<String, String>("", "")); // error
}

```

---

```

error: cannot find symbol
symbol   : method add(int,Pair<java.lang.Integer,java.lang.Integer>)
location: interface java.util.List<capture of ?>
    objArr.add(0,new Pair<Integer,Integer>(0,0));
            ^

error: cannot find symbol
symbol   : method add(int,Pair<java.lang.String,java.lang.String>)
location: interface java.util.List<capture of ?>
    objArr.add(1,new Pair<String,String>("", ""));
            ^

```

A collection of a concrete parameterized type is a *homogenous* sequence of elements and the compiler prevents any attempt to add alien elements by means of static type checks. To this regard it is semantically similar to the illegal array, but otherwise collections are very different from arrays. They have different operations; no index operator, but `get` and `add` methods. They have different type relationships; arrays are covariant, while collections are not. They are not as efficient as arrays; they add overhead in terms of memory footprint and performance. By using collections of concrete parameterized types as a workaround for the illegal array type many things change in your implementation.

The different type relationships, for instance, can be observed in the example above and it renders method `addElements` pointless. Using arrays we declared the argument type of the `addElements` method as type `Object[]` so that the method would accept all types of arrays. For the collections there is no such supertype as an `Object[]`. Type `Collection<?>`, or type `List<?>` in our example, comes closest to what the `Object[]` is for arrays. But wildcard instantiations of the collection types give only limited access to the collections' operations. In our example, we cannot insert any elements into the collection of integer pairs through a reference of type `List<?>`. A method like `addElements` does not make any sense any longer; we would need a method specifically for a collection of `Pair<Integer, Integer>` instead. In essence, you must design your APIs differently, when you work with collections instead of arrays.

The most compelling argument against collections is efficiency; arrays are without doubt more efficient. The argument in favor of collections is type safety; the compiler performs all necessary type checks to ensure that the collection is a homogenous sequence.

LINK TO THIS [GenericTypes.FAQ105](#)

#### REFERENCES

[What is a reifiable type?](#)  
[What is an unbounded wildcard?](#)  
[What is an unbounded wildcard parameterized type?](#)  
[What is the raw type?](#)  
[Can I create an array whose component type is a wildcard parameterized type?](#)  
[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)  
[What is the difference between the unbounded wildcard parameterized type and the raw type?](#)

---

## Why is there no class literal for concrete parameterized types?

*Because parameterized type has no exact runtime type representation.*

A class literal denotes a `Class` object that represents a given type. For instance, the class literal `String.class` denotes the `Class` object that represents the type `String` and is identical to the `Class` object that is returned when method `getClass` is invoked on a `String` object. A class literal can be used for runtime type checks and for reflection.

Parameterized types lose their type arguments when they are translated to byte code during compilation in a process called *type erasure*. As a side effect of type erasure, all instantiations of a generic type share the same runtime representation, namely that of the corresponding *raw type*. In other words, parameterized types do not have type representation of their own. Consequently, there is no point in forming class literals such as `List<String>.class`, `List<Long>.class` and `List<?>.class`, since no such `Class` objects exist. Only the raw type `List` has a `Class` object that represents its runtime type. It is referred to as `List.class`.

LINK TO THIS [GenericTypes.FAQ106](#)

REFERENCES [What is type erasure?](#)  
[What is the raw type?](#)

---

## Raw Types

---

### What is the raw type?

*The generic type without any type arguments.*

The generic type without any type arguments, like `Collection`, is called *raw type*.

The raw type is assignment compatible with all instantiations of the generic type. Assignment of an instantiation of a generic type to the corresponding raw type is permitted without warnings; assignment of the raw type to an instantiation yields an "unchecked conversion" warning.

Example (of assignment compatibility):

```
ArrayList      rawList      = new ArrayList();
ArrayList<String> stringList = new ArrayList<String>();
rawList        = stringList;
stringList     = rawList;    // unchecked warning
```

The "unchecked" warning indicates that the compiler does not know whether the raw type `ArrayList` really contains strings. A raw type `ArrayList` can in principle contain any type of object and is similar to a `ArrayList<Object>`.

LINK TO THIS [GenericTypes.FAQ201](#)

REFERENCES [Why are raw types permitted?](#)  
[Can I use a raw type like any other type?](#)  
[How does the raw type relate to instantiations of the corresponding generic type?](#)

---

### Why are raw types permitted?

*To facilitate interfacing with non-generic (legacy) code.*

Raw types are permitted in the language predominantly to facilitate interfacing with non-generic (legacy) code.

If, for instance, you have a non-generic legacy method that takes a `List` as an argument, you can pass a parameterized type such as `List<String>` to that method. Conversely, if you have a method that returns a `List`, you can assign the result to a reference variable of type `List<String>`, provided you know for some reason that the returned list really is a list of strings.

Example (of interfacing with legacy code using raw types):

```
class SomeLegacyClass {
    public void setNames(List c) { ... }
    public List getNames() { ... }
}

final class Test {
    public static void main(String[] args) {
        SomeLegacyClass obj = new SomeLegacyClass();
        List<String> names = new LinkedList<String>();
        ... fill list ...

        obj.setNames(names);

        names = obj.getNames();    // unchecked warning
    }
}
```

A `List<String>` is passed to the `setNames` method that asks for an argument of the raw type `List`. The conversion from a `List<String>` to a `List` is safe because a method that can handle a heterogeneous list of objects can certainly cope with a list of strings.

The `getNames` method returns a raw type `List`, which we assign to a variable of type `List<String>`. The compiler has not enough information to ensure that the list returned really is a list of strings. Despite of that, the compiler permits the conversion from the raw type `List` to the more specific type `List<String>`, in order to allow this kind of mixing of non-generic and generic Java code. Since the conversion from `List` to `List<String>` is not type-safe, the assignment is flagged as an "unchecked assignment".

The use of raw types in code written after the introduction of genericity into the Java programming language is discouraged. According to the Java Language Specification, it is possible that future versions of the Java programming language will disallow the use of raw types.

LINK TO THIS [GenericTypes.FAQ202](#)

REFERENCES [What are raw types?](#)  
[Can I use a raw type like any other type?](#)  
[How does the raw type relate to instantiations of the corresponding generic type?](#)

---

## Can I use a raw type like any other type?

*Yes, but certain uses will result in "unchecked" warnings.*

Raw types can be used like regular types without any restrictions, except that certain uses will result in "unchecked" warnings.

Example (of a parameterized type):

```
interface Copyable<T> {
    T copy();
}

final class Wrapped<Elem extends Copyable<Elem>> {
    private Elem theObject;
```



```

public Wrapped(Elem arg) { theObject = arg.copy(); }

public void setObject(Elem arg) { theObject = arg.copy(); }

public Elem getObject() { return theObject.copy(); }

public boolean equals(Object other) {
    if (other == null) return false;
    if (! (other instanceof Wrapped)) return false;
    return (this.theObject.equals(((Wrapped)other).theObject));
}
}

```

Methods or constructors of a raw type have the signature that they would have after type erasure. A method or constructor call to a raw type generates an unchecked warning if the erasure changes the argument types.

Example (same as above - after type erasure):

```

interface Copyable {
    Object copy();
}

final class Wrapped {
    private Copyable theObject;

    public Wrapped(Copyable arg) { theObject = arg.copy(); }

    public void setObject(Copyable arg) { theObject = arg.copy(); }

    public Copyable getObject() { return theObject.copy(); }

    public boolean equals(Object other) {
        if (other == null) return false;
        if (! (other instanceof Wrapped)) return false;
        return (this.theObject.equals(((Wrapped)other).theObject));
    }
}

```

Invocation of a method or constructor, whose argument type changed in the course of type erasure is unsafe and is flagged as an "unchecked" operation. For instance, the method `setObject` has the signature `void setObject(Copyable)` after type erasure and its invocation results in an "unchecked" warning. The invocation is unsafe because the compiler cannot ensure that the argument passed to the method is compatible to the "erased" type that the type parameter `Elem` stands for.

Example (using the raw type):

```

class MyString implements Copyable<MyString> {
    private StringBuilder buffer;
    public MyString(String s) { buffer = new StringBuilder(s); }
    public MyString copy() { return new MyString(buffer.toString()); }
    ...
}

class Test {
    private static void test(Wrapped wrapper) {
        wrapper.setObject(new MyString("Deutsche Bank")); // unchecked warning
        Object s = wrapper.getObject();
    }

    public static void main(String[] args) {
        Wrapped<MyString> wrapper = new Wrapped<MyString>(new MyString("Citibank"));
        test(wrapper);
    }
}

```



If the method's argument type is not changed by type erasure, then the method call is safe. For instance, the method `getObject` has the signature `Copyable getObject(void)` after type erasure and its invocation is safe and warning-free.

Fields of a raw type have the type that they would have after type erasure. A field assignment to a raw type generates an unchecked warning if erasure changes the field type. In our example, the field `theObject` of the raw type `Wrapped` is changed by type erasure and is of type `Copyable` after type erasure.

If the `theObject` field were public and we could assign to it, the assignment would be unsafe because the compiler cannot ensure that the value being assigned really is of type `Elem`. Yet the assignment is permitted and flagged as an "unchecked" assignment. Reading the field is safe and does not result in a warning.

LINK TO THIS [GenericTypes.FAQ203](#)

REFERENCES [What is type erasure?](#)  
[How does the raw type relate to instantiations of the corresponding generic type?](#)  
[Can I use a type parameter as part of its own bounds?](#)

---

## Wildcard Instantiations

---

### What is a wildcard parameterized type?

*An instantiation of a generic type where the type argument is a wildcard (as opposed to a concrete type).*

A wildcard parameterized type is an instantiation of a generic type where at least one type argument is a wildcard. Examples of wildcard parameterized types are `Collection<?>`, `List<? extends Number>`, `Comparator<? super String>` and `Pair<String, ?>`. A wildcard parameterized type denotes a family of types comprising concrete instantiations of a generic type. The kind of the wildcard being used determines which concrete parameterized types belong to the family. For instance, the wildcard parameterized type `Collection<?>` denotes the family of all instantiations of the `Collection` interface regardless of the type argument. The wildcard parameterized type `List<? extends Number>` denotes the family of all list types where the element type is a subtype of `Number`. The wildcard parameterized type `Comparator<? super String>` is the family of all instantiations of the `Comparator` interface for type argument types that are supertypes of `String`.

A wildcard parameterized type is not a concrete type that could appear in a new expression. A wildcard parameterized type is similar to an interface type in the sense that reference variables of a wildcard parameterized type can be declared, but no objects of the wildcard parameterized type can be created. The reference variables of a wildcard parameterized type can refer to an object that is of a type that belongs to the family of types that the wildcard parameterized type denotes.

Examples:

```
Collection<?> coll = new ArrayList<String>();
List<? extends Number> list = new ArrayList<Long>();
Comparator<? super String> cmp = new RuleBasedCollator("< a< b< c< d");
Pair<String, ?> pair = new Pair<String, String>();
```

Counter Example:

```
List<? extends Number> list = new ArrayList<String>(); // error
```

Type `String` is not a subtype of `Number` and consequently `ArrayList<String>` does not belong to the family of types denoted by `List<? extends Number>`. For this reason the compiler issues an error message.

LINK TO THIS [GenericTypes.FAQ301](#)

REFERENCES [What is a wildcard?](#)  
[Can I use a wildcard parameterized type like any other type?](#)

---

## What is the unbounded wildcard parameterized type?

*An instantiation of a generic type where all type arguments are the unbounded wildcard "?".*

Examples of unbounded wildcard parameterized types are `Pair<?, ?>` and `Map<?, ?>`.

The unbounded wildcard parameterized type is assignment compatible with *all* instantiations of the corresponding generic type. Assignment of another instantiation to the unbounded wildcard instantiation is permitted without warnings; assignment of the unbounded wildcard instantiation to another instantiation is illegal.

Example (of assignment compatibility):

```
ArrayList<?>    anyList    = new ArrayList<Long>();
ArrayList<String> stringList = new ArrayList<String>();
anyList        = stringList;
stringList     = anyList;    // error
```

The unbounded wildcard parameterized type is kind of the supertype of all other instantiations of the generic type: "subtypes" can be assigned to the "unbounded supertype", not vice versa.

LINK TO THIS [GenericTypes.FAQ302](#)

REFERENCES [How do unbounded wildcard instantiations of a generic type relate to other instantiations of the same generic type?](#)

---

## What is the difference between the unbounded wildcard parameterized type and the raw type?

*The compiler issues error messages for an unbounded wildcard parameterized type while it only reports "unchecked" warnings for a raw type.*

In code written after the introduction of genericity into the Java programming language you would usually avoid use of raw types, because it is discouraged and raw types might no longer be supported in future versions of the language (according to the Java Language Specification). Instead of the raw type you can use the unbounded wildcard parameterized type.

The raw type and the unbounded wildcard parameterized type have a lot in common. Both act as kind of a supertype of all instantiations of the corresponding generic type. Both are so-called *reifiable types*. Reifiable types can be used in `instanceof` expressions and as the component type of arrays, where non-reifiable types (such as concrete and bounded wildcard parameterized type) are not permitted.

In other words, the raw type and the unbounded wildcard parameterized type are semantically equivalent. The only difference is that the compiler applies stricter rules to the unbounded wildcard parameterized type than to the corresponding raw type. Certain operations performed on the raw type yield "unchecked" warnings. The same operations, when performed on the corresponding unbounded wildcard parameterized type, are rejected as errors.

LINK TO THIS [GenericTypes.FAQ303](#)

## REFERENCES

[What is the raw type?](#)

[What is the unbounded wildcard parameterized type?](#)

[What is a reifiable type?](#)

[How do parameterized types fit into the Java type system?](#)

[How does the raw type relate to instantiations of the corresponding generic type?](#)

[How do instantiations of a generic type relate to instantiations of other generic types?](#)

[How do unbounded wildcard instantiations of a generic type relate to other instantiations of the same generic type?](#)

[How do wildcard instantiations with an upper bound relate to other instantiations of the same generic type?](#)

[How do wildcard instantiations with a lower bound relate to other instantiations of the same generic type?](#)

---

## Which methods and fields are accessible/inaccessible through a reference variable of a wildcard parameterized type?

*It depends on the kind of wildcard.*

Using an object through a reference variable of a wildcard parameterized type is restricted. Consider the following class:

Example (of a generic class):

```
class Box<T> {
    private T t;
    public Box(T t) { this.t = t; }
    public void put(T t) { this.t = t; }
    public T take() { return t; }
    public boolean equalTo(Box<T> other) { return this.t.equals(other.t); }
    public Box<T> copy() { return new Box<T>(t); }
}
```

When we use a reference variable of a wildcard instantiation of type `Box` to access methods and fields of the referenced object the compiler would reject certain invocations.

Example (of access through a wildcard parameterized type):

```
class Test {
    public static void main(String[] args) {
        Box<?> box = new Box<String>("abc");

        box.put("xyz"); // error
        box.put(null); // ok

        String s = box.take(); // error
        Object o = box.take(); // ok

        boolean equal = box.equalTo(box); // error
        equal = box.equalTo(new Box<String>("abc")); // error

        Box<?> box1 = box.copy(); // ok
        Box<String> box2 = box.copy(); // error
    }
}
```

In a wildcard parameterized type such as `Box<?>` the type of the field and the argument and the return types of the methods would be unknown. It is like the field `t` would be of type `"?"` and the `put` method would take an argument of type `"?"` and the `take` method would return a `"?"` and so on.

In this situation the compiler does not let us assign anything to the field or pass anything to the `put` method. The reason is that the compiler cannot make sure that the object that we are trying to pass as an argument to a

method is of the expected type, since the expected type is unknown. Similarly, the compiler does not know of which type the field is and cannot check whether we are assigning an object of the correct type, because the correct type is not known.

In contrast, the `take` method can be invoked and it returns an object of an unknown type, which we can assign to a reference variable of type `Object`.

Similar effects can be observed for methods such as `like` `equalTo` and `copy`, which have a parameterized argument or return type and the type parameter `T` appears as type argument of the parameterized argument or return type.

Consider a generic class with methods that use the type parameter in the argument or return type of its methods:

Example (of a generic class):

```
class Box<T> {
    private T t;
    public Box(T t) { this.t = t; }
    public Box(Box<? extends T> box) { t = box.t; }
    ...
    public boolean equalTo(Box<T> other) { return this.t.equals(other.t); }
    public Box<T> copy() { return new Box<T>(t); }

    public Pair<T,T> makePair() { return new Pair<T,T>(t,t); }
    public Class<? extends T> getContentType() { ... }
    public int compareTo(Comparable<? super T> other) { return other.compareTo(t); }
}
```

The type parameter `T` can appear as the type argument of a parameterized argument or return type, like in method `makePair`, which returns a `Pair<T,T>`. But it can also appear as part of the type argument of a parameterized argument or return type, namely as bound of a wildcard, like in method `getContentType`, which returns a value of type `Class<? extends T>`. Which methods can or must not be invoked through a wildcard instantiation depends not only on the type of the wildcard instantiation (unbounded or bounded with upper or lower bound), but also on the use of the type parameter (as type argument or as wildcard bound).

The restriction are fairly complex in detail, because they depend on the type of the wildcard (unbounded or bounded with upper or lower bound). So far we have only seen `Box<?>`, that is, the unbounded wildcard instantiation. Which fields and methods are accessible through references of other wildcard instantiations? In addition, the rules depend on the way in which a method uses the type parameter in the method signatures (as the type of an argument or the return type or as the type argument of a parameterized argument or return type). A comprehensive discussion can be found in the FAQ entries listed in the reference section below.

LINK TO THIS [GenericTypes.FAQ304](#)

#### REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard parameterized type?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)  
[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)  
[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)  
[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)  
[In a wildcard parameterized type, can I read and write fields whose type is the type parameter?](#)

## Can I use a wildcard parameterized type like any other type?

*No. A wildcard parameterized type is not a type in the regular sense (different from a non-parameterized class/interface or a raw type).*

Wildcard parameterized types can be used for typing (like non-parameterized classes and interfaces):

- as argument and return types of methods
- as type of a field or local reference variable
- as component type of an array
- as type argument of other parameterized types
- as target type in casts

Wildcard parameterized type can NOT be used for the following purposes (different from non-parameterized classes and interfaces):

- for creation of objects
- for creation of arrays (except unbounded wildcard)
- in exception handling
- in instanceof expressions (except unbounded wildcard)
- as supertypes
- in a class literal

LINK TO THIS [GenericTypes.FAQ305](#)

### REFERENCES

---

## Can I create an object whose type is a wildcard parameterized type?

*No, not directly.*

Objects of a wildcard parameterized type are not particularly useful, mainly because there is not much you can do with the object. You can access an object of a wildcard parameterized type only through a reference of that wildcard parameterized type, and such a reference gives only restricted access to the referenced object. Basically, the wildcard parameterized type is too abstract to be useful. For this reason, the creation of objects of a wildcard parameterized type is discouraged: it is illegal that a wildcard parameterized type appears in a new expression.

Example (of illegal creation of objects of a wildcard parameterized type):

```
ArrayList<String> list = new ArrayList<String>();  
... populate the list ...  
  
ArrayList<?> coll1 = new ArrayList<?>(); // error  
ArrayList<?> coll2 = new ArrayList<?>(10); // error  
ArrayList<?> coll3 = new ArrayList<?>(list); // error
```

The compiler rejects all attempts to create an object of the wildcard type `ArrayList<?>`.

In a way, a wildcard parameterized type is like an interface type: you can declare reference variables of the type, but you cannot create objects of the type. A reference variable of an interface type or a wildcard parameterized type can refer to an object of a compatible type. For an interface, the compatible types are the class (or enum) types that implement the interface. For a wildcard parameterized type, the compatible types are the concrete instantiations of the corresponding generic type that belong to the family of instantiations that the wildcard denotes.

Example (comparing interface and wildcard parameterized type):

```
Cloneable clon1 = new Date();
Cloneable clon2 = new Cloneable(); // error

ArrayList<?> coll1 = new ArrayList<String>();
ArrayList<?> coll2 = new ArrayList<?>(); // error
```

The code snippet above illustrates the similarity between an interface and a wildcard parameterized type, using the interface `Cloneable` and the wildcard parameterized type `ArrayList<?>` as examples. We can declare reference variables of type `Cloneable` and `ArrayList<?>`, but we must not create objects of type `Cloneable` and `ArrayList<?>`.

Interestingly, the compiler's effort to prevent the creation of objects of a wildcard parameterized type can be circumvented. It is unlikely that you will ever want to create an object of a wildcard parameterized type, but should you ever need one, there's the workaround (see [TechnicalDetails.FAQ609](#)).

LINK TO THIS [GenericTypes.FAQ306](#)

#### REFERENCES

[What is a wildcard parameterized type?](#)

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type? Can I use a wildcard instantiation like any other type?](#)

[What is type argument inference?](#)

[Is it really impossible to create an object whose type is a wildcard parameterized type?](#)

---

## Can I create an array whose component type is a wildcard parameterized type?

*No, because it is not type-safe.*

The rationale is the same as for concrete parameterized types: a wildcard parameterized type, unless it is an unbounded wildcard parameterized type, is a non-reifiable type and arrays of non-reifiable types are not type-safe.

The array store check cannot be performed reliably because a wildcard parameterized type that is not an unbounded wildcard parameterized type has a non-exact runtime type.

Example (of the consequences):

```
Object[] numPairArr = new Pair<? extends Number,? extends Number>[10]; // illegal
numPairArr[0] = new Pair<Long,Long>(0L,0L); // fine
numPairArr[0] = new Pair<String,String>("", ""); // should fail, but would succeed
```

The array store check would have to check whether the pair added to the array is of type `Pair<? extends Number,? extends Number>` or of a subtype thereof. Obviously, a `Pair<String,String>` is not of a matching type and should be rejected with an `ArrayStoreException`. But the array store check does not detect any type mismatch, because the JVM can only check the array's runtime component type, which is `Pair[]` after type erasure, against the element's runtime type, which is `Pair` after type erasure.

LINK TO THIS [GenericTypes.FAQ307](#)

#### REFERENCES

[What does type-safety mean?](#)

[Can I create an array whose component type is a concrete parameterized type?](#)

[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)

[What is a reifiable type?](#)

---

## Can I declare a reference variable of an array type whose component type is a bounded wildcard parameterized type?

*Yes, you can, but you should not, because it is neither helpful nor type-safe.*

The rationale is the same as for concrete parameterized types: a wildcard parameterized type, unless it is an unbounded wildcard parameterized type, is a non-reifiable type and arrays of non-reifiable types must not be created. Hence it does not make sense to have a reference variable of such an array type because it can never refer to array of its type. All that it can refer to is `null`, an array whose component type is a non-parameterized subtype of the instantiations that belong to the type family denoted by the wildcard, or an array whose component type is the corresponding raw type. Neither of these cases is overly useful, yet they are permitted.

Example (of an array reference variable with wildcard parameterized component type):

```
Pair<? extends Number,? extends Number>[] arr = null; // fine
arr = new Pair<? extends Number,? extends Number>[2]; // error: generic array
creation
```

The code snippet shows that a reference variable of type `Pair<? extends Number,? extends Number>[]` can be declared, but the creation of such an array is illegal. But we can have the reference variable of type `Pair<? extends Number,? extends Number>[]` refer to an array of a non-parameterized subtype of any of the concrete instantiations that belong to the type family denoted by `Pair<? extends Number,? extends Number>`. (Remember, wildcard parameterized types cannot be used as supertypes; hence a non-parameterized subtype must be a subtype of a concrete parameterized type.)

Example (of another array reference variable with parameterized component type):

```
class Point extends Pair<Double,Double> { ... }

Pair<? extends Number,? extends Number>[] arr = new Point[2]; // fine
```

Using a reference variable of type `Pair<? extends Number,? extends Number>[]` offers no advantage over using a variable of the actual type `Point[]`. Quite the converse; it is an invitation for making mistakes.

Example (of an array reference variable referring to array of subtypes; not recommended):

```
Pair<? extends Number,? extends Number>[] arr = new Point[2];
arr[0] = new Point(-1.0,1.0); // fine
arr[1] = new Pair<Number,Number>(-1.0,1.0); // fine (causes ArrayStoreException)
arr[2] = new Pair<Integer,Integer>(1,2); // fine (causes ArrayStoreException)
```

The compiler permits code for insertion of elements of type `Pair<Number,Number>` or `Pair<Integer,Integer>` into the array through the reference variable of type `Pair<? extends Number,? extends Number>[]`. Yet, at runtime, this insertion will always fail with an `ArrayStoreException` because we are trying to insert a `Pair` into a `Point[]`. The debatable insertions would be flagged as errors and thereby prevented if we used the actual type of the array, namely `Point[]` instead of `Pair<? extends Number,? extends Number>[]`.

In essence, you should better refrain from using array reference variable whose component type is a wildcard parameterized type. Note, that the same holds for array reference variable whose component type is a concrete parameterized type. Only an array reference variable whose component type is an unbounded wildcard parameterized type make sense. This is because an unbounded wildcard parameterized type is a reifiable type and arrays with a reifiable component type can be created; the array reference variable can refer to an array of its type and the deficiencies discussed above simply do not exist for unbounded wildcard arrays.

LINK TO THIS [GenericTypes.FAQ307A](#)

#### REFERENCES

[What does type-safety mean?](#)

[Can I create an array whose component type is a wildcard parameterized type?](#)

[Can I declare a reference variable of an array type whose component type is a concrete parameterized type?](#)

[Can I create an array whose component type is a concrete parameterized type?](#)



[Can I declare a reference variable of an array type whose component type is an unbounded wildcard parameterized type?](#)  
[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)  
[What is a reifiable type?](#)

---

## Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?

*Because it is type-safe.*

The rationale is related to the rule for other instantiations of a generic type: an unbounded wildcard parameterized type is a reifiable type and arrays of reifiable types are type-safe, in contrast to arrays of non-reifiable types, which are not safe and therefore illegal. The problem with the unreliable array store check (the reason for banning arrays with a non-reifiable component type) does not occur if the component type is reifiable.

Example (of array of unbounded wildcard parameterized type):

```
Object[] pairArr = new Pair<?,?>[10]; // fine
pairArr[0] = new Pair<Long,Long>(0L,0L); // fine
pairArr[0] = new Pair<String,String>("",""); // fine
pairArr[0] = new ArrayList<String>(); // fails with ArrayStoreException
```

The array store check must check whether the element added to the array is of type `Pair<?,?>` or of a subtype thereof. In the example the two pairs, although of different type, are perfectly acceptable array elements. And indeed, the array store check, based on the non-exact runtime type `Pair`, accepts the two pairs and correctly sorts out the "alien" `ArrayList` object as illegal by raising an `ArrayStoreException`. The behavior is exactly the same as for an array of the raw type, which is not at all surprising because the raw type is a reifiable type as well.

LINK TO THIS [GenericTypes.FAQ308](#)

### REFERENCES

[What is a reifiable type?](#)  
[What does type-safety mean?](#)  
[What is the raw type?](#)  
[Can I create an array whose component type is a concrete parameterized type?](#)  
[Can I create an array whose component type is a wildcard parameterized type?](#)

---

## Can I declare a reference variable of an array type whose component type is an unbounded wildcard parameterized type?

*Yes.*

An array reference variable whose component type is an unbounded wildcard parameterized type (such as `Pair<?,?>[]`) is permitted and useful. This is in contrast to array reference variables with a component type that is a concrete or bounded wildcard parameterized type (such as `Pair<Long,Long>[]` or `Pair<? extends Number,? extends Number>[]`); the array reference variable is permitted, but not overly helpful.

The difference stems from the fact that an unbounded wildcard parameterized type is a reifiable type and arrays with a reifiable component type can be created. Concrete and bounded wildcard parameterized types are non-reifiable types and arrays with a non-reifiable component type cannot be created. As a result, an array variable with a reifiable component type can refer to array of its type, but this is not possible for the non-reifiable component types.

Example (of array reference variables with parameterized component types):

```
Pair<?,?>[] arr
```



```

= new Pair<?,?>[2];           // fine

Pair<? extends Number,? extends Number>[] arr
= new Pair<? extends Number,? extends Number>[2]; // error: generic array
creation

Pair<Double,Double>[] arr
= new Pair<Double,Double>[2]; // error: generic array creation

```

The examples above demonstrate that unbounded wildcard parameterized types are permitted as component type of an array, while other instantiations are not permitted. In the case of a non-reifiable component type the array reference variable can be declared, but it cannot refer to an array of its type. At most it can refer to an array of a non-parameterized subtype (or an array of the corresponding raw type), which opens opportunities for mistakes, but does not offer any advantage.

LINK TO THIS [GenericTypes.FAQ308A](#)

#### REFERENCES

[What is a reifiable type?](#)  
[Can I create an array whose component type is a wildcard parameterized type?](#)  
[Can I declare a reference variable of an array type whose component type is a concrete parameterized type?](#)  
[Can I create an array whose component type is a concrete parameterized type?](#)  
[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)  
[Can I declare a reference variable of an array type whose component type is a bounded wildcard parameterized type?](#)

## Can I derive from a wildcard parameterized type?

*No, a wildcard parameterized type is not a supertype.*

Let us scrutinize an example and see why a wildcard parameterized type cannot be a supertype. Consider the generic interface `Comparable`.

Example (of a generic interface):

```

interface Comparable<T> {
    int compareTo(T arg);
}

```

If it were allowed to subtype from a wildcard instantiation of `Comparable`, neither we nor the compiler would know what the signature of the `compareTo` method would be.

Example (of illegal use of a wildcard parameterized type as a supertype):

```

class MyClass implements Comparable<?> { // error
    public int compareTo(??? arg) { ... }
}

```

The signatures of methods of a wildcard parameterized type are undefined. We do not know what type of argument the `compareTo` method is supposed to accept. We can only subtype from concrete instantiations of the `Comparable` interface, so that the signature of the `compareTo` method is well-defined.

Example (of legal use of a concrete parameterized type as a supertype):

```

class MyClass implements Comparable<MyClass> { // fine
    public int compareTo(MyClass arg) { ... }
}

```

Note that the raw type is, of course, acceptable as a supertype, different from the wildcard parameterized types including the unbounded wildcard parameterized type.

Example (of legal use of a raw type as a supertype):

```
class MyClass implements Comparable { // fine
    public int compareTo(Object arg) { ... }
}
```

LINK TO THIS [GenericTypes.FAQ309](#)

#### REFERENCES

[What is the raw type?](#)

[What is a wildcard parameterized type?](#)

[What is the unbounded wildcard parameterized type?](#)

[What is the difference between the unbounded wildcard parameterized type and the raw type?](#)

---

## Why is there no class literal for wildcard parameterized types?

*Because a wildcard parameterized type has no exact runtime type representation.*

The rationale is the same as for concrete parameterized types.

Wildcard parameterized types lose their type arguments when they are translated to byte code in a process called *type erasure*. As a side effect of type erasure, all instantiations of a generic type share the same runtime representation, namely that of the corresponding *raw type*. In other words, parameterized types do not have type representation of their own. Consequently, there is no point to forming class literals such as `List<?>.class`, `List<? extends Number>.class` and `List<Long>.class`, since no such `Class` objects exist. Only the raw type `List` has a `Class` object that represents its runtime type. It is referred to as `List.class`.

LINK TO THIS [GenericTypes.FAQ310](#)

#### REFERENCES

[What is type erasure?](#)

[What is the raw type?](#)

[Why is there no class literal for concrete parameterized types?](#)

# Generic Methods

© Copyright 2004-2022 by Angelika Langer. All Rights Reserved.

## [Fundamentals](#)

- [What is a generic method?](#)
- [How do I invoke a generic method?](#)

---

## Generic Methods

---

### What is a generic method?

*A method with type parameters.*

Not only types can be generic, but methods can be generic, too. Static and non-static methods as well as constructors can have type parameters. The syntax for declaration of the formal type parameters is similar to the syntax for generic types. The type parameter section is delimited by angle brackets and appears before the method's return type. Its syntax and meaning is identical to the type parameter list of a generic type.

Here is the example of a generic `max` method that computes the greatest value in a collection of elements of an unknown type `A`.

Example (of a generic method):

```
class Collections {
    public static <A extends Comparable<A>> A max(Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
```

The `max` method has one type parameter, named `A`. It is a place holder for the element type of the collection that the method works on. The type parameter has a bound; it must be a type `A` that is a subtype of `Comparable<A>`, i.e., a type that can be compared to elements of itself.

**LINK TO THIS**      [GenericMethods.FAQ001](#)

**REFERENCES**

- [What is a generic type?](#)
- [How do I define a generic type?](#)
- [What is a type parameter?](#)
- [What is a bounded type parameter?](#)

---

### How do I invoke a generic method?

*Usually by calling it. Type arguments for generic methods need not be provided explicitly; they are almost*

*always automatically inferred.*

Generic methods are invoked like regular non-generic methods. The type parameters are inferred from the invocation context.

Example (of invocation of a generic method; taken from the preceding item):

```
class Collections {
    public static <A extends Comparable<A>> A max (Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}

final class Test {
    public static void main (String[ ] args) {
        LinkedList<Long> list = new LinkedList<Long>();
        list.add(0L);
        list.add(1L);
        Long y = Collections.max(list);
    }
}
```

In our example, the compiler would automatically invoke an instantiation of the `max` method with the type argument `Long`, that is, the formal type parameter `A` is replaced by type `Long`. Note that we do not have to explicitly specify the type argument. The compiler automatically infers the type argument by taking a look at the type of the arguments provided to the method invocation. The compiler finds that a `Collection<A>` is asked for and that a `LinkedList<Long>` is provided. From this information the compiler concludes at compile time that `A` must be replaced by `Long`.

**LINK TO THIS**      [GenericMethods.FAQ002](#)

#### REFERENCES

[What is type argument inference?](#)

[What explicit type argument specification?](#)

[What happens if a type parameter does not appear in the method parameter list?](#)

[Why doesn't type argument inference fail when I provide inconsistent method arguments?](#)

# Type Parameters

© Copyright 2004-2011 by Angelika Langer. All Rights Reserved.

## Fundamentals

- [What is a type parameter?](#)
- [What is a bounded type parameter?](#)

## Type Parameter Bounds

- [What is a type parameter bound?](#)
- [Which types are permitted as type parameter bounds?](#)
- [Can I use a type parameter as a type parameter bound?](#)
- [Can I use different instantiations of a same generic type as bounds of a type parameter?](#)
- [How can I work around the restriction that a type parameter cannot have different instantiations of a same generic type as its bounds?](#)
- [Does a bound that is a class type give access to all its public members?](#)
- [How do I decrypt "Enum<E extends Enum<E>>"?](#)
- [Why is there no lower bound for type parameters?](#)

## Usage

- [Can I use a type parameter like a type?](#)
- [Can I create an object whose type is a type parameter?](#)
- [Can I create an array whose component type is a type parameter?](#)
- [Can I cast to the type that the type parameter stands for?](#)
- [Can I use a type parameter in exception handling?](#)
- [Can I derive from a type parameter?](#)
- [Why is there no class literal for a type parameter?](#)

## Scope

- [Where is a type parameter visible \(or invisible\)?](#)
- [Can I use a type parameter as part of its own bounds?](#)
- [Can I use the type parameter of an outer type as part of the bounds of the type parameter of an inner type or a method?](#)

## Static Context

- [Is there one instances of a static field per instantiation of a generic type?](#)
- [Why can't I use a type parameter in any static context of a generic class?](#)

---

# Type Parameters

---

---

## Fundamentals

---

**What is a type parameter?**

## *A place holder for a type argument.*

Generic types have one or more type parameters.

Example of a parameterized type:

```
interface Comparable<E> {
    int compareTo(E other);
}
```

The identifier `E` is a type parameter. Each type parameter is replaced by a type argument when an instantiation of the generic type, such as `Comparable<Object>` or `Comparable<? extends Number>`, is used.

LINK TO THIS [TypeParameters.FAQ001](#)

REFERENCES [How is a generic type defined?](#)  
[What is a bounded type parameter?](#)  
[Where is a type parameter visible \(or invisible\)?](#)

---

## What is a bounded type parameter?

*A type parameter with one or more bounds. The bounds restrict the set of types that can be used as type arguments and give access to the methods defined by the bounds.*

When you declare a type parameter `T` and use it in the implementation of a generic type or method, the type parameter `T` still denotes an unknown type. The compiler knows that `T` is a place holder for a type, but it does not know anything about the type. This is okay in some implementations, but insufficient in others.

Example (of a generic type without bounds):

```
public class Hashtable<Key,Data> {
    ...
    private static class Entry<Key,Data> {
        private Key key;
        private Data value;
        private int hash;
        private Entry<Key,Data> next;
        ...
    }
    private Entry<Key,Data>[] table;
    ...
    public Data get(Key key) {
        int hash = key.hashCode();
        for (Entry<Key,Data> e = table[hash & hashMask]; e != null; e = e.next) {
            if ((e.hash == hash) && e.key.equals(key)) {
                return e.value;
            }
        }
        return null;
    }
}
```

The implementation of class `Hashtable` invokes the methods `hashCode` and `equals` on the unknown `Key` type. Since `hashCode` and `equals` are methods defined in class `Object` and available for all reference types, not much need to be known about the unknown `Key` type. This changes substantially, when we look into the implementation of sorted sequence.

Example (of a generic type, so far without bounds):

```
public interface Comparable<T> {
    public int compareTo(T arg);
}
public class TreeMap<Key,Data>{
    private static class Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left;
        Entry<K,V> right;
        Entry<K,V> parent;
    }
    private transient Entry<Key,Data> root;
    ...
    private Entry<Key,Data> getEntry(Key key) {
        Entry<Key,Data> p = root;
        Key k = key;
        while (p != null) {
            int cmp = k.compareTo(p.key); // error
            if (cmp == 0)
                return p;
            else if (cmp < 0)
                p = p.left;
            else
                p = p.right;
        }
        return null;
    }
    public boolean containsKey(Key key) {
        return getEntry(key) != null;
    }
    ...
}
```

The implementation of class `TreeMap` invokes the method `compareTo` on the unknown `Key` type. Since `compareTo` is not defined for arbitrary types the compiler refuses to invoke the `compareTo` method on the unknown type `Key` because it does not know whether the key type has a `compareTo` method.

In order to allow the invocation of the `compareTo` method we must tell the compiler that the unknown `Key` type has a `compareTo` method. We can do so by saying that the `Key` type implements the `Comparable<Key>` interface. We can say so by declaring the type parameter `Key` as a bounded parameter.

Example (of the same generic type, this time *with* bounds):

```
public interface Comparable<T> {
    public int compareTo(T arg);
}
public class TreeMap<Key extends Comparable<Key>,Data>{
    private static class Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left = null;
        Entry<K,V> right = null;
        Entry<K,V> parent;
    }
    private transient Entry<Key,Data> root = null;
    ...
}
```

```

private Entry<Key,Data> getEntry(Key key) {
    Entry<Key,Data> p = root;
    Key k = key;
    while (p != null) {
        int cmp = k.compareTo(p.key);
        if (cmp == 0)
            return p;
        else if (cmp < 0)
            p = p.left;
        else
            p = p.right;
    }
    return null;
}
public boolean containsKey(Key key) {
    return getEntry(key) != null;
}
...
}

```

In the example above, the type parameter `Key` has the bound `Comparable<Key>`. Specification of a bound has two effects:

- *It gives access to the methods that the bound specifies.* In the example, the bound `Comparable<Key>` gives access to the `compareTo` method that we want to invoke in the implementation of our `TreeMap` class.
- *Only types "within bounds" can be used for instantiation of the generic type.* In the example, a parameterized type such as `TreeMap<Number, String>` would be rejected, because the type `Number` is not a subtype of `Comparable<Number>`. A parameterized type like `TreeMap<String, String>` would be accepted, because the type `String` is within bounds, i.e. is a subtype of `Comparable<String>`.

Note that the suggested bound `Comparable<Key>` in this example is not the best conceivable solution. A better bound, that is more relaxed and allows a larger set of type arguments, would be `Comparable<? super Key>`. A more detailed discussion can be found in a separate FAQ entry (click [here](#)).

LINK TO THIS [TypeParameters.FAQ002](#)

#### REFERENCES

[When would I use a wildcard parameterized with a lower bound?](#)  
[What is a type parameter bound?](#)  
[Which types are permitted as type parameter bounds?](#)  
[Can I use different instantiations of a same generic type as bounds of a type parameter?](#)  
[Does a bound that is a class type give access to all its public members?](#)  
[Can I use a type parameter as part of its own bounds or in the declaration of other type parameters?](#)

---

## Type Parameter Bounds

---

### What is a type parameter bound?

*A reference type that is used to further describe a type parameter. It restricts the set of types that can be used as type arguments and gives access to the non-static methods that it defines.*

A type parameter can be unbounded. In this case any reference type can be used as type argument to replace



the unbounded type parameter in an instantiation of a generic type.

Alternatively can have one or several bounds. In this case the type argument that replaces the bounded type parameter in an instantiation of a generic type must be a subtype of all bounds.

The syntax for specification of type parameter bounds is:

```
<TypeParameter extends Class & Interface1 & ... & InterfaceN>
```

A list of bounds consists of one class and/or several interfaces.

Example (of type parameters with several bounds):

```
class Pair<A extends Comparable<A> & Cloneable,
    B extends Comparable<B> & Cloneable>
    implements Comparable<Pair<A,B>>, Cloneable { ... }
```

This is a generic class with two type arguments A and B, both of which have two bounds.

LINK TO THIS [TypeParameters.FAQ101](#)

#### REFERENCES

[What is the difference between a wildcard bound and a type parameter bound?](#)  
[Which types are permitted as type parameter bounds?](#)  
[Can I use different instantiations of a same generic type as bounds of a type parameter?](#)

---

## Which types are permitted as type parameter bounds?

*All classes, interfaces and enum types including parameterized types, but no primitive types and no array types.*

All classes, interfaces, and enum types can be used as type parameter bound, including nested and inner types. Neither primitive types nor array types be used as type parameter bound.

Examples (of type parameter bounds):

```
class X0 <T extends int> { ... } // error
class X1 <T extends Object[]> { ... } // error
class X2 <T extends Number> { ... }
class X3 <T extends String> { ... }
class X4 <T extends Runnable> { ... }
class X5 <T extends Thread.State> { ... }
class X6 <T extends List> { ... }
class X7 <T extends List<String>> { ... }
class X8 <T extends List<? extends Number>> { ... }
class X9 <T extends Comparable<? super Number>> { ... }
class X10<T extends Map.Entry<?,?>> { ... }
```

The code sample shows that primitive types such as `int` and array types such as `Object[]` are not permitted as type parameter bound.

Class types, such as `Number` or `String`, and interface types, such as `Runnable`, are permitted as type parameter bound.

Enum types, such as `Thread.State` are also permitted as type parameter bound. `Thread.State` is an example of a nested type used as type parameter bound. Non-static inner types are also permitted.

Raw types are permitted as type parameter bound; `List` is an example.

Parameterized types are permitted as type parameter bound, including concrete parameterized types such as `List<String>`, bounded wildcard parameterized types such as `List<? extends Number>` and `Comparable<? super Long>`, and unbounded wildcard parameterized types such as `Map.Entry<?, ?>`. A bound that is a wildcard parameterized type allows as type argument all types that belong to the type family that the wildcard denotes. The wildcard parameterized type bound gives only restricted access to fields and methods; the restrictions depend on the kind of wildcard.

Example (of wildcard parameterized type as type parameter bound):

```
class X<T extends List<? extends Number>> {
    public void someMethod(T t) {
        t.add(new Long(0L));    // error
        Number n = t.remove(0);
    }
}
class Test {
    public static void main(String[] args) {
        X<ArrayList<Long>> x1 = new X<ArrayList<Long>>();
        X<ArrayList<String>> x2 = new X<ArrayList<String>>(); // error
    }
}
```

Reference variables of type `T` (the type parameter) are treated like reference variables of a wildcard type (the type parameter bound). In our example the consequence is that the compiler rejects invocation of methods that take an argument of the "unknown" type that the type parameter stands for, such as `List.add`, because the bound is a wildcard parameterized type with an upper bound.

At the same time the bound `List<? extends Number>` determines the types that can be used as type arguments. The compiler accepts all type arguments that belong to the type family `List<? extends Number>`, that is, all subtypes of `List` with a type argument that is a subtype of `Number`.

Note, that even types that do not have subtypes, such as final classes and enum types, can be used as upper bound. In this case there is only one type that can be used as type argument, namely the type parameter bound itself. Basically, the parameterization is pointless then.

Example (of nonsensical parameterization):

```
class Box<T extends String> {
    private T theObject;
    public Box(T t) { theObject = t; }
    ...
}
class Test {
    public static void main(String[] args) {
        Box<String> box1 = Box<String>("Jack");
        Box<Long> box2 = Box<Long>(100L);    // error
    }
}
```

The compiler rejects all type arguments except `String` as "not being within bounds". The type parameter `T` is not needed and the `Box` class would better be defined as a non-parameterized class.

LINK TO THIS [TypeParameters.FAQ102](#)

#### REFERENCES

[What is a type parameter bound?](#)

[Can I use a type parameter as a type parameter bound?](#)

[Can I use different instantiations of a same generic type as bounds of a type parameter?](#)

[Can I use a type parameter as part of its own bounds or in the declaration of other type parameters?](#)

[How do unbounded wildcard instantiations of a generic type relate to other instantiations of the same generic type?](#)

## Can I use a type parameter as a type parameter bound?

*Yes.*

A type parameter can be used as the bound of another type parameter.

Example (of a type parameter used as a type parameter bound):

```
class Triple<T> {
    private T fst, snd, trd;
    public <U extends T, V extends T, W extends T> Triple(U arg1, V arg2, W arg3) {

        fst = arg1;
        snd = arg2;
        trd = arg3;
    }
}
```

In this example the type parameter `T` of the parameterized class is used as bound of the type parameters `U`, `V` and `W` of a parameterized instance method of that class.

Further opportunities for using type parameters as bounds of other type parameters include situations where a nested type is defined inside a generic type or a local class is defined inside a generic method. It is even permitted to use a type parameter as bound of another type parameter in the same type parameter section.

LINK TO THIS [TypeParameters.FAQ102A](#)

### REFERENCES

[Can I use a type parameter as part of its own bounds or in the declaration of other type parameters?](#)

[Which types are permitted as type parameter bounds?](#)

[Where is a type parameter visible \(or invisible\)?](#)

[Can I use different instantiations of a same generic type as bounds of a type parameter?](#)

[What is the difference between a wildcard bound and a type parameter bound?](#)

---

## Can I use different instantiations of a same generic type as bounds of a type parameter?

*No, at most one instantiation of the same generic type can appear in the list of bounds of a type parameter.*

Example (of illegal use of two instantiations of the same generic type as bounds of a type parameter):

```
class ObjectStore<T extends Comparable<T> & Comparable<String>> { // error
    private Set<T> theObjects = new TreeSet<T>();
    ...
    public boolean equals(ObjectStore<String> other) {
        if (theObjects.size() != other.size()) return false;
        Iterator<T> iterThis = theObjects.iterator();
        Iterator<String> iterOther = other.theObjects.iterator();
        while (iterThis.hasNext() && iterOther.hasNext()) {
            T t = iterThis.next();
            String string = iterOther.next();
            if (t.compareTo(string) != 0) return false;
        }
        return true;
    }
}
```

```
}  
}
```

---

error: java.lang.Comparable cannot be inherited with different arguments: <T> and <java.lang.String>

```
class ObjectStore<T> extends Comparable<T> & Comparable<String> {  
    ^
```

In the example the type parameter `T` is required to be `Comparable<T>`, that is, comparable to its own type. This is needed for storing objects of type `T` in a `TreeSet<T>`. At the same time the type parameter `T` is required to be `Comparable<String>`, because we want to invoke the type parameter's `compareTo(String)` method.

Remember, type parameter bounds are needed to give the compiler access to the type parameter's non-static methods. In this (admittedly contrived) example, we need to specify two instantiations of the `Comparable` interface as bound of the type parameter, but the compiler rejects it.

The reason for this restriction is that there is no type that is a subtype of two different instantiations of the `Comparable` interface and could serve as a type argument. It is prohibited that a type implements or extends two different instantiations of the same interface. This is because the bridge method generation process cannot handle this situation. Details are discussed in a separate FAQ entry (click [here](#)). If no class can ever implement both instantiations of `Comparable`, there is no point to a bounds list that requires it. The class in our example would not be instantiable because no type can ever be within bounds, except perhaps class `String`.

In practice, you will need to work around this restriction. Sadly, there might be situations in which there is no workaround at all.

**LINK TO THIS**     [TypeParameters.FAQ103](#)

**REFERENCES**

[Can a class implement different instantiations of the same generic interface?](#)

[What is type erasure?](#)

[What is a bridge method?](#)

[How does type erasure work when a type parameter has several bounds?](#)

[How can work around the restriction that a type parameter cannot have different instantiations of a same generic type as its bounds?](#)

---

## How can I work around the restriction that a type parameter cannot have different instantiations of a same generic type as its bounds?

*Usually there is no satisfactory workaround.*

Let us use the example from the previous question for our search of a workaround.

Example (of illegal use of two instantiations of the same generic type as bounds of a type parameter):

```
class ObjectStore<T> extends Comparable<T> & Comparable<String> { // error  
    private Set<T> theObjects = new TreeSet<T>();  
    ...  
    public boolean equals(ObjectStore<String> other) {  
        if (theObjects.size() != other.size()) return false;  
        Iterator<T> iterThis = theObjects.iterator();  
        Iterator<String> iterOther = other.theObjects.iterator();  
        while (iterThis.hasNext() && iterOther.hasNext()) {  
            T t = iterThis.next();  
            String string = iterOther.next();  
            if (t.compareTo(string) != 0) return false;  
        }  
        return true;  
    }  
}
```

```
}
```

In the example the type parameter `T` is required to be `Comparable<T>`, because objects of type `T` are stored in a `TreeSet<T>`. At the same time the type parameter `T` is required to be `Comparable<String>`, because we invoke the type parameter's `compareTo(String)` method. The compiler rejects the attempt of specifying two instantiations of the `Comparable` interface as bound of the type parameter.

One workaround for the example above could be the following: we could drop the requirement that the parameter `T` must be `Comparable<T>`, because the corresponding `compareTo(T)` method is not invoked in the implementation of the generic class itself, but in the operations of the `TreeSet<T>`. By dropping the requirement we would risk that a type argument is supplied that is not `Comparable<T>` and will cause `ClassCastExceptions` when operations of the `TreeSet<T>` are invoked. Clearly not a desirable solution, but perhaps a viable one for this particular example.

However, this might not be a solution if the class uses the type parameter in a slightly different way. For instance, if both `compareTo` methods were called in the implementation of the generic class, then we could not drop any of the bounds.

Example (another class with illegal use of two instantiations of the same generic type as bounds of a type parameter):

```
class SomeClass<T extends Comparable<T> & Comparable<String>> { // error
    ...
    private void method(T t1, T t2) {
        ... t1.compareTo(t2) ...
        ... t1.compareTo("string") ...
    }
}
```

If the methods of the bounds are invoked in the class implementation, then dropping one of the conflicting bounds does not solve the problem. One could consider use of an additional interface, such as a `CombinedComparable` interface that combines the two required interfaces into one interface.

Example (conceivable work-around; does not work):

```
interface CombinedComparable<T> {
    int compareTo(T other);
    int compareTo(String other);
}
class SomeClass<T extends CombinedComparable<T>> {
    ...
    private void m(T t1, T t2) {
        ... t1.compareTo(t2) ...
        ... t1.compareTo("string") ...
    }
    public boolean equals(SomeClass<String> other) { // error
        ...
    }
}
```

However, this is not really a viable solution, because it excludes class `String` as a type argument. `String` is a class that is comparable to itself and to `String`, but it does not implement a `CombinedComparable` interface. Hence type `String` is not within bounds. Another conceivable alternative is definition of one new interface per instantiation needed, such as a parameterized `SelfComparable` and a non-parameterized `StringComparable` interface. Again, this excludes class `String` as a potential type argument. If it acceptable that class `String` is excluded as a potential type argument then the definition of additional interfaces might be a viable workaround.

But there remain some situations, in which additional interfaces do not help. For instance, if the type parameter

is used as type argument of another parameterized method, then it must itself be within the bounds of that other type parameter.

Example (another class with illegal use of two instantiations of the same generic type as bounds of a type parameter):

```
class AnUnrelatedClass {
    public static <T extends Comparable<String>> void f(T t) { ... }
}
class AnotherUnrelatedClass {
    public static <T extends Comparable<T>> void g(T t) { ... }
}
class SomeClass<T extends Comparable<T> & Comparable<String>> { // error
    ...
    private void h(T t) {
        AnUnrelatedClass.f(t);
        AnotherUnrelatedClass.g(t);
    }
}
```

No solution sketched out above would address this situation appropriately. If we required that the type parameter be `CombinedComparable`, it would not be within the bounds of at least one of the two invoked methods. Note, that the `CombinedComparable` interface can be a subinterface of only one of the two instantiations of `Comparable`, but not both.

Example (conceivable work-around; does not work):

```
interface CombinedComparable<T> extends Comparable<String> {
    int compareTo(T other);
}
class ObjectStore<T extends CombinedComparable<T>> {
    ...
    private void h(T t) {
        AnUnrelatedClass.f(t);
        AnotherUnrelatedClass.g(t); // error
    }
}
```

The same happens when we require that the type parameter be `SelfComparable` and `StringComparable`. Even if both were subinterfaces of the respective instantiation of `Comparable`, there cannot be a class that implements both, because that class would indirectly implement the two instantiations of `Comparable`.

Ultimately the realization is that, depending on the circumstances, there might not be a work around at all.

LINK TO THIS [TypeParameters.FAQ104](#)

REFERENCES [Can I use different instantiations of a same generic type as bounds of a type parameter?](#)  
[Can a class implement different instantiations of the same parameterized interface?](#)

---

## Does a bound that is a class type give access to all its public members?

*Yes, except any constructors.*

A bound that is a class gives access to all its public members, that is, public fields, methods, and nested type. Only constructors are not made accessible, because there is no guarantee that a subclass of the bound has the same constructors as the bound.

Example (of a class used as bound of a type parameter):

```
public class SuperClass {

    // static members
    public enum EnumType {THIS, THAT}
    public static Object staticField;
    public static void staticMethod() { ... }

    // non-static members
    public class InnerClass { ... }
    public Object nonStaticField;
    public void nonStaticMethod() { ... }

    // constructors
    public SuperClass() { ... }

    // private members
    private Object privateField;

    ...
}

public final class SomeClass<T extends SuperClass> {
    private T object;
    public SomeClass(T t) { object = t; }

    public String toString() {
        return
            "static nested type      : "+T.EnumType.class+"\n"
            +"static field            : "+T.staticField+"\n"
            +"static method              : "+T.staticMethod()+"\n"
            +"non-static nested type: "+T.InnerClass.class+"\n"
            +"non-static field          : "+object.nonStaticField+"\n"
            +"non-static method        : "+object.nonStaticMethod()+"\n"
            +"constructor                : "+(new T())+"\n"           // error
            +"private member          : "+object.privateField+"\n"   // error
        ;
    }
}
```

The bound `SuperClass` gives access to its nested types, static fields and methods and non-static fields and methods. Only the constructor is not accessible. This is because constructors are not inherited. Every subclass defines its own constructors and need not support its superclass's constructors. Hence there is no guarantee that a subclass of `SuperClass` will have the same constructor as its superclass.

---

Although a superclass bound gives access to types, fields and methods of the type parameter, only the non-static methods are dynamically dispatched. In the unlikely case that a subclass redefines types, fields and static methods of its superclass, these redefinitions would not be accessible through the superclass bound.

Example (of a subclass of the bound used for instantiation):

```
public final class SubClass extends SuperClass {

    // static members
    public enum Type {FIX, FOXI}
    public static Object staticField;
    public static Object staticMethod() { ... }

    // non-static members
    public class Inner { ... }
    public Object nonStaticField;
```

```

public Object nonStaticMethod() { ... }

// constructors
public SubClass(Object o) { ... }
public SubClass(String s) { ... }

...
}

```

---

```

SomeClass<SubClass> ref = new SomeClass<SubClass>(new SubClass("xxx"));
System.out.println(ref);

```

---

```

prints:
static nested type      : SuperClass.EnumType
static field            : SuperClass.staticField
static method          : SuperClass.staticMethod => SuperClass.staticField
non-static nested type: SuperClass.InnerClass
non-static field       : SuperClass.nonStaticField
non-static method      : SubClass.nonStaticMethod => SubClass.nonStaticField

```

Calling the `nonStaticMethod` results in invocation of the subclass's overriding version of the `nonStaticMethod`. In contrast, the subclass's redefinitions of types, fields and static methods are not accessible through the bounded parameter. This is nothing unusual. First, it is poor programming style to redefine in a subclass any of the superclass's nested types, fields and static methods. Only non-static methods are overridden. Second, the kind of hiding that we observe in the example above also happens when a subclass object is used through a superclass reference variable.

LINK TO THIS

[TypeParameters.FAQ105](#)

REFERENCES

---

## How do I decrypt "Enum<E extends Enum<E>>"?

*As a type that can only be instantiation for its subtypes, and those subtypes will inherit some useful methods, some of which take subtype arguments (or otherwise depend on the subtype).*

The context in which "Enum<E extends Enum<E>>" appears is the declaration of the `Enum` class in package `java.lang`:

```

public abstract class Enum<E extends Enum<E>> {
    ...
}

```

The type `Enum` is the common base class of all enumeration types. In Java an enumeration type such as `Color` is translated into a class `Color` that extends `Enum<Color>`. The purpose of the superclass `Enum` is to provide functionality that is common to all enumeration types.

Here is a sketch of class `Enum`:

```

public abstract class Enum<E extends Enum<E>> implements Comparable<E>,
Serializable {
    private final String name;
    public final String name() { ... }

    private final int ordinal;
    public final int ordinal() { ... }
}

```



```

protected Enum(String name, int ordinal) { ... }

public String          toString() { ... }
public final boolean   equals(Object other) { ... }
public final int       hashCode() { ... }
protected final Object clone() throws CloneNotSupportedException { ... }
public final int       compareTo(E o) { ... }

public final Class<E> getDeclaringClass() { ... }
public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name) {
... }
}

```

The surprising feature in the declaration "Enum<E extends Enum<E>>" is the fact that the newly defined class Enum and its newly defined type parameter E appear in the bound of that same type parameter. It means that the Enum type must be instantiated for one of its subtypes. In order to understand why this makes sense, consider that every enum type is translated into a subtype of Enum.

Here is the contrived enum type Color:

```
enum Color {RED, BLUE, GREEN}
```

The compiler translates it into the following class:

```

public final class Color extends Enum<Color> {
    public static final Color[] values() { return (Color[])$VALUES.clone(); }
    public static Color valueOf(String name) { ... }

    private Color(String s, int i) { super(s, i); }

    public static final Color RED;
    public static final Color BLUE;
    public static final Color GREEN;

    private static final Color $VALUES[];

    static {
        RED = new Color("RED", 0);
        BLUE = new Color("BLUE", 1);
        GREEN = new Color("GREEN", 2);
        $VALUES = (new Color[] { RED, BLUE, GREEN });
    }
}

```

The inheritance has the effect that the Color type inherits all the methods implemented in Enum<Color>. Among them is the compareTo method. The Color.compareTo method should probably take a Color as an argument. In order to make this happen class Enum is generic and the Enum.compareTo method takes Enum's type parameter E as an argument. As a result, type Color derived from Enum<Color> inherits a compareTo method that takes a Color as an argument, exactly as it should.

If we dissect the declaration "Enum<E extends Enum<E>>" we can see that this pattern has several aspects.

First, there is the fact that the type parameter bound is the type itself: "Enum<E extends Enum<E>>". It makes sure that only subtypes of type Enum are permitted as type arguments. (Theoretically, type Enum could be instantiated on itself, like in Enum<Enum>, but this is certainly not intended and it is hard to imagine a situation in which such an instantiation would be useful.)

Second, there is the fact that the type parameter bound is the parameterized type Enum<E>, which uses the type parameter E as the type argument of the bound. This declaration makes sure that the inheritance relationship between a subtype and an instantiation of Enum is of the form "X extends Enum<X>". A subtype such as "X extends Enum<Y>" cannot be declared because the type argument Y would not be within bounds; only subtypes

of `Enum<X>` are within bounds.

Third, there is the fact that `Enum` is generic in the first place. It means that some of the methods of class `Enum` take an argument or return a value of an unknown type (or otherwise depend on an unknown type). As we already know, this unknown type will later be a subtype `X` of `Enum<X>`. Hence, in the parameterized type `Enum<X>`, these methods involve the subtype `X`, and they are inherited into the subtype `X`. The `compareTo` method is an example of such a method; it is inherited from the superclass into each subclass and has a subclass specific signature in each case.

To sum it up, the declaration "`Enum<E> extends Enum<E>>`" can be decyphered as: `Enum` is a generic type that can only be instantiated for its subtypes, and those subtypes will inherit some useful methods, some of which take subtype specific arguments (or otherwise depend on the subtype).

LINK TO THIS [TypeParameters.FAQ106](#)

REFERENCES [How is a generic type defined?](#)  
[What is a bounded type parameter?](#)  
[Where is a type parameter visible \(or invisible\)?](#)

---

## Why is there no lower bound for type parameters?

*Because it does not make sense for type parameters of classes; it would occasionally be useful in conjunction with method declarations, though.*

Type parameters can have several upper bounds, but no lower bound. This is mainly because lower bound type parameters of classes would be confusing and not particularly helpful. In conjunctions with method declarations, type parameters with a lower bound would occasionally be useful. In the following, we first discuss lower bound type parameters of [classes](#) and subsequently lower bound type parameters of [methods](#).

---

### Lower Bound Type Parameters of Classes

Type parameters can have several bounds, like in `class Box<T extends Number> {...}`. But a type parameter can have no lower bound, that is, a construct such as `class Box<T super Number> {...}` is not permitted. Why not? The answer is: it is pointless because it would not buy you anything, were it allowed. Let us see why lower bound type parameters of classes are confusing by exploring what an upper bound on a type parameter means.

The upper bound on a type parameter has three effects:

1. **Restricted Instantiation.** The upper bound restricts the set of types that can be used for instantiation of the generic type. If we declare a `class Box<T extends Number> {...}` then the compiler would ensure that only subtypes of `Number` can be used as type argument. That is, a `Box<Number>` or a `Box<Long>` is permitted, but a `Box<Object>` or `Box<String>` would be rejected.

Example (of restricted instantiation due to an upper bound on a type parameter):

```
class Box<T extends Number> {  
    private T value;  
    public Box(T t) { value = t; }  
}
```

```

    ...
}
class Test {
    public static void main(String[] args) {
        Box<Long>    boxOfLong    = new Box<Long>(0L);    // fine
        Box<String> boxOfString = new Box<String>(""); // error: String is not
        within bounds
    }
}

```

2. **Access To Non-Static Members.** The upper bound gives access to all public non-static methods and fields of the upper bound. In the implementation of our class `Box<T extends Number> {...}` we can invoke all public non-static methods defined in class `Number`, such as `intValue()` for instance. Without the upper bound the compiler would reject any such invocation. Example (of access to non-static members due to an upper bound on a type parameter):

```

class Box<T extends Number> {
    private T value;
    public Box(T t) { value = t; }
    public int increment() { return value.intValue()+1; } // <= would be an error
    without the Number bound
    ...
}

```

1. **Type Erasure.** The leftmost upper bound is used for type erasure and replaces the type parameter in the byte code. In our class `Box<T extends Number> {...}` all occurrences of `T` would be replaced by the upper bound `Number`. For instance, if class `Box` has a private field of type `T` and a method `void set(T content)` for setting this private field, then the field would be of type `Number` after type erasure and the method would be translated to a method `void set(Number content)`.

Example (of use of upper bound on a type parameter in type erasure - *before* type erasure):

```

class Box<T extends Number> {
    private T value;
    public Box(T t) { value = t; }
    ...
}

```

Example (of use of upper bound on a type parameter in type erasure - *after* type erasure):

```

class Box {
    private Number value;
    public Box(Number t) { value = t; }
    ...
}

```

In addition, the leftmost upper bound appears in further locations, such as automatically inserted casts and bridge methods.

If lower bounds were permitted on type parameters, which side effects would or should they have? If a construct such as `class Box<T super Number> {...}` were permitted, what would it mean? What would the 3 side effects of an upper type parameter bound - restricted instantiation, access to non-static member, type erasure - mean for a lower bound?

**Restricted Instantiations.** The compiler could restrict the set of types that can be used for instantiation of the generic type with a lower bound type parameter. For instance, the compiler could permit instantiations such as `Box<Number>` and `Box<Object>` from a `Box<T super Number>` and reject instantiations such as `Box<Long>` or `Box<Short>`. This would be an effect in line with the restrictive side-effect described for upper type parameter

bounds.

*Access To Non-Static Members.* A lower type parameter bound does not give access to any particular methods beyond those inherited from class `Object`. In the example of `Box<T super Number>` the supertypes of `Number` have nothing in common, except that they are reference types and therefore subtypes of `Object`. The compiler cannot assume that the field of type `T` is of type `Number` or a subtype thereof. Instead, the field of type `T` can be of any supertype of `Number`, such as `Serializable` or `Object`. The invocation of a method such as `intValue()` is no longer type-safe and the compiler would have to reject it. As a consequence, the lower type parameter bound would not give access to an non-static members beyond those defined in class `Object` and thus has the same effect as "no bound".

*Type Erasure.* Following this line of logic, it does not make sense to replace the occurrences of the type parameter by its leftmost lower bound. Declaring a method like the constructor `Box(T t)` as a constructor `Box(Number t)` does not make sense, considering that `T` is replaced by a supertype of `Number`. An `Object` might be rightly passed to the constructor in an instantiation `Box<Object>` and the constructor would reject it. This would be dead wrong. So, type erasure would replace all occurrences of the type variable `T` by type `Object`, and not by its lower bound. Again, the lower bound would have the same effect as "no bound".

Do you want to figure out what it would mean if both lower *and* upper bounds were permitted? Personally, I do not even want to think about it and would prefer to file it under "not manageable", if you permit.

The bottom line is: all that a "super" bound would buy you is the restriction that only supertypes of `Number` can be used as type arguments. And even that is frequently misunderstood. It would NOT mean, that `class Box<T super Number> {...}` contains only instances of supertypes of `Number`. Quite the converse - as the example below demonstrates!

Example (of use of upper bound on a type parameter in type erasure - *before* type erasure):

```
class Box<T super Number> {
    private T value;
    public Box(T t) { value = t; }
    ...
}
```

Example (of use of upper bound on a type parameter in type erasure - *after* type erasure):

```
class Box {
    private Object value;
    public Box(Object t) { value = t; }
    ...
}
```

A `class Box<T super Number> {...}` would be translated by type erasure to a `Box` containing an `Object` field and its constructor would be translated to `Box(Object t)`. That's fundamentally different from a `class Box<T extends Number> {...}`, which would be translated to a `Box` containing a `Number` field and its constructor would be translated to `Box(Number t)`. Consequently, a `Box<Number>` instantiated from a `class Box<T extends Number> {...}` would be different from a `Box<Number>` instantiation from a `class Box<T super Number> {...}`, which is likely to cause confusion. For this reason lower bounds do not make sense on type parameters of classes.

---

## Lower Bound Type Parameters of Methods

In conjunction with methods and their argument types, a type parameter with a lower bound can occasionally be useful.

Example (of a method that would profit from a type parameter with a lower bound):

```
class Pair<X,Y> {
    private X first;
```

```

private Y second;
...
public <A super X,B super Y> B addToMap(Map<A,B> map) { // error: type
parameter cannot have lower bound
    return map.put(first, second);
}
}
class Test {
    public static void main(String[] args) {
        Pair<String,Long> pair = new Pair<>("ABC",42L);
        Map<CharSequence, Number> map = HashMap<CharSequence, Number>();
        Number number = pair.addToMap(map);
    }
}

```

The `addToMap()` method adds the content of the pair to a map. Any map that can hold supertypes of `x` and `y` would do. The map's `put()` method returns the value found in the map for the given key, if there already is a key-value entry for the key in the map. The return value of the map's `put()` method shall be returned from the `addToMap()` method. Under these circumstances one would like to declare the method as shown above: The map is parameterized with supertypes of the pair's type parameters and the `addToMap()` method's return type is the map's value type.

Since the compiler does not permit lower bounds on type parameters we need a work-around.

One work-around that comes to mind is use of a wildcard, because wildcards can have a lower bound. Here is a work-around using a wildcard.

Example (of a work-around for the previous example using wildcards):

```

class Pair<X,Y> {
    private X first;
    private Y second;
    ...
    public Object addToMap(Map<? super X, ? super Y> map) {
        return map.put(first, second);
    }
}
class Test {
    public static void main(String[] args) {
        Pair<String,Long> pair = new Pair<>("ABC",42L);
        Map<CharSequence, Number> map = HashMap<CharSequence, Number>();
        Number number = (Number)pair.addToMap(map);
    }
}

```

It works, except that there is no way to declare the return type as desired. It would be the supertype of `y` that the compiler captures from the map type, but there is no syntax for specifying it. We must not declare the return type a `"? super Y"`, because `"? super Y"` is a wildcard and not a type and therefore not permitted as a return type. We have no choice and must use `Object` instead as our method's return type. This rather unspecific return type in turn forces callers of the `addToMap()` method into casting the return value down from `Object` to its actual type. This is not exactly what we had in mind.

Another work-around is use of static methods. Here is a work-around with a static instead of a non-static method.

Example (of a work-around for the previous example using a static method):

```

class Pair<X,Y> {

```

```

private X first;
private Y second;
...
public static <A,B,X extends A,Y extends B> B addToMap(Pair<X,Y> pair,
Map<A,B> map) {
    return map.put(pair.first,pair.second);
}
}
class Test {
    public static void main(String[] args) {
        Pair<String,Long> pair = new Pair<>("ABC",42L);
        Map<CharSequence, Number> map = HashMap<CharSequence, Number>();
        Number number = Pair.addToMap(pair,map);
    }
}

```

The generic `addToMap()` method has four type parameters: two placeholders `x` and `y` for the pair's type and two placeholders `A` and `B` for the map's type. `A` and `B` are supertypes of `x` and `y`, because `x` and `y` are declared with `A` and `B` as their upper bounds. (Note, that the generic method's type parameters `x` and `y` have nothing to do with the `Pair` class's `x` and `y` parameters. The names `x` and `y` are reused for the generic method to make them easily recognizable as the pair's type parameters.) Using four type parameters we can declare the precise return type as desired: it is the same type as the value type of the map.

---

The bottom line is that the usefulness of lower bounds on type parameters is somewhat debatable. They would be confusing and perhaps even misleading when used as type parameters of a generic class. On the other hand, generic methods would occasionally profit from a type parameter with a lower bound. For methods, a work-around for the lack of a lower bound type parameter can often be found. Such a work-around typically involves a static generic method or a lower bound wildcard.

LINK TO THIS [TypeParameters.FAQ107](#)

#### REFERENCES

[What is a bounded type parameter?](#)  
[Does a bound that is a class type give access to all its public members?](#)  
[What is a bounded wildcard?](#)  
[What is the difference between a wildcard bound and a type parameter bound?](#)

---

## Usage

---

### Can I use a type parameter like a type?

*No, a type parameter is not a type in the regular sense (different from a regular type such as a non-generic class or interface).*

Type parameters can be used for typing (like non-generic classes and interfaces)::

- as argument and return types of methods
- as type of a field or local reference variable
- as type argument of other parameterized types
- as target type in casts
- as explicit type argument of parameterized methods

Type parameters can NOT be used for the following purposes (different from non-generic classes and

interfaces)::

- for creation of objects
- for creation of arrays
- in exception handling
- in static context
- in instanceof expressions
- as supertypes
- in a class literal

LINK TO THIS

[TypeParameters.FAQ200](#)

REFERENCES

---

## Can I create an object whose type is a type parameter?

*No, because the compiler does not know how to create objects of an unknown type.*

Each object creation is accompanied by a constructor call. When we try to create an object whose type is a type parameter then we need an accessible constructor of the unknown type that the type parameter is a place holder for. However, there is no way to make sure that the actual type arguments have the required constructors.

Example (illegal generic object creation):

```
public final class Pair<A,B> {
    public final A fst;
    public final B snd;

    public Pair() {
        this.fst = new A(); // error
        this.snd = new B(); // error
    }
    public Pair(A fst, B snd) {
        this.fst = fst;
        this.snd = snd;
    }
}
```

In the example above, we are trying to invoke the no-argument constructors of two unknown types represented by the type parameters A and B. It is not known whether the actual type arguments will have an accessible no-argument constructor.

In situations like this - when the compiler needs more knowledge about the unknown type in order to invoke a method - we use type parameter bounds. However, the bounds only give access to methods of the type parameter. Constructors cannot be made available through a type parameter bound.

If you need to create objects of unknown type, you can use reflection as a workaround. It requires that you supply type information, typically in form of a `Class` object, and then use that type information to create objects via reflection.

Example (workaround using reflection):

```
public final class Pair<A,B> {
    public final A fst;
    public final B snd;

    public Pair(Class<A> typeA, Class<B> typeB) {
```

```

        this.fst = typeA.newInstance();
        this.snd = typeB.newInstance();
    }
    public Pair(A fst, B snd) {
        this.fst = fst;
        this.snd = snd;
    }
}

```

LINK TO THIS [TypeParameters.FAQ201](#)

#### REFERENCES

[Does a bound that is a class type give access to all its methods and fields?](#)  
[How do I generically create objects and arrays?](#)  
[How do I pass type information to a method so that it can be used at runtime?](#)

## Can I create an array whose component type is a type parameter?

*No, because the compiler does not know how to create an object of an unknown component type.*

We can declare array variables whose component type is a type parameter, but we cannot create the corresponding array objects. The compiler does not know how to create an array of an unknown component type.

Example (before type erasure):

```

class Sequence<T> {
    ...
    public T[] asArray() {
        T[] array = new T[size]; // error
        ...
        return array;
    }
}

```

Example (after a conceivable translation by type erasure):

```

class Sequence {
    ...
    public Object[] asArray() {
        Object[] array = new Object[size];
        ...
        return array;
    }
}

```

The type erasure of a type parameter is its leftmost bound, or type `Object` if no bound was specified. As a result, the compiler would create an array of `Objects` in our example. This is not what we want. If we later invoked the `asArray` method on a `Sequence<String>` a `Object[]` would be returned, which is incompatible to the `String[]` that we expect.

Example (invocation of illegal method):

```

Sequence<String> seq = new Sequence<String>();
...
String[] arr = seq.asArray(); // compile-time error
String[] arr = (String[])seq.asArray(); // runtime failure: ClassCastException

```



Not even a cast would help because the cast is guaranteed to fail at runtime. The returned array is really an array of Objects, not just a reference of type Object[] referring to a String[].

If you need to create arrays of an unknown component type, you can use reflection as a workaround. It requires that you supply type information, typically in form of a Class object, and then use that type information to create arrays via reflection.

Example (workaround using reflection):

```
class Sequence<T> {
    ...
    public T[] asArray(Class<T> type) {
        T[] array = (T[])Array.newInstance(type,size); // unchecked cast
        ...
        return array;
    }
}
```

By the way, the unchecked warning is harmless and can be ignored. It stems from the need to cast to the unknown array type, because the newInstance method returns an Object[] as a result.

LINK TO THIS [TypeParameters.FAQ202](#)

REFERENCES [What is type erasure?](#)  
[How do I generically create objects and arrays?](#)  
[How do I pass type information to a method so that it can be used at runtime?](#)

---

## Can I cast to the type that the type parameter stands for?

*Yes, you can, but it is not type-safe and the compiler issues an "unchecked" warning.*

Type parameters do not have a runtime type representation of their own. They are represented by their leftmost bound, or type Object in case of an unbounded type parameter. A cast to a type parameter would therefore be a cast to the bound or to type Object.

Example (of unchecked cast):

```
class Twins<T> {
    public T fst,snd;
    public Twins(T s, T t) { fst = s; snd = t; }
    ...
}
class Pair<S,T> {
    private S fst;
    private T snd;
    public Pair(S s, T t) { fst = s; snd = t; }
    ...
    public <U> Pair(Twins<U> twins) {
        fst = (S) twins.fst; // unchecked warning
        snd = (T) twins.snd; // unchecked warning
    }
}
```

The two casts to the type parameters are pointless because they will never fail; at runtime they are casts to type Object. As a result any type of Pair can be constructed from any type of Twins. We could end up with a Pair<Long,Long> that contains Strings instead of Longs. This would be a blatant violation of the type-safety

principle, because we would later trigger an unexpected `ClassCastException`, when we use this offensive `Pair<Long,Long>` that contains `Strings`. In order to draw attention to the potentially unsafe casts the compiler issues "unchecked" warnings.

LINK TO THIS [TypeParameters.FAQ203](#)

REFERENCES [What does type-safety mean?](#)  
[What is type erasure?](#)  
[What is the type erasure of a type parameter?](#)

---

## Can I use a type parameter in exception handling?

*It depends.*

Type parameters can appear in `throws` clauses, but not in `catch` clauses.

LINK TO THIS [TypeParameters.FAQ204](#)

REFERENCES [Can I use a type parameter in a catch clause?](#)  
[Can I use a type parameter in in a throws clause?](#)  
[Can I throw an object whose type is a type parameter?](#)

---

## Can I derive from a type parameter?

*No, because a type parameter does not have a runtime type representation of its own.*

As part of the translation by type erasure, all type parameters are replaced by their leftmost bound, or `Object` if the type parameter is unbounded. Consequently, there is no point to deriving from a type parameter, because we would be deriving from its bound, not from the type that the type parameter stands for. In addition, the actual type argument can be a final class or an enum type, from which we must not derive anyway.

Example (of illegal derivation from type parameter; before type erasure):

```
class Printable<T extends Collection<?>> extends T { // illegal
    public void printElements(PrintStream out) {
        for (Object o : this) out.println(o);
    }
    public void printElementsInReverseOrder(PrintStream out) {
        ...
    }
}
final class Test {
    public static void main(String[] args) {
        Printable<LinkedList<String>> list = new Printable<LinkedList<String>>();
        list.add(2, "abc");
        list.printElements(System.out);
    }
}
```

The idea of this generic subclass is that it adds `print` functionality to all collection classes by means of derivation. A `Printable<LinkedList<String>>` would have all the functionality of `LinkedList<String>` plus the `print` functionality. (This idiom is known in C++ as the *curiously recurring template pattern*). Since it is illegal to derive from a type parameter, this kind of programming technique is not possible in Java. Consider what the subclass would look like after type erasure.

Example (same example after a conceivable translation by type erasure):

```
class Printable extends Collection {    // error: Collection is an interface, not
class
    public void printElements(PrintStream out) {
        for (Object o : this)    out.println(o);
    }
    public void printElementsInReverseOrder(PrintStream out) {
        ...
    }
}

final class Test {
    public static void main(String[] args) {
        Printable list = new Printable();
        list.add(2, "abc");          // error: no such method can be found in
class Printable
        list.printElements(System.out);
    }
}
```

After type erasure the subclass `Printable` would not be a subclass of `LinkedList`, but a subclass of `Collection`, which is not even possible, because `Collection` is an interface, not a class. Even if we used a class as the bound of the type parameter, such as `<T extends AbstractCollection>`, none of the list-specific methods would be available in the subclass, which entirely defeats the purpose of this programming pattern.

LINK TO THIS [TypeParameters.FAQ205](#)

#### REFERENCES

[What is type erasure?](#)

[Which types are permitted as type arguments?](#)

[The Curiously Recurring Template Pattern in C++](#) (James O. Coplien. A Curiously Recurring Template Pattern. In C++ Gems, 135-144. Cambridge University Press, New York, 1996)

---

## Why is there no class literal for a type parameter?

*Because a type parameter does not have a runtime type representation of its own.*

As part of the translation by type erasure, all type parameters are replaced by their leftmost bound, or `Object` if the type parameter is unbounded. Consequently, there is no point to forming class literals such as `T.class`, where `T` is a type parameter, because no such `Class` objects exist. Only the bound has a `Class` object that represents its runtime type.

Example (before type erasure):

```
<T extends Collection> Class<?> someMethod(T arg){
    ...
    return T.class; // error
}
```

The compiler rejects the expression `T.class` as illegal, but even if it compiled it would not make sense. After type erasure the method above could at best look like this:

Example (after type erasure):

```
Class someMethod(Collection arg){
    ...
    return Collection.class;
}
```

The method would always return the bound's type representation, no matter which instantiation of the generic method was invoked. This would clearly be misleading.

The point is that type parameters are *non-reifiable*, that is, they do not have a runtime type representation. Consequently, there is no `Class` object for type parameters and no `class` literal for them.

LINK TO THIS [TypeParameters.FAQ206](#)

REFERENCES [What is type erasure?](#)  
[What is a reifiable type?](#)

---

## Scope

---

### Where is a type parameter visible (or invisible)?

*Everywhere in the definition of a generic type or method, except any static context of a type.*

#### *Generic Classes*

The scope of a class's type parameter is the entire definition of the class, except any static members or static initializers of the class. This means that the type parameters cannot be used in the declaration of static fields or methods or in static nested types or static initializers.

Example (of illegal use of type parameter in static context of a generic class):

```
class SomeClass<T> {
    // static initializer, static field, static method
    static {
        SomeClass<T> test = new SomeClass<T>(); // error
    }
    private static T globalInfo;           // error
    public static T getGlobalInfo() {      // error
        return globalInfo;
    }
    // non-static initializer, non-static field, non-static method
    {
        SomeClass<T> test = new SomeClass<T>();
    }
    private T localInfo;
    public T getLocalInfo() {
        return localInfo;
    }
    // static nested types
    public static class Failure extends Exception {
        private final T info;             // error
        public Failure(T t) { info = t; } // error
        public T getInfo() { return info; } // error
    }
    private interface Copyable {
        T copy();                          // error
    }
    private enum State {
```

```

    VALID, INVALID;
    private T info; // error
    public void setInfo(T t) { info = t; } // error
    public T getInfo() { return info; } // error
}
// non-static nested types
public class Accessor {
    public T getInfo() { return localInfo; }
}
}

```

The example illustrates that the type parameter cannot be used in the static context of a generic class. It also shows that nested interfaces and enum types are considered static type members of the class. Only inner classes, that is, non-static nested classes, can use the type parameter of the enclosing generic class.

---

### Generic Interfaces

The scope of an interface's type parameter is the entire definition of the interface, except any fields or nested types. This is because fields and nested types defined in an interface are implicitly static.

Example (of illegal use of type parameter in a generic interface):

```

interface SomeInterface<T> {
    // field
    SomeClass<T> value = new SomeClass<T>(); // error
    // nested type
    class Accessor {
        public T getInfo() { // error
            return value.getGlobalInfo();
        }
    }
    // methods
    T getValue();
}

```

The example shows that fields of an interface are implicitly static, so that the type parameter cannot be used anywhere in the declaration of a field of a generic interface. Similarly, the nested class is considered a static nested class, not an inner class, and for this reason use of the type parameter anywhere in the nested class is illegal.

---

### Generic Methods

The scope of a method's or constructor's type parameter is the entire definition of the method; there is no exception, because a method has no static parts.

Example (of use of type parameter in a generic method):

```

private interface Copyable<T> {
    T copy();
}
// non-static method
<T extends Copyable<T>> void nonStaticMethod(T t) {
    final T copy = t.copy();

    class Task implements Runnable {
        public void run() {
            T tmp = copy;
            System.out.println(tmp);
        }
    }
}

```

```

    }
}
(new Task()).run();
}
// static method
static <T extends Copyable<T>> void staticMethod(T t) {
    final T copy = t.copy();

    class Task implements Runnable {
        public void run() {
            T tmp = copy;
            System.out.println(tmp);
        }
    }
    (new Task()).run();
}

```

The example illustrates that the type parameter can be used any place in the definition of a generic method. The type parameter can appear in the return and argument type. It can appear in the method body and also in local (or anonymous) classes defined inside the method. Note, that it does not matter whether the generic method itself is static or non-static. Methods, different from types, do not have any "static context"; there is no such thing as a static local variable or static local class.

LINK TO THIS [TypeParameters.FAQ301](#)

REFERENCES [Why can't I use a type parameter in any static context of the generic class?](#)  
[Can I use a type parameter as part of its own bounds or in the declaration of other type parameters?](#)

## Can I use a type parameter as part of its own bounds?

*Yes, the scope of a type parameter includes the type parameter section itself.*

The type parameters of a generic type or method are visible in the entire declaration of the type or method, including the type parameter section itself. Therefore, type parameters can appear as parts of their own bounds, or as bounds of other type parameters declared in the same section.

Example (of use of a type parameter in the type parameter section itself):

```

public final class Wrapper<T extends Comparable<T>> implements
Comparable<Wrapper<T>> {
    private final T theObject;
    public Wrapper(T t) { theObject = t; }
    public T getWrapper() { return theObject; }
    public int compareTo(Wrapper<T> other) {
        return theObject.compareTo(other.theObject);
    }
}

```

In the example above, the type parameter `T` is used as type argument of its own bound `Comparable<T>`.

Example (of use of a type parameter in the type parameter section itself):

```

<S, T extends S> T create(S arg) { ... }

```

In the example above, the first type parameter `s` is used as bound of the second type parameter `T`.

Forward references to type parameters are not permitted. The type parameter cannot be used in the entire type

parameter section, but only after its point of declaration.

Example (of an illegal forward reference to a type parameter):

```
<S extends T, T extends Comparable<S>> T create(S arg) { ... } // error
```

In the example above, the type parameter `T` is used in the type parameter section before it has been defined in the same type parameter section. This kind of forward reference is illegal.

Forward references to types, not type parameters, are permitted, though.

Example (of an forward reference to a type):

```
interface Edge<N extends Node<? extends Edge<N>>> {
    N getBeginNode();
    void setBeginNode(N n);
    N getEndNode();
    void setEndNode(N n);
}
interface Node<E extends Edge<? extends Node<E>>> {
    E getOutEdge();
    void setOutEdge(E e);
    E getInEdge();
    void setInEdge(E e);
}
```

In the example above, the type `Node` is used (in the type parameter section of type `Edge`) before it has been defined (probably in a different source file). This kind of forward reference is permitted, which is not surprising. It is the usual way of defining and using types in Java.

LINK TO THIS [TypeParameters.FAQ302](#)

REFERENCES [Where is a type parameter visible \(or invisible\)?](#)  
[Can I use the type parameter of an outer type as part of the bounds of the type parameter of an inner type or a method?](#)

---

## Can I use the type parameter of an outer type as part of the bounds of the type parameter of an inner type or a method?

*Yes, the type parameter of an enclosing generic type or method can be used in the type parameter section of an inner generic type or method.*

The type parameters of a generic type or method can appear as parts of the bounds of the type parameters of any generic type or methods in that scope.

Example (of use of type parameter of enclosing class in the type parameter section of a method):

```
public final class Wrapper<T> {
    private final T theObject;
    public Wrapper(T t) { theObject = t; }
    public <U extends T> Wrapper(Wrapper<U> w) { theObject = w.theObject; }
    public T getWrapper() { return theObject; }
}
```

In the example above, the type parameter `T` of the class `Wrapper<T>` is used as bound of the type parameter `U` of the class's generic constructor.

In principle, you can use the type parameters of a generic class anywhere in the class scope, including the type

parameter sections of any generic methods or nested and inner types. For instance, the type parameters can appear in the type parameter declaration of an inner class.

Example (of use of type parameter of enclosing class in the type parameter section of an inner class):

```
public final class Wrapper<T> {
    private final T theObject;
    public Wrapper(T t) { theObject = t; }
    public T getWrapper() { return theObject; }

    private final class WrapperComparator<W extends Wrapper<? extends
Comparable<T>>>
        implements Comparator<W> {
            public int compare(W lhs, W rhs) {
                return lhs.theObject.compareTo((T)(rhs.theObject));
            }
        }
    public <V extends Wrapper<? extends Comparable<T>>> Comparator<V> comparator() {
        return this.new WrapperComparator<V>();
    }
}
```

In this example, the type parameter `T` of the class `Wrapper<T>` is used as part of the bound of the type parameter `W` of inner class `WrapperComparator`. In addition, it is also used as part of the bound of the type parameter `v` of the comparator method.

Similar rules apply to generic interfaces. Even the type parameters of a generic method can be used in the declaration of the type parameters of a local generic type.

Example (of use of type parameter of a method in the type parameter section of a local class):

```
class Test {
    private static <T> void method() {
        class Local<A extends T> { ... }
        ...
    }
}
```

However, generic local classes are rather rare in practice.

The type parameters of a generic type or method can appear anywhere in the declaration of the type parameters of any generic type or methods in that scope. A type parameter `T` can appear

- as the bound, as in `<U extends T>`, OR
- as part of the bounds, as in `<U extends Comparable<T>>`, OR
- as the bound of a wildcard, as in `<U extends <Comparable<? super T>>`, OR
- as part of the bound of a wildcard, as in `<U extends Wrapper<? extends Comparable<T>>>`.

There is only one restriction: if a type parameter is used as the bound of another type parameter then there must not follow any further bounds.

Example (of illegal use of type parameter as a bound)::

```
class Wrapper<T> implements Cloneable {
    private final T theObject;
    ...
    public <U extends T & Cloneable> Wrapper<U> clone() { ... } // error
}
```

The type parameter `T` is followed by another bound, which is illegal.



LINK TO THIS

[TypeParameters.FAQ303](#)

REFERENCES

[Where is a type parameter visible \(or invisible\)?](#)  
[Can I use a type parameter as part of its own bounds?](#)

---

---

## Static Context

---

### Is there one instances of a static field per instantiation of a generic type?

*No, there is only one instance of a static field for all instantiations of a generic type.*

If a generic type has a static field, how many instances of this static field exist?

Example (of a generic class with a static field):

```
class SomeClass<T> {
    public static int count;
    ...
}
```

The generic type can be instantiated for an arbitrary number of type arguments. Is there a different instance of the static field for each instantiation of the generic type?

Example (of several instantiations and usage of the static field(s)):

```
SomeClass<String> ref1 = new SomeClass<String>();
SomeClass<Long>    ref2 = new SomeClass<Long>();

ref1.count++;
ref2.count++;
```

The question is: are we accessing two different static fields in the code snippet above? The answer is: no, there is only one instance of a static field per parameterized type, not several ones per instantiation of the generic type.

The reason is that the compiler translates the definition of a generic type into one unique byte code representation of that type. The different instantiations of the generic type are later mapped to this unique representation by means of *type erasure*. The consequence is that there is only one static `count` field in our example, despite of the fact that we can work with as many instantiations of the generic class as we like.

Example (showing the syntax for access to a static field of a generic type):

```
SomeClass<String> ref1 = new SomeClass<String>();
SomeClass<Long>    ref2 = new SomeClass<Long>();

ref1.count++;           // discouraged, but legal
ref2.count++;           // discouraged, but legal
SomeClass.count++; // fine, recommended
SomeClass<String>.count++; // error
SomeClass<Long>.count++;   // error
```

Although we can refer to the static field through reference variables of different type, namely of type `SomeClass<String>` and `SomeClass<Long>` in the example, we access the same unique static `count` field. The

uniqueness of the static field is more clearly expressed when we refer to the static field using the enclosing scope instead of object references. Saying `SomeClass.count` makes clear that there is only one static `count` field that is independent of the type parameters of the enclosing class scope. Since the static field is independent of the enclosing class's type parameters it is illegal to use any instantiation of the generic enclosing class as scope qualifier.

LINK TO THIS [TypeParameters.FAQ401](#)

#### REFERENCES

[What is type erasure?](#)

[How do I refer to static members of a parameterized type?](#)

[Where is a type parameter visible \(or invisible\)?](#)

[Why can't I use a type parameter in any static context of the generic class?](#)

---

## Why can't I use a type parameter in any static context of a generic class?

*Because the static context is independent of the type parameters and exists only once per raw type, that is, only once for all instantiations of a generic type.*

Type parameters must not appear in any static context of a generic type, which means that type parameters cannot be used in the declaration of static fields or methods or in static nested types or static initializers.

Example (of illegal use of a type parameter in static context):

```
public final class X<T> {
    private static T field; // error
    public static T getField() { return field; } // error
    public static void setField(T t) { field = t; } // error
}
```

The attempt of declaring a static field of the unknown type `T` is non-sensical and rightly rejected by the compiler. There is only one instance of the static field for *all* instantiations of the generic class. Of which type could that static field possibly be? The declaration of a static field, whose type is the type parameter, makes it look like there were several instances of different types, namely one per instantiation, which is misleading and confusing. For this reason, the use of type parameters for declaration of static fields is illegal.

As static methods often operate on static fields it makes sense to extend the rule to static methods: the type parameter must not appear in a static method.

Interestingly, the same rule applies to static nested types defined in a generic class. There is no compelling technical reason for this restriction. It's just that static nested types are considered independent of any instantiations of the generic class, like the static fields and methods. For this reason, use of the type parameter in a static nested type is illegal. (Note, static nested types include nested static classes, nested interfaces and nested enum types.)

Example (of illegal use of a type parameter in static context):

```
class Wrapper<T> {
    private final T theObject;

    public Wrapper(T t) { theObject = t; }
    public T getWrappedItem() { return theObject; }

    public Immutable makeImmutable() {
        return new Immutable(theObject);
    }
    public Mutable makeMutable() {
        return new Mutable(theObject);
    }
}
```

```

private static <A> A makeClone(A theObject) { ... }

public static final class Immutable {
    private final T theObject;           // error

    public Immutable(T arg) {           // error
        theObject = makeClone(arg);
    }
    public T getWrappedItem() {         // error
        return makeClone(theObject);
    }
}
public static class Mutable {
    ... similar ...
}
}

```

In the example above the type parameter is used in the context of a nested class type. The compiler rejects the use of the type parameter because the class type is a nested *static* class.

---

### ***Workaround for nested static classes and interfaces:***

In case of static nested classes and interfaces this is not a major calamity. As a workaround we can generify the static class or interface itself.

Example (workaround - generify the nested static type):

```

class Wrapper<T> {
    private final T theObject;

    public Wrapper(T t) { theObject = t; }
    public T getWrapper() { return theObject; }

    public Immutable<T> makeImmutable() {
        return new Immutable<T>(theObject);
    }
    public Mutable<T> makeMutable() {
        return new Mutable<T>(theObject);
    }
}
private static <A> A makeClone(A theObject) { ... }

public static final class Immutable<A> { // is a generic class now
    private final A theObject;

    public Immutable(A arg) {
        theObject = makeClone(arg);
    }
    public A getWrappedItem() {
        return makeClone(theObject);
    }
}
public static class Mutable<A> {
    ... similar ...
}
}

```

There is no such workaround for nested enum type because they cannot be generic.

### ***Workaround for nested static classes:***

If the nested static type is a class, an alternative workaround would be turning the static class into a non-static inner class.

Example (workaround - use an inner class):

```
class Wrapper<T> {
    private final T theObject;

    public Wrapper(T t) { theObject = t; }
    public T getWrapper() { return theObject; }

    public Mutable makeMutable() {
        return this.new Mutable(theObject);
    }
    public Immutable makeImmutable() {
        return this.new Immutable(theObject);
    }
    private static <A> A makeClone(A theObject) { ... }

    public final class Immutable { // is no longer a static class
        private final T theObject;

        public Immutable(T arg) {
            theObject = makeClone(arg);
        }
        public T getWrappedItem() {
            return makeClone(theObject);
        }
    }
    public class Mutable
        ... similar ...
}
}
```

This workaround comes with a certain amount of overhead because all inner classes have a hidden reference to an instance of the outer type, which in this example they neither need nor take advantage of; the hidden reference is just baggage. Often, inner classes are combined with interfaces in order to keep the inner class a private implementation detail of the enclosing class. We can do the same here.

Example (the previous workaround refined):

```
class Wrapper<T> {
    private final T theObject;

    public Wrapper(T t) { theObject = t; }
    public T getWrapper() { return theObject; }

    public Mutable<T> makeMutable() {
        return this.new Mutable(theObject);
    }
    public Immutable<T> makeImmutable() {
        return this.new Immutable(theObject);
    }
    private static <A> A makeClone(A theObject) { ... }

    public interface Immutable<S> {
        S getWrappedItem();
    }
    public interface Mutable<S> {
        S getWrappedItem();
        void setWrappedItem(S arg);
    }
}
```

```
}  
private final class ImmutableImplementation implements Immutable<T> {  
    private final T theObject;  
  
    public ImmutableImplementation(T arg) {  
        theObject = makeClone(arg);  
    }  
    public T getWrappedItem() {  
        return makeClone(theObject);  
    }  
}  
private class MutableImplementation implements Mutable<T> {  
    ... similar ...  
}  
}
```

As you can see, the nested public interfaces need to be generic whereas the inner private classes can use enclosing class's the type parameter.

**LINK TO THIS**      [TypeParameters.FAQ402](#)

**REFERENCES**      [Where is a type parameter visible \(or invisible\)?](#)  
                     [Is there one instances of a static field per instantiation of a generic type?](#)

# Type Arguments

© Copyright 2004-2022 by Angelika Langer. All Rights Reserved.

## Fundamentals

- [What is a type argument?](#)
- [Which types are permitted as type arguments?](#)
- [Are primitive types permitted as type arguments?](#)
- [Are wildcards permitted as type arguments?](#)
- [Are type parameters permitted as type arguments?](#)
- [Do type parameter bounds restrict the set of types that can be used as type arguments?](#)
- [Do I have to specify a type argument when I want to use a generic type?](#)
- [Do I have to specify a type argument when I want to invoke a generic method?](#)

## Wildcards

- [What is a wildcard?](#)
- [What is an unbounded wildcard?](#)
- [What is a bounded wildcard?](#)
- [What do multi-level \(or nested\) wildcards mean?](#)
- [If a wildcard appears repeatedly in a type argument section, does it stand for the same type?](#)

## Wildcard Bounds

- [What is a wildcard bound?](#)
- [Which types are permitted as wildcard bounds?](#)
- [What is the difference between a wildcard bound and a type parameter bound?](#)

---

# Type Arguments

---

---

## Fundamentals

---

### What is a type argument?

*A reference type that is used for the instantiation of a generic type or for the instantiation of a generic method, or a wildcard that is used for the instantiation of a generic type . An actual type argument replaces the formal type parameter used in the declaration of the generic type or method.*

Generic types and methods have formal type parameters, which are replaced by actual type arguments when the parameterized type or method is instantiated.

Example (of a generic type):

```
class Box<T> {
    private T theObject;
    public Box(T arg) { theObject = arg; }
    ...
}
```

```

class Test {
    public static void main(String[] args) {
        Box<String> box = new Box<String>("Jack");
    }
}

```

In the example we see a generic class `Box` with one formal type parameter `T`. This formal type parameter is replaced by actual type argument `String`, when the `Box` type is used in the test program.

There are few of rules for type arguments:

- The actual type arguments of a generic type are
  - reference types,
  - wildcards, or
  - parameterized types (i.e. instantiations of other generic types).
- Generic methods cannot be instantiated using wildcards as actual type arguments.
- Type parameters are permitted as actual type arguments.
- Primitive types are not permitted as type arguments.
- Type arguments must be within bounds.

LINK TO THIS [TypeArguments.FAQ001](#)

#### REFERENCES

[What is a wildcard?](#)  
[What is a type parameter?](#)  
[Which types are permitted as type arguments?](#)  
[Are primitive types permitted as type arguments?](#)  
[Are wildcards permitted as type arguments?](#)  
[Are type parameters permitted as type arguments?](#)  
[Do type parameter bounds restrict the set of types that can be used as type arguments?](#)

## Which types are permitted as type arguments?

*All references types including parameterized types, but no primitive types.*

All reference types can be used a type arguments of a parameterized type or method. This includes classes, interfaces, enum types, nested and inner types, and array types. Only primitive types cannot be used as type argument.

Example (of types as type arguments of a parameterized type):

```

List<int>                10;                // error
List<String>            11;
List<Runnable>         12;
List<TimeUnit>         13;
List<Comparable>       14;
List<Thread.State>     15;
List<int[]>             16;
List<Object[]>         17;
List<Callable<String>>  18;
List<Comparable<? super Long>> 19;
List<Class<? extends Number>> 110;
List<Map.Entry<?,?>>    111;

```

The code sample shows that a primitive type such as `int` is not permitted as type argument.

Class types, such as `String`, and interface types, such as `Runnable`, are permitted as type arguments. Enum types, such as `TimeUnit` (see [java.util.concurrent.TimeUnit](#)) are also permitted as type arguments.

Raw types are permitted as type arguments; `Comparable` is an example.

`Thread.State` is an example of a nested type; `Thread.State` is an enum type nested into the `Thread` class. Non-

static inner types are also permitted.

An array type, such as `int[]` and `Object[]`, is permitted as type arguments of a parameterized type or method.

Parameterized types are permitted as type arguments, including concrete parameterized types such as `Callable<String>`, bounded wildcard parameterized types such as `Comparable<? super Long>` and `Class<? extends Number>`, and unbounded wildcard parameterized types such as `Map.Entry<?, ?>`.

The same types are permitted as explicit type arguments of a generic method.

Example (of types as type arguments of a generic method):

```
List<?> list;
list = Collections.<int>emptyList();           // error
list = Collections.<String>emptyList();
list = Collections.<Runnable>emptyList();
list = Collections.<TimeUnit>emptyList();
list = Collections.<Comparable>emptyList();
list = Collections.<Thread.State>emptyList();
list = Collections.<int[]>emptyList();
list = Collections.<Object[]>emptyList();
list = Collections.<Callable<String>>emptyList();
list = Collections.<Comparable<? super Long>>emptyList();
list = Collections.<Class<? extends Number>>emptyList();
list = Collections.<Map.Entry<?, ?>>emptyList();
```

The code sample shows that primitive types such as `int` are not permitted as type argument of a generic method either.

LINK TO THIS [TypeArguments.FAQ002](#)

REFERENCES [What is a type argument?](#)

## Are primitive types permitted as type arguments?

***No. Only reference types can be used as type arguments.***

A parameterized type such as `List<int>` or `Set<short>` is illegal. Only reference types can be used for instantiation of generic types and methods. Instead of `List<int>` we must declare a `List<Integer>`, using the corresponding wrapper type as the type argument.

The lack of primitive type instantiations is not a major restriction in practice (except for performance reasons), because autoboxing and auto-unboxing hides most of the nuisance of wrapping and unwrapping primitive values into their corresponding wrapper types.

Example (of autoboxing):

```
int[] array = {1,2,3,4,5,6,7,8,9,10};
List<Integer> list = new LinkedList<Integer>();
for (int i : array)
    list.add(i);           // autoboxing
for (int i=0;i<list.size();i++)
    array[i] = list.get(i); // auto-unboxing
```

Here we insert primitive type `int` values to the list of `Integers`, relying on autoboxing, which is the automatic conversion from the primitive type to the corresponding wrapper type. Similarly, we extract primitive type `int` values from the list, relying on auto-unboxing, which is the automatic conversion from the wrapper type to the corresponding primitive type.



Note, that the lack of primitive type instantiations incurs a performance penalty. Autoboxing and -unboxing make the use of wrapper type instantiations of generic types very convenient and concise in the source code. But the concise notation hides the fact that behind the curtain the virtual machine creates and uses lots of wrapper objects, each of which must be allocated and later garbage collected. The higher performance of direct use of primitive type values cannot be achieved with generic types. Only a regular (i.e., non-generic) type can provide the optimal performance of using primitive type values.

Example:

```
class Box<T> {
    private T theObject;
    public Box(T arg) { theObject = arg; }
    public T get() { return theObject; }
    ...
}
class BoxOfLong {
    private long theObject;
    public BoxOfLong(long arg) { theObject = arg; }
    public long get() { return theObject; }
    ...
}
class Test {
    public static void main(String[] args) {
        long result;

        Box<Long> box = new Box<Long>(0L); // autoboxing
        result = box.get();                // auto-unboxing

        Box<long> box = new Box<long>(0L); // error
        result = box.get();

        BoxOfLong box = new BoxOfLong(0L);
        result = box.get();
    }
}
```

The example illustrates that the instantiation of the `Box` type for type `Long` leads to the inevitable overhead of boxing and unboxing. The instantiation on the primitive type `long` does not help, because it is illegal. Only a dedicated non-generic `BoxOfLong` type eliminates the overhead by using the primitive type `long`.

LINK TO THIS [TypeArguments.FAQ003](#)

REFERENCES [What is a type argument?](#)  
[Which types are permitted as type arguments?](#)

---

## Are wildcards permitted as type arguments?

*For instantiation of a generic type, yes. For instantiation of a generic method, no.*

A wildcard is a syntactic construct that denotes a family of types.

All wildcards can be used as type arguments of a parameterized type. This includes the unbounded wildcard as well as wildcards with an upper or lower bound.

Examples:

```
List<?>          10;
List<? extends Number> 11;
List<? super Long>  12;
```

Wildcards *cannot* be used as type arguments of a generic method.

Examples:

```
list = Collections.<?>emptyList(); //error
list = Collections.<? extends Number>emptyList(); //error
list = Collections.<? super Long>emptyList(); //error
```

LINK TO THIS [TypeArguments.FAQ004](#)

REFERENCES [What is a wildcard?](#)  
[What is an unbounded wildcard?](#)  
[What is a bounded wildcard?](#)

---

## Are type parameters permitted as type arguments?

*Yes.*

Type parameters of a generic type or method can be used as arguments of parameterized types or methods.

Example (of instantiations of a generic type using a type parameter as type argument):

```
class someClass<T> {
    public List<T> someMethod() {
        List<T> list = Collections.<T>emptyList();
        ...
        return list;
    }
    public static <S> void anotherMethod(S arg) {
        List<S> list = Collections.<S>emptyList();
        ...
    }
}
```

The example above demonstrates how the type parameter `T` of the enclosing generic class and the type parameter `S` of a generic method can be used as type arguments to both a parameterized type, namely `List`, and a generic method, namely `emptyList`.

LINK TO THIS [TypeArguments.FAQ005](#)

REFERENCES [What is a type argument?](#)  
[What is a type parameter?](#)  
[What is a parameterized \(or generic\) method?](#)  
[What is a parameterized \(or generic\) type?](#)

---

## Do type parameter bounds restrict the set of types that can be used as type arguments?

*Yes, type arguments must be within bounds.*

When a formal type parameter is declared with one or several bounds, then the actual type argument must be a subtype of all of the bounds specified for the respective formal type parameter.

Examples (using types from the packages [java.util](#) and [java.lang](#)):

```
class Wrapper<T extends Comparable<T>> implements Comparable<Wrapper<T>> {
    private final T theObject;
    public Wrapper(T t) { theObject = t; }
    public T getWrapper() { return theObject; }
    public int compareTo(Wrapper<T> other) { return theObject.compareTo(other.theObject); }
```

```
}
```

---

```
Wrapper<String> wrapper1 = new Wrapper<String>("Oystein");  
Wrapper<? extends Number> wrapper2 = new Wrapper<Long>(0L);  
Wrapper<?> wrapper3 = new Wrapper<Date>(new Date());  
Wrapper<Number> wrapper4 = new Wrapper<Number>(new Long(0L)); // error  
Wrapper<int> wrapper5 = new Wrapper<int>(5); // error
```

`Comparable<T>` uses a type parameter as its type argument.

`Comparable<Wrapper<T>>` uses an instantiation of a parameterized type as type argument.

`Wrapper<String>` and `Wrapper<Long>` have concrete reference types as type arguments.

`Wrapper<? extends Number>` and `Wrapper<?>` use wildcards as type arguments.

`Wrapper<Number>` is illegal because `Number` is not a subtype of `Comparable<Number>` and is not within bounds.

`Wrapper<int>` is illegal because primitive types are not allowed as type arguments.

LINK TO THIS

[TypeArguments.FAQ006](#)

REFERENCES

[What is a type parameter?](#)

[What is a bounded type parameter?](#)

---

## Do I have to specify a type argument when I want to use a generic type?

*No; you can use the so-called raw type, which is the generic type without type arguments.*

A generic type without any type arguments is called a *raw type*. Examples of raw types are `List`, `Set`, `Comparable`, `Iterable`, etc. (examples are taken from the packages [java.util](#) and [java.lang](#)).

Raw types are permitted for compatibility between generic and non-generic (legacy) Java APIs. The use of raw types in code written after the introduction of genericity into the Java programming language is strongly discouraged.

According to the Java Language Specification, it is possible that future versions of the Java programming language will disallow the use of raw types.

LINK TO THIS

[TypeArguments.FAQ007](#)

REFERENCES

[What is the raw type?](#)

[Can I use a raw type like any other type?](#)

[How does the raw type relate to instantiations of the corresponding generic type?](#)

[Where can I find a specification of the Java generics language features?](#)

---

## Do I have to specify a type argument when I want to invoke a generic method?

*No; a generic method can be used without type arguments.*

A generic method can be invoked like a regular method, that is, without specification of the type arguments. In such a case the compiler will automatically infer the type arguments from the static types of the method arguments or the context of the method invocation. This process is known as *type argument inference*.

LINK TO THIS

[TypeArguments.FAQ008](#)

REFERENCES

[What is type argument inference?](#)

[What is explicit type argument specification?](#)

---

## Wildcards

---

## What is a wildcard?

*A syntactic construct that denotes a family of types.*

A wildcard describes a family of types. There are 3 different flavors of wildcards:

- "?" - the unbounded wildcard. It stands for the family of *all* types.
- "? extends Type" - a wildcard with an upper bound. It stands for the family of all types that are subtypes of Type, type Type being included.
- "? super Type" - a wildcard with a lower bound. It stands for the family of all types that are supertypes of Type, type Type being included.

Wildcards are used to declare so-called *wildcard parameterized types*, where a wildcard is used as an argument for instantiation of generic types. Wildcards are useful in situations where no or only partial knowledge about the type argument of a parameterized type is required.

LINK TO THIS [TypeArguments.FAQ101](#)

### REFERENCES

[What is a wildcard parameterized type?](#)  
[If a wildcard appears repeatedly in a type argument section, does it stand for the same type?](#)  
[What is a wildcard bound?](#)  
[What is an unbounded wildcard?](#)  
[What is a bounded wildcard?](#)  
[Which super-subset relationships exist among wildcards?](#)

---

## What is an unbounded wildcard?

*A wildcard without a bound.*

The unbounded wildcard looks like "?" and stands for the family of *all* types.

The unbounded wildcard is used as argument for instantiations of generic types. The unbounded wildcard is useful in situations where no knowledge about the type argument of a parameterized type is needed.

Example:

```
void printCollection(Collection<?> c){ // an unbounded wildcard parameterized type
    for (Object o : c){
        System.out.println(o);
    }
}
```

The `printCollection` method does not require any particular properties of the elements contained in the collection that it prints. For this reason it declares its argument using an unbounded wildcard parameterized type, saying that any type of collection regardless of the element type is welcome.

LINK TO THIS [TypeArguments.FAQ102](#)

### REFERENCES

[What is a wildcard?](#)  
[What is a wildcard parameterized type?](#)  
[What is an unbounded wildcard parameterized type?](#)  
[How do unbounded wildcard instantiations of a parameterized type relate to other instantiations of the same generic type?](#)  
[Which super-subset relationships exist among wildcards?](#)

---

## What is a bounded wildcard?

### *A wildcard with either an upper or a lower bound.*

A wildcard with an upper bound looks like "`? extends Type`" and stands for the family of all types that are subtypes of `Type`, type `Type` being included. `Type` is called the *upper bound*.

A wildcard with a lower bound looks like "`? super Type`" and stands for the family of all types that are supertypes of `Type`, type `Type` being included. `Type` is called the *lower bound*.

Bounded wildcards are used as arguments for instantiation of generic types. Bounded wildcards are useful in situations where only partial knowledge about the type argument of a parameterized type is needed, but where unbounded wildcards carry too little type information.

Example:

```
public class Collections {
    public static <T> void copy
        (List<? super T> dest, List<? extends T> src) { // bounded wildcard parameterized
types
        for (int i=0; i<src.size(); i++)
            dest.set(i,src.get(i));
    }
}
```

The `copy` method copies elements from a source list into a destination list. The destination list must be capable of holding the elements from the source list. We express this by means of bounded wildcards: the output list is required to have an element type with a lower bound `T` and the input list must have an element type with an upper bound `T`.

---

Let's study an example to explore the typical use of bounded wildcards and to explain why unbounded wildcards do not suffice. It's the example of the `copy` method mentioned above. It copies elements from a source list into a destination list. Let's start with a naive implementation of such a `copy` method.

Example (of a restrictive implementation of a `copy` method):

```
public class Collections {
    public static <T> void copy(List<T> dest, List<T> src) { // uses no wildcards
        for (int i=0; i<src.size(); i++)
            dest.set(i,src.get(i));
    }
}
```

This implementation of a `copy` method is more restrictive than it need be, because it requires that both input and output collection must be lists with the exact same type. For instance, the following invocation - although perfectly sensible - would lead to an error message:

Example (of illegal use of the `copy` method):

```
List<Object> output = new ArrayList<Object>();
List<Long> input = new ArrayList<Long>();
...
Collections.copy(output,input); // error: illegal argument types
```

The invocation of the `copy` method is rejected because the declaration of the method demands that both lists must be of the same type. Since the source list is of type `List<Long>` and the destination list is of type `List<Object>` the compiler rejects the method call, regardless of the fact that a list of `Object` references can hold `Long`s. If *both* list were of type `List<Object>` or both were of type `List<Long>` the method call were accepted.

We could try to relax the method's requirements to the argument types and declare wildcard parameterized types as the method parameter types. Declaring wildcard parameterized types as method parameter types has the advantage of allowing a broader set of argument types. Unbounded wildcards allow the broadest conceivable argument set, because the unbounded wildcard `?` stands for any type without any restrictions. Let's try using an unbounded wildcard parameterized type. The method would then look as follows:

Example (of a relaxed `copy` method; does not compile):

```

public class Collections {
    public static void copy(List<?> dest, List<?> src) { // uses unbounded wildcards
        for (int i=0; i<src.size(); i++)
            dest.set(i,src.get(i)); // error: illegal argument types
    }
}

```

It turns out that this relaxed method signature does not compile. The problem is that the `get()` method of a `List<?>` returns a reference pointing to an object of unknown type. References pointing to objects of unknown type are usually expressed as a reference of type `Object`. Hence `List<?>.get()` returns an `Object`.

On the other hand, the `set()` method of a `List<?>` requires something unknown, and "unknown" does not mean that the required argument is of type `Object`. Requiring an argument of type `Object` would mean accepting everything that is derived of `Object`. That's not what the `set()` method of a `List<?>` is asking for. Instead, "unknown" in this context means that the argument must be of a type that matches the type that the wildcard `?` stands for. That's a much stronger requirement than just asking for an `Object`.

For this reason the compiler issues an error message: `get()` returns an `Object` and `set()` asks for a more specific, yet unknown type. In other words, the method signature is too relaxed. Basically, a signature such as `void copy(List<?> dest, List<?> src)` is saying that the method takes one type of list as a source and copies the content into another - totally unrelated - type of destination list. Conceptually it would allow things like copying a list of apples into a list of oranges. That's clearly not what we want.

What we really want is a signature that allows copying elements from a source list into a destination list with a specific property, namely that it is capable of holding the source list's elements. Unbounded wildcards are too relaxed for this purpose, as we've seen above, but bounded wildcards are suitable in this situation. A bounded wildcard carries more information than an unbounded wildcard.

In our example of a `copy` method we can achieve our goal of allowing all sensible method invocations by means of bounded wildcards, as in the following implementation of the `copy` method:

Example (of an implementation of the `copy` method that uses bounded wildcards):

```

public class Collections {
    public static <T> void copy
        (List<? super T> dest, List<? extends T> src) { // uses bounded wildcards
        for (int i=0; i<src.size(); i++)
            dest.set(i,src.get(i));
    }
}

```

In this implementation we require that a type `T` exists that is subtype of the output list's element type and supertype of the input list's element type. We express this by means of wildcards: the output list is required to have an element type with a lower bound `T` and the input list must have an element type with an upper bound `T`.

Example (of using the `copy` method with wildcards):

```

List<Object> output = new ArrayList<Object>();
List<Long> input = new ArrayList<Long>();
...
Collections.copy(output,input); // fine; T:= Number & Serializabe & Comparable<Number>

List<String> output = new ArrayList<String>();
List<Long> input = new ArrayList<Long>();
...
Collections.copy(output,input); // error

```

In the first method call `T` would have to be a supertype of `Long` and a subtype of `Object`, and luckily there is a number of types that fall into this category, namely `Number`, `Serializable` and `Comparable<Number>`. Hence the compiler can use any of the 3 types as type argument and the method invocation is permitted.

The second nonsensical method call is rejected by the compiler, because the compiler realizes that there is no type that is subtype of `String` and supertype of `Long`.

### Conclusion:

Bounded wildcards carry more information than unbounded wildcards. While an unbounded wildcard stands for a representative from the family of *all* types, a bounded wildcard stands for a representative of a family of either super- or subtypes of a type. Hence a bounded wildcard carries more type information than an unbounded wildcard. The supertype of such a family is called the upper bound, the subtype of such a family is called the lower bound.

LINK TO THIS [TypeArguments.FAQ103](#)

### REFERENCES

[What is a wildcard?](#)  
[What is a wildcard instantiation?](#)  
[How do wildcard instantiations with an upper bound relate to other instantiations of the same generic type?](#)  
[How do wildcard instantiations with a lower bound relate to other instantiations of the same generic type?](#)  
[Which super-subset relationships exist among wildcards?](#)  
[What is a wildcard bound?](#)  
[Which types are permitted as wildcard bounds?](#)  
[What is the difference between a wildcard bound and a type parameter bound?](#)  
[What is the difference between a `Collection<?>` and a `Collection<Object>`?](#)

---

## What do multi-level (or nested) wildcards mean?

*It depends.*

A multi-level wildcard is a wildcard that appears as the type argument of a type argument. The instantiations `Collection<Pair<String, ?>>` and `Collection<? extends Pair<String, ?>>` are examples of multi-level wildcards. Multi-level wildcard parameterized types are inherently difficult to interpret, because the wildcard can appear on different levels. For illustration, let us discuss the difference between a `Collection<Pair<String, ?>>` and a `Collection<? extends Pair<String, ?>>`. For sake of simplicity, let us assume `Pair` is a final class.

The type `Collection<Pair<String, ?>>` is a concrete instantiation of the generic `Collection` interface. It is a heterogeneous collection of pairs of different types. It can contain elements of type `Pair<String, Long>`, `Pair<String, Date>`, `Pair<String, Object>`, `Pair<String, String>`, and so on and so forth. In other words, `Collection<Pair<String, ?>>` contains a mix of pairs of different types of the form `Pair<String, ?>`.

The type `Collection<? extends Pair<String, ?>>` is a wildcard parameterized type; it does NOT stand for a concrete parameterized type. It stands for a representative from the family of collections that are instantiations of the `Collection` interface, where the type argument is of the form `Pair<String, ?>`. Compatible instantiations are `Collection<Pair<String, Long>>`, `Collection<Pair<String, String>>`, `Collection<Pair<String, Object>>`, or `Collection<Pair<String, ?>>`. In other words, we do not know which instantiation of `Collection` it stands for.

As a rule of thumb, you have to read multi-level wildcards top-down.

If the top-level type argument is a concrete type then the instantiation is a concrete type, probably a mixed bag of something, if the wildcard appears further down on a lower level. In this sense, a `Collection<Pair<String, ?>>` is a collection of pairs of a certain form, that has lots of concrete subtypes. It's similar to a `Collection<Object>`, which is a collection of a concrete type that has lots of subtypes and is a mixed bag of something that is a subtype of `Object`.

If the top-level type argument is a wildcard, then the type is not concrete. It is a placeholder for a representative from a family of types. In this sense, a `Collection<? extends Pair<String, ?>>` is a placeholder for a collection instantiated for a particular unknown pair type of a certain form. It's similar to a `Collection<?>`, which is a placeholder for a specific instantiation of the `Collection` interface, but it is not a concrete type.

---

Here is an example that illustrates the difference between `Collection<Pair<String, ?>>` and `Collection<? extends Pair<String, ?>>`.

Example:



```

Collection<Pair<String,Long>>      c1 = new ArrayList<Pair<String,Long>>();

Collection<Pair<String,Long>>      c2 = c1; // fine
Collection<Pair<String,?>>        c3 = c1; // error
Collection<? extends Pair<String,?>> c4 = c1; // fine

```

Of course, we can assign a `Collection<Pair<String,Long>>` to a `Collection<Pair<String,Long>>`. There is nothing surprising here.

But we *cannot* assign a `Collection<Pair<String,Long>>` to a `Collection<Pair<String,?>>`. The parameterized type `Collection<Pair<String,Long>>` is a homogenous collection of pairs of a `String` and a `Long`; the parameterized type `Collection<Pair<String,?>>` is a heterogenous collection of pairs of a `String` and something of unknown type. The heterogenous `Collection<Pair<String,?>>` could for instance contain a `Pair<String,Date>` and that clearly does not belong into a `Collection<Pair<String,Long>>`. For this reason the assignment is not permitted.

But then, we can assign a `Collection<Pair<String,Long>>` to a `Collection<? extends Pair<String,?>>`, because the type `Pair<String,Long>` belongs to the family of types denoted by the wildcard `? extends Pair<String,?>`. This is because the type family denoted by the wildcard `? extends Pair<String,?>` comprises all subtypes of `Pair<String,?>`. Since we assumed that `Pair` is a final type, this type family includes all instantiations of the generic type `Pair` where the first type argument is `String` and second type argument is an arbitrary type or wildcard. The type family includes members such as `Pair<String,Long>`, `Pair<String,Object>`, `Pair<String,? extends Number>`, and `Pair<String,?>` itself.

If we give up the simplification that `Pair` is a final class, then we must also consider subtypes of `Pair`.

`Collection<Pair<String,?>>` is then a heterogenous collection of pairs of different types of the form `Pair<String,?>`, or *subtypes thereof*. It can contain elements of type `Pair<String,Long>`, `Pair<String,Date>`, but also elements of type `SubTypeOfPair<String,Date>`, `SubTypeOfPair<String,Object>`, and so on and so forth.

`Collection<? extends Pair<String,?>>` stands for a representative from the family of collections that are instantiations of the `Collection` interface, where the type argument is of the form `Pair<String,?>`, or *subtypes thereof*. Compatible parameterized types are `Collection<Pair<String,Long>>`, `Collection<Pair<String,Object>>`, but also `Collection<SubTypeOfPair<String,Object>>`, or `Collection<SubTypeOfPair<String,?>>`.

Here is an example that illustrates the difference between the concrete parameterized type `Collection<Pair<String,?>>` and the wildcard parameterized type `Collection<? extends Pair<String,?>>`.

Example:

```

Collection<SubTypeOfPair<String,Long>> c1 = new ArrayList<SubTypeOfPair<String,Long>>();

Collection<Pair<String,Long>>          c2 = c1; // error
Collection<SubTypeOfPair<String,Long>> c3 = c1; // fine
Collection<Pair<String,?>>            c4 = c1; // error
Collection<? extends Pair<String,?>>  c5 = c1; // fine

```

In this case, we *cannot* assign a `Collection<SubTypeOfPair<String,Long>>` to a `Collection<Pair<String,Long>>`, because these two instantiations of the generic type `Collection` are unrelated and incompatible types. The `Collection<SubTypeOfPair<String,Long>>` contains `SubTypeOfPair<String,Long>` objects, whereas the `Collection<Pair<String,Long>>` contains a mix of objects of types that are subtypes of type `Pair<String,Long>`. This includes, but is not limited to objects of type `SubTypeOfPair<String,Long>`. For this reason a `Collection<SubTypeOfPair<String,Long>>` cannot be assigned to a `Collection<Pair<String,Long>>`.

Also, we *cannot* assign a `Collection<SubTypeOfPair<String,Long>>` to a `Collection<Pair<String,?>>` because the parameterized type `Collection<SubTypeOfPair<String,Long>>` is a homogenous collection of objects of type `SubTypeOfPair<String,Long>`, whereas the parameterized type `Collection<Pair<String,?>>` is a heterogenous collection. The heterogenous `Collection<Pair<String,?>>` could for instance contain a `Pair<String,Date>` and that clearly does not belong in a `Collection<SubTypeOfPair<String,Long>>`.

The assignment of a `Collection<SubTypeOfPair<String,Long>>` to a `Collection<? extends Pair<String,?>>` is



fine because the type `SubTypeOfPair<String,Long>` belongs to the family of types denoted by the wildcard `?` extends `Pair<String,?>`. This is because the type family denoted by the wildcard `?` extends `Pair<String,?>` comprises all subtypes of `Pair<String,?>`. Since we no longer assumed that `Pair` is a final type, this type family includes all instantiations of the generic type `Pair` and any of its subtypes where the first type argument is `String` and second type argument is an arbitrary type or wildcard. The type family includes members such as `Pair<String,Long>`, `Pair<String,Object>`, `Pair<String,? extends Number>`, and `Pair<String,?>` itself, but also `SubTypeOfPair<String,Long>`, `SubTypeOfPair<String,Object>`, `SubTypeOfPair<String,? extends Number>`, and `SubTypeOfPair<String,?>`.

LINK TO THIS [TypeArguments.FAQ104](#)

#### REFERENCES

[What is the difference between a Collection<?> and a Collection<Object>?](#)

[Which super-subset relationships exist among wildcards?](#)

[What is the difference between a Collection<Pair<String,Object>>, a Collection<Pair<String,?>> and a Collection<? extends Pair<String,?>>?](#)

---

## If a wildcard appears repeatedly in a type argument section, does it stand for the same type?

*No, each occurrence can stand for a different type.*

If the same wildcard appears repeatedly in a type argument section each occurrence of the wildcard refers to a potentially different type. It is similar to wildcards in a regular expression: in `"s??"` the wildcard need not stand for the same character. `"see"`, but also `"sea"` or `"sew"` or `"saw"` would be matching expressions. Each question mark stands for a potentially different character. The same holds for wildcards in Java generics.

Example (using the same wildcard repeatedly):

```
Pair<?,?> couple = new Pair<String,String>("Orpheus","Eurydike");
Pair<?,?> xmas   = new Pair<String,Date>("Xmas", new Date(104,11,24));
```

In the example above, the wildcard `"?"` can stand for the same type, such as `String`, but it need not do so. Each wildcard can stand for a different type, such as `String` and `Date` for instance.

Conversely, different wildcards need not stand for different types. If the type families denoted by the two different wildcards overlap, then the two different wildcards can stand for the same concrete type.

Example (using different wildcards):

```
Pair<? extends Appendable,? extends CharSequence> textPlusSuffix
    = new Pair<StringBuilder,String>(new StringBuilder("log"), ".txt");

Pair<? extends Appendable,? extends CharSequence> textPlusText
    = new Pair<StringBuilder,StringBuilder>(new StringBuilder("log"), new
    StringBuilder(".txt"));
```

In the examples above, the different wildcards `"? extends Appendable"` and `"? extends CharSequence"` can stand for different types, such as `StringBuilder` and `String`, but they can equally well stand for the same type, such as `StringBulder`, provided the bounds permit it.

---

Below are a couple of examples where the same wildcard appears repeatedly and where the compiler rightly issues an error message.

In the first example the wildcard appears twice in an parameterized type, namely in `Pair<?,?>`.

Example #1 (demonstrating the incompatibility of one wildcard to another):

```
class Pair<S,T> {
    private S first;
    private T second;
    public Pair(S s,T t) { first = s; second = t; }
    ...
    public static void flip(Pair<?,?> pair) {
```

```

    Object tmp = pair.first;
    pair.first = pair.second; // error: incompatible types
    pair.second = tmp;      // error: incompatible types
}
}
class Test {
    public static void test() {
        Pair<?,?> xmas = new Pair<String,Date>("Xmas",new Date(104,11,24));
        Pair.flip(xmas);

        Pair<?,?> name = new Pair<String,String>("Yves","Meyer");
        Pair.flip(name);
    }
}

```

The fields of the `Pair<?,?>` may be of different types. For instance, when the `flip` method is invoked with an argument of type `Pair<String,Date>`, then `first` would be of type `String`, while `second` is of type `Date`, and the fields cannot be assigned to each other. Even if the `flip` method is invoked with an argument of type `Pair<String,String>`, i.e. both wildcards stand for the same type, namely `String`, the compiler does not know this inside the implementation of the `flip` method. For this reason the compiler issues an error message in the implementation of the `flip` method when the two fields of unknown - and potentially different types - are assigned to each other.

In the second example the wildcard appears in an two instantiations of the same generic type, namely in `Collection<?>`.

Example #2 (demonstrating the incompatibility of one wildcard to another):

```

class Utilities {
    public static void add(Collection<?> list1, Collection<?> list2) {
        for (Object o : list1)
            list2.add(o); // error: incompatible types
    }
}
class Test {
    public static void test() {
        Collection<?> dates = new LinkedList<Date>();
        Collection<?> strings = new LinkedList<String>();
        add(dates,strings);
        add(strings,strings);
    }
}

```

The two collections contain elements of two potentially different unknown types. The compiler complains when elements from one collection are passed to the other collection, because the types of the elements might be incompatible.

In the third example the wildcard appears in an two instantiations of two different generic type, namely in `Collection<? extends CharSequence>` and `Class<? extends CharSequence>`.

Example #3 (demonstrating the incompatibility of one wildcard to another):

```

class Utilities {
    public static void method(Collection<? extends CharSequence> coll,
                             Class <? extends CharSequence> type) {
        ...
        coll.add(type.newInstance()); // error: incompatible types
        ...
    }
}
class Test {
    public static void test() {
        List<StringBuilder> stringList = new LinkedList<StringBuilder>();
    }
}

```

```
        method(stringList, String.class);
    }
}
```

The `newInstance` method of class `Class<? extends CharSequence>` creates an object of an unknown subtype of `CharSequence`, while the collection `Collection<? extends CharSequence>` holds elements of a potentially different subtype of `CharSequence`. The compiler complains when the newly created object is passed to the collection, because the collection might hold objects of an incompatible type.

**LINK TO THIS**      [TypeArguments.FAQ105](#)

**REFERENCES**

- [What is a wildcard instantiation?](#)
- [How can I make sure that the same wildcard stand for the same type?](#)
- [What is a wildcard bound?](#)
- [What is an unbounded wildcard?](#)
- [What is a bounded wildcard?](#)
- [Which super-subset relationships exist among wildcards?](#)

---

---

## Wildcard Bounds

---

### What is a wildcard bound?

*A reference type that is used to further describe the family of types denoted by a wildcard.*

A wildcard can be unbounded, in which case it is denoted as "?". The unbounded wildcard stands for the family of *all* types.

Alternatively a wildcard can have a bound. There are two types of bounds: upper and lower bounds. The syntax for a bounded wildcard is either `"? extends SuperType"` (wildcard with upper bound) or `"? super SubType"` (wildcard with lower bound). The terms "upper" and "lower" stem from the way, in which inheritance relationships between types are denoted in modeling languages such as UML for instance.



The bound shrinks the family of types that the wildcard stands for. For instance, the wildcard `"? extends Number"` stands for the family of subtypes of `Number`; type `Number` itself is included in the family. The wildcard `"? super Long"` stands for the family of supertypes of `Long`; type `Long` itself is included.

Note, a wildcard can have only one bound. It can neither have both an upper *and* a lower bound nor several upper or lower bounds. Constructs such as `"? super Long extends Number"` or `"? extends Comparable<String> & Cloneable"` are illegal.

**LINK TO THIS**      [TypeArguments.FAQ201](#)

**REFERENCES**

- [What is a wildcard?](#)
- [Which types are permitted as wildcard bounds?](#)
- [What is the difference between a wildcard bound and a type parameter bound?](#)

---

## Which types are permitted as wildcard bounds?

*All references types including parameterized types, but no primitive types.*

All reference types can be used as a wildcard bound. This includes classes, interfaces, enum types, nested and inner types, and array types. Only primitive types cannot be used as wildcard bound.

Example (of wildcard bounds):

```
List<? extends int>                10;           // error
List<? extends String>            11;
List<? extends Runnable>         12;
List<? extends TimeUnit>         13;
List<? extends Comparable>       14;
List<? extends Thread.State>     15;
List<? extends int[]>            16;
List<? extends Object[]>        17;
List<? extends Callable<String>> 18;
List<? extends Comparable<? super Long>> 19;
List<? extends Class<? extends Number>> 110;
List<? extends Map.Entry<?, ?>> 111;
List<? extends Enum<?>>         112;
```

The example only shows the various reference types as upper bound of a wildcard, but these type are permitted as lower bound as well.

We can see that primitive types such as `int` are not permitted as wildcard bound.

Class types, such as `String`, and interface types, such as `Runnable`, are permitted as wildcard bound. Enum types, such as `TimeUnit` (see [java.util.concurrent.TimeUnit](#)) are also permitted as wildcard bound. Note, that even types that do not have subtypes, such as final classes and enum types, can be used as upper bound. The resulting family of types has exactly one member then. For instance, "`? extends String`" stands for the type family consisting of the type `String` alone. Following the same line of logic, the wildcard "`? super Object`" is permitted, too, although class `Object` does not have a supertype. The resulting type family consists of type `Object` alone.

Raw types are permitted as wildcard bound; `Comparable` is an example.

`Thread.State` is an example of a nested type; `Thread.State` is an enum type nested into the `Thread` class. Non-static inner types are also permitted.

An array type, such as `int[]` and `Object[]`, is permitted as wildcard bound. Wildcards with an array type as a bound denote the family of all sub- or supertypes of the wildcard type. For instance, "`? extends Object[]`" is the family of all array types whose component type is a reference type. `int[]` does not belong to that family, but `Integer[]` does. Similarly, "`? super Number[]`" is the family of all supertypes of the array type, such as `Object[]`, but also `Object`, `Cloneable` and `Serializable`.

Parameterized types are permitted as wildcard bound, including concrete parameterized types such as `Callable<String>`, bounded wildcard parameterized types such as `Comparable<? super Long>` and `Class<? extends Number>`, and unbounded wildcard parameterized types such as `Map.Entry<?, ?>`. Even the primordial supertype of all enum types, namely class `Enum`, can be used as wildcard bound.

LINK TO THIS [TypeArguments.FAQ202](#)

REFERENCES [What is a wildcard?](#)  
[What is a wildcard bound?](#)

---

## What is the difference between a wildcard bound and a type parameter bound?

*A wildcard can have only one bound, while a type parameter can have several bounds.  
A wildcard can have a lower or an upper bound, while there is no such thing as a lower bound for a type parameter.*

Wildcard bounds and type parameter bounds are often confused, because they are both called bounds and have in part similar syntax.

Example (of type parameter bound and wildcard bound):

```
class Box<T extends Comparable<T> & Cloneable>
    implements Comparable<Box<T>>, Cloneable {

    private T theObject;
    public Box(T arg) { theObject = arg; }
    public Box(Box<? extends T> box) { theObject = box.theObject; }
    ...
    public int compareTo(Box<T> other) { ... }
    public Box<T> clone() { ... }
}
```

The code sample above shows a type parameter `T` with two bounds, namely `Comparable<T>` and `Cloneable`, and a wildcard with an upper bound `T`.

The type parameter bounds give access to their non-static methods. For instance, in the example above, the bound `Comparable<T>` makes it possible that the `compareTo` method can be invoked on variables of type `T`. In other words, the compiler would accept an expression such as `theObject.compareTo(other.theObject)`.

The wildcard bound describes the family of types that the wildcard stands for. In the example, the wildcard "`? extends T`" denotes the family of all subtypes of `T`. It is used in the argument type of a constructor and permits that `box` objects of a `Box` type from the family `Box<? extends T>` can be supplied as constructor arguments. It allows that a `Box<Number>` can be constructed from a `Box<Long>`, for instance.

The syntax is similar and yet different:

	Syntax
<b>type parameter bound</b>	<code>TypeParameter extends Class &amp; Interface<sub>1</sub> &amp; ... &amp; Interface<sub>N</sub></code>
<b>wildcard bound</b>	
<b>upper bound</b>	<code>? extends SuperType</code>
<b>lower bound</b>	<code>? super SubType</code>

A wildcard can have only one bound, either a lower or an upper bound. A list of wildcard bounds is not permitted.

A type parameter, in contrast, can have several bounds, but there is no such thing as a lower bound for a type parameter.

**LINK TO THIS**      [TypeArguments.FAQ203](#)

**REFERENCES**

- [What is a wildcard?](#)
- [What is a wildcard bound?](#)
- [What is a type parameter?](#)
- [What is a type parameter bound?](#)
- [Why is there no lower bound for type parameters?](#)

# Practicalities - Programming With Java Generics

© Copyright 2004-2022 by Angelika Langer. All Rights Reserved.

## Using Generic Types

- [Should I prefer parameterized types over raw types?](#)
- [Why shouldn't I mix parameterized and raw types, if I feel like it?](#)
- [Should I use the generic collections or better stick to the old non-generic collections?](#)
- [What is a checked collection?](#)
- [What is the difference between a `Collection<?>` and a `Collection<Object>`?](#)
- [How do I express that a collection is a mix of objects of different types?](#)
- [What is the difference between a `Collection<Pair<String, Object>>`, a `Collection<Pair<String, ?>>` and a `Collection<? extends Pair<String, ?>>`?](#)
- [How can I make sure that the same wildcard stands for the same type?](#)

## Using Generic Methods

- [Why doesn't method overloading work as I expect it?](#)
- [Why doesn't method overriding work as I expect it?](#)

## Coping With Legacy

- [What happens when I mix generic and non-generic legacy code?](#)
- [Should I re-engineer all my existing classes and generify them?](#)
- [How do I generify an existing non-generic type or method?](#)
- [Can I safely generify a supertype, or does it affect all subtypes?](#)
- [How do I avoid breaking binary compatibility when I generify an existing type or method?](#)

## Defining Generic Types and Methods

- [Which types should I design as generic types instead of defining them as regular non-generic types?](#)
- [Do generics help designing parallel class hierarchies?](#)
- [When would I use an unbounded wildcard instantiation instead of a bounded or concrete instantiation?](#)
- [When would I use a wildcard parameterized type instead of a concrete parameterized type?](#)
- [When would I use a wildcard parameterized type with a lower bound?](#)
- [How do I recover the actual type of the `this` object in a class hierarchy?](#)
- [What is the "getThis" trick?](#)
- [How do I recover the element type of a container?](#)
- [What is the "getTypeArgument" trick?](#)

## Designing Generic Methods

- [Why does the compiler sometimes issue an unchecked warning when I invoke a "varargs" method?](#)
- [What is a "varargs" warning?](#)
- [How can I suppress a "varargs" warning?](#)
- [When should I refrain from suppressing a "varargs" warning?](#)
- [Which role do wildcards play in method signatures?](#)
- [Which one is better: a generic method with type parameters or a non-generic method with wildcards?](#)
- [Under which circumstances are the generic version and the wildcard version of a method equivalent?](#)
- [Under which circumstances do the generic version and the wildcard version of a method mean different things?](#)
- [Under which circumstances is there no transformation to the wildcard version of a method possible?](#)
- [Should I use wildcards in the return type of a method?](#)
- [How do I implement a method that takes a wildcard argument?](#)
- [How do I implement a method that takes a multi-level wildcard argument?](#)
- [I want to pass a `U` and a `X<U>` to a method. How do I correctly declare that method?](#)

## Working With Generic Interfaces

- [Can a class implement different instantiations of the same generic interface?](#)
- [Can a subclass implement a parameterized interface other than any of its superclasses does?](#)
- [What happens if a class implements two parameterized interfaces that define the same method?](#)
- [Can an interface type nested into a generic type use the enclosing type's type parameters?](#)

## Implementing Infrastructure Methods

- [How do I best implement the `equals` method of a generic type?](#)
- [How do I best implement the `clone` method of a generic type?](#)

## Using Runtime Type Information

- [What does the type parameter of class `java.lang.Class` mean?](#)
- [How do I pass type information to a method so that it can be used at runtime?](#)
- [How do I generically create objects and arrays?](#)
- [How do I perform a runtime type check whose target type is a type parameter?](#)

## Reflection

- [Which information related to generics can I access reflectively?](#)
- [How do I retrieve an object's actual \(dynamic\) type?](#)
- [How do I retrieve an object's declared \(static\) type?](#)
- [What is the difference between a generic type and a parameterized type in reflection?](#)
- [How do I figure out whether a type is a generic type?](#)
- [Which information is available about a generic type?](#)
- [How do I figure out whether a type is a parameterized type?](#)
- [Which information is available about a parameterized type?](#)
- [How do I retrieve the representation of a generic method?](#)
- [How do I figure out whether a method is a generic method?](#)
- [Which information is available about a generic method?](#)
- [Which information is available about a type parameter?](#)
- [What is a generic declaration?](#)
- [What is a wildcard type?](#)
- [Which information is available about a wildcard?](#)

---

# Programming With Generics

---

---

## Using Generic Types and Methods

---

### Should I prefer parameterized types over raw types?

*Yes, using parameterized types has various advantages and is recommended, unless you have a compelling reason to prefer the raw type.*

It is permitted to use generic types without type arguments, that is, in their raw form. In principle, you can entirely ignore Java Generics and use raw types throughout your programs. It is, however, recommended that type arguments are provided when a generic type is used, unless there is a compelling reason not to do so.

Providing the type arguments rather than using the raw type has a couple of advantages:

- **Improved readability.** An instantiation with type arguments is more informative and improves the readability of the source code.
- **Better tool support.** Providing type arguments enables development tools to support you more effectively: IDEs (= integrated development environments) can offer more precise context-sensitive information; incremental compilers can flag type errors the moment you type in the incorrect source code. Without providing type arguments the errors would go undetected until you start testing your program.
- **Fewer ClassCastException.** Type arguments enable the compiler to perform static type checks to ensure type safety at compile time, as opposed to dynamic type checks performed by the virtual machine at runtime. As a result there are fewer opportunities for the program to raise a `ClassCastException`.
- **Fewer casts.** More specific type information is available when type arguments are provided, so that hardly any casts are needed compared to the substantial number of casts that clutter the source code when raw types are used.
- **No unchecked warnings.** Raw types lead to "unchecked" warning, which can be prevented by use of type arguments.
- **No future deprecation.** The Java Language Specification states that raw types might be deprecated in a future version of Java, and might ultimately be withdrawn as a language feature.

Raw types have an advantage, too:

- **Zero learning effort.** If you ignore Java Generics and use raw types everywhere in your program you need not familiarize yourself with new language features or learn how to read any puzzling error messages.

Advantages that are no advantages:

- **Improved Performance.** Especially C++ programmers might expect that generic programs are more efficient than non-generic programs, because C++ templates can boost runtime efficiency. However, if you take a look under the hood of the Java compiler and study how the compiler translates generic source code to byte code you realize that Java code using parameterized types does not perform any faster than non-generic programs.



## REFERENCES

[How does the compiler translate Java generics?](#)  
[What is an "unchecked" warning?](#)  
[What is the benefit of using Java generics?](#)

---

## Why shouldn't I mix parameterized and raw types, if I feel like it?

*Because it is poor style and highly confusing to readers of your source code.*

Despite of the benefits of parameterized types you might still prefer use of raw types over using pre-defined generic types in their parameterized form, perhaps because the raw types look more familiar. To some extent it is a matter of style and taste and both styles are permitted. No matter what your preferences are: be consistent and stick to it. Either ignore Java generics and use raw type in all places, or take advantage of the improved type-safety and provide type arguments in all places. Mixing both styles is confusing and results in "unchecked" warnings that can and should be avoided.

Naturally, you have to mix both styles when you interface with source code that was written before the advent of Java generics. In these cases you cannot avoid the mix and the inevitable "unchecked" warnings. However, one should never have any "unchecked" warnings in code that is written in generic style and does not interface with non-generic APIs.

Here is a typical beginner's mistake for illustration.

Example (of poor programming style):

```
List<String> list = new ArrayList<String>();
Iterator iter = list.iterator();
String s = (String) iter.next();
...
```

Beginners often start out correctly providing type arguments and suddenly forget, in the heat of the fighting, that methods of parameterized types often return other parameterized types. This way they end up with a mix of generic and non-generic programming style, where there is no need for it. Avoid mistakes like this and provide type arguments in *all* places.

Example (corrected):

```
List<String> list = new ArrayList<String>();
Iterator<String> iter = list.iterator();
String s = iter.next();
...
```

Here is an example of a code snippet that produces avoidable "unchecked" warnings.

Example (of avoidable "unchecked" warning):

```
void f(Object obj) {
    Class type = obj.getClass();
    Annotation a = type.getAnnotation(Documented.class); // unchecked warning
    ...
}
```

---

```
warning: [unchecked] unchecked call to <A>getAnnotation(java.lang.Class<A>) as a member of the raw type
java.lang.Class
```

```
Annotation a = type.getAnnotation(Documented.class);
                ^
```

The `getClass` method returns an instantiation of class `Class`, namely `Class<? extends X>`, where `X` is the erasure of the static type of the expression on which `getClass` is called. In the example, the parameterization of the return type is ignored and the raw type `Class` is used instead. As a result, certain method calls, such as the invocation of `getAnnotation`, are flagged with an "unchecked" warning.

In general, it is recommended that type arguments are provided unless there is a compelling reason not to do so. In case of doubt, often the unbounded wildcard parameterized type is the best alternative to the raw type. It is semantically equivalent, eliminates "unchecked" warnings and yields to error messages if their use is unsafe.

Example (corrected):

```
void f(Object obj) {
    Class<?> type = obj.getClass();
    Annotation a = type.getAnnotation(Documented.class);
    ...
}
```

## LINK TO THIS

[Practicalities.FAQ002](#)

## REFERENCES

[What is the benefit of using Java generics?](#)  
[What does type-safety mean?](#)  
[What is an "unchecked" warning?](#)



[What is the raw type?](#)  
[What is a parameterized or generic type?](#)  
[How is a generic type instantiated?](#)  
[What is an unbounded wildcard parameterized type?](#)

## Should I use the generic collections or stick to the old non-generic collections?

*Provide type arguments when you use collections; it improves clarity and expressiveness of your source code.*

The JDK collection framework has been re-engineered. All collections are generic types since Java 5.0. In principle, you can choose whether you want to use the pre-defined generic collections in their parameterized or raw form. Both is permitted, but use of the parameterized form is recommended because it improves the readability of your source code.

Let us compare the generic and non-generic programming style and see how they differ.

Example (of non-generic style):

```
final class HtmlProcessor {
    public static Collection process(Collection files) {
        Collection imageFileNames = new TreeSet();
        for (Iterator i = files.iterator(); i.hasNext(); ) {
            URI uri = (URI)i.next();
            Collection tokens = HtmlTokenizer.tokenize(new File(uri));
            imageFileNames.addAll(ImageCollector.collect(tokens)); // unchecked warning
        }
        return imageFileNames;
    }
}

final class ImageCollector {
    public static Collection collect(Collection tokens) {
        Set images = new TreeSet();
        for (Iterator i = tokens.iterator(); i.hasNext(); ) {
            HtmlToken tok = (HtmlToken)i.next();
            if (tok.getTag().3("img") && tok.hasAttribute("src")) {
                Attribute attr = tok.getAttribute("src");
                images.add(attr.getValue()); // unchecked warning
            }
        }
        return images;
    }
}
```

From the code snippet above it is relatively difficult to tell what the various collections contain. This is typical for non-generic code. The raw type collections do not carry information regarding their elements. This lack of type information also requires that we cast to the alleged element type each time an element is retrieved from any of the collections. Each of these casts can potentially fail at runtime with a `ClassCastException`. `ClassCastExceptions` are a phenomenon typical to non-generic code.

If we translate this non-generic source code with a Java 5.0 compiler, we receive "unchecked" warnings when we invoke certain operations on the raw type collections. We would certainly ignore all these warnings, or suppress them with the `SuppressWarnings` annotation.

Example (of generic counterpart):

```
final class HtmlProcessor {
    public static Collection<String> process(Collection<URI> files) {
        Collection<String> imageFileNames = new TreeSet<String>();
        for (URI uri : files) {
            Collection<HtmlToken> tokens = HtmlTokenizer.tokenize(new File(uri));
            imageFileNames.addAll(ImageCollector.collect(tokens));
        }
        return imageFileNames;
    }
}

final class ImageCollector {
    public static Collection<String> collect(Collection<HtmlToken> tokens) {
        Set<String> images = new TreeSet<String>();
        for (HtmlToken tok : tokens) {
            if (tok.getTag().equals("img") && tok.hasAttribute("src")) {
                Attribute attr = tok.getAttribute("src");
                images.add(attr.getValue());
            }
        }
        return images;
    }
}
```

From the generic source code we can easily tell what type of elements are stored in the various collections. This is one of the benefits of generic Java: the source code is substantially more expressive and captures more of the programmer's intent. In addition it enables the compiler to perform lots of type checks at compile time that would otherwise be performed at runtime. Note that we got rid of all casts. As a consequence there will be no runtime failure due to a `ClassCastException`.

This is a general rule in Java 5.0: if your source code compiled without any warnings then there will be no unexpected `ClassCastExceptions` at runtime. Of course, if your code contains explicit cast expressions any exceptions resulting from these casts are not considered unexpected. But the number of casts in your source code will drop substantially with the use of generics.

LINK TO THIS [Practicalities.FAQ003](#)

REFERENCES

- [package java.util](#)
- [Should I prefer parameterized types over raw types?](#)
- [What is the benefit of using Java generics?](#)
- [What is an "unchecked" warning?](#)
- [How can I disable or enable unchecked warnings?](#)
- [What is the SuppressWarnings annotation?](#)
- [What is the raw type?](#)
- [What is a parameterized or generic type?](#)
- [How is a generic type instantiated?](#)

---

## What is a checked collection?

*A view to a regular collection that performs a runtime type check each time an element is inserted.*

Despite of all the type checks that the compiler performs based on type arguments in order to ensure type safety it is still possible to smuggle elements of the wrong type into a generic collection. This can happen easily when generic and non-generic code is mixed.

Example (of smuggling an alien into a collection):

```
class Legacy {
    public static List create() {
        List rawList = new ArrayList();
        rawList.add("abc");    // unchecked warning
        ...
        return rawList;
    }
    public static void insert(List rawList) {
        ...
        rawList.add(new Date()); // unchecked warning
        ...
    }
}
class Modern {
    private void someMethod() {
        List<String> stringList = Legacy.create(); // unchecked warning
        Legacy.insert(stringList);
        Unrelated.useStringList(stringList);
    }
}
class Unrelated {
    public static void useStringList(List<String> stringList) {
        ...
        String s = stringList.get(1);    // ClassCastException
        ...
    }
}
```

An "alien" `Date` object is successfully inserted into a list of strings. This can happen inadvertently when a parameterized type is passed to a piece of legacy code that accepts the corresponding raw type and then adds alien elements. The compiler can neither detect nor prevent this kind of violation of the type safety, beyond issuing an "unchecked" warning when certain methods of the raw type are invoked. The inevitable type mismatch will later show up in a potentially unrelated part of the program and will manifest itself as an unexpected `ClassCastException`.

For purposes of diagnostics and debugging JDK 5.0 adds a set of "checked" views to the collection framework (see [java.util.Collections](#)), which can detect the kind of problem explained above. If a checked view is used instead of the original collection then the error is reported at the correct location, namely when the "alien" element is inserted.

Example (of using a checked collection):

```
class Legacy {
    public static List create() {
        List rawList = new ArrayList();
        rawList.add("abc");    // unchecked warning
```

```

    ...
    return rawList;
}
public static void insert(List rawList) {
    ...
    rawList.add(new Date());    // ClassCastException
    ...
}
}
class Modern {
    private void someMethod() {
        List<String> stringList
            = Collections.checkedList(Legacy.create(),String.class); // unchecked warning
        Legacy.insert(stringList);
        Unrelated.useStringList(stringList);
    }
}
class Unrelated
    public static void useStringList(List<String> stringList) {
        ...
        String s = stringList.get(1);
        ...
    }
}

```

The checked collection is a view to an underlying collection, similar to the unmodifiable and synchronized views provided by class `Collections`. The purpose of the checked view is to detect insertion of "aliens" and prevent it by throwing a `ClassCastException` in case the element to be inserted is of an unexpected type. The expected type of the elements is provided by means of a `Class` object when the checked view is created. Each time an element is added to the checked collection a runtime type check is performed to make sure that element is of an acceptable type. Here is a snippet of the implementation of the checked view for illustration.

Example (excerpt from a checked view implementation):

```

public class Collections {
    public static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type) {
        return new CheckedCollection<E>(c, type);
    }
    private static class CheckedCollection<E> implements Collection<E> {
        final Collection<E> c;
        final Class<E> type;

        CheckedCollection(Collection<E> c, Class<E> type) {
            this.c = c;
            this.type = type;
        }
        public boolean add(E o){
            if (!type.isInstance(o))
                throw new ClassCastException();
            return c.add(o);
        }
    }
}

```

The advantage of using a checked view is that the error is reported at the correct location. The downside of using a checked collection is the performance overhead of an additional dynamic type check each time an element is inserted into the collection.

The error detection capabilities of the checked view are somewhat limited. The type check that is performed when an element is inserted into a checked collection is performed at runtime - using the runtime type representation of the expected element type. If the element type is a parameterized type the check cannot be exact, because only the raw type is available at runtime. As a result, aliens can be inserted into a checked collection, although the checked collection was invented to prevent exactly that.

Example (of limitations of checked collections):

```

class Legacy {
    public static List legacyCreate() {
        List rawList = new ArrayList();
        rawList.add(new Pair("abc","xyz")); // unchecked warning
        ...
        return rawList;
    }
    public static void legacyInsert(List rawList) {
        ...
        rawList.add(new Pair(new Date(),"Xmas")); // unchecked warning
        ...
    }
}

```

```

class Modern {
    private void someModernMethod() {
        List<Pair<String,String>> stringPairs
            = Collections.checkedList(legacyCreate(),Pair.class); // unchecked warning
        Legacy.insert(stringPairs);
        Unrelated.useStringPairs(stringPairs);
    }
}
class Unrelated {
    public static void useStringPairs(List<Pair<String,String>> stringPairList) {
        ...
        String s = stringPairList.get(1).getFirst(); // ClassCastException
        ...
    }
}

```

The checked view can only check against the raw type `Pair` and cannot prevent that an alien pair of type `Pair<Date,String>` is inserted into the checked view to a collection of `Pair<String,String>`. Remember, parameterized types do not have an exact runtime type representation and there is not class literal for a parameterized type that we could provide for creation of the checked view.

Note, that a checked view to a collection of type `Pair<String,String>` cannot be created without a warning.

Example:

```

List<Pair<String,String>> stringPairs
    = Collections.checkedList
      (new ArrayList<Pair<String,String>>(),Pair.class); // error

List<Pair<String,String>> stringPairs
    = Collections.checkedList
      ((List<Pair>)(new ArrayList<Pair<String,String>>()),Pair.class); // error

List<Pair<String,String>> stringPairs
    = Collections.checkedList
      ((List)(new ArrayList<Pair<String,String>>()),Pair.class); // unchecked warning

```

We cannot create a checked view to a parameterized type such as `List<Pair<String,String>>`, because it is required that we supply the runtime type representation of the collection's element type as the second argument to the factory method `Collections.checkedList`. The element type `Pair<String,String>` does not have a runtime type representation of its own; there is no such thing as `Pair<String,String>.class`. At best, we can specify the raw type `Pair` as the runtime type representation of the collection's element type. But this is the element type of a collection of type `List<Pair>`, not of a `List<Pair<String,String>>`.

This explains why we have to add a cast. The natural cast would be to type `List<Pair>`, but the conversion from `ArrayList<Pair<String,String>>` to `List<Pair>` is not permitted. These two types are incompatible because they are instantiations of the same generic type for different type arguments.

As a workaround we resort to the raw type `List`, because the conversion `ArrayList<Pair<String,String>>` to `List` is permitted for reasons of compatibility. Use of the raw type results in the usual "unchecked" warnings. In this case the compiler complains that we pass a raw type `List` as the first argument to the `Collections.checkedList` method, where actually a `List<Pair>` is expected.

In general, we cannot create a checked view to an instantiation of a collection whose type argument is a parameterized type (such as `List<Pair<String,String>>`). This is only possible using debatable casts, as demonstrated above. However, it is likely that checked collections are used in cases where generic and non-generic legacy code is mixed, because that is the situation in which alien elements can be inserted into a collection inadvertently. In a mixed style context, you might not even notice that you work around some of the compiler's type checks, when you create a checked view, because you have to cope with countless "unchecked" warnings anyway.

The point to take home is that checked views provide a certain safety net for collections whose element type is a raw type, but fails to provide the same kind of safety for collections whose element type is a parameterized type.

LINK TO THIS [Practicalities.FAQ004](#)

REFERENCES

- [class java.util.Collections](#)
- [What is an "unchecked" warning?](#)
- [What is the raw type?](#)
- [What happens when I mix generic and non-generic code?](#)
- [How do I pass type information to a method so that it can be used at runtime?](#)
- [How do I perform a runtime type check whose target type is a type parameter?](#)
- [Why is there no class literal for concrete parameterized types?](#)
- [How does the compiler translate Java generics?](#)
- [What is type erasure?](#)
- [What is the type erasure of a parameterized type?](#)

## What is the difference between a `Collection<?>` and a `Collection<Object>`?

`Collection<Object>` is a heterogenous collection, while `Collection<?>` is a homogenous collection of elements of the same unknown type.

The type `Collection<Object>` is a *heterogenous* collection of objects of different types. It's a mixed bag and can contain elements of all reference types.

The type `Collection<?>` stands for a representative from the family of types that are instantiations of the generic interface `Collection`, where the type argument is an arbitrary reference type. For instance, it refers to a `Collection<Date>`, or a `Collection<String>`, or a `Collection<Number>`, or even a `Collection<Object>`.

A `Collection<?>` is a *homogenous* collection in the sense that it can only contain elements that have a common unknown supertype, and that unknown supertype might be more restrictive than `Object`. If the unknown supertype is a `final` class then the collection is truly homogenous. Otherwise, the collection is not really homogenous because it can contain objects of different types, but all these types are subtypes of the unknown supertype. For instance, the `Collection<?>` might stand for `Collection<Number>`, which is homogenous in the sense that it contains numbers and not apples or pears, yet it can contain a mix of elements of type `Short`, `Integer`, `Long`, etc.

A similar distinction applies to bounded wildcards, not just the unbounded wildcard `"?"`.

A `List<Iterable>` is a concrete parameterized type. It is a mixed list of objects whose type is a subtype of `Iterable`. I can contain an `ArrayList` and a `TreeSet` and a `SynchronousQueue`, and so on.

A `List<? extends Iterable>` is a wildcard parameterized type and stands for a representative from the family of types that are instantiations of the generic interface `List`, where the type argument is a subtype of `Iterable`, or `Iterable` itself. Again, the list is truly homogenous if the unknown subtype of `Iterable` is a `final` class. Otherwise, it is a mix of objects with a common unknown supertype and that supertype itself is a subtype of `Iterable`. For example, `List<? extends Iterable>` might stand for `List<Set>`, which is homogenous in the sense that it contains sets and not lists or queues. Yet the `List<Set>` can be heterogenous because it might contain a mix of `TreeSets` and `HashSets`.

LINK TO THIS [Practicalities.FAQ005](#)

REFERENCES [What is a concrete parameterized type?](#)  
[What is a wildcard parameterized type?](#)

## How do I express that a collection is a mix of objects of different types?

### Using wildcard instantiations of the generic collections.

Occasionally, we want to refer to sequences of objects of different types. An example would be a `List<Object>` or a `Object[]`. Both denote sequences of objects of arbitrary types, because `Object` is the supertype of all reference types.

How do we express a sequence of objects not of arbitrary different types, but of different instantiations of a certain generic type? Say, we need to refer to a sequence of pairs of arbitrary elements. We would need the supertype of all instantiations of the generic `Pair` type. This supertype is the unbounded wildcard instantiation `Pair<?,?>`. Hence a `List<Pair<?,?>>` and a `Pair<?,?>[]` would denote sequences of pairs of different types.

	of any type	of any pair type
collection	<code>List&lt;Object&gt;</code>	<code>List&lt;Pair&lt;?,?&gt;&gt;</code>
array	<code>Object[]</code>	<code>Pair&lt;?,?&gt;[]</code>

When we want to refer to a mixed sequence of certain types, instead of all arbitrary types, we use the supertype of those "certain types" to express the mixed sequence. Examples are `List<Number>` or `Number[]`. The corresponding mixed sequences of instantiations of a generic type is expressed in a similar way. A mixed sequences of pairs of numbers can be expressed as `List<Pair<? extends Number, ? extends Number>>` or as `Pair<? extends Number, ? extends Number>[]`.

	of any number type	of any type of pair of numbers
collection	<code>List&lt;Number&gt;</code>	<code>List&lt;Pair&lt;? extends Number, ? extends Number&gt;&gt;</code> *) Legal as the type of reference variable, but illegal in a new expression.
array	<code>Number[]</code>	<code>Pair&lt;? extends Number, ? extends Number&gt;[]</code> *)

The array type `Pair<? extends Number, ? extends Number>[]` needs further explanation. This type would in principle denote a mixed sequence of pairs of different type, but this array type is not overly useful. It can only be used for declaration of reference variables, while it must not appear in new expressions. That is, we can declare reference variables of type `Pair<? extends Number, ? extends Number>[]`, but the reference can never refer to an array of its type, because no such array can be created.

Example (of illegal array creation):

```
Pair<? extends Number, ? extends Number>[] makeNumberPairs(int size) {  
    return new Pair<? extends Number, ? extends Number>[size]; // error  
}
```

```
error: generic array creation  
return new Pair<? extends Number, ? extends Number>[size];  
    ^
```

By and large an array type such as `Pair<? extends Number, ? extends Number>[]` is not particularly useful, because it cannot refer to an array of its type. It can refer to an array of the corresponding raw type, i.e. `Pair[]`, or to an array of a non-generic subtype, e.g. `Point[]`, where `Point` is a subclass of `Pair<Double,Double>` for instance. In each of these cases using a reference variable of type `Pair<? extends Number, ? extends`

Number>[] offers

no advantage over using a reference variable that matches the type of the array being referred to. Quite the converse; it is error prone and should be avoided. This rule applies to all array types with a component type that is a concrete or bounded wildcard parameterized type. For details see [ParameterizedTypes.FAQ104A](#) and [ParameterizedTypes.FAQ307A](#).

Note that arrays of unbounded wildcard parameterized types do not suffer from this restriction. The creation of an array of an unbounded wildcard parameterized type is permitted, because the unbounded wildcard parameterized type is a so-called reifiable type, so that an array reference variable with an unbounded wildcard parameterized type as its component type, such as `Pair<?,?>[]`, can refer to an array of its type.

Example (of legal array creation):

```
Pair<?,?>[] makeNumberPairs(int size) {
    return new Pair<?,?>[size]; // fine
}
```

LINK TO THIS  
REFERENCES

[Practicalities.FAQ006](#)

[Can I create an object whose type is a wildcard parameterized type?](#)

[Can I create an array whose component type is a wildcard parameterized type?](#)

[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)

[Can I declare a reference variable of an array type whose component type is a concrete parameterized type?](#)

[Can I declare a reference variable of an array type whose component type is a bounded wildcard parameterized type?](#)

[Can I declare a reference variable of an array type whose component type is an unbounded wildcard parameterized type?](#)

[What is a reifiable type?](#)

---

## What is the difference between a `Collection<Pair<String, Object>>`, a `Collection<Pair<String, ?>>` and a `Collection<? extends Pair<String, ?>>`?

*All three types refer to collections that hold pairs where the first part is a `String` and the second part is of an arbitrary type. The differences are subtle.*

The three parameterized types are relatively similar. They all refer to collections that hold pairs where the first part is a `String` and the second part is of an arbitrary type.

Let us start with a comparison of the two concrete parameterized types `Collection<Pair<String, Object>>` and `Collection<Pair<String, ?>>`. The both contain pairs where the first part is a `String`. The individual pairs stored in the collection can for instance contain a `String` and a `Date`, or a `String` and an `Object`, or a `String` and a `String`. The difference lies in the types of the pairs that can be added to the two collections.

Example (using a `Collection<Pair<String, Object>>`):

```
Collection<Pair<String, Object>> c = new ArrayList<Pair<String, Object>>();

c.add(new Pair<String, Date> ("today", new Date())); // error: illegal argument type
c.add(new Pair<String, Object>("today", new Date())); // fine

c.add(new Pair<String, String>("name", "Pete Becker")); // error: illegal argument type
c.add(new Pair<String, Object>("name", "Pete Becker")); // fine
```

The example demonstrates that only pairs of type `Pair<String, Object>` can be added to a `Collection<Pair<String, Object>>`. A `Collection<Pair<String, Object>>` is a homogenous collections of elements of the same type. The individual pairs may contain different things, as long as the type of the pair is `Pair<String, Object>`. For instance, a pair may consist of a `String` and a `Date`, but it must not be of type `Pair<String, Date>`.

Example (using a `Collection<Pair<String, ?>>`):

```
Collection<Pair<String, ?>> c = new ArrayList<Pair<String, ?>>();

c.add(new Pair<String, Date> ("today", new Date())); // fine
c.add(new Pair<String, Object>("today", new Date())); // fine

c.add(new Pair<String, String>("name", "Pete Becker")); // fine
c.add(new Pair<String, Object>("name", "Pete Becker")); // fine
```

The example illustrates that a `Collection<Pair<String, ?>>` accepts all types of pairs as long as the first type argument is `String`. For instance, a pair of type `Pair<String, Date>` is accepted. A `Collection<Pair<String, ?>>` is a heterogenous collections of elements of the similar types.

The key difference between a `Collection<Pair<String, Object>>` and a `Collection<Pair<String, ?>>` is that the first contains elements of the same type and the latter contains elements of different similar types.

The type `Collection<? extends Pair<String, ?>>` is fundamentally different. It is a wildcard parameterized type, not a concrete parameterized type. We simply do not know what exactly a reference variable of the wildcard type refers to.

Example (using a `Collection<? extends Pair<String, ?>>`):

```
Collection<? extends Pair<String, ?>> c = new ArrayList<Pair<String, ?>>();
```

```

c.add(new Pair<String,Date> ("today", new Date())); // error: add method must not be called
c.add(new Pair<String,Object>("today", new Date())); // error: add method must not be called

c.add(new Pair<String,String>("name","Pete Becker")); // error: add method must not be called
c.add(new Pair<String,Object>("name","Pete Becker")); // error: add method must not be called

```

The type `Collection<? extends Pair<String,?>>` stands for a representative from the family of all instantiations of the generic type `Collection` where the type argument is a subtype of type `Pair<String,?>`. This type family includes members such as `Pair<String,String>`, `Pair<String,Object>`, `Pair<String,? extends Number>`, and `Pair<String,?>` itself.

Methods like `add` must not be invoked through a reference of a wildcard type. This is because the `add` method takes an argument of the unknown type that the wildcard stands for. Using the variable `c` of the wildcard type `Collection<? extends Pair<String,?>>`, nothing can be added to the collection. This does not mean that the collection being referred to does not contain anything. We just do not know what exactly the type of the collection is and consequently we do not know what type of elements it contains. All we know is that it contains pairs where the first part is a `String`. But we do not know of which type the second part of the pair is, or whether or not all pairs are of the same type.

So far, we've silently assumed that `Pair` is a final class. What if it has subtypes? Say, it has a subtype class `SubTypeOfPair<X,Y>` extends `Pair<X,Y>`.

In that case, a `Collection<Pair<String,Object>>` may not only contain objects of type `Pair<String,Object>`, but also objects of type `SubTypeOfPair<String,Object>`.

A `Collection<Pair<String,?>>` may not only contain objects of different pair types such as `Pair<String,Date>` and `Pair<String,Object>`, but also objects of subtypes of those, such as `SubTypeOfPair<String,Date>` and `SubTypeOfPair<String,Object>`.

The type `Collection<? extends Pair<String,?>>` stands for a representative from the family of all instantiations of the generic type `Collection` where the type argument is a subtype of type `Pair<String,?>`. This type family is now even larger. It does not only include members such as `Pair<String,String>`, `Pair<String,Object>`, `Pair<String,? extends Number>`, and `Pair<String,?>` itself, but also type such as `SubTypeOfPair<String,String>`, `SubTypeOfPair<String,Object>`, `SubTypeOfPair<String,? extends Number>`, and `SubTypeOfPair<String,?>`.

LINK TO THIS [Practicalities.FAQ006A](#)

REFERENCES [What is a bounded wildcard?](#)  
[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)  
[What is the difference between a Collection<?> and a Collection<Object>?](#)  
[Which super-subset relationships exist among wildcards?](#)

## How can I make sure that a wildcard that occurs repeatedly in the same scope stands for the same type?

*In general you can't.*

If the same wildcard appears repeatedly, each occurrence of the wildcard stands for a potentially different type. There is no way to make sure that the same wildcard represents the same type.

Example (using the same wildcard repeatedly):

```

Pair<?,?> couple = new Pair<String,String>("Orpheus","Eurydike");
Pair<?,?> xmas    = new Pair<String,Date>("Xmas", new Date(104,11,24));

```

There is nothing you can do to make sure that a reference variable of type `Pair<?,?>` represents a pair of elements of the same type.

Depending on the circumstances there might be work-arounds that achieve this goal. For instance, if the type `Pair<?,?>` is the type of a method argument, it might be possible to generify the method to ensure that the method argument is a pair of elements of the same type.

For instance, the following method

```
void someMethod(Pair<?,?> pair) { ... }
```

accepts all types of pairs. It is mostly equivalent to the following generic method:

```
<X,Y> void someMethod(Pair<X,Y> pair) { ... }
```

In order to make sure that only pairs of elements of the same type are passed to the method, the method can be generified as follows:

```
<T> void someMethod(Pair<T,T> pair) { ... }
```

Now it is guaranteed that the method accepts only pairs of elements of the same type.

LINK TO THIS [Practicalities.FAQ007](#)

REFERENCES [What is a wildcard parameterized type?](#)  
[If a wildcard appears repeatedly in a type argument section, does it stand for the same type?](#)



---

## Using Generic Methods

---

### Why doesn't method overloading work as I expect it?

*Because there is only one byte code representation of each generic type or method.*

When you invoke an overloaded method and pass an argument to the method whose type is a type variable or involves a type variable, you might observe surprising results. Let us study an example.

Example (of invocation of an overloaded method):

```
static void overloadedMethod(Object o) {
    System.out.println("overloadedMethod(Object) called");
}
static void overloadedMethod(String s) {
    System.out.println("overloadedMethod(String) called");
}
static void overloadedMethod(Integer i) {
    System.out.println("overloadedMethod(Integer) called");
}

static <T> void genericMethod(T t) {
    overloadedMethod(t); // which method is called?
}

public static void main(String[] args) {
    genericMethod("abc");
}
```

We have several overloaded versions of a method. The overloaded method is invoked by a generic method which passes an argument of type `T` to the overloaded method. Eventually the generic method is called and a string is passed as an argument to the generic method. One might expect that inside the generic method the string version of the overloaded method is invoked, because the method argument is a string. This, however, is wrong.

The program prints:

```
overloadedMethod(Object) called
```

How can this happen? We pass an argument of type `String` to the overloaded method and yet the version for type `Object` is called. The reason is that the compiler creates only one byte code representation per generic type or method and maps all instantiations of the generic type or method to that one representation.

In our example the generic method is translated to the following representation:

```
void genericMethod(Object t) {
    overloadedMethod(t);
}
```

Considering this translation, it should be obvious why the `Object` version of the overloaded method is invoked. It is entirely irrelevant what type of object is passed to the generic method and then passed along to the overloaded method. We will always observe a call of the `Object` version of the overloaded method.

More generally speaking: overload resolution happens at compile time, that is, the compiler decides which overloaded version must be called. The compiler does so when the generic method is translated to its unique byte code representation. During that translation type erasure is performed, which means that type parameters are replaced by their leftmost bound or `Object` if no bound was specified. Consequently, the leftmost bound or `Object` determines which version of an overloaded method is invoked. What type of object is passed to the method at runtime is entirely irrelevant for overload resolution.

---

Here is another even more puzzling example.

Example (of invocation of an overloaded method):

```
public final class GenericClass<T> {

    private void overloadedMethod(Collection<?> o) {
        System.out.println("overloadedMethod(Collection<?>)");
    }
}
```



```

    }
    private void overloadedMethod(List<Number> s) {
        System.out.println("overloadedMethod(List<Number>");
    }
    private void overloadedMethod(ArrayList<Integer> i) {
        System.out.println("overloadedMethod(ArrayList<Integer>");
    }

    private void method(List<T> t) {
        overloadedMethod(t); // which method is called?
    }

    public static void main(String[] args) {
        GenericClass<Integer> test = new GenericClass<Integer>();
        test.method(new ArrayList<Integer>());
    }
}

```

The program prints:

```
overloadedMethod(Collection<?>)
```

One might have expected that version for `ArrayList<Integer>` would be invoked, but that again is the wrong expectation. Let us see what the compiler translates the generic class to.

Example (after type erasure):

```

public final class GenericClass {

    private void overloadedMethod(Collection o) {
        System.out.println("overloadedMethod(Collection<?>");
    }
    private void overloadedMethod(List s) {
        System.out.println("overloadedMethod(List<Number>");
    }
    private void overloadedMethod(ArrayList i) {
        System.out.println("overloadedMethod(ArrayList<Integer>");
    }

    private void method(List t) {
        overloadedMethod(t);
    }

    public static void main(String[] args) {
        GenericClass test = new GenericClass();
        test.method(new ArrayList());
    }
}

```

One might mistakenly believe that the compiler would decide that the `List` version of the overloaded method is the best match. But that would be wrong, of course. The `List` version of the overloaded method was originally a version that takes a `List<Number>` as an argument, but on invocation a `List<T>` is passed, where `T` can be any type and need not be a `Number`. Since `T` can be any type the only viable version of the overloaded method is the version for `Collection<?>`.

### Conclusion:

Avoid passing type variables to overloaded methods. Or, more precisely, be careful when you pass an argument to an overloaded method whose type is a type variable or involves a type variable.

LINK TO THIS [Practicalities.FAQ050](#)

REFERENCES

- [How does the compiler translate Java generics?](#)
- [What is type erasure?](#)
- [What is method overriding?](#)
- [What is method overloading?](#)
- [What is a method signature?](#)
- [What is the @Override annotation?](#)
- [What are override-equivalent signatures?](#)
- [When does a method override its supertype's method?](#)
- [What is overload resolution?](#)

## Why doesn't method overriding work as I expect it?

*Because the decision regarding overriding vs. overloading is based on the generic type, not on any instantiation thereof.*

Sometimes, when you believe you override a method inherited from a supertype you inadvertently overload instead of override the inherited method. This can lead to surprising effects. Let us study an example.

Example (of overloading):

```
class Box<T> {
    private T theThing;
    public Box(T t)      { theThing = t; }
    public void reset(T t) { theThing = t; }
    ...
}
class WordBox<S extends CharSequence> extends Box<String> {
    public WordBox(S t)      { super(t.toString().toLowerCase()); }
    public void reset(S t) { super.reset(t.toString().toLowerCase()); }
    ...
}
class Test {
    public static void main(String[] args) {
        WordBox<String> city = new WordBox<String>("Skogland");
        city.reset("Stavanger"); // error: ambiguous
    }
}
```

---

```
error: reference to reset is ambiguous,
both method reset(T) in Box<String> and method reset(T) in WordBox<String> match
    city.reset("Stavanger");
        ^
```

In this example, one might be tempted to believe that the method `WordBox<String>.reset(String)` overrides the superclass method `Box<String>.reset(String)`. After all, both methods have the same name and the same parameter types. Methods with the same name and the same parameter types in a super- and a subtype are usually override-equivalent. For this reason, we might expect that the invocation of the `reset` method in the `Test` class leads to the execution of the `WordBox<String>.reset(String)` method. Instead, the compiler complains about an ambiguous method call. Why?

The problem is that the subclass's `reset` method does not override the superclass's `reset` method, but overloads it instead. You can easily verify this by using the `@Override` annotation.

Example (of overloading):

```
class Box<T> {
    private T theThing;
    public Box(T t)      { theThing = t; }
    public void reset(T t) { theThing = t; }
    ...
}
class WordBox<S extends CharSequence> extends Box<String>{
    public WordBox(S t)      { super(t.toString().toLowerCase()); }
    @Override
    public void reset(S t) { super.reset(t.toString().toLowerCase()); }
    ...
}
```

---

```
error: method does not override a method from its superclass
    @Override
    ^
```

When a method is annotated by an `@Override` annotation, the compiler issues an error message if the annotated method does not override any of its supertype's methods. If it does not override, then it overloads or hides methods with the same name inherited from its supertype. In our example the `reset` method in the generic `WordBox<S extends CharSequence>` class overloads the `reset` method in the parameterized `Box<String>` class.

The overloading happens because the two methods have different signatures. This might come as a surprise, especially in the case of the instantiation `WordBox<String>`, where the two `reset` methods have the same name and the same parameter type.

The point is that the compiler decides whether a subtype method overrides or overloads a supertype method when it compiles the generic subtype, independently of any instantiations of the generic subtype. When the compiler compiles the declaration of the generic `WordBox<S extends CharSequence>` class, then there is no knowledge regarding the concrete type by which the type parameter `S` might later be replaced. Based on the declaration of the generic subtype the two `reset` methods have different signatures, namely `reset(String)` in the supertype and `reset(S_extends_CharSequence)` in the generic subtype. These are two completely different signatures that are not override-equivalent. Hence the compiler considers them overloading versions of each other.

In a certain instantiation of the subtype, namely in `WordBox<String>`, the type parameter `S` might be replaced by the concrete type `String`. As a result both `reset` methods visible in `WordBox<String>` suddenly have the same argument type. But that does not change the fact that the two methods still have different signatures and therefore overload rather than override each other.

The identical signatures of the two overloading version of the `reset` method that are visible in `WordBox<String>` lead to the ambiguity that we observe in our example. When the `reset` method is invoked through a reference of type `WordBox<String>`, then the compiler finds both

overloading versions. Both versions are perfect matches, but neither is better than the other, and the compiler rightly reports an ambiguous method call.

---

### Conclusion:

Be careful when you override methods, especially when generic types or generic methods are involved. Sometimes the intended overriding turns out to be considered overloading by the compiler, which leads to surprising and often confusing results. In case of doubt, use the `@Override` annotation.

LINK TO THIS [Practicalities.FAQ051](#)

REFERENCES

- [How does the compiler translate Java generics?](#)
- [What is type erasure?](#)
- [What is method overriding?](#)
- [What is method overloading?](#)
- [What is a method signature?](#)
- [What is the @Override annotation?](#)
- [When does a method override its supertype's method?](#)
- [Can a method of a generic subtype override a method of a generic supertype?](#)

---

## Coping With Legacy

### What happens when I mix generic and non-generic legacy code?

*The compiler issues lots of "unchecked" warnings.*

It is permitted that a generic class or method is used in both its parameterized and its raw form. Both forms can be mixed freely. However, all uses that potentially violate the type-safety are reported by means of an "unchecked warning". In practice, you will see a lot of unchecked warnings when you use generic types and methods in their raw form.

Example (of mixing parameterized and raw use of a generic type):

```
interface Comparable<T> {
    int compareTo(T other);
}
class SomeClass implements Comparable {
    public int compareTo(Object other) {
        ...
    }
}
class Test {
    public static void main(String[] args) {
        Comparable x = new SomeClass();
        x.compareTo(x);    // "unchecked" warning
    }
}
```

---

```
warning: [unchecked] unchecked call to compareTo(T) as a member of the raw type java.lang.Comparable
    x.compareTo(x);
      ^
```

The `Comparable` interface is a generic type. Its raw use in the example above leads to "unchecked" warnings each time the `compareTo` method is invoked.

The warning is issued because the method invocation is considered a potential violation of the type-safety guarantee. This particular invocation of `compareTo` is not unsafe, but other methods invoked on raw types might be.

Example (of type-safety problem when mixing parameterized and raw use):

```
class Test {
    public static void someMethod(List list) {
        list.add("xyz");    // "unchecked" warning
    }
    public static void test() {
        List<Long> list = new ArrayList<Long>();
        someMethod(list);
    }
}
```

---

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type java.util.List
    list.add("xyz");
      ^
```

Similar to the previous example, the invocation of the `add` method on the raw type `List` is flagged with an "unchecked" warning. The invocation is indeed unsafe, because it inserts a string into a list of long values.

The compiler cannot distinguish between invocations that are safe and those that are not. It reports "unchecked" warnings just in case that a call might be unsafe. It applies a simple rule: every invocation of a method of a raw type that takes an argument of the unknown type that the class's type parameter stands for, is potentially unsafe. That does not mean, it must be unsafe (see `Comparable.compareTo`), but it can be unsafe (see `List.add`).

If you find that you must intermix legacy and generic code, pay close attention to the unchecked warnings. Think carefully how you can justify the safety of the code that gives rise to the warning. Once you've made sure the warning is harmless suppress it using the `SuppressWarnings` annotation.

If you can re-engineer existing code or if you write new code from scratch you should use generic types and methods in their parameterized form and avoid any raw use. For instance, the examples above can be "repaired" as follows:

Example #1 (corrected):

```
interface Comparable<T> {
    int compareTo(T other);
}
class SomeClass implements Comparable<Object> {
    public int compareTo(Object other) {
        ...
    }
}
class Test {
    public static void main(String[] args) {
        Comparable<Object> x = new SomeClass();
        x.compareTo(x);    // fine
    }
}
```

No "unchecked" warning occurs if the `Comparable` interface is used in its parameterized form in all places.

Example #2 (corrected):

```
class Test {
    public static void someMethod(List<String> list) {
        list.add("xyz");    // fine
    }
    public static void test() {
        List<Long> list = new ArrayList<Long>();
        someMethod(list);  // error
    }
}
```

---

```
error: someMethod(java.util.List<java.lang.String>) cannot be applied to java.util.List<java.lang.Long>
    someMethod(list);
      ^
```

The "unchecked" warning in `someMethod` is no longer necessary if the generic type `List` is used in its parameterized form as `List<String>`. With this additional type information the compiler is now capable of flagging the formerly undetected type-safety problem in method `test` as an error.

LINK TO THIS [Practicalities.FAQ101](#)

REFERENCES [What does type-safety mean?](#)  
[What is the raw type?](#)  
[Can I use a raw type like any other type?](#)  
[What is an "unchecked" warning?](#)  
[How can I disable or enable unchecked warnings?](#)  
[What is the SuppressWarnings annotation?](#)

---

## Should I re-engineer all my existing types and generify them?

*No, most likely not.*

Not all types are inherently generic. There is no point to turning a type into a generic type if the type does not semantically depend on a particular unknown type that can be more adequately be expressed by means of a type parameter.

Example (of an arbitrary non-generic type taken from package `org.w3c.dom`):

```

public interface NameList {
    boolean  contains(String str);
    boolean  containsNS(String namespaceURI, String name);
    int      getLength();
    String   getName(int index);
    String   getNamespaceURI(int index);
}

```

The `NameList` interface takes and returns either strings or primitive types and there is no reason why this class should be generic in any form.

---

Other non-generic types would benefit from generics.

Example (of another arbitrary non-generic type):

```

public interface Future {
    boolean  cancel(boolean mayInterruptIfRunning);
    Object   get();
    Object   get(long timeout, TimeUnit unit);
    boolean  isCancelled();
    boolean  isDone();
}

```

This interface has `get` methods that return `Object` references. If these methods return the same type of object for a given instance of type `Future`, then the interface is more precisely declared as a generic interface.

Example (of corresponding generic type):

```

public interface Future<V> {
    boolean  cancel(boolean mayInterruptIfRunning);
    V        get();
    V        get(long timeout, TimeUnit unit);
    boolean  isCancelled();
    boolean  isDone();
}

```

---

Occasionally, the generification of one type leads to the generification of other related types.

Example (of non-generic types taken from package `java.lang.ref` in JDK 1.4):

```

public class ReferenceQueue {
    public ReferenceQueue() { }
    public Reference poll() { ... }
    public Reference remove(long timeout)
        throws IllegalArgumentException, InterruptedException { ... }
    public Reference remove()
        throws InterruptedException { ... }
}
public abstract class Reference {
    private Object referent;
    ReferenceQueue queue;
    Reference next;
    Reference(Object referent) { ... }
    Reference(Object referent, ReferenceQueue queue) { ... }
    public void clear() { ... }
    public boolean enqueue() { ... }
    public Object get() { ... }
    public boolean isEnqueued() { ... }
}

```

The abstract class `Reference` internally holds a reference of type `Object` and has methods that take and return `Object` references. If these methods take and return the same type of object that is held internally, then the class is more precisely declared as a generic class, namely as `Reference<T>` where `T` is the type of the referent.

When we decide to parameterize class `Reference` then we must provide type arguments in all places where type `Reference` is used. This affects class `ReferenceQueue` because it has methods that return references of type `Reference`. Consequently, we would declare class `ReferenceQueue` as a generic class, too.

Once we have generified class `ReferenceQueue` then we must return to class `Reference` and provide type arguments in all places where type `ReferenceQueue` is used.

Example (of corresponding generic type in JDK 5.0):

```

public class ReferenceQueue<T> {
    public ReferenceQueue() { }
    public Reference<? extends T> poll() { ... }
    public Reference<? extends T> remove(long timeout)
}

```

```

    throws IllegalArgumentException, InterruptedException { ... }
public Reference<? extends T> remove()
    throws InterruptedException { ... }
}
public abstract class Reference<T> {
    private T referent;
    ReferenceQueue<? super T> queue;
    Reference next;
    Reference(T referent) { ... }
    Reference(T referent, ReferenceQueue<? super T> queue) { ... }
    public void clear() { ... }
    public boolean enqueue() { ... }
    public T get() { ... }
    public boolean isEnqueued() { ... }
}

```

This is an example where a class, namely `ReferenceQueue`, is turned into a generic class because the types it uses are generic. This propagation of type parameters into related types is fairly common. For instance, the subtypes of type `Reference` (namely `PhantomReference`, `SoftReference`, and `WeakReference`) are generic types as well.

LINK TO THIS [Practicalities.FAQ102](#)

REFERENCES [How do I generify an existing non-generic class?](#)

## How do I generify an existing non-generic type or method?

*There are no carved-in-stone rules. It all depends on the intended semantics of the generified type or method.*

Modifying an existing type that was non-generic in the past so that it becomes usable as a parameterized type in the future is a non-trivial task. The generification must not break any existing code that uses the type in its old non-generic form and it must preserve the original non-generic type's semantic meaning.

For illustration, we study a couple of examples from the collection framework (see package `java.util` in [J2SE 1.4.2](#) and [J2SE 5.0](#)). We will generify the traditional non-generic interface `Collection`. From the semantics of a collection it is obvious that for a homogenous collection of elements of the same type the element type would be the type parameter of a generic `Collection` interface.

Example (from JDK 1.4; before generification):

```

interface Collection {
    boolean add    (Object o);
    boolean contains(Object o);
    boolean remove (Object o);
    ...
}

```

These methods take an element as an argument and insert, find or extract the element from the collection. In a generic collection the method parameters would be of type `E`, the interface's type parameter.

Example (from JDK 5.0; after generification):

```

interface Collection<E> {
    boolean add    (E o);
    boolean contains(E o);
    boolean remove (E o);
    ...
}

```

However, this modification does not exactly preserve the semantics of the old class. Before the generification it was possible to pass an arbitrary type of object to these methods. After the generification only objects of the "right" type are accepted as method arguments.

Example (of modified semantics):

```

class ClientRepository {
    private Collection<Client> clients = new LinkedList<Client>();
    ...
    boolean isClient(Object c) {
        return clients.contains(c); // error
    }
}

```

Passing an `Object` reference to method `contains` used to be permitted before the generification, but no longer compiles after generification. Seemingly, our generified type is not semantically compatible with the original non-generic type. A more relaxed generification would look like this.

Example (from JDK 5.0; after an alternative generification):

```

interface Collection<E> {
    boolean add      (E o);
    boolean contains(Object o);
    boolean remove  (Object o);
    ...
}

```

Only for the `add` method now would accept the more restrictive method parameter type `E`. Since a `Collection<E>` is supposed to contain only elements of type `E`, it is expected and desired that insertion of an alien element is rejected at compile time.

This seemingly trivial example illustrates that decisions regarding a "correct" generification are largely a matter of taste and style. Often, there are several viable approaches for a generification. Which one is "correct" depends on the specific requirements to and expectations of the semantics of the resulting generified type.

LINK TO THIS [Practicalities.FAQ103](#)

REFERENCES [How do I avoid breaking binary compatibility when I generify an existing type or method?](#)

## Can I safely generify a supertype, or does it affect all subtypes?

*Yes, we can generify non-generic legacy supertypes without affecting the non-generic legacy subtypes - provided the subtype method's signature is identical to the erasure of the supertype method's signature.*

Assume we have a class hierarchy of legacy types and we want to generify the supertype. Must we also generify all the subtypes? Fortunately not. Let us consider an example.

Example (of a hierarchy of legacy types):

```

class Box {
    private Object theThing;
    public Box(Object t)      { theThing = t; }
    public void reset(Object t) { theThing = t; }
    public Object get()      { return theThing; }
    ...
}
class NamedBox extends Box {
    private String theName;
    public NamedBox(Object t,String n) { super(t); theName = n; }
    public void reset(Object t)      { super.reset(t); }
    public Object get()              { return super.get(); }
    ...
}

```

Now we decide to generify the supertype.

Example (same as before, but with generified superclass):

```

class Box<T> {
    private T theThing;
    public Box(T t)      { theThing = t; }
    public void reset(T t) { theThing = t; }
    public T get()      { return theThing; }
    ...
}
class NamedBox extends Box {
    private String theName;
    public NamedBox(Object t,String n) { super(t); theName = n; }
    public void reset(Object t)      { super.reset(t); }
    public Object get()              { return super.get(); }
    ...
}

```

---

```

warning: [unchecked] unchecked call to Box(T) as a member of the raw type Box
    public NamedBox(Object t,String n) { super(t); theName = n; }
                                   ^

```

```

warning: [unchecked] unchecked call to reset(T) as a member of the raw type Box
    public void reset(Object t)      { super.reset(t); }
                                   ^

```

The subclass is still considered a subtype of the now generic supertype where the `reset` and `get` method override the corresponding supertype methods. Inevitably, we now receive unchecked warnings whenever we invoke certain methods of the supertype because we are now using methods of a raw type. But other than that, the subtype is not affected by the re-engineering of the supertype. This is possible because the signatures of the subtype methods are identical to the erasures of the signatures of the supertype methods.

Let us consider a slightly different generification. Say, we re-engineer the superclass as follows.

Example (same as before, but with a different generification):

```
class Box<T> {
    private T theThing;
    public <S extends T> Box(S t)          { theThing = t; }
    public <S extends T> void reset(S t) { theThing = t; }
    public T get() { return theThing; }
    ...
}
class NamedBox extends Box {
    private String theName;
    public NamedBox(Object t,String n) { super(t); theName = n; }
    public void reset(Object t)        { super.reset(t); }
    public Object get()                 { return super.get(); }
    ...
}
```

This time the `reset` method is a generic method. Does the subtype method `reset` still override the generic supertype method? The answer is: yes. The subtype method's signature is still identical to the erasure of the supertype method's signature and for this reason the subtype method is considered an overriding method. Naturally, we still receive the same unchecked warnings as before, but beyond that there is no need to modify the subtype although we re-engineered the supertype.

The point to take home is that methods in a legacy subtype can override (generic and non-generic) methods of a generic supertype as long as the subtype method's signature is identical to the erasure of the supertype method's signature.

LINK TO THIS [Practicalities.FAQ103A](#)

REFERENCES [Can a method of a non-generic subtype override a method of a generic supertype?](#)

How does the compiler translate Java generics?

[What is type erasure?](#)

[What is method overriding?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

[When does a method override its supertype's method?](#)

---

## How do I avoid breaking binary compatibility when I generify an existing type or method?

*Sometimes a dummy bound does the trick.*

Occasionally, one must pay attention to the fact that a generification might change the signature of some methods in the byte code. Changing the signature will break existing code that cannot be recompiled and relies on the binary compatibility of the old and new version of the `.class` file.

Example (before generification, taken from package [java.util](#)):

```
class Collections {
    public static Object max(Collection coll) {...}
    ...
}
```

The `max` method finds the largest element in a collection and obviously the declared return type of the method should match the element type of the collection passed to the method. A conceivable generification could look like this.

Example (after a naive generification):

```
class Collections {
    public static <T extends Comparable<? super T>>
        T max(Collection<? extends T> coll) {...}
    ...
}
```

While this generification preserves the semantics of the method, it changes the signature of the `max` method. It is now a method with return type `Comparable`, instead of `Object`.

Example (after type erasure):

```
class Collections {
    public static Comparable max(Collection coll) {...}
    ...
}
```

This will break existing code that relies on the binary compatibility of the `.class` files. In order to preserve the signature and thus the binary compatibility, an otherwise superfluous bound can be used.



Example (after binary compatible generification, as available in package [java.util](#)):

```
class Collections {
    public static <T extends Object & Comparable<? super T>>
        T max(Collection<? extends T> coll) {...}
    ...
}
```

The leftmost bound of the type parameter is now type `Object` instead of type `Comparable`, so that the type parameter `T` is replaced by `Object` during type erasure.

Example (after type erasure):

```
class Collections {
    public static Object max(Collection coll) {...}
    ...
}
```

---

Afterthought:

Perhaps you wonder why the hack described in this FAQ entry is needed. Indeed, had the `Collections.max` method been defined as returning a `Comparable` in the first place, no further measures, such as adding `Object` as a type parameter bound, had been required to preserve binary compatibility. Basically, the declared return type `Object` is a mistake in the design of this method.

If you carefully study the specification of the `Collections.max` method's functionality then you realize that all elements of the collection are required to implement the `Comparable` interface. Consequently, the returned object is `Comparable`, too. There is no reason why the method should return an `Object` reference.

The only explanation one can think of is that in pre-generic Java there was no way of ensuring by compile-time type checks that the `Collection` contains only `Comparable` objects. However, this was ensured via runtime type checks, namely an explicit downcast in the implementation of the method. Hence this is not really an excuse for the bug.

Note, that the runtime time type check in the pre-generic version of the `Collections.max` method still exists in the generic version. The former explicit cast is now an implicit one generated by the compiler. In the generic version, this cast can never fail (unless there are unchecked warnings), because the type parameter bound `Comparable` ensures at compile-time that the elements in the `Collection` are `Comparable`.

LINK TO THIS [Practicalities.FAQ104](#)

REFERENCES

---

---

## Defining Generic Types and Methods

---

### Which types should I design as generic types instead of defining them as regular non-generic types?

*Types that use supertype references in several places and where there is a correspondence between the occurrences of these supertype references.*

Not all types are inherently generic, not even the majority of the types in a program is. The question is: which types profit from being generic types and which ones do not. This FAQ entry tries to sketch out some guidelines.

Obvious candidates for generic types are those types that work closely with existing generic types. For instance, when you derive from a generic type, such as `WeakReference<T>`, then the derived class is often generic as well.

Example (of a generic subclass):

```
class WeakKey<T> extends java.lang.ref.WeakReference<T> {
    private int hash;
    public WeakKey(T ref) { super(t); hash = t.hashCode(); }
    ...
    public int hashCode() { return hash; }
}
```

The subclass `WeakKey` can be used in lieu of its superclass `WeakReference` and therefore is as generic as the superclass is.

Classes that use generic types are sometimes generic as well. For instance, if you want to build a cache abstraction as a map of a key and an associated value that is referred to by a soft reference, then this new `Cache` type will naturally be a generic type.

Example (of a generic cache):

```

class Cache<K,V> {
    private HashMap<K,SoftReference<V>> theCache;
    ...
    public V get(K key) { ... }
    public V put(K key, V value) { ... }
    ...
}

```

The `Cache` class is built on top of a `Map` and can be seen as a wrapper around a `Map` and therefore is as generic as the `Map` itself.

On the other hand, a cache type need not necessarily be a generic type. If you know that all keys are strings and you do not want to have different types of caches for different types of cached values, then the `Cache` type might be a non-generic type, despite of the fact that it works closely with the generic `Map` type.

Example (of a non-generic cache):

```

class Cache {
    private HashMap<String,SoftReference<Object>> theCache;
    ...
    public Object get(String key) { ... }
    public Object put(String key, Object value) { ... }
    ...
}

```

Both abstractions are perfectly reasonable. The first one is more flexible. It includes the special case of a `Cache<String,Object>`, which is the equivalent to the non-generic `Cache` abstraction. In addition, the generic `Cache` allows for different cache types. You could have a `Cache<Link,File>`, a `Cache<CustomerName,CustomerRecord>`, and so on. By means of the parameterization you can put a lot more information into the type of the cache. Other parts of your program can take advantage of the enhanced type information and can do different things for different types of caches - something that is impossible if you have only one non-generic cache type.

Another indication for a generic type is that a type uses the same supertype in several places. Consider a `Triple` class. Conceptually, it contains three elements of the same type. It could be implemented as a non-generic class.

Example (of a non-generic triple):

```

class Triple {
    private Object t1, t2, t3;
    public Triple(Object a1, Object a2, Object a3) {
        t1 = a1;
        t2 = a2;
        t3 = a3;
    }
    public void reset(Object a1, Object a2, Object a3) {
        t1 = a1;
        t2 = a2;
        t3 = a3;
    }
    public void setFirst(Object a1) {
        t1 = a1;
    }
    public void setSecond(Object a2) {
        t2 = a2;
    }
    public void setThird(Object a3) {
        t3 = a3;
    }
    public Object getFirst() {
        return a1;
    }
    public Object getSecond() {
        return a2;
    }
    public Object getThird() {
        return a3;
    }
    ...
}

```

A triple is expected to contain three elements of the same type, like three strings, or three dates, or three integers. It is usually not a triple of objects of different type, and its constructors enforce these semantics. In addition, a certain triple object will probably contain the same type of members during its entire lifetime. It will not contain strings today, and integers tomorrow. This, however, is not enforced in the implementation shown above, perhaps mistakenly so.

The point is that there is a correspondence between the types of the three fields and their type `Object` does not convey these semantics. This correspondence - all three fields are of the same type - can be expressed more precisely by a generic type.

Example (of a generic triple):

```

class Triple<T> {
    private T t1, t2, t3;
    public Triple(T a1, T a2, T a3) {
        t1 = a1;
        t2 = a2;
        t3 = a3;
    }
    public void reset(T a1, T a2, T a) {
        t1 = a1;
        t2 = a2;
        t3 = a3;
    }
    public void setFirst(T a1) {
        t1 = a1;
    }
    public void setSecond(T a2) {
        t2 = a2;
    }
    public void setThird(T a3) {
        t3 = a3;
    }
    public T getFirst() {
        return a1;
    }
    public T getSecond() {
        return a2;
    }
    public T getThird() {
        return a3;
    }
    ...
}

```

Now we would work with a `Triple<String>`, saying that all members are strings and will remain strings. We can still permit variations like in a `Triple<Number>`, where the members can be of different number types like `Long`, `Short` and `Integer`, and where a `Short` member can be replaced by a `Long` member or vice versa. We can even use `Triple<Object>`, where everything goes. The point is that the generification allows to be more specific and enforces homogeneity.

### Conclusion:

When a type uses a supertype in several places and there is a correspondence among the difference occurrences, then this is an indication that perhas the type should be generic.

Note, that the supertype in question need not be `Object`. The same principle applies to supertypes in general. Consider for instance an abstraction that uses character sequences in its implementation and refers to them through the supertype `CharSequence`. Such an abstraction is a candidate for a generic type.

Example (of a non-generic class using character sequences):

```

class CharacterStore {
    private CharSequence theChars;
    ...
    public CharacterProcessingClass(CharSequence s) { ... }
    public void set(CharSequence s) { ... }
    public CharSequence get() { ... }
    ...
}

```

The idea of this abstraction is: whatever the `get` method receives is stored and later returned by the `set` method. Again there is a correspondence between the argument type of the `set` method, the return type of the `get` method, and the type of the private field. If they are supposed to be of the same type then the abstraction could be more precisely expressed as a generic type.

Example (of a generic class using character sequences):

```

class CharacterStore<C extends CharSequence> {
    private C theChars;
    ...
    public CharacterStore(C s) { ... }
    public void set(C s) { ... }
    public C get() { ... }
    ...
}

```

This class primarily serves as a store of a character sequence and we can create different types of stores for different types of character sequences, such as a `CharacterStore<String>` or a `CharacterStore<StringBuilder>`.

If, however, the semantics of the class is different, then the class might be better defined as a non-generic type. For instance, the purpose might

be to provide a piece of functionality, such as checking for a suffix, instead of serving as a container. In that case it does not matter what type of character sequence is used and a generification would not make sense.

Example (of a non-generic class using character sequences):

```
class SuffixFinder {
private CharSequence theChars;
...
public CharacterProcessingClass(CharSequence s) { ... }
public boolean hasSuffix(CharSequence suffix) { ... }
}
```

In this case, the character sequence being examined could be a `CharBuffer` and the suffix to be searched for could be a `StringBuilder`, or vice versa. It would not matter. There is no correspondence implied between the types of the various character sequences being used in this abstraction. Under these circumstances, the generification does not provide any advantage.

Ultimately, it all depends on the intended semantics, whether a type should be generic or not. Some indicators were illustrated above: a close relationship to an existing generic type, correspondences among references of the same supertype, the need for distinct types generated from a generic type, and the need for enhanced type information. In practice, most classes are non-generic, because most classes are defined for one specific purpose and are used in one specific context. Those classes hardly ever profit from being generic.

LINK TO THIS

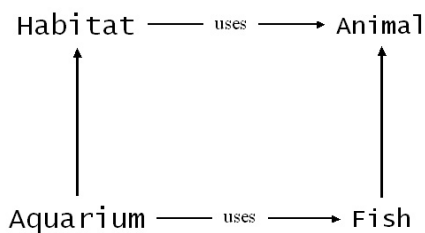
[Practicalities.FAQ201](#)

REFERENCES

## Do generics help designing parallel class hierarchies?

Yes.

Some hierarchies of types run in parallel in the sense that a supertype refers to another type and the subtype refers to a subtype of that other type. Here is an example, where the supertype `Habitat` refers to `Animal` and the subtype `Aquarium` refers to `Fish`.



Overriding methods in the subtype often have to perform a type check in this situation, like in the example below.

Example (of parallel type hierarchies leading to dynamic type check):

```
abstract class Habitat {
protected Collection theAnimals;
...
public void addInhabitant(Animal animal) {
theAnimals.add(animal);
}
}
class Aquarium extends Habitat {
...
public void addInhabitant(Animal fish) {
if (fish instanceof Fish)
theAnimals.add(fish);
else
throw new IllegalArgumentException(fish.toString());
}
}
Aquarium a = new Aquarium();
a.addInhabitant(new Cat()); // ClassCastException
```

In order to ensure that the aquarium only contains fish, the `addInhabitant` method performs an `instanceof` test. The test may fail at runtime with a `ClassCastException`. It would be nice if the `addInhabitant` method could be declared as taking a `Fish` argument; the `instanceof` test would be obsolete then. The problem is that a `addInhabitant(Fish)` method in the `Aquarium` class would be an overloading version of the `Habitat`'s `addInhabitant(Animal)` method rather than an overriding version thereof and this is neither intended nor correct. Hence, we cannot get rid of the `instanceof` test - unless we consider generics.

This kind of type relationship among parallel type hierarchies can be more elegantly expressed by means of generics. If the supertype `Habitat` were a generic type, then the subtype `Aquarium` would no longer need the type check. Here is a re-engineered version of the example above.

Example (same as above, re-engineered using generics):

```
abstract class Habitat<A extends Animal> {
    protected Collection<A> theAnimals;
    ...
    public void addInhabitant(A animal) {
        theAnimals.add(animal);
    }
}
class Aquarium extends Habitat<Fish> {
    ...
    public void addInhabitant(Fish fish) {
        // no test necessary
        theAnimals.add(fish);
    }
}

Aquarium a = new Aquarium();
a.addInhabitant(new Cat()); // error: illegal argument type
```

When the supertype is generic, then the subtype can derive from a certain instantiation of the supertype. The advantage is that overriding methods in the subtype can now be declared to take the intended type of argument rather than a supertype argument. In the example, the `Aquarium` is a `Habitat<Fish>`, which means that the `addInhabitant` method now takes a `Fish` argument instead of an `Animal` argument. This way, the `instanceof` test is no longer necessary and any attempt to add a non-`Fish` to the `Aquarium` will be detected at compile-time already.

Note, that the generic version of the type hierarchy has further advantages.

Example (of parallel type hierarchies):

```
abstract class Habitat {
    protected Collection theAnimals;
    ...
    public Habitat(Collection animals) {
        theAnimals = animals;
    }
}
class Aquarium extends Habitat {
    ...
    public Aquarium(Collection fish) {
        // no type check possible
        super(fish);
    }
}

ArrayList animals = new ArrayList();
animals.add(new Cat());
Aquarium a = new Aquarium(animals); // no error or exception here
```

In the `Aquarium` constructor there is no way to check whether the collection contains `Fish` or not. Compare this to the generic solution.

Example (of parallel type hierarchies using generics):

```
abstract class Habitat<A extends Animal> {
    protected Collection<A> theAnimals;
    ...
    public Habitat(Collection<A> animals) {
        theAnimals = animals;
    }
}
class Aquarium extends Habitat<Fish> {
    ...
    public Aquarium(Collection<Fish> fish) {
        // no type check necessary
        super(fish);
    }
}

ArrayList<Animal> animals = new ArrayList<Animal>();
animals.add(new Cat());
Aquarium a = new Aquarium(animals); // error: illegal argument type
```

In this generic version of the type hierarchy, the `Aquarium` constructor requires a `Collection<Fish>` as a constructor argument and this collection of fish can be passed along to the supertype's constructor because `Aquarium` extends the supertype's instantiation `Habitat<Fish>` whose constructor requires exactly that type of collection.

**Conclusion:** Type hierarchies that run in parallel are more easily and more reliably implemented by means of generics.

## When would I use an unbounded wildcard parameterized type instead of a bounded wildcard or concrete parameterized type?

### *When you need a reifiable type.*

Occasionally, an unbounded wildcard parameterized type is used because it is a so-called reifiable type and can be used in situations where non-reifiable types are not permitted.

- One of these situations are type checks (i.e., `cast` or `instanceof` expressions). Non-reifiable types (i.e., concrete or bounded wildcard parameterized type) are not permitted as the target type of a type check or lead to "unchecked" warnings.
- Another situation is the use of arrays. Non-reifiable types (i.e., concrete or bounded wildcard parameterized type) are not permitted as the component type of an array.

Depending on the situation, the unbounded wildcard parameterized type can substitute a concrete or bounded wildcard parameterized type in a type check or an array in order to avoid errors or warning.

Non-reifiable types (i.e., concrete or bounded wildcard parameterized type) are not permitted as the target type of a type check or lead to "unchecked" warnings. A typical situation, in which such a cast would be needed, is the implementation of methods such as the `equals` method, that take `Object` reference and where a cast down to the actual type must be performed.

Example (not recommended):

```
class Triple<T> {
    private T fst, snd, trd;
    ...
    public boolean equals(Object other) {
        ...
        Triple<T> otherTriple = (Triple<T>)other; // warning; unchecked cast
        return (this.fst.equals(otherTriple.fst)
            && this.snd.equals(otherTriple.snd)
            && this.trd.equals(otherTriple.trd));
    }
}
```

When we replace the cast to `Triple<T>` by a cast to `Triple<?>` the warning disappears, because unbounded wildcard parameterized type are permitted as target type of a cast without any warnings.

Example (implementation of `equals`):

```
class Triple<T> {
    private T fst, snd, trd;
    ...
    public boolean equals(Object other) {
        ...
        Triple<?> otherTriple = (Triple<?>)other;
        return (this.fst.equals(otherTriple.fst)
            && this.snd.equals(otherTriple.snd)
            && this.trd.equals(otherTriple.trd));
    }
}
```

Note, that replacing the concrete parameterized type by the wildcard parameterized type works in this example only because we need no write access to the fields of the referenced object referred and we need not invoke any methods. Remember, use of the object that a wildcard reference variable refers to is restricted. In other situations, use of a wildcard parameterized type might not be a viable solution, because full access to the referenced object is needed. (Such a situation can arise, for instance, when you implement the `clone` method of a generic class.)

Non-reifiable types (i.e., concrete or bounded wildcard parameterized type) are not permitted as the component type of an array. Here is an example:

Example (of illegal array type):

```
static void test() {
    Pair<Integer,Integer>[] arr = new Pair<Integer,Integer>[10]; // error
    arr[0] = new Pair<Integer,Integer>(0,0);
    arr[1] = new Pair<String,String>("", ""); // would fail with ArrayStoreException

    Pair<Integer,Integer> pair = arr[0];
    Integer i = pair.getFirst();
    pair.setSecond(i);
}
```

The concrete parameterized type `Pair<Integer,Integer>` is illegal. As a workaround one might consider using an array of the corresponding unbounded wildcard parameterized type.

Example (of array of unbounded wildcard parameterized type):

```
static void test() {
    Pair<?,?>[] arr = new Pair<?,?>[10];
    arr[0] = new Pair<Integer,Integer>(0,0);
    arr[1] = new Pair<String,String>("","");    // succeeds

    Pair<Integer,Integer> pair1 = arr[0];    // error
    Pair<?,?> pair2 = arr[0];    // ok

    Integer i = pair2.getFirst();    // error
    Object o = pair2.getFirst();    // ok

    pair2.setSecond(i);    // error
}
```

However, a `Pair<?,?>[]` is semantically different from the illegal `Pair<Integer,Integer>[]`. It is not homogenous, but contains a mix of arbitrary pair types. The compiler does not and cannot prevent that they contain different instantiations of the generic type. In the example, I can insert a pair of strings into what was initially supposed to be a pair of integers.

When we retrieve elements from the array we receive references of type `Pair<?,?>`. This is demonstrated in our example: we cannot assign the `Pair<?,?>` taken from the array to the more specific `Pair<Integer,Integer>`, that we really wanted to use.

Various operations on the `Pair<?,?>` are rejected as errors, because the wildcard type does not give access to all operations of the referenced object. In our example, invocation of the `set`-methods is rejected with error messages.

Depending on the situation, an array of a wildcard parameterized type may be a viable alternative to the illegal array of a concrete (or bounded wildcard) parameterized type. If full access to the referenced element is needed, this approach does not work and a better solution would be use of a collection instead of an array.

LINK TO THIS [Practicalities.FAQ202](#)

REFERENCES [What is a reifiable type?](#)  
[How can I avoid "unchecked cast" warnings?](#)  
[How can I work around the restriction that there are no arrays whose component type is a concrete parameterized type?](#)

---

## When would I use a wildcard parameterized type instead of a concrete parameterized type?

*Whenever you need the supertype of all or some instantiations of a generic type.*

There are two typical situations in which wildcard parameterized types are used because they act as supertype of all instantiations of a given generic type:

- relaxing a method signature to allow a broader range of argument or return types
- denoting a mix of instantiations of the same generic type

Details are discussed in the FAQ entries [Practicalities.FAQ301](#) and [Practicalities.FAQ006](#) listed in the reference section below.

LINK TO THIS [Practicalities.FAQ203](#)

REFERENCES [Which role do wildcards play in method signatures?](#)  
[How do I express a mixed sequence of instantiations of a given generic type?](#)

---

## When would I use a wildcard parameterized type with a lower bound?

*When a concrete parameterized type would be too restrictive.*

Consider a class hierarchy where a the topmost superclass implements an instantiation of the generic `Comparable` interface.

Example:

```
class Person implements Comparable<Person> {
    ...
}
class Student extends Person {
    ...
}
```

Note, the `Student` class does not and cannot implement `Comparable<Student>`, because it would be a subtype of two different instantiations of the same generic type then, and that is illegal (details [here](#)).

Consider also a method that tries to sort a sequence of subtype objects, such as a `List<Student>`.

Example:

```
class Utilities {
    public static <T extends Comparable<T>> void sort(List<T> list) {
        ...
    }
    ...
}
```

This `sort` method cannot be applied to a list of students.

Example:

```
List<Student> list = new ArrayList<Student>();
...
Utilities.sort(list); // error
```

The reason for the error message is that the compiler infers the type parameter of the `sort` method as `T:=Student` and that class `Student` is not `Comparable<Student>`. It is `Comparable<Person>`, but that does not meet the requirements imposed by the bound of the type parameter of method `sort`. It is required that `T` (i.e. `Student`) is `Comparable<T>` (i.e. `Comparable<Student>`), which in fact it is not.

In order to make the `sort` method applicable to a list of subtypes we would have to use a wildcard with a lower bound, like in the re-engineered version of the `sort` method below.

Example:

```
class Utilities {
    public static <T extends Comparable<? super T>> void sort(List<T> list) {
        ...
    }
    ...
}
```

Now, we can sort a list of students, because students are comparable to a supertype of `Student`, namely `Person`.

LINK TO THIS [Practicalities.FAQ204](#)

REFERENCES [Can a subclass implement another instantiation of a generic interface than any of its superclasses does?](#)

## How do I recover the actual type of the `this` object in a class hierarchy?

*With a `getThis()` helper method that returns the `this` object via a reference of the exact type.*

Sometimes we need to define a hierarchy of classes whose root class has a field of a super type and is supposed to refer to different subtypes in each of the subclasses that inherit the field. Here is an example of such a situation. It is a generic `Node` class.

Example (of a class with a type mismatch - does not compile):

```
public abstract class Node <N extends Node<N>> {
    private final List<N> children = new ArrayList<N>();
    private final N parent;

    protected Node(N parent) {
        this.parent = parent;
        parent.children.add(this); // error: incompatible types
    }
    public N getParent() {
        return parent;
    }
    public List<N> getChildren() {
        return children;
    }
}

public class SpecialNode extends Node<SpecialNode> {
    public SpecialNode(SpecialNode parent) {
        super(parent);
    }
}
```

The idea of this class design is: in the subclass `SpecialNode` the list of children will contain `SpecialNode` objects and in another subclass of `Node` the child list will contain that other subtype of `Node`. Each node adds itself to the child list at construction time. The debatable aspect in the design is the attempt to achieve this addition to the child list in the superclass constructor so that the subclass constructors can simply invoke the superclass constructor and thereby ensure the addition of this node to the child list.



The class designer overlooked that in the `Node` superclass the child list is of type `List<N>`, where `N` is a subtype of `Node`. Note, that the list is NOT of type `List<Node>`. When in the superclass constructor the `this` object is added to the child list the compiler detects a type mismatch and issues an error message. This is because the `this` object is of type `Node`, but the child list is declared to contain objects of type `N`, which is an unknown subtype of `Node`.

There are at least three different ways of solving the problem.

- Declare the child list as a `List<Node>` and add the `this` object in the superclass constructor.
- Declare the child list as a `List<N>` and add the `this` object in the subclass constructor.
- Declare the child list as a `List<N>`, recover the `this` object's actual type, and add the `this` object in the superclass constructor.

Below you find the source code for each of these solutions.

---

### Declare the child list as a `List<Node>` and add the `this` object in the superclass constructor.

If we want to add each node to the child list in the superclass constructor then we need to declare the child list as a `List<Node>`, because in the superclass constructor the `this` object is of type `Node`. The `Node` superclass is supposed to be used in a way that the `Node` reference will refer to an object of type `N`, but this is just a convention and not reflected in the types being used. Type-wise the `this` object is just a `Node` - at least in the context of the superclass.

Example (problem solved using a list of supertypes):

```
public abstract class Node <N extends Node<N>> {
    private final List<Node<?>> children = new ArrayList<Node<?>>();
    private final N parent;

    protected Node(N parent) {
        this.parent = parent;
        parent.children.add(this); // fine
    }
    public N getParent() {
        return parent;
    }
    public List<Node<?>> getChildren() {
        return children;
    }
}

public class SpecialNode extends Node<SpecialNode> {
    public SpecialNode(SpecialNode parent) {
        super(parent);
    }
}
```

---

### Declare the child list as a `List<N>` and add the `this` object in the subclass constructor.

Our type mismatch problem would be solved if refrained from adding the `this` object in the superclass constructor, but defer its addition to the subclass constructor instead. In the context of the subclass constructor the exact type of the `this` object is known and there would be no type mismatch any longer.

Example (problem solved by adding the `this` object in the subtype constructor):

```
public abstract class Node <N extends Node<N>> {
    protected final List<N> children = new ArrayList<N>();
    private final N parent;

    protected Node(N parent) {
        this.parent = parent;
    }
    public N getParent() {
        return parent;
    }
    public List<N> getChildren() {
        return children;
    }
}

public class SpecialNode extends Node<SpecialNode> {
    public SpecialNode(SpecialNode parent) {
        super(parent);
        parent.children.add(this); // fine
    }
}
```

---

**Declare the child list as a `List<N>`, recover the `this` object's actual type, and add the `this` object in the superclass constructor.**

The problem can alternatively be solved by means of an abstract helper method that each of the subclasses implements. The purpose of the helper method is recovering the `this` object's actual type.

Example (problem solved by recovering the `this` object's actual type):

```
public abstract class Node <N extends Node<N>> {
    private final List<N> children = new ArrayList<N>();
    private final N parent;

    protected abstract N getThis();

    protected Node(N parent) {
        this.parent = parent;
        parent.children.add(getThis()); // fine
    }
    public N getParent() {
        return parent;
    }
    public List<N> getChildren() {
        return children;
    }
}

public class SpecialNode extends Node<SpecialNode> {
    public SpecialNode(SpecialNode parent) {
        super(parent);
    }
    protected SpecialNode getThis() {
        return this;
    }
}
```

We added an abstract helper method `getThis()` that returns the `this` object with its exact type information. Each implementation of the `getThis()` method in one of the `Node` subtypes returns an object of the specific subtype `N`.

Usually, one would try to recover type information by means of a cast, but in this case the target type of the cast would be the unknown type `N`. Following this line of logic one might have tried this unsafe solution:

Example (problem solved by recovering the `this` object's actual type - not recommended):

```
public abstract class Node <N extends Node<N>> {
    ...
    protected Node(N parent) {
        this.parent = parent;
        parent.children.add((N)this); // warning: unchecked cast
    }
    ...
}
```

Casts whose target type is a type parameter cannot be verified at runtime and lead to an unchecked warning. This unsafe cast introduces the potential for unexpected `ClassCastException`s and is best avoided. The exact type information of the object referred to by the `this` reference is best recovered by means of overriding a `getThis()` helper method.

LINK TO THIS [Practicalities.FAQ205](#)

REFERENCES [What is the "getThis" trick?](#)

---

## What is the "getThis" trick?

*A way to recover the type of the `this` object in a class hierarchy.*

The "*getThis trick*" was first published by Heinz Kabutz in [Issue 123](#) of his Java Specialists' Newsletter in March 2006 and later appeared in the book [Java Generics and Collections](#) by Maurice Naftalin and Philip Wadler, who coined the term "*getThis*" trick. It is a way to recover the type of the `this` object - a recovery of type information that is sometimes needed in class hierarchies with a self-referential generic supertype.

Examples of self-referential generic types are

- abstract class `Enum<E extends Enum<E>>` in the `java.lang` package of the JDK, or
- abstract class `Node <N extends Node<N>>` from entry [FAQ205](#) above, or
- abstract class `TaxPayer<P extends TaxPayer<P>>` in the original example discussed by Heinz Kabutz.

Self-referential generic types are often - though not necessarily - used to express in a supertype that its subtypes depend on themselves. For

instance, all enumeration types are subtypes of class `Enum`. The idea is that an enumeration type `Color` extends `Enum<Color>`, an enumeration type `TimeUnit` extends `Enum<TimeUnit>`, and so on. Similarly in the example discussed in entry [FAQ205](#): each node type extends class `Node` parameterized on its own type, e.g. class `SpecialNode` extends `Node<SpecialNode>`. Heinz Kabutz's example uses the same idea: there is a class `Employee` that extends `TaxPayer<Employee>` and a class `Company` that extends `TaxPayer<Company>`.

Let us consider an arbitrary self-referential generic type `SelfReferentialType<T>` extends `SelfReferentialType<T><>`. In its implementation it may be necessary to pass the `this` reference to a method that expects an argument of type `T`, the type parameter. The attempt results in a compile-time error message, as illustrated below:

```
public abstract class SelfReferentialType<T> extends SelfReferentialType<T><> {
    private SomeOtherType<T> ref;
    public void aMethod() { ref.m(this); } // error: incompatible types
}
public interface SomeOtherType<E> {
    void m(E arg);
}
```

The problem is that the `this` reference is of type `SelfReferentialType<T>`, while the method `m` expects an argument of type `T`, which is a subtype of type `SelfReferentialType<T>`. Since we must not supply supertype objects where subtype objects are asked for, the compiler rightly complains. Hence the compiler is right.

However, we as developers know that conceptually all subtypes of type `SelfReferentialType` are subtypes of type `SelfReferentialType` parameterized on their own type. As a result, the type of the `this` reference is the type that the type parameter `T` stands for. This is illustrated below:

```
public class Subtype extends SelfReferentialType<Subtype> { ... }
```

When the inherited `aMethod` is invoked on a `Subtype` object, then the `this` reference refers to an object of type `Subtype` and a `Method` expects a argument of type `T:=Subtype`. This perfect match is true for all subtypes. Consequently, we wished that the compiler would accept the method invocation as is. Naturally, the compiler does not share our knowledge regarding the intended structure of the class hierarchy and there are no language means to express that each `Subtype` extends `SelfReferentialType<Subtype>`. Hence we need a work-around - and this is what the "getThis" trick provides.

The "getThis" trick provides a way to recover the exact type of the `this` reference. It involves an abstract method in the self-referential supertype that all subtypes must override. The method is typically named `getThis`. The intended implementation of the method in the subtype is `getThis()` { return this; }, as illustrated below:

```
public abstract class SelfReferentialType<T> extends SelfReferentialType<T><> {
    private SomeOtherType<T> ref;
    protected abstract T getThis();
    public void aMethod() { ref.m(getThis()); } // fine
}
public interface SomeOtherType<E> {
    void m(E arg);
}
public class Subtype extends SelfReferentialType<Subtype> {
    protected subtype getThis() { return this; }
}
```

As we discussed in entry [FAQ205](#), the "getThis" trick is not the only conceivable work-around.

LINK TO THIS [Practicalities.FAQ206](#)

REFERENCES [How do I recover the actual type of the this object in a class hierarchy?](#)

## How do I recover the element type of a container?

*By having the container carry the element type as a type token.*

Suppose that you are defining a pair of related interfaces which need to be implemented in pairs:

Example (of a pair of related interfaces):

```
interface Contained {}

interface Container<T> extends Contained {
    void add(T element);
    List<T> elements();
}
```

Example (of implementations of the related interfaces):

```
class MyContained implements Contained {
    private final String name;
    public MyContained(String name) {this.name = name;}
    public @Override String toString() {return name;}
}
```

```

}
class MyContainer implements Container<MyContained> {
    private final List<MyContained> _elements = new ArrayList<MyContained>();
    public void add(MyContained element) {_elements.add(element);}
    public List<MyContained> elements() {return _elements;}
}

```

Given these interfaces you need to write generic code which works on any instance of these interfaces.

Example (of generic code using the pair of interfaces):

```

class MetaContainer {
    private Container<? extends Contained> container;
    public void setContainer(Container<? extends Contained> container) {
        this.container = container;
    }
    public void add(Contained element) {
        container.add(element); // error
    }
    public List<? extends Contained> elements() {return container.elements();}
}

```

---

```

error: add(capture#143 of ? extends Contained) in Container<capture#143 of ? extends Contained> cannot be
applied to Contained
        container.add(element);
                   ^

```

The `MetaContainer` needs to handle an unknown parameterization of the generic `Container` class. For this reason it holds a reference of type `Container<? extends Contained>`. Problems arise when the container's `add()` method is invoked. Since the container's type is a wildcard parameterization of class `Container` the compiler does not know the container's exact type and cannot check whether the type of the element to be added is acceptable and the element can safely be added to the container. As the compiler cannot ensure type safety, it issues an error message. The problem is not at all surprising: wildcard parameterizations give only restricted access to the concrete parameterization they refer to (see entry [GenericTypes.FAQ304](#) for details).

In order to solve the problem, we would have to retrieve the container's exact type and in particular its element type. However, this is not possible statically at compile-time. A viable work-around is adding to the `Container` class a method that returns a type token that represents the container's element type so that we can retrieve the element type dynamically at run-time.

Example (of container with element type):

```

interface Container<T extends Contained> {
    void add(T element);
    List<T> elements();
    Class<T> getElementType();
}

class MyContainer implements Container<MyContained> {
    private final List<MyContained> _elements = new ArrayList<MyContained>();
    public void add(MyContained element) {_elements.add(element);}
    public List<MyContained> elements() {return _elements;}
    public Class<MyContained> getElementType() {return MyContained.class;}
}

```

The `MetaContainer` can then retrieve the element type from the container by means of the container's `getElementType()` method..

Example (first attempt of re-engineering the meta container):

```

class MetaContainer {
    private Container<? extends Contained> container;
    public void setContainer(Container<? extends Contained> container) {
        this.container = container;
    }
    public void add(Contained element) {
        container.add(container.getElementType().cast(element)); // error
    }
    public List<? extends Contained> elements() {return container.elements();}
}

```

---

```

error: add(capture#840 of ? extends Contained) in Container<capture#840 of ? extends Contained> cannot be
applied to (Contained)
        container.add(container.getElementType().cast(element));
                   ^

```

Unfortunately the container is still of a type that is a wildcard parameterization and we still suffer from the restrictions that wildcard parameterizations come with: we still cannot invoke the container's `add()` method. However, there is a common technique for working around this kind of restriction: using a generic helper method (see [Practicalities.FAQ304](#) for details).

Example (successfully re-engineered meta container):

```
class MetaContainer {
    private Container<? extends Contained> container;
    public void setContainer(Container<? extends Contained> container) {
        this.container = container;
    }
    public void add(Contained element) {
        _add(container, element);
    }
    private static <T extends Contained> void _add(Container<T> container, Contained element){
        container.add(container.getElementType().cast(element));
    }
    public List<? extends Contained> elements() {return container.elements();}
}
```

This programming technique relies on the fact that the compiler performs type argument inference when a generic method is invoked (see [Technicalities.FAQ401](#) for details). It means that the type of the `container` argument in the helper method `_add()` is not a wildcard parameterization, but a concrete parameterization for an unknown type that the compiler infers when the method is invoked. The key point is that the container is no longer of a wildcard type and we may eventually invoke its `add()` method.

LINK TO THIS [Practicalities.FAQ207](#)

REFERENCES [Which methods and fields are accessible/inaccessible through a reference variable of a wildcard parameterized type?](#)  
[How do I implement a method that takes a wildcard argument?](#)  
[What is the capture of a wildcard?](#)  
[What is type argument inference?](#)

## What is the "getTypeArgument" trick?

*A technique for recovering the type argument from a wildcard parameterized type at run-time.*

A reference of a wildcard type typically refers to a concrete parameterization of the corresponding generic type, e.g. a `List<?>` refers to a `LinkedList<String>`. Yet it is impossible to retrieve the concrete parameterization's type argument from the wildcard type. The "getTypeArgument" trick solves this problem and enables you to retrieve the type argument dynamically at run-time. The previous FAQ entry demonstrates an application of this technique (see [Practicalities.FAQ207](#)).

Consider a generic interface and a type that implements the interface.

Example (of generic interface and implementing class):

```
interface GenericType<T> {
    void method(T arg);
}
class ConcreteType implements GenericType<TypeArgument> {
    public void method(TypeArgument arg) {...}
}
```

Note that the interface has a method that takes the type variable as an argument.

When you later use a wildcard parameterization of the generic interface and need to invoke a method that takes the type variable as an argument, the compiler will complain. This is because wildcard parameterizations do not give full access to all methods (see entry [GenericTypes.FAQ304](#) for details).

Example (of using a wildcard parameterization of the generic interface):

```
class GenericUsage {
    private GenericType<?> reference;
    public void method(Object arg) {
        reference.method(arg); // error
    }
}
```

---

```
error: method(capture#143 of ? extends TypeArgument) in GenericType<capture#143 of ? extends TypeArgument>
cannot be applied to TypeArgument
    reference.method(arg);
                   ^
```

In order to solve the problem, you add a method to the implementation of the generic interface that return a type token . The type token represents the type argument of the parameterization of the generic interface that the class implements. This way you can later retrieve the type argument dynamically at run-time.

Example (of container with element type):

```

interface GenericType<T> {
    void method(T arg);
    Class<T> getTypeArgument();
}

class ConcreteType implements GenericType<TypeArgument> {
    public void method(TypeArgument arg) {...}
    public Class<TypeArgument> getTypeArgument() {return TypeArgument.class;}
}

```

Using the `getTypeArgument()` method you can then retrieve the type argument even from a wildcard parameterization.

Example (of retrieving the type argument via the "getTypeArgument" trick):

```

class GenericUsage {
    private GenericType<?> reference;
    public void method(Object arg) {
        _helper(reference, arg);
    }
    private static <T> void _helper(GenericType<T> reference, Object arg){
        reference.method(reference.getTypeArgument().cast(arg));
    }
}

```

Note that the generic helper method `_helper()` is needed because otherwise the interface's method would still be invoked through a reference of a wildcard type and you would still suffer from the restrictions that wildcard parameterizations come with. Using a generic helper method is a common technique for working around this kind of restriction (see [Practicalities.FAQ304](#) for details).

The work-around relies on the fact that the compiler performs type argument inference when a generic method is invoked (see [Technicalities.FAQ401](#) for details). It means that the type of the `reference` argument in the helper method is not a wildcard parameterization, but a concrete parameterization for an unknown type that the compiler infers when the method is invoked. The key point is that the reference is no longer of a wildcard type and we may eventually invoke its method.

The key point of the "getTypeArgument" trick is making available the type argument as a type token (typically by providing a method such as `getTypeArgument()`) so that you can retrieve the type argument at run-time even in situations where the static type information does not provide information about the type argument.

**LINK TO THIS**            [Practicalities.FAQ208](#)

**REFERENCES**

- [Which methods and fields are accessible/inaccessible through a reference variable of a wildcard parameterized type?](#)
- [How do I implement a method that takes a wildcard argument?](#)
- [What is the capture of a wildcard?](#)
- [What is type argument inference?](#)
- [How do I recover the element type of a container?](#)

## Designing Generic Methods

### Why does the compiler sometimes issue an unchecked warning when I invoke a "varargs" method?

*Because you pass in a variable argument list of reifiable types.*

When you invoke a method with a variable argument list (also called *varargs*) you will occasionally find that the compiler issues an unchecked warning. Here is an example:

Example (of a varargs method and its invocation):

```

public static <E> void addAll(List<E> list, E... array) {
    for (E element : array) list.add(element);
}

public static void main(String[] args) {
    addAll(new ArrayList<String>(), // fine
        "Leonardo da Vinci",
        "Vasco de Gama"
    );
    addAll(new ArrayList<Pair<String,String>>(), // unchecked warning
        new Pair<String,String>("Leonardo","da Vinci"),
        new Pair<String,String>("Vasco","de Gama")
    );
}

```

---

```
warning: [unchecked] unchecked generic array creation of type Pair<String,String>[] for varargs parameter
addAll(new ArrayList<Pair<String,String>>(),
      ^
```

The first invocation is fine, but the second invocation is flagged with an unchecked warning. This warning is confusing because there is no array creation expression anywhere in the source code. In order to understand, what the compiler complains about you need to keep in mind two things:

- Variable argument lists are translated by the compiler into an array.
- Creation of arrays with a non-reifiable component type is not permitted.

In the example above the compiler translates the varargs parameter in the method definition into an array parameter. Basically the method declaration is translated to the following:

Example (of varargs method after translation):

```
public static <E> void addAll(List<E> list, E[] array) {
    for (E element : array) list.add(element);
}
```

When the method is invoked, the compiler automatically takes the variable number of arguments, creates an array into which it stuffs the arguments, and passes the array to the method. The method invocations in the example above are translated to the following:

Example (of invocation of varargs method after translation):

```
public static void main(String[] args) {
    addAll(new ArrayList<String>(), // fine
          new String[] {
              "Leonardo da Vinci",
              "Vasco de Gama"
          }
    );
    addAll(new ArrayList<Pair<String,String>>(), // unchecked warning
          new Pair<String,String>[] {
              new Pair<String,String>("Leonardo","da Vinci"),
              new Pair<String,String>("Vasco","de Gama")
          }
    );
}
```

As you can see, the compiler creates a `String[]` for the first invocation and a `Pair<String,String>[]` for the second invocation. Creating a `String[]` is fine, but creating a `Pair<String,String>[]` is not permitted. `Pair<String,String>` is not a reifiable type, that is, it loses information as a result of type erasure and is at runtime represented as the raw type `Pair` instead of the exact type `Pair<String,String>`. The loss of information leads to problems with arrays of such non-reifiable component types. The reasons are illustrated in FAQ entry [ParameterizedTypes.FAQ104](#); as usual it has to do with type safety issues.

If you were trying to create such an array of type `Pair<String,String>[]` yourself, the compiler would reject the `new`-expression with an error message. But since it is the compiler itself that creates such a forbidden array, it chooses to do so despite of the type safety issues and gives you an unchecked warning to alert you to potential safety hazards.

---

You might wonder why the unchecked warning is needed and what peril it tries to warn about. The example above is perfectly type-safe, because in the method implementation the array is only read and nothing is stored in the array. However, if a method would store something in the array it could attempt to store an alien object in the array, like putting a `Pair<Long,Long>` into a `Pair<String,String>[]`. Neither the compiler nor the runtime system could prevent it.

Example (of corrupting the implicitly created varargs array; not recommended):

```
Pair<String,String>[] method(Pair<String,String>... lists) {
    Object[] objs = lists;
    objs[0] = new Pair<String,String>("x","y");
    objs[1] = new Pair<Long,Long>(0L,0L); // corruption !!!
    return lists;
}
public static void main(String[] args) {
    Pair<String,String>[] result
        = method(new Pair<String,String>("Vasco","da Gama"), // unchecked warning
                new Pair<String,String>("Leonard","da Vinci"));
    for (Pair<String,String> p : result) {
        String s = p.getFirst(); // ClassCastException
    }
}
```

The implicitly created array of `String` pairs is accessed through a reference variable of type `Object[]`. This way anything can be stored in the array; neither the compiler nor the runtime system can prevent that a `Pair<Long,Long>` is stored in the array of `Pair<String,String>`. What the compiler *can* do is warning you when the implicit varargs array is created. If you ignore the warning you will get an unexpected `ClassCastException` later at runtime.

---

Here is another example that illustrates the potential danger of ignoring the warning issued regarding array construction in conjunction with variable argument lists.

Example (of a varargs method and its invocation):

```
public final class Test {
    static <T> T[] method_1(T t1, T t2) {
        return method_2(t1, t2); // unchecked warning
    }
    static <T> T[] method_2(T... args) {
        return args;
    }
    public static void main(String... args) {
        String[] strings = method_1("bad", "karma"); // ClassCastException
    }
}
```

---

```
warning: [unchecked] unchecked generic array creation of type T[] for varargs parameter
        return method_2(t1, t2);
                ^
```

In this example the first method calls a second method and the second method takes a variable argument list. In order to invoke the varargs method the compiler creates an array and passes it to the method. In this example the array to be created is an array of type `T[]`, that is, an array whose component type is a type parameter. Creation of such arrays is prohibited in Java and you would receive an error message if you tried to create such an array yourself; see [TypeParameters.FAQ202](#) for details.

As in the previous example, the array's component type is non-reifiable and due to type erasure the compiler does not create a `T[]`, but an `Object[]` instead. Here is what the compiler generates:

Example (same as above, after translation by type erasure):

```
public final class Test {
    static Object[] method_1(Object t1, Object t2) {
        return method_2(new Object[] {t1, t2}); // unchecked warning
    }
    static Object[] method_2(Object[] args) {
        return args;
    }
    public static void main(String[] args) {
        String[] strings = (String[])method_1("bad", "karma"); // ClassCastException
    }
}
```

The unchecked warning is issued to alert you to the potential risk of type safety violations and unexpected `ClassCastException`s. In the example, you would observe a `ClassCastException` in the `main()` method where two strings are passed to the first method. At runtime, the two strings are stuffed into an `Object[]`; note, *not* a `String[]`. The second method accepts the `Object[]` as an argument, because after type erasure `Object[]` is its declared parameter type. Consequently, the second method returns an `Object[]`, *not* a `String[]`, which is passed along as the first method's return value. Eventually, the compiler-generated cast in the `main()` method fails, because the return value of the first method is an `Object[]` and no `String[]`.

Again, the problem is that calling the varargs method requires creation of an array with a non-reifiable component type. In the first example, the array in question was a `Pair<String,String>[]`; in the second example, it was a `T[]`. Both are prohibited in Java because they can lead to type safety problems.

---

**Conclusion:** It is probably best to avoid providing objects of non-reifiable types where a variable argument list is expected. You will always receive an unchecked warning and unless you know exactly what the invoked method does you can never be sure that the invocation is type-safe.

LINK TO THIS [Practicalities.FAQ300](#)

REFERENCES [What does type-safety mean?](#)  
[What is a reifiable type?](#)  
[Can I create an array whose component type is a concrete parameterized type?](#)  
[Can I create an array whose component type is a wildcard parameterized type?](#)  
[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)  
[Can I create an array whose component type is a type parameter?](#)

---

## What is a "varargs" warning?

*A warning that the compiler issues for the definition of certain methods with a variable argument list.*



Certain methods with a variable arguments list (called a *varargs method*) lead to unchecked warnings when they are invoked. This can occur if the declared type of the variable argument is non-reifiable, e.g. if it is a parameterized type or a type variable. Since Java 7 the compiler does not only give an unchecked warning when such a method is invoked, but also issues a warning for the definition of such a method. In order to distinguish between the warning issued for the definition of a debatable varargs method and the warning issued at the call site of such a method we will refer to the warning at the definition site as a *varargs warning*.

Here is an example:

Example (of a varargs warning):

```
public static <E> void addAll(List<E> list, E... array) { // varargs warning
    for (E element : array) list.add(element);
}

public static void main(String[] args) {
    addAll(new ArrayList<Pair<String,String>>(), // unchecked warning
           new Pair<String,String>("Leonardo","da Vinci"),
           new Pair<String,String>("Vasco","de Gama"));
};
```

---

```
warning: [unchecked] Possible heap pollution from parameterized vararg type E
    public static <E> void addAll(List<E> list, E... array) {
                                   ^
```

```
warning: [unchecked] unchecked generic array creation for varargs parameter of type Pair<String,String>[]
    addAll(new ArrayList<Pair<String,String>>(),
           ^
```

The `addAll()` method has a variable argument list `E...`. The type of the variable argument is `E` which is a type variable. When the `addAll()` method is invoked then the type variable `E` is replaced by the parameterized type `Pair<String,String>` in the example above, which leads to an unchecked warning. The details regarding this unchecked warning are explained in [Practicalities.FAQ300](#).

In order to alert the provider of the `addAll()` method (rather than its caller) to the trouble the method might later cause on invocation, the compiler gives a varargs warning for the method definition. This warning was introduced in Java 7.

The reason for the additional warning is that the caller of a varargs method cannot do anything about the unchecked warning. At best he can blindly suppress the unchecked warning with a `@SuppressWarnings("unchecked")` annotation, which is hazardous because the caller cannot know whether the unchecked warning is justified or not. Only the method's provider can judge whether the unchecked warning can safely be ignored or whether it will lead to subsequent errors due to heap pollution (see [Technicalities.FAQ050](#)). For this reason the provider of a varargs method is responsible for deciding whether the unchecked warning on invocation of the method can be ignored or not.

With a varargs warning the compiler tries to tell the provider of a varargs method: invocation of your method can lead to type safety issues and subsequent errors in form of unexpected `ClassCastException`s exceptions (collectively called *heap pollution*).

LINK TO THIS [Practicalities.FAQ300A](#)

REFERENCES

- [Why does the compiler sometimes issue an unchecked warning when I invoke a "varargs" method?](#)
- [What is a reifiable type?](#)
- [What is heap pollution?](#)
- [When does heap pollution occur?](#)
- [How can I suppress a "varargs" warning?](#)
- [What is the SuppressWarnings annotation?](#)

---

## How can I suppress a "varargs" warning?

*By using a `@SafeVarargs` annotation.*

A varargs warning can be suppressed using the `@SafeVarargs` annotation. When we use this annotation on a method with a variable argument list the compiler will not only suppress the varargs warning for the method definition, but also the unchecked warnings for the method invocations. Here is an example, first without the annotation:

Example (of a varargs warning):

```
public static <E> void addAll(List<E> list, E... array) { // varargs warning
    for (E element : array) list.add(element);
}

public static void main(String[] args) {
    addAll(new ArrayList<Pair<String,String>>(), // unchecked warning
           new Pair<String,String>("Leonardo","da Vinci"),
           new Pair<String,String>("Vasco","de Gama"));
};
```

```
warning: [unchecked] Possible heap pollution from parameterized vararg type E
public static <E> void addAll(List<E> list, E... array) {

warning: [unchecked] unchecked generic array creation for varargs parameter of type Pair<String,String>[]
addAll(new ArrayList<Pair<String,String>>(),
    ^
```

Here is the same example, this time with the annotation:

Example (of a suppressed varargs warning):

```
@SafeVarargs
public static <E> void addAll(List<E> list, E... array) { // fine
    for (E element : array) list.add(element);
}

public static void main(String[] args) {
    addAll(new ArrayList<Pair<String,String>>(), // fine
        new Pair<String,String>("Leonardo","da Vinci"),
        new Pair<String,String>("Vasco","de Gama")
    );
}
```

The `@SafeVarargs` annotation for the `addAll()` method eliminates both warnings.

As usual, you must not suppress warnings unless you are absolutely sure that they can safely be ignored. See [Practicalities.FAQ300C](#) for details on suppressing the varargs warnings.

LINK TO THIS [Practicalities.FAQ300B](#)

REFERENCES [What is the SuppressWarnings annotation?](#)  
[Why does the compiler sometimes issue an unchecked warning when I invoke a "varargs" method?](#)  
[What is a "varargs" warning?](#)  
[When should I refrain from suppressing a "varargs" warning?](#)

## When should I refrain from suppressing a "varargs" warning?

*When the varargs method in question can lead to heap pollution.*

Suppressing a warning is always hazardous and should only be attempted when the warning can with certainty be considered harmless and no heap pollution will ever occur. In all other situations you should refrain from suppressing any warnings.

Regarding suppression of a varargs warning: The provider of a varargs method may only suppress the warning if

- the varargs method does not add any elements to the array that the compiler creates for the variable argument list, or
- if the method adds an element to the array that the compiler creates for the variable argument list, the element must be type-compatible to the array's component type.

Example (of a harmless varargs method):

```
@SafeVarargs
public static <E> void addAll(List<E> list, E... array) {
    for (E element : array) list.add(element);
}

public static void main(String... args) {
    addAll(new ArrayList<Pair<String,String>>(),
        new Pair<String,String>("Leonardo","da Vinci"),
        new Pair<String,String>("Vasco","de Gama")
    );
}
```

The `addAll()` method only reads the array that the compiler created for the variable argument `E... array`. No heap pollution can occur; this method is harmless; the varargs warning can be ignored and therefore safely suppressed.

Example (of an incorrect and harmful varargs method):

```
public static Pair<String,String>[] modify(Pair<String,String>... lists) { // varargs warning
    Object[] objs = lists;
    objs[0] = new Pair<String,String>("x","y");
    objs[1] = new Pair<Long,Long>(0L,0L); // corruption !!!
    return lists;
}

public static void main(String... args) {
```

```

Pair<String,String>[] result
    = modify(new Pair<String,String>("Vasco","da Gama"), // unchecked warning
             new Pair<String,String>("Leonard","da Vinci"));
for (Pair<String,String> p : result) {
    String s = p.getFirst(); // ClassCastException
}

```

---

```

warning: [unchecked] Possible heap pollution from parameterized vararg type Pair<String,String>
private static Pair<String,String>[] modify(Pair<String,String>... lists) {
    ^

```

```

warning: [unchecked] unchecked generic array creation for varargs parameter of type Pair<String,String>[]
    = modify(new Pair<String,String>("Vasco","da Gama"),
             ^

```

The method `modify()` is plain wrong and should be corrected. It adds a `Pair<Long,Long>` to an array that is supposed to contain only elements of type `Pair<String,String>` and as a result the heap is polluted. The compiler issues a warning for the method definition as such, but does not flag the offending assignment as an error. The invocation of method `modify()` also leads to a warning. If all these warnings are ignored, an unexpected `ClassCastException` can occur.

While the varargs method in the example above is blatantly wrong, the situation can be far more subtle. Here is an example:

Example (of another incorrect and harmful varargs method):

```

public static <T> T[] method_1(T t1, T t2) {
    return method_2(t1, t2); // unchecked warning
}

public static <T> T[] method_2(T... args) { // varargs warning
    return args;
}

public static void main(String... args) {
    String[] strings = method_2("bad", "karma"); // fine
    strings = method_1("bad", "karma"); // ClassCastException
}

```

---

```

warning: [unchecked] unchecked generic array creation for varargs parameter of type T[]
    return method_2(t1, t2);
    ^

```

```

warning: [unchecked] Possible heap pollution from parameterized vararg type T
static <T> T[] method_2( T... args) { // varargs warning
    ^

```

Method `method_2()` is a generic method and has a variable argument list of type `T...`, where `T` is the type variable. As long as the varargs method is directly called, nothing bad will happen; the compiler infers that `T` is `String` in our example and returns an array of type `String[]`.

If the varargs method is called from another generic method such as `method_1()`, then the compiler will pass two arguments of type `Object` as arguments to method `method_2()` due to type erasure. It will then infer that `T` is `Object` in our example and returns an array of type `Object[]`, which subsequently leads to an unexpected `ClassCastException`. In this situation the question is: who is to blame? Is the varargs method incorrect, or is it incorrectly used? It is debatable whether it is a good idea to provide a method such as `method_2()` where a type variable appears in the variable argument list. In any case, suppressing the varargs warning is not advisable because this method can lead to heap pollution as demonstrated above.

In general, it is very difficult to decide whether the varargs warning can safely be suppressed. Whenever a non-reifiable type appears in the variable argument list, an array with a non-reifiable component type is created by the compiler. This is always hazardous. As soon as this array becomes accessible, heap pollution can occur. As a consequence, **you can only safely suppress a varargs warning if you can make sure that the automatically created array with a non-reifiable component type (or any copy thereof) never becomes accessible for modification.**

Here is an example of another unsafe varargs method:

Example (of another incorrect and harmful varargs method):

```

public class SomeClass<E> {
    private Pair<E,E>[] pairs;

    public SomeClass(Pair<E,E>... args) { // varargs warning
        pairs = args;
    }
    public Pair<E,E>[] getPairs() {
        List<Pair<E,E>> tmp = new ArrayList<Pair<E,E>>();
        for (Pair<E,E> p : pairs)

```

```

        tmp.add(p.clone());
    }
    return tmp.toArray(pairs);
}

... more methods ...
}

public static void main(String... args) {
    SomeClass<String> test = new SomeClass<String>(new Pair<String,String>("bad", "karma")); // unchecked
warning
    Pair<?,?>[] tmp = test.getPairs();
    tmp[0] = Pair.makePair(0L,0L);
    String s = test.pairs[0].getFirst(); // ClassCastException
}

```

---

```

warning: [unchecked] Possible heap pollution from parameterized vararg type Pair<E,E>
    public SomeClass(Pair<E,E>... args) {
        ^

```

```

warning: [unchecked] unchecked generic array creation for varargs parameter of type Pair<String,String>[]
    SomeClass<String> test = new SomeClass<String>(new Pair<String,String>("bad", "karma"));
        ^

```

The varargs method in question is the constructor of class `SomeClass`; it stores the automatically created array with a non-reifiable component type `Pair<E,E>` in a private field. Any modification of this array can create heap pollution. Even if the class itself does not modify the array in any of its methods, matters can go wrong. In the example, the `getPairs()` method creates a deep copy of the array and passes the copy to its caller. As soon as someone gets hold of an array with a non-reifiable component type (like the copy of the `pairs` field in the example), illegal elements can be added to the array without any error or warning from the compiler. The heap pollution and the resulting unexpected `ClassCastException` is shown in the `main()` method. Only if the automatically created array were confined to the class and the class were guaranteed to use the array sensibly in all situations, then the varargs warning could safely be ignored.

In essence, **there are very few situations in which you can safely suppress a varargs warning; usually the warning is justified.**

LINK TO THIS [Practicalities.FAQ300C](#)

REFERENCES [What is a "varargs" warning?](#)  
[How can I suppress a "varargs" warning?](#)  
[What is the SuppressWarnings annotation?](#)  
[What is heap pollution?](#)  
[When does heap pollution occur?](#)

## Which role do wildcards play in method signatures?

*They broaden the set of argument or return types that a method accepts or returns.*

Consider the problem of writing a routine that prints out all the elements in a collection. In non-generic Java it might look like this:

Example (of non-generic print method):

```

void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}

```

In generic Java the same method might be implemented like this.

Example (of generic print method):

```

void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}

```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes `Collection<Object>`, which is not a supertype of all kinds of collections. For instance, it is possible to invoke the old version supplying a `List<String>` as an argument, while the new version rejects the `List<String>` argument because it has an incompatible type.

So what we need here is the supertype of all kinds of collections and that's exactly what the unbounded wildcard parameterized type `Collection<?>` is.

Example (final version of print method):

```

void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}

```

Now, we can call the `print` method with any type of collection.

Bounded wildcards are used for similar purposes. The sole difference is that the set of types that they allow is smaller (because it's restricted by the respective bound). The key idea for use of wildcards in method signatures is to allow a broader set of argument or return types than would be possible with a concrete instantiation.

LINK TO THIS [Practicalities.FAQ301](#)

REFERENCES

## Which one is better: a generic method with type parameters or a non-generic method with wildcards?

*It depends. There is not one-size-fits-all rule.*

Often, we have two alternatives for the declaration of a method:

- We can declare the method as a non-generic method using wildcard parameterized types as argument and return types.

Example: `void reverse(List<?> list) { ... }`

or

- We can declare the method as a generic method with type parameters, that is, without using wildcards.

Example: `<T> void reverse(List<T> list) { ... }`

Whether one alternative is better than the other depends on the semantics of the method. In some situations there is no semantic difference between the two alternatives and it is mostly a matter of taste and style which technique is preferred. But there are also semantics that cannot be expressed with wildcards as well as cases that cannot be solved without wildcards. The subsequent entries explore the details and provide examples.

LINK TO THIS [Practicalities.FAQ302](#)

REFERENCES

[Under which circumstances are the generic version and the wildcard version of a method equivalent?](#)  
[Under which circumstances do the generic version and the wildcard version of a method mean different things?](#)  
[Under which circumstances is there no transformation to the wildcard version of a method possible?](#)

## Under which circumstances are the generic version and the wildcard version of a method equivalent?

*If there is a transformation between the generic and the wildcard version that maintains the semantics.*

In many situations we can replace wildcards by type parameters and vice versa. For example, the following two signatures are semantically equivalent:

```

void reverse(List<?> list) { ... }
<T> void reverse(List<?> list) { ... }

```

In this and the subsequent entries we aim to explore not only which of two versions is better than the other, but also how we can transform between a generic and a wildcard version of a method signature.

### The Transformation

**Wildcard => Generic:** The key idea for turning a method signature with a wildcard into a generic method signature is simple: replace each wildcard by a type variable. These type variables are basically the captures of the respective wildcards, that is, the generic method signature makes the captures visible as type parameters. For example, we can transform the following method signature

```

<T> void fill(List<? super T> list, T obj) { ... }

```

into this signature

```

<S, T extends S> void fill(List<S> list, T obj)

```

by replacing the wildcard "`? super T`" by an additional type parameter `s`. The type relationship, namely that the former wildcard is a supertype of `T`, is expressed by saying that "`T extends S`".

**Generic => Wildcard:** Conversely, if we prefer method signatures with fewer type parameters, then we can reduce the number of type parameters by means of wildcards: replace each type parameter that appears in a parameterized argument or return type by a wildcard. In the previous example, we would transform the method signature

```

<S, T extends S> void fill(List<S> list, T obj)

```

into the signature

```
<T> void fill(List<? super T> list, T obj) { ... }
```

by replacing the type variable `s` by a wildcard. The type relationship, namely that `T` is a subtype of `s`, is expressed by giving the wildcard a lower bound, that is, by saying `"? super T"`.

The transformations sketched out above do not always work. Especially the transformation from a generic version to a wildcard version is not always possible. Problems pop up, for instance, when the generic method signature has more than one type parameter and the type parameters have certain type relationships, such as super-subtype or same-type relationships. In such a situation it might be impossible to translate the type relationship among the type parameters into a corresponding relationship among the wildcards. In the example above, a semantically equivalent wildcard version could be found, because the type relationship could be expressed correctly by means of the wildcard bound. But this is not always possible, as is demonstrated in subsequent entries.

In this entry, we discuss only situations in which a transformation exists that allows for two semantically equivalent signature and the questions is: which one is better? For illustration let us study a couple of examples.

---

### Case Study #1

Let us consider the following `reverse` method. It can be declared as a generic method.

Example (of a method with type parameters):

```
public static <T> void reverse(List<T> list) {
    ListIterator<T> fwd = list.listIterator();
    ListIterator<T> rev = list.listIterator(list.size());
    for (int i = 0, mid = list.size() >> 1; i < mid; i++) {
        T tmp = fwd.next();
        fwd.set(rev.previous());
        rev.set(tmp);
    }
}
```

Alternatively, it can be declared as a non-generic method using a wildcard argument type instead. The transformation simply replaces the unbounded type parameter `T` by the unbounded wildcard `"?"`.

Example (of the same method with wildcards; does not compile):

```
public static void reverse(List<?> list) {
    ListIterator<?> fwd = list.listIterator();
    ListIterator<?> rev = list.listIterator(list.size());
    for (int i = 0, mid = list.size() >> 1; i < mid; i++) {
        Object tmp = fwd.next();
        fwd.set(rev.previous()); // error
        rev.set(tmp);           // error
    }
}
```

The wildcard version has the problem that it does not compile. The iterators of a `List<?>` are of type `ListIterator<?>`, a side effect of which is that their `next` and `previous` methods return an `Object`, while their `set` method requires a more specific type, namely the "capture of `?`".

We can find a workaround for this problem by using raw types, as shown below.

Example (of the same method with wildcards; not recommended):

```
public static void reverse(List<?> list) {
    ListIterator fwd = list.listIterator();
    ListIterator rev = list.listIterator(list.size());
    for (int i = 0, mid = list.size() >> 1; i < mid; i++) {
        Object tmp = fwd.next();
        fwd.set(rev.previous()); // unchecked warning
        rev.set(tmp);           // unchecked warning
    }
}
```

But even that workaround is not satisfying because the compiler gives us unchecked warnings, and rightly so. After all we are calling the `set` method on the raw type `ListIterator`, without knowing which type of object the iterator refers to.

The best implementation of the wildcard version of `reverse` would use a generic helper method, as shown below.

Example (of the same method with wildcards; uses helper method):

```
private static <T> void reverseHelper(List<T> list) {
```

```

ListIterator<T> fwd = list.listIterator();
ListIterator<T> rev = list.listIterator(list.size());
for (int i = 0, mid = list.size() >> 1; i < mid; i++) {
    T tmp = fwd.next();
    fwd.set(rev.previous());
    rev.set(tmp);
}
}
public static void reverse(List<?> list) {
    reverseHelper(list);
}

```

This solution compiles without warnings and works perfectly well, thanks to wildcard capture. However, it raises the question: why not use the generic version in the first place. The helper method is exactly the generic version of our `reverse` method. The wildcard version only adds overhead and does not buy the user anything.

---

### Case Study #2

Let us start with the wildcard version this time. We discuss the example of a `copy` method.

Example (of the a method with wildcards):

```

public static<T> void copy(List<? super T> dest, List<? extends T> src) {
    int srcSize = src.size();
    if (srcSize > dest.size())
        throw new IndexOutOfBoundsException("Source does not fit in dest");
    ListIterator<? super T> di=dest.listIterator();
    ListIterator<? extends T> si=src.listIterator();
    for (int i = 0; i < srcSize; i++) {
        di.next();
        di.set(si.next());
    }
}

```

It is a method that has one type parameter `T` and uses two different wildcard types as argument types. We can transform it into a generic method without wildcards by replacing the two wildcards by two type parameters. Here is the corresponding generic version without wildcards.

Example (of the same method without wildcards):

```

public static <U,T extends U,L extends T> void copy(List<U> dest, List<L> src) {
    int srcSize = src.size();
    if (srcSize > dest.size())
        throw new IndexOutOfBoundsException("Source does not fit in dest");
    ListIterator<U> di = dest.listIterator();
    ListIterator<L> si = src.listIterator();
    for (int i = 0; i < srcSize; i++) {
        di.next();
        di.set(si.next());
    }
}

```

The version without wildcards uses two additional type parameters `U` and `L`. `U` stands for a supertype of `T` and `L` stands for a subtype of `T`. Basically, `U` and `L` are the captures of the wildcards `"? extends T"` and `"? super T"` from the wildcard version of the `copy` method.

Semantically the two version are equivalent. The main difference is the number of type parameters. The version without wildcards expresses clearly that 3 unknown types are involved: `T`, a supertype of `T`, and a subtype of `T`. In the wildcard version this is less obvious. Which version is preferable is to the eye of the beholder.

---

### Case Study #3

Let's study the example of a `fill` method, which has been mentioned earlier, greater detail. Let's start with the generic version without wildcards and let's try to figure out whether we can get rid of the type parameters by means of wildcards.

Example (of the a method with type parameters):

```

public static <S, T extends S> void fill(List<S> list, T obj) {
    int size = list.size();
    ListIterator<S> itr = list.listIterator();
    for (int i = 0; i < size; i++) {
        itr.next();
        itr.set(obj);
    }
}

```

The method takes two type parameters `S` and `T` and two method parameters: an unknown instantiation of the generic type `List`, namely `List<S>`, and an object of unknown type `T`. There is a relationship between `S` and `T`: `S` is a supertype of `T`.

When we try to eliminate the type parameters we find that we can easily replace the type parameter `S` by a wildcard, but we cannot get rid of the type parameter `T`. This is because there is no way to express by means of wildcards that the `fill` method takes an argument of unknown type. We could try something like this:

Example (of the same method with wildcards; does not work):

```
public static void fill(List<?> list, Object obj) {
    int size = list.size();
    ListIterator<?> itr = list.listIterator();
    for (int i = 0; i < size; i++) {
        itr.next();
        itr.set(obj); // error
    }
}
```

The first problem is that this version does not compile; the problem can be reduced to an unchecked warning by using a raw type `ListIterator` instead of the unbounded wildcard `ListIterator<?>`. But the real issue is that this signature gives up the relationship between the element type of the list and the type of the object used for filling the list. A semantically equivalent version of the `fill` method would look like this:

Example (of the same method with wildcards):

```
public static <T> void fill(List<? super T> list, T obj) {
    int size = list.size();
    ListIterator<? super T> itr = list.listIterator();
    for (int i = 0; i < size; i++) {
        itr.next();
        itr.set(obj);
    }
}
```

Now, we have successfully eliminated the need for the type parameter `S`, which stands for the list's element type, by using the "`? super T`" wildcard, but we still need the type parameter `T`. To this regard the example is similar to the copy method discussed earlier, because we can reduce the number of type parameters by means of wildcards, but we cannot entirely eliminate the type parameters. Which version is better is a matter of style and taste.

---

Conclusion: In all these examples it is mostly a matter of taste and style whether you prefer the generic or the wildcard version. There is usually trade-off between ease of implementation (the generic version is often easier to implement) and complexity of signature (the wildcard version has fewer type parameters or none at all).

LINK TO THIS [Practicalities.FAQ302A](#)

#### REFERENCES

- [Should I use wildcards in the return type of a method?](#)
- [Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)
- [What is the capture of a wildcard?](#)
- [Under which circumstances do the generic version and the wildcard version of a method mean different things?](#)
- [Under which circumstances is there no transformation to the wildcard version of a method possible?](#)

---

## Under which circumstances do the generic version and the wildcard version of a method mean different things?

**When a type parameter appears repeatedly in a generic method signature and in case of multi-level wildcards.**

In many situations we can replace wildcards by type parameters and vice versa. For example, the following two signatures are semantically equivalent:

```
void reverse(List<?> list) { ... }
<T> void reverse(List<T> list) { ... }
```

In the previous entry we saw several examples of equivalent method signature, but there are also situations in which the generic version and the wildcard version of a method signature mean different things. These situations include generic method signature in which a type parameter appears repeated and method signatures in which multi-level wildcards, such as `List<Pair<?,?>>`, appear. In the following we study a couple of examples.

---

### Case Study #1

Let us consider the implementation of a `reverse` method. It is slightly different from the `reverse` method we discussed in the previous entry. The key difference is that the `List` type, and with it the type parameter `T`, appears twice in the method's signature: in the argument type and the return type. Let's start again with the generic version of the `reverse` method.

Example (of a method with type parameters):

```
public static <T>List<T> reverse(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
```



```

    for (int i = 0; i < list.size(); i++) {
        tmp.set(i, list.get(list.size() - i - 1));
    }
    return tmp;
}

```

If we tried to declare this method as a non-generic method using wildcards, a conceivable signature could look like this.

Example (of the same method with wildcards; does not compile):

```

public static List<?> reverse(List<?> list) {
    List<?> tmp = new ArrayList<?>(list); // error
    for (int i = 0; i < list.size(); i++) {
        tmp.set(i, list.get(list.size() - i - 1)); // error
    }
    return tmp;
}

```

The first problem is that this version does not compile; the problem can be reduced to an unchecked warning by using the raw types `List` and `ArrayList` instead of the unbounded wildcards `List<?>` and `ArrayList<?>`. Even the warnings can be eliminated by relying on wildcard capture and using a generic helper method. But one fundamental issue remains: the wildcard version has an entirely different semantic meaning compared to the generic version.

The generic version is saying: the reverse method accepts a list with a certain, unknown element type and returns a list of that same type. The wildcard version is saying: the reverse method accepts a list with a certain, unknown element type and returns a list of a potentially different type. Remember, each occurrence of a wildcard stands for a potentially different type. In principle, the reverse method could take a `List<Apple>` and return a `List<Orange>`. There is nothing in the signature or the implementation of the reverse method that indicates that "what goes in does come out". In other words, the wildcard signature does not reflect our intent correctly.

Conclusion: In this example it the generic version and the wildcard version have different meaning.

---

## Case Study #2

Another example where more than one wildcard occurs in the signature of the method.

Example (of a method with type parameters):

```

class Pair<S,T> {
    private S first;
    private T second;
    ...
    public Pair(S s,T t) { first = s; second = t; }

    public static <U> void flip(Pair<U,U> pair) {
        U tmp = pair.first;
        pair.first = pair.second;
        pair.second = tmp;
    }
}

```

When we try to declare a wildcard version of the generic `flip` method we find that there is no way of doing so. We could try the following:

Example (of the same method with wildcards; does not compile):

```

class Pair<S,T> {
    private S first;
    private T second;
    ...
    public Pair(S s,T t) { first = s; second = t; }

    public static void flip(Pair<?,?> pair) {
        Object tmp = pair.first;
        pair.first = pair.second; // error: incompatible types
        pair.second = tmp; // error: incompatible types
    }
}

```

But this wildcard version does not compile, and rightly so. It does not make sense to flip the two parts of a `Pair<?,?>`. Remember, each occurrence of a wildcard stands for a potentially different type. We do not want to flip the two parts of a pair, if the part are of different types. This additional requirement, that the parts of the pair must be of the same type, cannot be expressed by means of wildcards.

The wildcard version above would be equivalent to the following generic version:

Example (of the generic equivalent of the wildcard version; does not compile):

```

class Pair<S,T> {
    private S first;
    private T second;
    ...
    public Pair(S s,T t) { first = s; second = t; }

    public static <U,V>void flip(Pair<U,V> pair) {
        U tmp = pair.first;
        pair.first = pair.second; // error: incompatible types
        pair.second = tmp; // error: incompatible types
    }
}

```

```
} }  
}
```

Now it should be obvious that the wildcard version simply does not express our intent.

Conclusion: In this example only the generic version allows to express the intent correctly.

---

### Case Study #3

If a method signature uses multi-level wildcard types then there is always a difference between the generic method signature and the wildcard version of it. Here is an example. Assume there is a generic type `Box` and we need to declare a method that takes a list of boxes.

Example (of a method with a type parameter):

```
public static <T> void print1(List<Box<T>> list) {  
    for (Box<T> box : list) {  
        System.out.println(box);  
    }  
}
```

Example (of method with wildcards):

```
public static void print2(List<Box<?>> list) {  
    for (Box<?> box : list) {  
        System.out.println(box);  
    }  
}
```

Both methods are perfectly well behaved methods, but they are not equivalent. The generic version requires a homogenous list of boxes of the same type. The wildcard version accepts a heterogenous list of boxes of different type. This becomes visible when the two `print` methods are invoked.

Example (calling the 2 versions):

```
List<Box<?>> list1 = new ArrayList<Box<?>>();  
list1.add(new Box<String>("abc"));  
list1.add(new Box<Integer>(100));  
  
print1(list1); // error  
print2(list1); // fine  
  
List<Box<Object>> list2 = new ArrayList<Box<Object>>();  
list2.add(new Box<Object>("abc"));  
list2.add(new Box<Object>(100));  
  
print1(list2); // fine  
print2(list2); // error
```

---

```
error: <T>print1(Box<T>>) cannot be applied to (Box<?>>)  
    print1(list1);  
    ^
```

```
error: print2(Box<?>>) cannot be applied to (Box<Object>>)  
    print2(list2);  
    ^
```

First, we create a list of boxes of different types and stuff a `Box<String>` and a `Box<Integer>` into the list. This heterogenous list of type `List<Box<?>>` cannot be passed to the generic method, because the generic method expects a list of boxes of the same type.

Then, we create a list of boxes of the same type, namely of type `Box<Object>`, and we stuff two `Box<Object>` objects into the list. This homogenous list of type `List<Box<Object>>` cannot be passed to the wildcard method, because the wildcard method expects a list of boxes, where there is no restriction regarding the type of the boxes.

Let us consider a third version of the `print` method, again with wildcards, but more relaxed so that it accepts either type of list, the homogenous and the heterogenous list of boxes.

Example (of another wildcard version):

```
public static void print3(List<? extends Box<?>> list) {  
    for (Box<?> box : list) {  
        System.out.println(box);  
    }  
}
```

Example (calling all 3 versions):

```
List<Box<?>> list1 = new ArrayList<Box<?>>();  
list1.add(new Box<String>("abc"));
```

```
list1.add(new Box<Integer>(100));

print1(list1); // error
print2(list1); // fine
print3(list1); // fine

List<Box<Object>> list2 = new ArrayList<Box<Object>>();
list2.add(new Box<Object>("abc"));
list2.add(new Box<Object>(100));

print1(list2); // fine
print2(list2); // error
print3(list2); // fine
```

*No matter how we put it, the generic version and the wildcard versions are not equivalent.*

*Conclusion: In this example the generic version and the wildcard version have different meaning.*

uiLINK TO THIS [Practicalities.FAQ302B](#)

#### REFERENCES

[What do multi-level wildcards mean?](#)  
[If a wildcard appears repeatedly in a type argument section, does it stand for the same type?](#)  
[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)  
[What is the capture of a wildcard?](#)  
[Under which circumstances are the generic version and the wildcard version of a method equivalent?](#)  
[Under which circumstances is there no transformation to the wildcard version of a method possible?](#)

## Under which circumstances is there no transformation to the wildcard version of a method possible?

*If a type parameter has more than one bound.*

Wildcards can have at most one upper bound, while type parameters can have several upper bounds. For this reason, there is not wildcard equivalent for generic method signatures with type parameter with several bounds. Here is an example.

Example (of a method with a type parameter with more than one bound):

```
public interface State {
    boolean isIdle();
}

public static <T extends Enum<T> & State> boolean hasIdleState(EnumSet<T> set) {
    for (T state : set)
        if (state.isIdle()) return true;
    return false;
}
```

This `hasIdleState` method has a type parameter that must be a enum type that implements the `State` interface. The requirement of both being an enum type and implementing an interface cannot be expressed by means of wildcards. If we tried it it would look like this:

Example (of the same method without type parameters; does not compile):

```
public static boolean hasIdleState(EnumSet<? extends Enum<?> & State> set) { // error
    ...
}
```

This attempt fails because a wildcard cannot have two bounds and for this reason the expression `"? extends Enum<?> & State"` is illegal syntax.

Conclusion: In this example there is no way to find an equivalent version with wildcards and the generic version is the only viable solution.

uiLINK TO THIS [Practicalities.FAQ302C](#)

#### REFERENCES

[What is the difference between a wildcard bound and a type parameter bound?](#)  
[Under which circumstances are the generic version and the wildcard version of a method equivalent?](#)  
[Under which circumstances do the generic version and the wildcard version of a method mean different things?](#)

## Should I use wildcards in the return type of a method?

*Avoid it, if you can.*

Methods that return their result through a reference of a wildcard type are rarely a good idea. The key problem is that access to the result is restricted and there is often not much the caller can do with the result he receives. Remember, access to an object through a reference of a wildcard type is restricted; the restrictions depend on the sort of wildcard being used. For this reason wildcard return types are best avoided.

Example (of a method with a wildcard return type; not recommended):

```

List<?> modifyList(List<?> list) {
    ...
    return list;
}

List<String> names = ...
List<?> result = modifyList(names);
result.add("Bobby Anderson"); // error

```

Since the result is returned through a wildcard reference, a whole bunch of methods cannot be invoked on the result. A generic method would in this example be way more useful.

Example (alternative generic method; recommended):

```

<T> List<T> modifyList(List<T> list) {
    ...
    return list;
}

List<String> names = ...
List<String> result = modifyList(names);
result.add("Bobby Anderson"); // fine

```

It is hard to imagine that a method such as `modifyList` would be sensible in the first place. Most likely it is bad, if not buggy design. After all it is weird that a method received one unknown type of list as input and returns another unknown type of list as output. Does it turn a `List<Apples>` into a `List<Oranges>`? The generic version is more likely to be the more sensible signature to begin with. But there are examples, even in the JDK, where methods return wildcard types and the design looks reasonable at first sight, and yet suffers from the restrictions outlined above. The remainder of this FAQ entry discusses such a more realistic example. If you are not interested in further details, feel free to skip the rest of this entry. It's quite instructive though, if you're interested in learning how to design generic classes properly.

## A More Comprehensive Example

### The Problem

As promised, we are going to study an example from the JDK to illustrate the problems with methods that return wildcard types. The example is taken from the JDK package `java.lang.ref`. This package provides reference classes, which support a limited degree of interaction with the garbage collector. All these reference classes are subtypes of a super class named `Reference<T>`.

Example (sketch of class `java.lang.ref.Reference`):

```

public abstract class Reference<T> {
    public T get() { ... }
    public void clear() { ... }
}

```

There are two reference classes of interest `SoftReference<T>` and `WeakReference<T>`. Instances of the reference classes can be registered with a reference queue.

Example (sketch of class `java.lang.ref.WeakReference`):

```

public class WeakReference<T> extends Reference<T> {
    public WeakReference(T referent) { ... }
    public WeakReference(T referent, ReferenceQueue<? super T> q) { .. }
}

```

This reference queue is described by a type named `ReferenceQueue<T>`. Its `poll` and `remove` methods return elements from the queue through a wildcard reference of type `Reference<? extends T>`.

Example (sketch of class `java.lang.ref.ReferenceQueue`):

```

public class ReferenceQueue<T> {
    public Reference<? extends T> remove() { ... }
    public Reference<? extends T> remove(long timeout) { ... }
    public Reference<? extends T> poll() { ... }
}

```

The methods of the `ReferenceQueue<T>` type are examples of methods that return their result through a wildcard type. The purpose of the reference classes and the reference queue is of no relevance for our discussion. What we intend to explore are the consequences of the wildcard return type of the reference queue's methods.

Let's consider a use case for these reference classes. It is common that the actual reference types are subtypes of the reference classes from the JDK. This is because a reference type often must maintain additional data. In our example this subtype is called `DateReference` and it is weak reference to a date object. It caches the representation of the referenced date as a time value and has a couple of additional methods.

Example (of a user-defined reference class):

```

public class WeakDateReference<T extends Date> extends WeakReference<T> {

```

```

    long time;

    public WeakDateReference(T t) {
        super(t);
        time = t.getTime();
    }
    public WeakDateReference(T t, ReferenceQueue<? super T> q) {
        super(t, q);
        time = t.getTime();
    }
    public long getCacheTime() { return time; }

    public boolean isEquivalentTo(DateReference<T> other) {
        return this.time == other.getCacheTime();
    }
    public boolean contains(T t) {
        return this.get() == t;
    }
}

```

Let's now create such a weak date reference and register it with a reference queue.

Example (of using a user-defined reference class with a reference queue):

```

ReferenceQueue<Date> queue = new ReferenceQueue<Date>();
Date date = new Date();
WeakDateReference<Date> dateRef = new WeakDateReference<Date>(date, queue);

```

The reference queue will later contain weak date references that have been cleared by the garbage collector. When we retrieve entries from the reference queue, they are passed to us through a reference of a the wildcard type `Reference<? extends Date>`, because this is the way the reference queue's methods are declared.

Example (of using a user-defined reference class with a reference queue):

```

WeakDateReference<Date> deadRef = queue.poll(); // error
Reference<? extends Date> deadRef = queue.poll(); // fine

```

---

```

error: incompatible types
found   : Reference<capture of ? extends Date>
required: WeakDateReference<.Date>
    WeakDateReference<Date> deadRef = queue.poll();
                                             ^

```

What is returned is a reference of type `Reference<? extends Date>` pointing to an object of type `WeakDateReference<Date>`. If we now try to use the returned object we find that we cannot access the object as would like to. In particular, some of the methods of my weak date reference type cannot be called.

Example (of using the returned reference object):

```

Reference<? extends Date> deadRef = queue.poll();

long time = deadRef.getCacheTime(); // error
long time = ((WeakDateReference<Date>)deadRef).getCacheTime(); // unchecked warning
long time = ((WeakDateReference<? extends Date>)deadRef).getCacheTime(); // fine

```

---

```

error: cannot find symbol
symbol  : method getCacheTime()
location: class Reference<capture of ? extends Date>
    time = deadRef.getCacheTime();
                    ^

warning: [unchecked] unchecked cast
found   : Reference<capture of ? extends Date>
required: WeakDateReference<Date>
    time = ((WeakDateReference<Date>)deadRef).getCacheTime();
                    ^

```

Before we can access any of the weak date reference type's methods we must cast down from its super-type `Reference` to its own type `WeakDateReference`. This explains why the first invocation of the `getCacheTime` method fails; the super-type `Reference` does not have any such method.

So, we must cast down. We would like to cast the returned reference variable of type `Reference<? extends Date>` to the object's actual type `WeakDateReference<Date>`, but the compiler issues an unchecked warning. This warning is justified because the reference queue can potentially hold a mix of weak and soft references of all sorts as long as they refer to a `Date` object or a subtype thereof. We know that the reference queue only holds objects of our weak date reference type, because we know the context of our little sample program. But the compiler can't know this and rejects the cast to `WeakDateReference<Date>` based on the static type information as an unchecked cast.

We can safely cast down from the type `Reference<? extends Date>` to the wildcard type `WeakDateReference<? extends Date>` though. This is safe because the two types have the same type argument `"? extends Date"`. The compiler can ensure that type `WeakDateReference<? extends Date>` is a subtype of `Reference<? extends Date>` and the JVM can check at runtime based on the raw types that the referenced object really is a `WeakDateReference`.

So, we invoke the weak date reference methods through a reference of the wildcard type `WeakDateReference<? extends Date>`. This fine for the `getCachedTime` method, but fails when we try to invoke methods in whose argument type the type parameter `T` of our type `WeakDateReference<T>` appears.

Example (of using the returned reference object):

```
Reference<? extends Date> deadRef = queue.poll();

long time = ((WeakDateReference<? extends Date>)deadRef).getCachedTime(); // fine
boolean eqv = ((WeakDateReference<? extends Date>)deadRef).isEquivalentTo(dateRef); // error
boolean cont = ((WeakDateReference<? extends Date>)deadRef).contains(date); // error
```

---

```
error: isEquivalentTo(WeakDateReference<capture of ? extends Date>)
in WeakDateReference<capture of ? extends Date>
cannot be applied to (WeakDateReference<Date>)
boolean eqv = ((WeakDateReference<? extends Date>)deadRef).isEquivalentTo(dateRef);
                ^
error: contains(capture of ? extends Date)
in WeakDateReference<capture of ? extends Date>
cannot be applied to (Date)
boolean cont = ((WeakDateReference<? extends Date>)deadRef).contains(date);
                ^
```

This illustrates the problems that wildcard return types introduce: certain methods cannot be invoked through the returned wildcard reference. In other word, there is not much you can do with the result. How severe the restrictions are, depends on the nature of the wildcard type, the type of the returned object and the signatures of the methods that shall be invoked on the returned object. In our example we are forced to access the result through a reference of type `WeakDateReference<? extends Date>`. As a consequence, we cannot invoke the methods `boolean isEquivalentTo(DateReference<T> other)` and `boolean contains(T t)`, because the type parameter `T` appears in their argument types.

## Conclusion

Can or should we conclude that methods with wildcard return types are always wrong? Not quite. There are other examples in the JDK, where the wildcard return type does not impose any problems. The most prominent example is the generic class `java.lang.Class<T>`. It has a number of methods that return wildcard such as `Class<?>`, `Class<? super T>`, and `Class<? extends U>`, but at the same time class `Class<T>` does not have a single method in whose argument type the type parameter `T` would appear. The restriction illustrated above exists in principle, but in practice it is irrelevant, because the type in question does not have any methods whose inaccessibility would hurt.

This is different for the generic `ReferenceQueue<T>` type discussed above. The super type `Reference<T>` does not have any methods in whose argument type the type parameter `T` would appear, pretty much like class `Class<T>`. But, it is common that subtypes of type `Reference<T>` are defined and used, and there is no reason why those subtypes shouldn't be generic and have method in whose argument type the type parameter `T` would appear. And there we are ... and hit the limits.

## A Conceivable Solution

The recommendation is: avoid wildcard return types if you can. The question is: can we avoid the wildcard return type in the reference queues's methods? The answer is: yes, but it comes at a cost. In order to understand what the trade-off is we need to find out why the reference queue returns a wildcard type instead of a concrete parameterized type. After all, no other queue type returns a wildcard from any of its methods; consider for instance `java.util.Queue` or `java.util.concurrent.BlockingQueue`.

The crux in case of the `ReferenceQueue` is its interaction with the `Reference` type. Class `Reference` and all its subclasses have constructors that permit attachment of a reference queue to a reference object. In class `Reference` this constructor is package visible, in the subclasses it is `public`.

Example (excerpt from class `java.lang.ref.Reference`):

```
public abstract class Reference<T> {
    ReferenceQueue<? super T> queue;
    Reference(T referent) { ... }
    Reference(T referent, ReferenceQueue<? super T> queue) { ... }

    public T get() { ... }
    public void clear() { ... }
    ...
}
```

The package visible constructor takes the wildcard instantiation `ReferenceQueue<? super T>` as the argument type and thereby allows to attach a reference queue for references of a supertype, say `Date`, to a reference for a subtype, say `NamedDate`.

Example (of using a reference with a reference queue):

```
ReferenceQueue<Date> queue = new ReferenceQueue<Date>();
NamedDate date = new NamedDate("today");
WeakReference<NamedDate> dateRef = new WeakReference<NamedDate>(date, queue);
```

Thanks to the wildcard argument type in the reference's constructor we can place references of type `Reference<NamedDate>` into a reference queue of type `ReferenceQueue<Date>`.

Inside class `Reference`, at some point in time, the reference puts itself into its attached reference queue. For this purpose the type `ReferenceQueue` has a package visible `enqueue` method.

Example (excerpt from class `java.lang.ref.ReferenceQueue`):

```

public class ReferenceQueue<T> {
    boolean enqueue(Reference<? extends T> ref) { ... }
    public Reference<? extends T> remove() { ... }
    public Reference<? extends T> remove(long timeout) { ... }
    public Reference<? extends T> poll() { ... }
}

```

This enqueue method must accept a `Reference<? extends T>` as an argument, because it is permitted that a reference of a subtype can be put into the reference queue. Like in the example above, where we registered a `Reference<NamedDate>` with a `ReferenceQueue<Date>`. If the enqueue method required an argument of the concrete type `Reference<T>` then we could never store a `Reference<NamedDate>` in a `ReferenceQueue<Date>`.

A consequence of accepting references of type `Reference<? extends T>` in the constructor is that the exact type of the references in the queue is unknown. All retrieval methods, such as `poll` and `remove`, have no choice and must return the same wildcard type that was accepted in the constructor. This is the reason why the reference queue's `poll` and `remove` methods return wildcard types instead of concrete type.

If we want to get rid of the wildcard return type we must give up the ability to attach a reference queue for references of a supertype, say `Date`, to a reference for a subtype, say `NamedDate`. An alternative design would look like this:

Example (sketch of a revised `Reference` class; different from JDK version):

```

public abstract class Reference<T> {
    ReferenceQueue<T> queue;
    Reference(T referent) { ... }
    Reference(T referent, ReferenceQueue<T> queue) { ... }

    public T get() { ... }
    public void clear() { ... }
    ...
}

```

Example (sketch of a revised `ReferenceQueue` class; different from JDK version):

```

public class ReferenceQueue<T> {
    boolean enqueue(Reference<T> ref) { ... }

    public Reference<T> remove() { ... }
    public Reference<T> remove(long timeout) { ... }
    public Reference<T> poll() { ... }
}

```

After such a redesign we can not longer place references to `NamedDate` into a reference queue for reference to `Date`.

Example (of using a reference with a reference queue; different from JDK version):

```

ReferenceQueue<Date> queue = new ReferenceQueue<Date>();
NamedDate date = new NamedDate("today");
WeakReference<NamedDate> dateRef = new WeakReference<NamedDate>(date, queue); // error

```

In return we now receive a concrete parameterized type when we take references out of the queue and the concrete type gives us full access to the reference type. The restrictions resulting from wildcard return types are eliminated.

Example (of using a user-defined reference type; different from JDK version):

```

Reference<Date> deadRef = queue.poll();

long time = ((WeakDateReference<Date>)deadRef).getCacheTime(); // fine
boolean equiv = ((WeakDateReference<Date>)deadRef).isEquivalentTo(dateRef); // fine
boolean cont = ((WeakDateReference<Date>)deadRef).contains(date); // fine

```

As you can see, there is a trade-off: the flexibility to put references to a subtype into a reference queue of references to a supertype costs us limited access to the references retrieved from the queue, and vice versa. The design decisions made for the reference queue are certainly reasonable, because user-defined reference types with sophisticated functionality are probably rare and hence the restrictions from the wildcard return type will not hit too many programmers.

Nonetheless, the case study illustrates that design decisions made in one place have consequences in other places. As a general rule, be aware of the restrictions that come with wildcard return types and avoid them if you can, unless you have a compelling reason to use them anyway.

LINK TO THIS [Practicalities.FAQ303](#)

REFERENCES

[Under which circumstances are the generic version and the wildcard version of a method equivalent?](#)  
[Under which circumstances do the generic version and the wildcard version of a method mean different things?](#)  
[Under which circumstances is there no transformation to the wildcard version of a method possible?](#)  
[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard instantiation?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard instantiation?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard instantiation?](#)  
[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard instantiation?](#)

## How do I implement a method that takes a wildcard argument?

### *Using a generic helper method and wildcard capture.*

Consider the situation where you decided that a certain method should take arguments whose type is a wildcard parameterized type. When you start implementing such a method you will find that you do not have full access to the argument. This is because wildcards do not permit certain operations on the wildcard parameterized type.

Example (implementation of a `reverse` method with wildcards; does not work):

```
public static void reverse(List<?> list) {
    List<?> tmp = new ArrayList<?>(list); // error
    for (int i=0;i<list.size();i++){
        tmp.set(i,list.get(list.size()-i-1)); // error
    }
    list = tmp;
}
```

Using the wildcard type `List<?>` we can neither create a temporary copy of the argument nor can we invoke the `set` method. A workaround, that works in this particular case, is use of wildcard capture and a generic helper method.

Example (corrected implementation of a `reverse` method with wildcards):

```
public static void reverse(List<?> list) {
    rev(list);
}
private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i=0;i<list.size();i++){
        tmp.set(i,list.get(list.size()-i-1));
    }
    list = tmp;
}
```

Wildcard capture makes it possible to invoke a generic helper method. The helper method does not use any wildcards; it is generic and has a type parameter instead. It has unrestricted access to its arguments' methods and can provide the necessary implementation.

Since the helper method has the exact same functionality as the original method and permits the same set of argument types, one might consider using it instead of the method with the wildcard argument in the first place.

Example (generic version of the `reverse` method):

```
public static <T> void reverse(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i=0;i<list.size();i++){
        tmp.set(i,list.get(list.size()-i-1));
    }
    list = tmp;
}
```

LINK TO THIS [Practicalities.FAQ304](#)

REFERENCES [What is the capture of a wildcard?](#)  
[What is a parameterized \(or generic\) method?](#)  
[Can I use a wildcard parameterized type like any other type?](#)  
[Can I create an object whose type is a wildcard parameterized type?](#)

## How do I implement a method that takes a multi-level wildcard argument?

### *Using several generic helper methods and wildcard capture.*

Here is an example of a method whose argument and return type is a multi-level wildcard. It is a method that takes a list whose element type is an arbitrary pair type and return such a list. The `swapAndReverse` method reverses the order all the list elements and swaps the members of each pair. It is a contrived example for the purpose of illustrating the implementation technique.

Example:

```
class Pair<E> {
    private E fst, snd;
    public E getFirst() { return fst; }
}
```



```

    public void setFirst(S s) { fst = s; }
    ...
}
class Test {
    public static ArrayList<? extends Pair<?>> swapAndReverse(ArrayList<? extends Pair<?>> l) {
        ...
    }
    public static void main(String[] args) {
        ArrayList<Pair<Integer>> list = new ArrayList<Pair<Integer>>();
        list.add(new Pair<Integer>(-1,1,0));
        list.add(new Pair<Integer>(1,0,0));
        ...
        List<?> result = swapAndReverse(list);

        ArrayList<Pair<?>>list = new ArrayList<Pair<?>>();
        list.add(new Pair<String>("a","b","c"));
        list.add(new Pair<Integer>(1,0,-1));
        list.add(new Pair<Object>(new Date(),Thread.State.NEW,5));
        ...
        List<?> result = swapAndReverse(list);
    }
}

```

The `swapAndReverse` method can be invoked on homogenous lists of pairs of the same type, such as `ArrayList<Pair<Integer>>`, but also on a heterogenous list of pairs of different types, such as `ArrayList<Pair<?>>`.

When we try to implement the method we find that the wildcard argument type does not permit invocation of the operations that we need.

Example (implementation of a `swapAndReverse` method with wildcards; does not work):

```

public static ArrayList<? extends Pair<?>> swapAndReverse(ArrayList<? extends Pair<?>> l) {
    ArrayList<? extends Pair<?>> list
        = new ArrayList<? extends Pair<?>>(l); // error
    for (int i=0;i<l.size();i++){
        list.set(i,l.get(l.size()-i-1)); // error
    }
    for (Pair<?> pair : list) {
        Object e = pair.getFirst();
        pair.setFirst(pair.getSecond()); // error
        pair.setSecond(e); // error
    }
    return list;
}

```

We cannot create a temporary copy of the list and cannot access the individual pairs in the list. Hence we apply the capture-helper technique from above.

Example (implementation of a `swapAndReverse` method with helper method; does not work):

```

public static ArrayList<? extends Pair<?>> swapAndReverse(ArrayList<? extends Pair<?>> l) {
    return capturePairType(l);
}
private static <T extends Pair<?>> ArrayList<T> capturePairType(ArrayList<T> l) {
    ArrayList<T> list = new ArrayList<T>(l);
    for (int i=0;i<l.size();i++){
        list.set(i,l.get(l.size()-i-1));
    }
    for (T pair : list) {
        Object e = pair.getFirst();
        pair.setFirst(pair.getSecond()); // error
        pair.setSecond(e); // error
    }
    return list;
}

```

The compiler will capture the type of the pairs contained in the list, but we still do not know what type of members the pairs have. We can use the capture-helper technique again to capture the pairs' type argument.

Example (corrected implementation of a `swapAndReverse` method with wildcards):

```

public static ArrayList<? extends Pair<?>> swapAndReverse(ArrayList<? extends Pair<?>> l) {
    return capturePairType(l);
}
private static <T extends Pair<?>> ArrayList<T> capturePairType(ArrayList<T> l) {
    ArrayList<T> list = new ArrayList<T>(l);
    for (int i=0;i<l.size();i++){
        list.set(i,l.get(l.size()-i-1));
    }
}

```

```

    }
    for (T pair : list) {
        captureMemberType(pair);
    }
    return list;
}
private static <E> void captureMemberType(Pair<E> pair) {
    E e = pair.getFirst();
    pair.setFirst(pair.getSecond());
    pair.setSecond(e);
}

```

In this case there is no alternative to the stepwise application of the capture-helper technique. A generic version of the `swapAndReverse` method would have slightly different semantics.

Example (parameterized version of the `swapAndReverse` method):

```

public static <E,T extends Pair<E>> ArrayList<T> swapAndReverse(ArrayList<T> l) {
    ArrayList<T> list = new ArrayList<T>(l);
    for (int i=0;i<l.size();i++){
        list.set(i,l.get(l.size()-i-1));
    }
    for (T pair : list) {
        E e = pair.getFirst();
        pair.setFirst(pair.getSecond());
        pair.setSecond(e);
    }
    return list;
}

```

This version of the `swapAndReverse` method has one disadvantage: it does *not* accept a mixed list of pairs of arbitrary types, such as `ArrayList<Pair<?>>`.

Example:

```

class Test {
    public static void main(String[] args) {
        ArrayList<Pair<Integer>> list = new ArrayList<Pair<Integer>>();
        list.add(new Pair<Integer>(-1,1,0));
        list.add(new Pair<Integer>(1,0,0));
        ...
        List<?> result = swapAndReverse(list);

        ArrayList<Pair<?>> list = new ArrayList<Pair<?>>();
        list.add(new Pair<String>("a","b","c"));
        list.add(new Pair<Integer>(1,0,0));
        list.add(new Pair<Object>(new Date(),Thread.State.NEW,5));
        ...
        List<?> result = swapAndReverse(list); // error
    }
}

```

---

```

error: <E,T>swapAndReverse(java.util.ArrayList<T>) in Test cannot be applied to (java.util.ArrayList<Pair<?>>)
    List<?> result = swapAndReverse(list);
                        ^

```

On the other hand, the generic `swapAndReverse` method has the advantage that it returns a concrete instantiation of `ArrayList`, that does not suffer from the limitations that come with the wildcard instantiation that is returned from the wildcard version of the `swapAndReverse` method.

LINK TO THIS [Practicalities.FAQ305](#)

REFERENCES

- [How do I implement a method that takes a wildcard argument?](#)
- [What do multi-level wildcards mean?](#)
- [What is the capture of a wildcard?](#)
- [What is a parameterized or generic method?](#)
- [What is a bounded type parameter?](#)
- [Which types are permitted as type parameter bounds?](#)
- [Can I use a type parameter as part of its own bounds or in the declaration of other type parameters?](#)
- [Can I use a wildcard parameterized type like any other type?](#)
- [Can I create an object whose type is a wildcard parameterized type?](#)

**I want to pass a `U` and a `X<U>` to a method. How do I correctly declare that method?**

*Using an upper bound wildcard parameterized type instead of a concrete parameterized type as the argument type.*

Example (has a bug):

```
interface Acceptor<V> {
    void accept(Task<V> task, V v);
}
interface Task<U> {
    void go(Acceptor<? super U> acceptor);
}
class AcceptingTask<U> implements Task<U> {
    public void go(Acceptor<? super U> acceptor) {
        U result = null;
        ... produce result ...
        acceptor.accept(this, result); // error
    }
}
```

---

```
error: accept(Task<capture of ? super U>,capture of ? super U)
in Acceptor<capture of ? super U> cannot be applied to (AcceptingTask<U>,U)
    acceptor.accept(this, result);
                   ^
```

This is the example of a callback interface `Acceptor` and its `accept` method which takes result-producing task and the result. Note that the `accept` method takes a result of type `v` and a corresponding task of type `Task<V>`.

The task is described by an interface `Task`. It has a method `go` that is supposed to produce a result and takes an `Acceptor`, to which it passes the result.

The class `AcceptingTask` is an implementation of the `Task` interface and in its implementation of the `go` method we see an invocation of the `accept` method. This invocation fails.

The problem with this invocation is that the `accept` method is invoked on a wildcard instantiation of the `Acceptor`, namely `Acceptor<? super U>`. Access to methods through wildcard parameterized types is restricted. The error message clearly indicates the problem. Method `accept` in `Acceptor<? super U>` expects a `Task<capture of ? super U>` and a `capture of ? super U`. What we pass as arguments are a `AcceptingTask<U>` and a `U`. The argument of type `U` is fine because the declared argument type is an unknown supertype of `U`. But the argument of type `AcceptingTask<U>` is a problem. The declared argument type is an instantiation of `Task` for an unknown supertype of `U`. The compiler does not know which supertype and therefore rejects all argument types.

The crux is that the signature of the `accept` method is too restrictive. If we would permit instantiations of `Task` for subtypes of `U`, then it would work.

Example (corrected):

```
interface Acceptor<V> {
    void accept(Task<? extends V> task, V v);
}
interface Task<U> {
    void go(Acceptor<? super U> acceptor);
}
class AcceptingTask<U> implements Task<U> {
    public void go(Acceptor<? super U> acceptor) {
        U result = null;
        ... produce result ...
        acceptor.accept(this, result); // fine
    }
}
```

With this relaxed signature the `accept` method in `Acceptor<? super U>` expects a `Task<? extends capture of ? super U>`, that is, an instantiation of `Task` for a subtype of a supertype of `U` and `Task<U>` meets this requirement.

The common misunderstanding here is that the signature `accept(Task<V> task, V v)` looks that I can pass a `Task<U>` whenever I can pass a `U`. This is true for concrete instantiations of the enclosing type, but not when wildcard instantiations are used. The accessibility rules for methods that take the type parameter such as `v` as an argument and methods that take a parameterized type instantiated on the type parameter such as `Task<V>` are very different.

The solution to the problem is relaxing the signature by using a wildcard parameterized type as an argument type instead of a concrete parameterized type.

LINK TO THIS [Practicalities.FAQ306](#)

#### REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard parameterized type?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)  
[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)  
[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)  
[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard instantiation?](#)

---

## Working With Generic Interfaces

---

### Can a class implement different instantiations of the same generic interface?

*No, a type must not directly or indirectly derive from two different instantiations of the same generic interface.*

The reason for this restriction is the translation by type erasure. After type erasure the different instantiations of the same generic interface collapse to the same raw type. At runtime there is no distinction between the different instantiations any longer.

Example (of illegal subtyping from two instantiations of the same generic interface):

```
class X implements Comparable<X>, Comparable<String> { // error
    public int compareTo(X arg)    { ... }
    public int compareTo(String arg) { ... }
}
```

During type erasure the compiler would not only remove the type arguments of the two instantiations of `Comparable`, it would also try to create the necessary bridge methods. Bridge methods are synthetic methods generated by the compiler; they are needed when a class has a parameterized supertype.

Example (same as above, after a conceivable translation by type erasure):

```
class X implements Comparable, Comparable {
    public int compareTo(X arg)    { ... }
    public int compareTo(String arg) { ... }
    public int compareTo(Object arg) { return compareTo((X)arg); }
    public int compareTo(Object arg) { return compareTo((String)arg); }
}
```

The bridge method generation mechanism cannot handle this.

LINK TO THIS [Practicalities.FAQ401](#)

REFERENCES

- [What is type erasure?](#)
- [What is a bridge method?](#)
- [Can I use different instantiations of a same generic type as bounds of a type parameter?](#)
- [Can a subclass implement another instantiation of a generic interface than any of its superclasses does?](#)
- [What happens if a class implements two parameterized interfaces that define the same method?](#)

---

### Can a subclass implement a different instantiation of a generic interface than any of its superclasses does?

*No, the superclass determines which instantiation of a generic interface the entire class hierarchy must implement.*

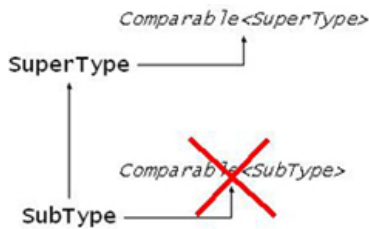
Example:

```
class Person implements Comparable<Person> {
    public int compareTo(Person arg) { ... }
}
class Student extends Person implements Comparable<Student> { // error
    public int compareTo(Student arg) { ... }
}
```

---

```
error: java.lang.Comparable cannot be inherited with different arguments: <Student> and <Person>
class Student extends Person implements Comparable<Student> {
  ^
```

The `Student` subclass would be implementing two different instantiations of the generic `Comparable` interface, which is illegal. The consequence is that a superclass that implement a certain instantiation of a generic interface determines for all its subclasses which instantiation of the interface they must implement. No subclass can ever implement another instantiation of the generic interface.



This consequence makes proper use of generic interfaces fairly challenging. Here is another example of the effect, using the `Delayed` interface from the [java.util.concurrent](http://java.util.concurrent) package.

Example (interface `java.util.concurrent.Delayed`):

```
public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

The `Delayed` interface is a sub-interface of an instantiation of the `Comparable` interface and thereby takes away the chance that any implementing class can ever be comparable to anything else but a `Delayed` object.

Example:

```
class SomeClass implements Delayed, Comparable<SomeClass> { // error
    public long getDelay(TimeUnit unit) { ... }
    public int compareTo(Delayed other) { ... }
    public int compareTo(SomeClass other) { ... }
}
```

---

error: java.lang.Comparable cannot be inherited with different arguments: <java.util.concurrent.Delayed> and <SomeClass>

```
class SomeClass implements Delayed, Comparable<SomeClass> {
    ^
```

LINK TO THIS [Practicalities.FAQ402](http://Practicalities.FAQ402)

REFERENCES [Can a class implement different instantiations of the same generic interface?](http://Can a class implement different instantiations of the same generic interface?)

## What happens if a class implements two parameterized interfaces that both define a method with the same name?

*If the two method have the same erasure then the class is illegal and rejected with a compile-time error message.*

If, after type erasure, two inherited methods happen to have the same erasure, then the compiler issues an error message.

Example (of illegal class definition; before type erasure):

```
interface Equivalent<T> {
    boolean equalTo(T other);
}
interface EqualityComparable<T> {
    boolean equalTo(T other);
}
class SomeClass implements Equivalent<Double>, EqualityComparable<SomeClass> { // error
    public boolean equalTo(Double other) { ... }
    public boolean equalTo(SomeClass other) { ... }
}
```

---

error: name clash: equalTo(T) in EqualityComparable<SomeClass> and equalTo(T) in Equivalent<java.lang.String> have the same erasure, yet neither overrides the other

```
class SomeClass implements EqualityComparable<SomeClass>, Equivalent<Double> {
    ^
```

During type erasure the compiler does not only create the type erased versions of the two colliding interfaces, it would also try to create the necessary bridge methods. Bridge methods are synthetic methods generated by the compiler when a class has a parameterized supertype.

Example (after a conceivable translation by type erasure):

```
interface Equivalent {
    boolean equalTo(Object other);
}
interface EqualityComparable {
    boolean equalTo(Object other);
}
```

```

class SomeClass implements Equivalent, EqualityComparable {
    public boolean equalTo(Double other)    { ... }
    public boolean equalTo(Object other)   { return equalTo((Double)other); }
    public boolean equalTo(SomeClass other) { ... }
    public boolean equalTo(Object other)   { return equalTo((SomeClass)other); }
}

```

The bridge methods would have the same signature. Instead of resolving the conflict the compiler reports an error.

By the way, the problem is *not* that the class has several overloaded versions of the `equalTo` method. The problem stems from the fact that the interfaces are generic and the methods have the same type erasure. No problem occurs when the two interfaces have no type parameter.

Example (of legal class definition):

```

interface Equivalent {
    boolean equalTo(Double other);
}
interface EqualityComparable {
    boolean equalTo(SomeClass other);
}
class SomeClass implements Equivalent, EqualityComparable {
    public boolean equalTo(Double other) { ... }
    public boolean equalTo(SomeClass other) { ... }
}

```

In the example above the compiler need not generate any bridge methods because the interfaces are not generic.

Note, that there is no problem if the two interfaces are generic and the conflicting methods have *different type erasures*.

Example (of legal class definition):

```

interface Equivalent<T extends Number> {
    boolean equalTo(T other);
}
interface EqualityComparable<T> {
    boolean equalTo(T other);
}
class SomeClass implements Equivalent<Double>, EqualityComparable<SomeClass> {
    public boolean equalTo(Double other) { ... }
    public boolean equalTo(SomeClass other) { ... }
}

```

Example (after a conceivable translation by type erasure):

```

interface Equivalent {
    boolean equalTo(Number other);
}
interface EqualityComparable {
    boolean equalTo(Object other);
}
class SomeClass implements Equivalent, EqualityComparable {
    public boolean equalTo(Double other)    { ... }
    public boolean equalTo(Number other)   { return equalTo((Double)other); }
    public boolean equalTo(SomeClass other) { ... }
    public boolean equalTo(Object other)   { return equalTo((SomeClass)other); }
}

```

The two `equalTo` methods have different erasures and then the bridge method generation mechanism create two bridge methods with different signatures and no problem occurs.

Effects similar to ones illustrated above can be observed with a parameterized superclass and a parameterized interface if they have a method with the same type erasure.

Last but not least, a legal way of implementing two interfaces with methods that have the same type erasure: as long as the colliding methods are instantiated for the same type argument there is no problem at all.

Example (of legal class definition):

```

class SomeClass implements Equivalent<SomeClass>, EqualityComparable<SomeClass> {
    public boolean equalTo(SomeClass other) { ... }
}

```

The class provide exactly one method, namely the matching one from both interfaces and the compiler generates one synthetic bridge method. No problem.

Example (after type erasure):

```

class SomeClass implements Equivalent, EqualityComparable {

```

```

    public boolean equalTo(SomeClass other) { ... }
    public boolean equalTo(Object other) { return equalTo((SomeClass)other); }
}

```

LINK TO THIS

[Practicalities.FAQ403](#)

REFERENCES

[What is type erasure?](#)  
[What is a bridge method?](#)

## Can an interface type nested into a generic type use the enclosing type's type parameters?

*No, but as workaround you can generify the nested interface itself.*

Nested interfaces are implicitly static. This is sometimes overlooked because the interface looks like it were a non-static member of its enclosing class, while in fact it is static. Since type parameters must not be used in any static context of a generic type, a nested interface cannot use its enclosing type's type parameters.

Example (of a nested interface):

```

interface Action {
    void run();
}
final class SomeAction implements Action {
    public void run() { &hellip; }
}
final class Controller<A extends Action>{
    public interface Command {
        void doIt(A action);    // error
        void undoIt(A action); // error
    }
    &hellip;
}

```

```

error: non-static class A cannot be referenced from a static context
    void doIt(A action);
           ^

```

```

error: non-static class A cannot be referenced from a static context
    void undoIt(A action);
                ^

```

The `Command` interface is nested into the generic `Controller` class. Inside the nested interface we cannot refer to the type parameter `A` of the enclosing class, because the nested interface is implicitly static and type parameters must not appear in any static context.

So, how do we express that the `Command` interface mandates do/undo methods for different types of actions? The solution is to generify the interface itself independently of the generic enclosing class.

Example (same as above, but corrected):

```

interface Action {
    void run();
}
final class SomeAction implements Action {
    public void run() { &hellip; }
}
final class Controller<A extends Action> {
    public interface Command<B extends Action> {
        void doIt(B action);
        void undoIt(B action);
    }
    &hellip;
}

```

LINK TO THIS

[Practicalities.FAQ404](#)

REFERENCES

[Why can't I use a type parameter in any static context of the generic class?](#)  
[How do I refer to an interface type nested into a generic type?](#)

## Implementing Infrastructure Methods

### How do I best implement the equals method of a generic type?

**Override `Object.equals(Object)` as usual and perform the type check using the unbounded wildcard instantiation.**

The recommended implementation of the `equals` method of a generic type looks like the one shown in the example below. Conceivable alternatives are discussed and evaluated later.

Example (recommended implementation of `equals`):

```
class Triple<T> {
    private T fst, snd, trd;
    public Triple(T t1, T t2, T t3) {fst = t1; snd = t2; trd = t3;}
    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (this.getClass() != other.getClass()) return false;
        Triple<?> otherTriple = (Triple<?>)other;
        return (this.fst.equals(otherTriple.fst)
            && this.snd.equals(otherTriple.snd)
            && this.trd.equals(otherTriple.trd));
    }
}
```

Perhaps the greatest difficulty is the downcast to the triple type, after the check for type match has been passed successfully. The most natural approach would be a cast to `Triple<T>`, because only objects of the same type are comparable to each other.

Example (not recommended):

```
class Triple<T> {
    private T fst, snd, trd;
    public Triple(T t1, T t2, T t3) {fst = t1; snd = t2; trd = t3;}
    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (this.getClass() != other.getClass()) return false;
        Triple<T> otherTriple = (Triple<T>)other; // unchecked warning
        return (this.fst.equals(otherTriple.fst)
            && this.snd.equals(otherTriple.snd)
            && this.trd.equals(otherTriple.trd));
    }
}
```

The cast to `Triple<T>` results in an "unchecked cast" warning, because the target type of the cast is a parameterized type. Only the cast to `Triple<?>` is accepted without a warning. Let us try out a cast to `Triple<?>` instead of `Triple<T>`.

Example (better, but does not compile):

```
class Triple<T> {
    private T fst, snd, trd;
    public Triple(T t1, T t2, T t3) {fst = t1; snd = t2; trd = t3;}
    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (this.getClass() != other.getClass()) return false;
        Triple<T> otherTriple = (Triple<?>)other; // error
        return (this.fst.equals(otherTriple.fst)
            && this.snd.equals(otherTriple.snd)
            && this.trd.equals(otherTriple.trd));
    }
}
```

---

```
error: incompatible types
found   : Triple<capture of ?>
required: Triple<T>
        Triple<T> otherTriple = (Triple<?>)other;
                                   ^
```

This implementation avoids the "unchecked" cast, but does not compile because the compiler refuses to assign a `Triple<?>` to a `Triple<T>`. This is because the compiler cannot ensure that the unbounded wildcard parameterized type `Triple<?>` matches the concrete parameterized type `Triple<T>`. To make it compile we have to change the type of the local variable `otherTriple` from `Triple<T>` to `Triple<?>`. This change leads us to the first implementation shown in this FAQ entry, which is the recommended way of implementing the `equals` method of a generic type.



## Evaluation of the alternative implementations.

How do the two alternative implementations, the recommended one casting to `Triple<?>` and the not recommended one casting to `Triple<T>`, compare? The recommended implementation compiles without warnings, which is clearly preferable when we strive for warning-free compilation of our programs. Otherwise there is no difference in functionality or behavior, despite of the different cast expressions in the source code. At runtime both casts boils down to a cast to the raw type `Triple`.

If there is no difference in functionality and behavior and one of the implementations raises a warning, isn't there a type-safety problem? After all, "unchecked" warnings are issued to alert the programmer to potentially unsafe code. It turns out that in this particular cases all is fine. Let us see why.

With both implementations of `equals` it might happen that triples of different member types, like a `Triple<String>` and a `Triple<Number>`, pass the check for type match via `getClass()` and the cast to `Triple<?>` (or `Triple<T>`). We would then compare members of different type with each other. For instance, if a `Triple<String>` and a `Triple<Number>` are compared, they would pass the type check, because they are both triples and we would eventually compare the `Number` members with the `String` members. Fortunately, the comparison of a `String` and a `Number` always yields `false`, because both `String.equals` and `Number.equals` return `false` in case of comparison with an object of an incompatible type.

In general, every implementation of an `equals` method is responsible for performing a check for type match and to return `false` in case of mismatch. This rule is still valid, even in the presence of Java generics, because the signature of `equals` is still the same as in pre-generic Java: the `equals` method takes an `Object` as an argument. Hence, the argument can be of any reference type and the implementation of `equals` must check whether the argument is of an acceptable type so that the actual comparison for equality makes sense and can be performed.

## Yet another alternative.

It might seem natural to provide an `equals` method that has a more specific signature, such as a version of `equals` in class `Triple` that takes a `Triple<T>` as an argument. This way we would not need a type check in the first place. The crux is that a version of `equals` that takes a `Triple<T>` as an argument would not be an overriding version of `Object.equals(Object)`, because the `equals` method in `Object` is not generic and the compiler would not generate the necessary bridge methods. We would have to provide the bridge method ourselves, which again would result in an "unchecked" warning.

Example (not recommended):

```
class Triple<T> {
    private T fst, snd, trd;
    public Triple(T t1, T t2, T t3) {fst = t1; snd = t2; trd = t3;}
    ...
    public boolean equals(Triple<T> other) {
        if (this == other) return true;
        if (other == null) return false;
        return (this.fst.equals(other.fst)
            && this.snd.equals(other.snd)
            && this.trd.equals(other.trd));
    }
    public boolean equals(Object other) {
        return equals((Triple<?>) other);    // unchecked warning
    }
}
```

This implementation has the flaw of raising an "unchecked" warning and offers no advantage of the recommended implementation to make up for this flaw.

LINK TO THIS [Practicalities.FAQ501](#)

REFERENCES [What is a bridge method?](#)  
[What is an "unchecked" warning?](#)  
[What is the capture of a wildcard?](#)  
[What is a wildcard capture assignment-compatible to?](#)

---

## How do I best implement the clone method of a generic type?

*Override `Object.clone()` as usual and ignore the inevitable unchecked warnings.*

The recommended implementation of the `clone` method of a generic type looks like the one shown in the example below.

Example (implementation of `clone`):

```
class Triple<T> implements Cloneable {
    private T fst, snd, trd;
    public Triple(T t1, T t2, T t3) {fst = t1; snd = t2; trd = t3;}
    ...
    public Triple<T> clone() {
        Triple<T> clon = null;
        try {
```

```

    clon = (Triple<T>)super.clone(); // unchecked warning
} catch (CloneNotSupportedException e) {
    throw new InternalError();
}
try {
    Class<?> clzz = this.fst.getClass();
    Method meth = clzz.getMethod("clone", new Class[0]);
    Object dupl = meth.invoke(this.fst, new Object[0]);
    clon.fst = (T)dupl; // unchecked warning
} catch (Exception e) {
    ...
}
try {
    Class<?> clzz = this.snd.getClass();
    Method meth = clzz.getMethod("clone", new Class[0]);
    Object dupl = meth.invoke(this.snd, new Object[0]);
    clon.snd = (T)dupl; // unchecked warning
} catch (Exception e) {
    ...
}
try {
    Class<?> clzz = this.trd.getClass();
    Method meth = clzz.getMethod("clone", new Class[0]);
    Object dupl = meth.invoke(this.trd, new Object[0]);
    clon.trd = (T)dupl; // unchecked warning
} catch (Exception e) {
    ...
}
return clon;
}
}

```

*Return type.*

In our implementation we declared the return type of the `clone` method not as type `Object`, but of the more specific generic type. This is possible, since the overriding rules have been relaxed and an overriding method in a subclass need no longer have the exact same signature as the superclass's method that it overrides. Since Java 5.0 it is permitted that the subclass version of a method returns a type that is a subtype of the return type of the superclass's method. In our example, the method `clone` in class `Triple<T>` returns a `Triple<T>` and overrides the `clone` method in class `Object`, which returns an `Object`.

The more specific return type is largely a matter of taste. One might equally well stick to the traditional technique of declaring the return type of all `clone` methods as type `Object`. The more specific return type is beneficial for the users of our triple class, because it saves them a cast from `Object` down to `Triple<T>` after a call to `Triple<T>.clone`.

*"unchecked cast" warnings.*

The most annoying aspect of implementing `clone` for a generic type are the inevitable "unchecked" warnings. The warning stem from two categories of casts that are needed.

- Casting the result of `super.clone` to the generic type.
- Casting the result of cloning any fields to the type that the type parameter stands for.

*Casting the result of `super.clone` to the generic type.*

Part of every implementation of `clone` is the invocation of the superclass's `clone` method. The result of `super.clone` is either of the supertype itself or of type `Object`. In our example `super.clone` is `Object.clone`, whose return type is `Object`. In order to access the fields of the clone returned from `super.clone` a cast to own type is needed. In our example this is a cast to the type `Triple<T>`. The target type of this cast is the generic type itself and the compiler issues the usual "unchecked cast" warning.

In some cases the cast is not needed at all, namely when the clone produced by `super.clone` is already deep enough so that the fields of the clone need not be accessed. This would be the case if all fields are either of primitive type or of an immutable reference type.

In all other cases, there is no way to avoid the unchecked warning. A cast to `Triple<?>` instead of `Triple<T>` would eliminate the unchecked warning, but does not give the required access to the fields. The two fields in our example would be of type "capture of ?" to which we cannot assign the result of cloning the individual fields. Alternatively we might consider a cast to the raw type `Triple` instead of `Triple<T>`, but that would give us "unchecked assignment" warnings instead of "unchecked cast" warnings. The compiler would issue the warnings when we access the fields of our raw triple class. No matter how we put it, we cannot avoid the unchecked warnings the cast after `super.clone`. The warnings are harmless and hence best suppressed by means of the standard annotation `@annotation.SuppressWarnings`.

*Cloning the individual fields.*

We must invoke the fields' clone method via reflection because we do not know whether the respective field has an accessible `clone` method. Two factor play a role:

- Every class inherits a `clone` method from class `Object`, but `Object.clone` is a protected method and for this reason not part of the public interface of a class. In essence, all classes have a `clone` method, but only a private one, unless they explicitly provide a public `clone`

method.

- Most classes that have a `clone` method also implement the `Cloneable` interface. The `Cloneable` interface is an empty marker interface and does not mandate that a `Cloneable` class must have a public `clone` method. Even if we could successfully cast down to `Cloneable` we would not have access to a `clone` method. Hence, for purposes of invoking a `clone` method the `Cloneable` interface is totally irrelevant.

In the example we use reflection to find out whether the field has a public `clone` method. If it has a `clone` method, we invoke it.

*Casting the result of cloning any fields to the type that the type parameter stands for.*

If individual fields must be cloned, the `clone` method of the respective fields' type must be invoked. The result of this invocation of the `clone` method is often type `Object`, so that another cast is necessary. If the field in question has the type that the enclosing class's type parameter stands for then the target of this cast is the type variable and the compiler issues the usual "unchecked cast" warning. In our example we must clone the two fields of the unknown type `T`, which requires that we invoke the field's `clone` method via reflection. The result of the reflective call is of type `Object` and we must cast from `Object` to the type parameter `T`. Again, there is no way to avoid the unchecked casts after cloning the fields and the warnings are best suppressed by means of the standard annotation `@annotation.SuppressWarnings`.

*More "unchecked" warnings.*

If a class has fields that are of a parameterized type and these fields must be cloned then a cast from `Object` to the parameterized type might be necessary and the compiler issues the usual "unchecked cast" warning.

Example:

```
class Store {
    private ArrayList<String> store = new ArrayList<String>();
    ...
    public Store clone() {
        Store clon = (Store)super.clone();
        clon.store = (ArrayList<String>)this.store.clone(); // unchecked warning
    }
}
```

Again there is no chance to avoid the "unchecked cast" warnings and they are best suppressed by means of the standard annotation `@annotation.SuppressWarnings`.

The reason for the undesired unchecked warnings in conjunction with the `clone` method stem from the fact that the `clone` method is a non-generic legacy method. In situations where generic and non-generic code is mixed, unchecked warnings cannot be avoided.

*Exception Handling.*

In the example, we left open how the exceptions from reflective invocation of the members' `clone` methods should be handled. Should we suppress the exceptions, or should we map them to a `CloneNotSupportedException`, or perhaps simply propagate the exceptions to the caller?

Example (excerpt from implementation of `clone`):

```
public Triple<T> clone() {
    ...
    try {
        Class<?> clzz = this.fst.getClass();
        Method meth = clzz.getMethod("clone", new Class[0]);
        Object dupl = meth.invoke(this.fst, new Object[0]);
        clon.fst = (T)dupl;
    } catch (Exception e) {
        ... ??? what should be done here ??? ...
    }
    ...
}
```

Usually, a `clone` method does not throw any exceptions; at least it does not through a `CloneNotSupportedException`. The point in implementing a `clone` method is to support cloning. Why should a `clone` method throw a `CloneNotSupportedException` then? It is equally unusual that a `clone` method would throw any other exception, because a class knows its fields and their types well enough to successfully produce a clone of each field.

For a generic class the situation is more complex. We do not know anything about those fields of the class whose type is a type parameter. In particular, we do not know whether those fields are `Cloneable` and/or have a `clone` method, as was explained above. The attempted invocation of the members' `clone` method via reflection bears the risk of failure, indicated by a number of exceptions raised by `Class.getMethod` and `Method.invoke` such as `NoSuchMethodException`, `IllegalArgumentException`, etc. In this situation the `clone` method might in fact fail to produce a clone and it might make sense to indicate this failure by mapping all (or some) exceptions to a `CloneNotSupportedException`.

Example (throwing a `CloneNotSupportedException`):

```
public Triple<T> clone() throws CloneNotSupportedException{
    ...
    try {
        Class<?> clzz = this.fst.getClass();
        Method meth = clzz.getMethod("clone", new Class[0]);
        Object dupl = meth.invoke(this.fst, new Object[0]);
    }
}
```

```

        clon.fst = (T)dupl;
    } catch (Exception e) {
        throw new CloneNotSupportedException(e.toString());
    }
    ...
}

```

On the other hand, one might argue that a type that does not have a clone method probably needs no cloning because objects of the type can safely be referenced from many other objects at the same time. Class `String` is an example. Class `String` is neither `Cloneable` nor has it a `clone` method. Class `String` does not support the cloning feature, because `String` objects are immutable, that is, they cannot be modified. An immutable object is never copied, but simply shared among all objects that hold a reference to it. With our exception handling above the `clone` method of a `Triple<String>` would throw a `CloneNotSupportedException`, which is not quite appropriate. It would be preferable to let the original triple and its clone hold references to the shared string members.

Example (suppressing the `NoSuchMethodException`):

```

public Triple<T> clone() {
    ...
    try {
        Class<?> clzz = this.fst.getClass();
        Method meth = clzz.getMethod("clone", new Class[0]);
        Object dupl = meth.invoke(this.fst, new Object[0]);
        clon.fst = (T)dupl;
    } catch (NoSuchMethodException e) {
        // exception suppressed
    } catch (Exception e) {
        throw new InternalError(e.toString());
    }
    ...
}

```

In the exception handling suggested above we suppress the `NoSuchMethodException` under the assumption that an object without a `clone` method need not be cloned, but can be shared.

Note, that we cannot ascertain statically by means of type argument bounds, that the members of a triple have a `clone` method. We could define the type parameter with `Cloneable` as a bound, that is, as `class Triple<T extends Cloneable>`, but that would not avoid any of the issues discussed above. The `Cloneable` interface is an empty tagging interface and does not demand that a cloneable type has a `clone` method. We would still have to invoke the `clone` method via reflection and face the exception handling issues as before.

LINK TO THIS [Practicalities.FAQ502](#)

REFERENCES [What is an "unchecked" warning?](#)  
[What is the SuppressWarnings annotation?](#)

## Using Runtime Type Information

### What does the type parameter of class `java.lang.Class` mean?

*The type parameter is the type that the class object represents, e.g. `Class<String>` represents `String`.*

An object of type `java.lang.Class` represents the runtime type of an object. Such a `Class` object is usually obtained via the `getClass` method defined in class `Object`. Alternative ways of obtaining a `Class` object representing a certain type are use of a class literal or the static method `forName` defined in class `Class`.

Since Java 5.0 class `java.lang.Class` is a generic class with one unbounded type parameter. The type parameter is the type that the `Class` object represents. For instance, type `Number` is represented by a `Class` object of type `Class<Number>`, type `String` by a `Class` object of type `Class<String>`, and so forth.

Parameterized types share the same runtime type and as a result they are represented by the same `Class` object, namely the `Class` object that represents the raw type. For instance, all instantiations of `List`, such as `List<Long>`, `List<String>`, `List<?>`, and the raw type `List` itself are represented by the same `Class` object; this `Class` object is of type `Class<List>`.

In general, the type argument of a `Class` object's type is the erasure of the type that the `Class` object represents.

Note that the methods `Object.getClass` and `Class.forName` return references of a wildcard type. A side effect is that they cannot be assigned to a `Class` object of the actual type.

Example (using `Class` objects):

```

Number n = new Long(0L);
Class<Number> c1 = Number.class;
Class<Number> c2 = Class.forName("java.lang.Number"); // error
Class<Number> c3 = n.getClass(); // error

```

The `forName` method returns a reference of type `Class<?>`, not of type `Class<Number>`. Returning an object of any `Class` type makes sense because the method can return a `Class` object representing any type.

The `getClass` method returns a reference of type `Class<? extends X>`, where `X` is the erasure of the static type of the expression on which `getClass` is called. Returning `Class<? extends X>` makes sense because the type `X` might be a supertype referring to a subtype object. The `getClass` method would then return the runtime type representation of the subclass and not the representation of the supertype. In the example above the reference of type `Number` refers to an object of type `Long`, so that the `getClass` method returns a `Class` object of type `Class<Long>` instead of `Class<Number>`.

Example (corrected):

```

Number n = new Long(0L);
Class<Number> c1 = Number.class;
Class<?> c2 = Class.forName("java.lang.Number");
Class<? extends Number> c3 = n.getClass();

```

The easiest way of passing around type representations is via a reference of type `Class<?>`.

LINK TO THIS [Practicalities.FAQ601](#)

REFERENCES [What is type erasure?](#)  
[How do I pass type information to a method so that it can be used at runtime?](#)

## How do I pass type information to a method so that it can be used at runtime?

*By means of a class object.*

The type information that is provided by a type parameter is static type information that is no longer available at runtime. When we need type information that is available at runtime we must explicitly supply the runtime time information to the method. Below are a couple of situations where the static type information provided by a type parameter does not suffice.

Example (of illegal or pointless use of type parameter):

```

public static <T> void someMethod() {
    ... new T() ... // error
    ... new T[SIZE] ... // error
    ... ref instanceof T ... // error
    ... (T) ref ... // unchecked warning
}

```

```
Utilities.<String>someMethod();
```

The type parameter `T` of the method does not provide any type information that would still be accessible at runtime. At runtime the type parameter is represented by the raw type of it leftmost bound or type `Object`, if no bound was specified. For this reason, the compiler refuses the accept type parameters in `new` expressions, and type checks based on the type parameter are either illegal or nonsensical.

If we really need runtime type information we must pass it to the method explicitly. There are 3 techniques for supplying runtime type information to a method:

- supply an object
- supply an array
- supply a `Class` object

The 3 alternative implementations of the method above would look like this:

Example (of passing runtime type information):

```

public static <T> void someMethod(T dummy) {
    Class<?> type = dummy.getClass();
    ... use type reflectively ...
}
public static <T> void someMethod(T[] dummy) {
    ... use type reflectively ...
    Class<?> type = dummy.getClass().getComponentType();
}
public static <T> void someMethod(Class<T> type) {
    ... use type reflectively ...
    ... (T)type.newInstance() ...
}

```

```

... (T[])Array.newInstance(type,SIZE) ...
... type.isInstance(ref) ...
... type.cast(tmp) ...
}

```

---

```

Utilities.someMethod(new String());
Utilities.someMethod(new String[0]);
Utilities.someMethod(String.class);

```

The first two alternatives are wasteful, because dummy objects must be created for the sole purpose of supplying their type information. In addition, the first approach does not work when an abstract class or an interface must be represented, because no objects of these types can be created.

The second technique is the classic approach; it is the one taken by the `toArray` methods of the collection classes in package `java.util` (see [java.util.Collection.toArray\(T\[\]\)](#)).

The third alternative is the recommended technique. It provides runtime type information by means of a `Class` object.

Here are the corresponding operations based on the runtime type information from the example above, this time performed using reflection.

Example (of reflective use of runtime type information):

```

public static <T> void someMethod(Class<T> type) {
... (T)type.newInstance() ...
... (T[])Array.newInstance(type,SIZE) ...
... type.isInstance(ref) ...
... type.cast(tmp) ...
}

```

Examples using class `Class` to provide type information can be found in the subsequent two FAQ entries (see REFERENCES or click [here](#) and [here](#)).

LINK TO THIS [Practicalities.FAQ602](#)

REFERENCES [What does the type parameter of class java.lang.Class mean?](#)  
[How do I generically create objects and arrays?](#)  
[How do I perform a runtime type check whose target type is a type parameter?](#)

## How do I generically create objects and arrays?

### *Using reflection.*

The type information that is provided by a type parameter is static type information that is no longer available at runtime. It does not permit generic creation of objects or arrays.

Example (of failed generic array creation based on static type information):

```

class Utilities {
private static final int SIZE = 1024;

public static <T> T[] createBuffer() {
return new T[SIZE]; // error
}
}

public static void main(String[] args) {
String[] buffer = Utilities.<String>createBuffer();
}

```

The type parameter `T` of method `createBuffer` does not provide any type information that would still be accessible at runtime. At runtime the type parameter is represented by the raw type of it leftmost bound or type `Object`, if no bound was specified. For this reason, the compiler refuses to accept type parameters in `new` expressions.

If we need to generically create an object or array, then we must pass type information to the `createBuffer` method that persists until runtime. This runtime type information can then be used to perform the generic object or array creation via reflection. The type information is best supplied by means of a `Class` object. (A `Class` object used this way is occasionally called a *type token*.)

Example (of generic array creation based on runtime type information):

```

public static <T> T[] createBuffer(Class<T> type) {
return (T[])Array.newInstance(type,SIZE);
}

public static void main(String[] args) {
String[] buffer = Utilities.createBuffer(String.class);
}

```

Note that the parameterization of class `Class` allows to ensure at compile time that no arbitrary types of `Class` objects are passed to the `createBuffer` method. Only a `Class` object that represents a runtime type that matches the desired component type of the created array is

permitted.

Example:

```
String[] buffer = Utilities.createBuffer(String.class);
String[] buffer = Utilities.createBuffer(Long.class); // error
Number[] buffer = Utilities.createBuffer(Long.class);
```

---

Note also, that arrays of primitive type elements cannot be created using the aforementioned technique.

Example (of a failed attempt to create an array of primitive type):

```
class Utilities {
    @SuppressWarnings("unchecked")
    public static <T> T[] slice(T[] src, Class<T> type, int start, int length) {
        T[] result = (T[])Array.newInstance(type,length);
        System.arraycopy(src, start, result, 0, length);
        return result;
    }
}
class Test {
    public static void main(String[] args) {
        double[] avg = new double[]{1.0, 2.0, 3.0};
        double[] res = Utilities.slice(avg, double.class, 0, 2); // error
    }
}
```

---

```
error: <T>slice(T[],java.lang.Class<T>,int,int) cannot be applied to
(double[],java.lang.Class<java.lang.Double>,int,int)
    double[] res = Utilities.slice(avg, double.class, 0, 2);
                                ^
```

Since primitive types are not permitted as type arguments, we cannot invoke the `slice` method using `double.class` as the type token. The compiler would have to infer `T:=double`, which is not permitted because `double` is a primitive type and cannot be used as the type argument of a generic method. The `slice` method can only create arrays of reference type elements, which means that we have to convert back and forth between `double[]` and `Double[]` in the example.

Example (of a successful attempt to create an array of reference type):

```
class Test {
    public static void main(String[] args) {
        double[] avg = new double[]{1.0, 2.0, 3.0};
        Double[] avgdup = new Double[avg.length];
        for (int i=0; i<avg.length;i++) avgdup[i] = avg[i]; // auto-boxing
        Double[] tmp = Utilities.slice(avgdup, Double.class, 0, 2); // fine
        avg = new double[tmp.length];
        for (int i=0; i<tmp.length;i++) avg[i] = tmp[i]; // auto-unboxing
    }
}
```

LINK TO THIS [Practicalities.FAQ603](#)

REFERENCES [What does the type parameter of class java.lang.Class mean?](#)  
[How do I pass type information to a method so that it can be used at runtime?](#)  
[Are primitive types permitted as type arguments?](#)

---

## How do I perform a runtime type check whose target type is a type parameter?

### *Using reflection.*

The type information that is provided by a type parameter is static type information that is no longer available at runtime. It does not permit any generic type checks.

Consider a method that is supposed to extract from a sequence of objects of arbitrary types all elements of a particular type. Such a method must at runtime check for a match between the type of each element in the sequence and the specific type that it is looking for. This type check cannot be performed by means on the type parameter.

Example (of failed generic type check based on static type information):

```
class Utilities {
    public static <T> Collection<T> extract(Collection<?> src) {
        HashSet<T> dest = new HashSet<T>();
        for (Object o : src)
```



```

        if (o instanceof T) // error
            dest.add((T)o); // unchecked warning
    return dest;
}
}
public static void test(Collection<?> coll) {
    Collection<Integer> coll = Utilities.<Integer>extract(coll);
}

```

Type parameters are not permitted in instanceof expressions and the cast to the type parameter is nonsensical, because it is a cast to type Object after type erasure.

For a type check at runtime we must explicitly provide runtime type information so that we can perform the type check and cast by means of reflection. The type information is best supplied by means of a Class object.

Example (of generic type check based on runtime type information):

```

class Utilities {
    public static <T> Collection<T> extract(Collection<?> src, Class<T> type) {
        HashSet<T> dest = new HashSet<T>();
        for (Object o : src)
            if (type.isInstance(o))
                dest.add(type.cast(o));
        return dest;
    }
}
public static void test(Collection<?> coll) {
    Collection<Integer> coll = Utilities.extract(coll, Integer.class);
}

```

LINK TO THIS [Practicalities.FAQ604](#)

REFERENCES [What does the type parameter of class java.lang.Class mean?](#)  
[How do I pass type information to a method so that it can be used at runtime?](#)

## Reflection

### Which information related to generics can I access reflectively?

*The exact static type information, but only inexact dynamic type information.*

Using the reflection API of package `java.lang.reflect` you can access the exact declared type of fields, method parameters and method return values. However, you have no access to the exact dynamic type of an object that a reference variable refers to.

Below are a couple of examples that illustrate which information is available by means of reflection. Subsequent FAQ entries discuss in greater detail the ways and means of extracting the information. Here is the short version of how the static and dynamic type information is retrieved reflectively.

For illustration, we consider the field of a class:

Example (of a class with a field):

```

class SomeClass {
    static Object field = new ArrayList<String>();
    ...
}

```

The information regarding the declared type of a field (*static type information*) can be found like this:

Example (find declared type of a field):

```

class Test {
    public static void main(String[] args) {
        Field f = SomeClass.class.getDeclaredField("field");
        Type t = f.getGenericType();
    }
}

```

In order to retrieve the declared type of a field you need a representation of the field in question as an object of type `java.lang.reflect.Field`. Such a representation can be found by invoking either the method `getField()` or `getDeclaredField()` of class `java.lang.Class`. Class `java.lang.reflect.Field` has a method named `getGenericType()`; it returns an object of type `java.lang.reflect.Type`, which represents the declared type of the field.



The information regarding the type of the object that a reference refers to (*dynamic type information*) can be found like this:

Example (find actual type of a field):

```
class Test {
    public static void main(String[] args) {
        Class<?> c = SomeClass.field.getClass();
    }
}
```

In order to retrieve the actual type of an object you need a representation of its type as an object of type `java.lang.Class`. This type representation can be found by invoking the method `getClass()` of class `java.lang.Object`.

In the example above, the field `SomeClass.field` is declared as a field of type `Object`; for this reason `Field.getGenericType()` yields the type information `Object`. This is the static type information of the field as declared in the class definition.

At runtime the field variable `SomeClass.field` refers to an object of any subtype of `Object`. The actual type of the referenced object is retrieved using the object's `getClass()` method, which is defined in class `Object`. If the field refers to an object of type `ArrayList<String>` then `getClass()` yields the raw type information `ArrayList`, but not the exact type information `ArrayList<String>`.

The table below shows further examples of the type information that is available for the field of a class using `Field.getGenericType()` and `Object.getClass()`.

Declaration of Field (retrieved via <code>Class.getField()</code> )	Static Type Information (retrieved via <code>Field.getGenericType()</code> )	Dynamic Type Information (retrieved via <code>Object.getClass()</code> )
<pre>class SomeClass {     Object field     = new ArrayList&lt;String&gt;();     ... }</pre>	Object <i>regular type</i>	ArrayList <i>generic type</i>
<pre>class SomeClass {     List&lt;String&gt; field     = new ArrayList&lt;String&gt;();     ... }</pre>	List<String> <i>parameterized type</i>	ArrayList <i>generic type</i>
<pre>class SomeClass {     Set&lt;? extends Number&gt; field     = new TreeSet&lt;Long&gt;();     ... }</pre>	Set<? extends Number> <i>parameterized type</i>	TreeSet <i>generic type</i>
<pre>class SomeClass&lt;T&gt; {     T field;     SomeClass(T t) { field = t; }     ... } SomeClass&lt;CharSequence&gt; object = new SomeClass&lt;CharSequence&gt;("a");</pre>	T <i>type variable</i>	String <i>non-generic type</i>
<pre>class SomeClass {     Iterable&lt;?&gt;[] field     = new Collection&lt;?&gt;[0];     ... }</pre>	Iterable<?>[] <i>generic array type</i>	[LCollection <i>non-generic type</i>
<pre>class SomeClass&lt;T&gt; {     T[] array;     SomeClass(T... arg) { array = arg; }     ... } SomeClass&lt;String&gt; object = new SomeClass&lt;String&gt;("a");</pre>	T[] <i>generic array type</i>	[LString <i>non-generic type</i>

LINK TO THIS

[Practicalities.FAQ701](#)

REFERENCES

[java.lang.reflect.Field.getGenericType\(\)](#)  
[java.lang.Object.getClass\(\)](#)

## How do I retrieve an object's actual (dynamic) type?

*By calling its `getClass()` method.*

When you want to retrieve an object's actual type (as opposed to its declared type) you use a reference to the object in question and invoke its `getClass()` method.

Example (of retrieving an object's actual type):

```

class Test {
    public static void main(String[] args) {
        Object tmp = java.util.EnumSet.allOf(java.util.concurrent.TimeUnit.class);
        Class<?> clazz = tmp.getClass();
        System.out.println("actual type of Object tmp is: "+clazz);
    }
}

```

---

```

actual type of Object tmp is: class java.util.RegularEnumSet

```

The *actual* type of the object that the local `tmp` variable refers to is unknown at compile time. It is some class type that extends the abstract `EnumSet` class; we do not know which type exactly. It turns out that in our example the actual type is `java.util.RegularEnumSet`, which is an implementation specific class type defined by the JDK implementor. The class is a private implementation detail of the JDK and is not even mentioned in the API description of the `java.util` package. Nonetheless the virtual machine can retrieve the actual type of the object via reflection by means of the `getClass()` method.

In contrast, the *declared* type of the object in question is type `Object`, because the reference variable `tmp` is of type `Object`.

In this example the declared type is not available through reflection, because `tmp` is a local variable. The declared type is available reflectively solely for fields of types, and for return types or parameter types or exception types of methods. The actual type of an object, however, can be retrieved for all objects regardless of their declaration: for local variables, fields of classes, return types of methods, arguments passed to method, etc.

The `getClass()` method of class `Object` returns an object of type `java.lang.Class`, which means that the actual type of each object is represented by a `Class` object. You can extract various information about the type represented by the `Class` object, such as "is it a primitive type?", "is it an array type?", "is it an interface, or a class, or an enum type?", "which fields does the type have?", "which methods does the type have?", etc. You can additionally find out whether the `Class` object represents a generic type by asking it: "does it have type parameters?".

#### LINK TO THIS

[Practicalities.FAQ702](#)

#### REFERENCES

[How do I figure out whether a type is a generic type?](#)  
[What is a parameterized or generic type?](#)  
[How do I retrieve an object's declared type?](#)  
[java.lang.Class](#)

---

## How do I retrieve an object's declared (static) type?

*By finding the declaration's reflective representation and calling the appropriate `getGeneric...()` method.*

When you want to retrieve an object's declared type (as opposed to its actual type) you first need a representation of the declaration.

- *Field.* For a field of a type you need a representation of that field in terms of an object of type `java.lang.reflect.Field`. This can be obtained by one of the methods `getField()`, `getFields()`, `getDeclaredField()`, or `getDeclaredFields()` of class `Class`.
- *Return Value.* For the return value of a method you need a representation of the method in terms of an object of type `java.lang.reflect.Method`. This can be obtained by one of the methods `getMethod()`, `getMethods()`, `getDeclaredMethod()`, or `getDeclaredMethods()` of class `Class`. Then you invoke the `getGenericReturnType()` method of class `Method`.
- *Method Parameter.* Same as for the return value. Once you have a representation of the method, you invoke the `getGenericParameterTypes()` method of class `Method`.
- *Method Exceptions.* Same as for the return value. Once you have a representation of the method, you invoke the `getGenericExceptionTypes()` method of class `Method`.

Note, that there is no representation of the declaration of a local variable on the stack of a method. Only the declarations of fields declared in classes, interfaces or enumeration types, and return types, parameter types, and exception types of methods have a reflective representation.

Example (of retrieving a field's declared type):

```

class Test {
    private static EnumSet<TimeUnit> set = EnumSet.allOf(TimeUnit.class);

    public static void main(String[] args) {
        Field field = Test.class.getDeclaredField("set");
        Type type = field.getGenericType();
        System.out.println("declared type of field set is: "+type);
    }
}

```

---

```

declared type of field set is: java.util.EnumSet<java.util.concurrent.TimeUnit>

```

The declared return type, argument type, or exception type of a method is retrieved similarly by invoking the corresponding `getGeneric...Type()` method.

All these methods return an object of type `java.reflect.Type`, which means that the declared type of an object is represented by a `Type` object.

is an interface and represents all type-like constructs in Java reflection. It has five subtypes, as shown in the subsequent diagram.

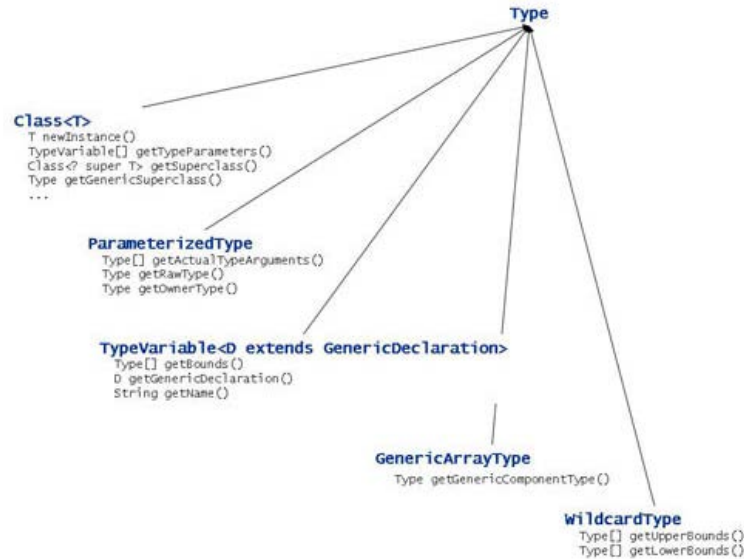


Figure: Subtypes of Interface `java.lang.reflect.Type`

As you can tell from the diagram, class `Class` is a subtype of interface `Type`, but it is not the only subtype. A `Type` can represent one of the following type-like things:

- A *regular type*. In this case the `Type` variable refers to a `Class` object. Examples of regular types are non-generic types such `String` or `CharSequence`, enumeration types such as `TimeUnit`, array types with regular component types such as `String[]` (but *not* `Class<?>[]`, because the component type is a parameterized type), and raw types such as `List` or `Set`. In other words, a regular type is a type that has nothing to do with generics.
- A *parameterized type*. In this case the `Type` variable refers to an object of the subtype `ParameterizedType`. Examples of parameterized types are `List<String>` or `Set<? extends Number>` or `Iterator<E>`, that is, all types that are instantiations of generic types and have type arguments.
- A *type variable*. In this case the `Type` variable refers to an object of the subtype `TypeVariable`. Examples of type variables are `T`, `E`, `K`, `V`, that is, the type parameters of generic types and generic methods.
- A *generic array type*. In this case the `Type` variable refers to an object of the subtype `GenericArrayType`. Examples of generic array types are `Class<?>[]` or `T[]` or `Future<Object>[]` or `Iterator<?>[][]`, that is, all array types with a non-regular component type.
- A *wildcard type*. In this case the `Type` variable refers to an object of the subtype `WildcardType`. Examples of wildcard types are `?` or `? extends Number` or `? super T`, that is, all wildcard expressions. If you retrieved the declared type of a field or the return type, argument type or exception type of a method, the resulting `Type` variable can never refer to a `WildcardType`, because wildcards are not types; they can only be used as type arguments. Hence, the subtype's name "wildcard type" is slightly misleading. Only when you retrieve the type argument of a parameterized type you might come across a `Type` variable that refers to a `WildcardType`. This would, for instance, happen if you ask for the type argument of the type `Class<?>`.

Extracting information from the `Type` object returned by `Field.getGenericType()` or a similar method is not as easy as it is to retrieve information from a `Class` object. When you have a `Class` variable you simply invoke methods of class `Class`. When you have a `Type` variable you cannot invoke any methods, because the `Type` interface is an empty interface. Before you can extract any information you must figure out to which of the 5 subtypes discussed above the `Type` variable refers. This is usually done by a cascade of `instanceof` tests.

Example (of analyzing `java.lang.reflect.Type`):

```
void analyzeType(Type type) {
    if (type instanceof Class) {
        // regular type, e.g. String or Date[]
    } else if (type instanceof ParameterizedType) {
        // parameterized type, e.g. List<String> or Set<? extends Number>
    } else if (type instanceof TypeVariable) {
        // type variable, e.g. T
    } else if (type instanceof GenericArrayType) {
        // generic array, e.g. List<?>[] or T[]
    } else if (type instanceof WildcardType) {
        // wildcard, e.g. ? extends Number or ? super Long
    } else {
        // we should never get here
        throw new InternalError("unknown type representation "+type);
    }
}
```

Once you know what subtype of type `Type` the variable refers to, you simply cast down to the respective subtype and start retrieving information by invocation of the subtype's methods. Just browse the respective type's JavaDoc; most methods are self-explanatory. Here are some examples:

- If it is a `Class` then you can pose the usual questions such as "are you a primitive type?", "are you an array type?", "are you an interface, or a class, or an enum type?", "which fields do you have?", "which methods do you have?", "do you have type parameters?", etc.

- If it is a `ParameterizedType` you can ask "what type arguments do you have?", "what is your raw type?", etc.
- If it is a `TypeVariable` you can ask "which bounds do you have?", "which generic type do you belong to?", etc.
- If it is a `GenericArrayType` you can ask "what is your component type?".
- If it is a `WildcardType` you can ask "what is your upper and lower bound?".

LINK TO THIS [Practicalities.FAQ703](#)

REFERENCES [How do I retrieve an object's actual type?](#)  
[java.lang.reflect.Type](#)

## What is the difference between a generic type and a parameterized type in reflection?

*A generic type is represented by a class object; a parameterized type is represented by a ParameterizedType object.*

Generic and parameterized types are easily confused when you access them via reflection.

- We say that a type is a *generic type* (as opposed to *non-generic type*) when it declares formal type parameters, that is, placeholders that can be replaced by type arguments. For instance, `java.util.List` is a generic type because it is declared as `interface List<E> { ... }` and has one type parameter `E`. In contrast, class `java.util.Date` is a non-generic type, because it is a plain, regular class type that does not have formal type parameters.
- We talk of a *parameterized type* (as opposed to a *raw type*) when we mean an instantiation of a generic type where the formal type parameters are replaced by actual type arguments. For instance, `List<String>` is a parameterized type where the type parameter `E` is replaced by `String`. In contrast, `List` is a raw type. The same is true for `Date`.

In order to illustrate the difference between generic and parameterized type, let us consider an example. Say, we want to retrieve the declared and actual type of the private field `header` of class `java.util.LinkedList`. The field is declared as follows:

```
public class LinkedList<E> {
    private transient Entry<E> header = new Entry<E>(null, null, null);
    ...
    private static class Entry<T> { ... }
}
```

where `Entry` is a nested generic type defined in class `LinkedList` and `E` is the `LinkedList`'s type parameter.

The `header` field's *declared* type is `Entry<E>` and its *actual* type is `Entry`. This might be confusing at first sight, because the `header` field is declared as field of type `Entry<E>` and it actually refers to an object of type `Entry<E>`. However, due to type erasure, actual types are always raw types, because type erasure drops all information regarding type arguments. This mismatch between declared type and actual type adds to the confusion regarding the distinction between parameterized and generic types.

In our example, the `header` field's *declared* type is `Entry<E>` and `Entry<E>` is a parameterized type (as opposed to a raw type). This is because `Entry<E>` is an instantiation of the generic type `Entry` rather than the raw type `Entry`.

The `header` field's *actual* type is the raw type `Entry` (as a side effect of type erasure) and `Entry` is a generic type (as opposed to a non-generic type). This is because class `Entry` has a formal type parameter `T`.

Declaration of Field (retrieved via <code>Class.getField()</code> )	Static Type Information (retrieved via <code>Field.getGenericType()</code> )	Dynamic Type Information (retrieved via <code>Object.getClass()</code> )
<pre>public class LinkedList&lt;E&gt; {     private transient <b>Entry&lt;E&gt;</b> header     = new <b>Entry&lt;E&gt;</b>(null, null, null);     ...     private static class Entry&lt;T&gt; { ... } }</pre>	<pre>Entry&lt;E&gt;      parameterized type</pre>	<pre>Entry      generic type</pre>

Let us consider another example. Say, we want to retrieve the declared and actual type of the public field `EMPTY_LIST` of class `Collections`. The field is declared as follows:

```
public class Collections {
    public static final List EMPTY_LIST = new EmptyList();
    ...
    private static class EmptyList extends AbstractList<Object> { ... }
}
```

where `EmptyList` is a nested type defined in class `Collections`.

The `EMPTY_LIST` field's *declared* type is `List` and its *actual* type is `LinkedList.EmptyList`.

The `EMPTY_LIST` field's *declared* type `List` is not a parameterized type, because it does not have any type arguments; it is a raw type. In turn, the raw type `List` is a generic type, because interface `List` has a formal type parameter `E`.

The `EMPTY_LIST` field's *actual* type `LinkedList.EmptyList` is a non-generic type (as opposed to a generic type), because it does not have any formal type parameters; it is just a plain, regular class type.

Declaration of Field	Static Type Information	Dynamic Type Information
----------------------	-------------------------	--------------------------

(retrieved via <code>Class.getField()</code> )	(retrieved via <code>Field.getGenericType()</code> )	(retrieved via <code>Object.getClass()</code> )
<pre>public class Collections {     public static final List EMPTY_LIST         = new EmptyList();     ...     private static class EmptyList         extends AbstractList&lt;Object&gt; { ... } }</pre>	List <i>regular (raw) type</i>	EmptyList <i>non-generic type</i>

The starting point for retrieving information regarding parameterized and generic types is different. Being generic or non-generic is a property of a type that is represented by a `Class` object. In contrast, whether a type is parameterized or raw is a property of a type represented by a `Type` object. As a result, we need a `Class` object to distinguish between *generic* or *non-generic* and we need a `Type` object to distinguish between *parameterized* and *raw*.

The method below distinguishes between a parameterized and a raw type. It needs a `Type` object for this distinction.

Example (of distinction between parameterized and raw type):

```
static boolean isParameterizedType(Type type) {
    if (type instanceof ParameterizedType)
        return true;
    else
        return false;
}
```

The methods below distinguish between a generic and a non-generic type. The distinction regarding generic and non-generic requires a `Class` object.

Example (of distinction between generic and non-generic type):

```
static boolean isGenericType(Class<?> clazz) {
    TypeVariable<?>[] params = clazz.getTypeParameters();
    if (params != null && params.length > 0) {
        return true;
    }
    else {
        return false;
    }
}

static boolean isGenericType(Type type) {
    if (type instanceof Class && isGenericType((Class<?>)type))
        return true;
    else
        return false;
}
```

The overloaded version of the method that takes a `Type` object delegates to the other version of the method that takes a `Class` object, because only `Class` objects provide the information whether the type in question has type parameters (i.e. is generic), or not.

LINK TO THIS      [Practicalities.FAQ704](#)

REFERENCES

- [What is a parameterized or generic type?](#)
- [Which information related to generics can I access reflectively?](#)
- [What is type erasure?](#)
- [How do I figure out whether a type is a generic type?](#)
- [Which information is available about a generic type?](#)
- [How do I figure out whether a type is a parameterized type?](#)
- [Which information is available about a parameterized type?](#)

## How do I figure out whether a type is a generic type?

*By asking it whether it has type parameters.*

When you have the type representation of a type in form of a `Class` object then you can find out whether the type represents a generic type by retrieving its type parameters. If it does not have any type parameters then the type is a non-generic type, otherwise it is a generic type. Here is an example:

Example (of distinction between generic and non-generic type):

```
Object object = new LinkedHashMap<String,Number>();
Class<?> clazz = object.getClass();
TypeVariable<?>[] params = clazz.getTypeParameters();
if (params != null && params.length > 0) {
    System.out.println(clazz + " is a GENERIC TYPE");
    // generic type, e.g. HashSet
```

```

}
else {
    System.out.println(clazz + " is a NON-GENERIC TYPE");
    // non-generic type, e.g. String
}

```

---

```

class java.util.LinkedHashMap is a GENERIC TYPE

```

We obtain the `Class` object by calling the `getClass()` of an object. The `Class` object represents the type `LinkedHashMap` in our example. Note that `getClass()` returns the actual dynamic type of an object and the actual dynamic type is always a raw type because of type erasure.

Then we retrieve the type parameters by calling `getTypeParameters()`. If type parameters are returned then the type is a generic type, otherwise it is non-generic.

**LINK TO THIS**            [Practicalities.FAQ705](#)

**REFERENCES**            [What is a parameterized or generic type?](#)  
[What is the difference between a generic type and a parameterized type in reflection?](#)

---

## Which information is available about a generic type?

*All the information that is available for regular types plus information about the generic type's type parameters.*

A generic type is represented by a `Class` object. For this reason we can retrieve all the information about a generic type that is also available for regular non-generic types, such as fields, methods, supertypes, modifiers, annotations, etc. Different from a non-generic type a generic type has type parameters. They can be retrieved by means of the `getTypeParameters()` method.

Let us take a look at an example, namely the generic class `EnumSet<E extends Enum<E>>`.

Example (of retrieving information about a generic type):

```

Object object = new EnumMap<TimeUnit,Number>(TimeUnit.class);
Class<?> clazz = object.getClass();
TypeVariable<?>[] params = clazz.getTypeParameters();
if (params != null && params.length > 0) {
    System.out.println(clazz + " is a GENERIC TYPE with "+params.length+" type parameters");
    System.out.println();

    for (TypeVariable<?> typeparam : params) {
        System.out.println("\t"+typeparam);
    }
}
else {
    System.out.println(clazz + " is a NON-GENERIC TYPE");
}

```

---

```

class java.util.EnumMap is a GENERIC TYPE with 2 type parameters
TYPE PARAMETERS:
K
V

```

**LINK TO THIS**            [Practicalities.FAQ706](#)

**REFERENCES**            [How do I figure out whether a type is a generic type?](#)  
[Which information is available about a type parameter?](#)  
[java.lang.Class](#)  
[java.lang.reflect.GenericDeclaration](#)  
[java.lang.reflect.Type](#)  
[java.lang.reflect.TypeVariable](#)

---

## How do I figure out whether a type is a parameterized type?

*By asking whether the type representation is a `ParameterizedType`.*

When you have the type representation of a type in form of a `Type` object then you can find out whether the type represents a parameterized type (as opposed to a raw type) by checking whether the type representation refers to an object of a type that implements the `ParameterizedType` interface. Here is an example:

Example (of distinction between parameterized and regular (raw) type):

```

Method method = EnumSet.class.getMethod("clone");
System.out.println("METHOD: "+method.toGenericString());
Type returnType = method.getGenericReturnType();
if (returnType instanceof ParameterizedType) {
    System.out.println(returnType + " is a PARAMETERIZED TYPE");
} else if (returnType instanceof Class) {
    System.out.println(returnType + " is a RAW TYPE");
} else {
    System.out.println(returnType + " is something else");
}

```

---

```

METHOD: public java.util.EnumSet<E> java.util.EnumSet.clone()
java.util.EnumSet<E> is a PARAMETERIZED TYPE

```

First we retrieve the declared return type of the `clone()` method of class `EnumSet`. by calling the `getGenericReturnType()` method of class `java.lang.reflect.Method`. The resulting `Type` object represents the `clone()` method's return type, which in our example is `EnumSet<E>`. Then we verify that the return type is a parameterized type by means of an `instanceof` test.

LINK TO THIS [Practicalities.FAQ707](#)

REFERENCES [Which information is available about a parameterized type?](#)

## Which information is available about a parameterized type?

*Information about the parameterized type's type arguments, its corresponding raw type, and its enclosing type if it is a nested type or inner class.*

A parameterized type is represented by a `ParameterizedType` object. A parameterized type has actual type arguments, a corresponding raw type, and you can find out which enclosing type the parameterized type belongs to if it is a nested type or inner class.

Let us take a look at an example, namely the parameterized type `EnumMap<K, V>`, which we retrieve as the return type of the `clone()` method of class `EnumMap`.

Example (of retrieving information about a parameterized type):

```

Method method = EnumMap.class.getMethod("clone");
System.out.println("METHOD: "+method.toGenericString());
Type returnType = method.getGenericReturnType();

if (returnType instanceof ParameterizedType) {
    System.out.println(returnType + " is a PARAMETERIZED TYPE");

    ParameterizedType type = (ParameterizedType) returnType;

    Type rawType = type.getRawType();
    System.out.println("raw type : " + rawType);

    Type ownerType = type.getOwnerType();
    System.out.println("owner type: " + ownerType
        + ((ownerType != null) ? " : ", i.e. is a top-level type"));

    Type[] typeArguments = type.getActualTypeArguments();
    System.out.println("actual type arguments: ");
    for (Type t : typeArguments)
        System.out.println("\t" + t);
}

```

---

```

METHOD: public java.util.EnumMap<K, V> java.util.EnumMap.clone()
java.util.EnumMap<K, V> is a PARAMETERIZED TYPE
raw type : class java.util.EnumMap
owner type: null, i.e. is a top-level type
actual type arguments:
K
V

```

LINK TO THIS [Practicalities.FAQ708](#)

REFERENCES [How do I figure out whether a type is a parameterized type?](#)  
[java.lang.reflect.ParameterizedType](#)

## How do I retrieve the representation of a generic method?



### ***By retrieving the type erasure of the generic method.***

Generic methods are retrieved like non-generic methods: the `getMethod()` method of class `Class` is invoked providing a description of the method's type signature, that is, the name of the method and the raw types of the parameter types. What we supply is a description of the method's type erasure; we need not specify in any way, that the method is a generic method.

As an example let us retrieve the representation of the generic `toArray()` method of interface `Collection`. It is declared as:

```
interface Collection<E> {  
    ...  
    <T> T[] toArray(T[] a) { ... }  
}
```

Example (of retrieving the representation of a generic method):

```
Method method = Collection.class.getMethod("toArray", Object[].class);  
System.out.println("METHOD: "+method.toGenericString());
```

---

```
METHOD: public abstract <T> T[] java.util.Collection.toArray(T[])
```

Note, that we did not mention whether we are looking for a generic or a non-generic method. We just supplied the method name "toArray" and specified its parameter type as `Object[]`, which is the type erasure of the declared parameter type `T[]`.

Note, that there is some minor potential for confusion regarding the method description that is delivered by the resulting `Method` object. In the example above, we retrieved the method description using the `toGenericString()` method of class `Method`.

```
System.out.println("METHOD: "+method.toGenericString());
```

---

```
METHOD: public abstract <T> T[] java.util.Collection.toArray(T[])
```

It describes the generic method's signature including information regarding its type parameter `T`. Had we used the `toString()` method instead, the resulting method description had described the type erasure of the method.

```
System.out.println("METHOD: "+method.toString());
```

---

```
METHOD: public abstract java.lang.Object[] java.util.Collection.toArray(java.lang.Object[])
```

The confusing element here is the fact that `toString()` does not deliver a description of the method as it is declared, but of its type erasure.

**LINK TO THIS**                      [Practicalities.FAQ709](#)

**REFERENCES**                      [How do I figure out whether a method is a generic method?](#)  
[What is a generic declaration?](#)

---

## **How do I figure out whether a method is a generic method?**

### ***By asking it whether it has type parameters.***

Starting with the reflective representation of a method in form of a `Method` object you can find out whether the method is generic or non-generic by retrieving its type parameters. (Note, we are looking for *type* parameters, not *method* parameters.) If the method does not have any type parameters then it is a non-generic method, otherwise it is a generic method. Here is an example:

Example (of distinction between generic and non-generic method):

```
Method method = Collection.class.getMethod("toArray", Object[].class);  
TypeVariable[] typeParams = method.getTypeParameters();  
if (typeParams!=null && typeParams.length>0) {  
    System.out.println(method.getName()+" is a GENERIC METHOD");  
} else {  
    System.out.println(method.getName() + " is a NON-GENERIC METHOD");  
}
```

---

```
toArray is a GENERIC METHOD
```

We obtain the `Method` object by calling the `getMethod()` method of the `Class` object that represents the type whose method we are looking for. In our example the `Method` object represents the generic `toArray()` of interface `Collection`.

Then we retrieve the type parameters by calling `getTypeParameters()`. If type parameters are returned then the method is a generic method, otherwise it is non-generic.

**LINK TO THIS**                      [Practicalities.FAQ710](#)

**REFERENCES**



## Which information is available about a generic method?

*All the information that is available for regular methods plus information about the generic method's type parameters.*

A generic method is represented by a `Method` object. For this reason we can retrieve all the information about a generic method that is also available for regular non-generic methods, such as return type, method parameter types, exception types, declaring class, modifiers, annotations, etc. Different from a non-generic method a generic method has type parameters. They can be retrieved by means of the `getTypeParameters()` method. Type parameters are represented by `TypeVariable` objects.

Let us take a look at an example, namely the generic method

```
<T extends Object & Comparable<? super T>> T Collections.max(Collection<? extends T>).
```

Example (of retrieving information about a generic method):

```
Method theMethod = Collections.class.getMethod("max",Collection.class);
System.out.println("analyzing method: ");
System.out.println(theMethod.toGenericString()+"\n");

TypeVariable[] typeParams = theMethod.getTypeParameters();
if (typeParams!=null && typeParams.length>0) {
    System.out.println("GENERIC METHOD");
    System.out.println("type parameters: ");
    for (TypeVariable v : typeParams) {
        System.out.println("\t"+v);
    }
} else {
    System.out.println("NON-GENERIC METHOD");
}
System.out.println();

Type type = theMethod.getGenericReturnType();
System.out.println("generic return type of method "+theMethod.getName()+": " + type);
System.out.println();

Type[] genParamTypes = theMethod.getGenericParameterTypes();
if (genParamTypes == null || genParamTypes.length == 0) {
    System.out.println("no parameters");
} else {
    System.out.println("generic parameter types: ");
    for (Type t : genParamTypes) {
        System.out.println("\t"+t);
    }
}
System.out.println();

Type[] genExcTypes = theMethod.getGenericExceptionTypes();
if (genExcTypes == null || genExcTypes.length == 0) {
    System.out.println("no exceptions");
} else {
    System.out.println("generic exception types: ");
    for (Type t : genExcTypes) {
        System.out.println("\t"+t);
    }
}
}
```

---

```
analyzing method:
public static <T> T java.util.Collections.max(java.util.Collection<? extends T>)

GENERIC METHOD
type parameters:
 T

generic return type of method max: T

generic parameter types:
 java.util.Collection<? extends T>

no exceptions
```

Do not confuse `getParameterTypes()` with `getTypeParameters()`. The methods `getParameterTypes()` and `getGenericParameterTypes()` return the types of the method parameters; in our example the type `Collection<? extends T>`. The method `getTypeParameters()` returns a generic method's type parameters; in our example the type parameter `T`.

**LINK TO THIS**[Practicalities.FAQ711](#)**REFERENCES**[How do I figure out whether a method is a generic method?](#)[What is a generic declaration?](#)[java.lang.reflect.Method](#)[Which information is available about a type parameter?](#)**Which information is available about a type parameter?**

*The type parameter's name, its bounds, and the generic type or method that the type parameter belongs to.*

Type parameters of generic types and methods are represented by `TypeVariable` objects. A type parameter has a name, bounds, and you can find out which generic type or method the type parameter belongs to.

Let us take a look at an example, namely the type parameter of the generic class `EnumSet<E>` extends `Enum<E>>`.

Example (of retrieving information about a generic type):

```
Object object = new EnumMap<TimeUnit,Number>(TimeUnit.class);
Class<?> clazz = object.getClass();
TypeVariable<?>[] params = clazz.getTypeParameters();
if (params != null && params.length > 0) {
    System.out.println(clazz + " is a GENERIC TYPE with "+params.length+" type parameters");
    System.out.println();

    for (TypeVariable<?> typeparam : params) {
        System.out.println(typeparam + " is a TYPE VARIABLE");
        System.out.println("name   : " + typeparam.getName());

        GenericDeclaration genDecl = typeparam.getGenericDeclaration();
        System.out.println("is type parameter of generic declaration: " + genDecl);

        Type[] bounds = typeparam.getBounds();
        System.out.println("bounds: ");
        for (Type bound : bounds)
            System.out.println("\t" + bound + "\n");
        System.out.println();
    }
}
else {
    System.out.println(clazz + " is a NON-GENERIC TYPE");
}
```

```
class java.util.EnumMap is a GENERIC TYPE with 2 type parameters

K is a TYPE VARIABLE
name   : K
is type parameter of generic declaration: class java.util.EnumMap
bounds:
    java.lang.Enum<K>

V is a TYPE VARIABLE
name   : V
is type parameter of generic declaration: class java.util.EnumMap
bounds:
    class java.lang.Object
```

**LINK TO THIS**[Practicalities.FAQ712](#)**REFERENCES**[How do I figure out whether a method is a generic method?](#)[What is a generic declaration?](#)[java.lang.reflect.Method](#)**What is a generic declaration?**

*Either a generic type or a generic method or a generic constructor.*

In Java reflection a *generic declaration* is something that has type parameters, that is, either a generic type or a generic method or a generic constructor. A generic declaration is represented by the interface `GenericDeclaration` from the `java.lang.reflect` package. It provides access to the `getTypeParameters()` method, which is used to retrieve the type parameters of generic types, methods and constructors. Consequently, the classes `Class`, `Method` and `Constructor` implement this interface.

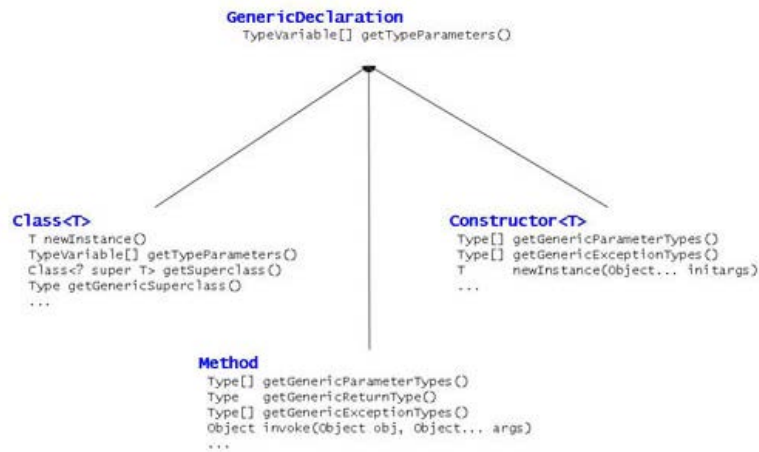


Figure: Subtypes of Interface `java.lang.reflect.GenericDeclaration`

LINK TO THIS [Practicalities.FAQ713](#)

REFERENCES [How do I figure out whether a method is a generic method?](#)  
[java.lang.reflect.GenericDeclaration](#)  
[java.lang.Class](#)  
[java.lang.reflect.Method](#)  
[java.lang.reflect.Constructor](#)

## What is a wildcard type?

*A wildcard expression; it appears as the type argument of a parameterized type.*

In Java reflection a *wildcard type* is a wildcard expression such as "`? extends Number`". It is represented by an object of type `java.lang.reflect.WildcardType` and can appear solely as a type argument of a parameterized type. The term "wildcard type" is slightly misleading, because a wildcard is not a type like the return type of a method or the type of a field. More correctly it is a type argument of a parameterized type.

Let us take a look at an example, namely the wildcards that appear in the signature of the generic method

```
<T extends Object & Comparable<? super T>> T Collections.max(Collection<? extends T>).
```

The first wildcard appears in the bounds of the method's type parameter `T`; its second bound is `Comparable<? super T>`, which is a parameterized type, and its type argument is the wildcard "`? super T`". The second wildcard appears in the method's declared argument type `Collection<? extends T>`, which is a parameterized type, and its type argument is the wildcard "`? extends T`".

Here is how the wildcard in the bound is retrieved:

Example (of a wildcard in Java reflection):

```
Method method = Collections.class.getMethod("max",Collection.class);
System.out.println("METHOD: "+method.toGenericString());

TypeVariable<Method> typeParameter = method.getTypeParameters()[0];
System.out.println("TYPE PARAMETER: "+typeParameter);

ParameterizedType bound = (ParameterizedType)typeParameter.getBounds()[1];
System.out.println("TYPE PARAMETER BOUND: "+bound);

WildcardType wildcard = (WildcardType)bound.getActualTypeArguments()[0];
System.out.println("WILDCARD: "+wildcard);
```

---

```
METHOD: public static <T> T java.util.Collections.max(java.util.Collection<? extends T>)
TYPE PARAMETER: T
TYPE PARAMETER BOUND: java.lang.Comparable<? super T>
WILDCARD: ? super T
```

We retrieve the method `Collections.max` via `Class.getMethod()` and its type parameter `T` via `GenericDeclaration.getTypeParameters()`. The result is the representation of the generic method's type parameter `T` as an object of type `java.lang.reflect.TypeVariable`. We retrieve the type variable's two bounds via `TypeVariable.getBounds()`. The second bound is `Comparable<? super T>` and it is represented by an object of type `java.lang.reflect.ParameterizedType`. We retrieve its type argument `? super T` via `ParameterizedType.getActualTypeArguments()` and check whether the type argument is a wildcard expression by checking whether it is represented by an object of type `java.lang.reflect.WildcardType`.

Here is how the wildcard in the declared method parameter type is retrieved:

Example (of a wildcard in Java reflection):

```
Method method = Collections.class.getMethod("max",Collection.class);
System.out.println("METHOD: "+method.toGenericString());

ParameterizedType methodParameterType = (ParameterizedType)method.getGenericParameterTypes()[0];
System.out.println("METHOD PARAMETER TYPE: "+methodParameterType);

WildcardType wildcard = (WildcardType)methodParameterType.getActualTypeArguments()[0];
System.out.println("WILDCARD: "+wildcard);
```

---

```
METHOD: public static <T> T java.util.Collections.max(java.util.Collection<? extends T>)
METHOD PARAMETER TYPE: java.util.Collection<? extends T>
WILDCARD: ? extends T
```

We obtain a representation of the method as before and this time retrieve the type of its method parameter `Collection<? extends T>` via `Method.getGenericParameterTypes()`. The result is the representation of the parameterized type `Collection<? extends T>` as an object of type `java.lang.reflect.ParameterizedType`. We retrieve its type argument `? extends T` via `ParameterizedType.getActualTypeArguments()` and check whether the type argument is a wildcard expression by checking whether it is represented by an object of type `java.lang.reflect.WildcardType`.

LINK TO THIS [#FAQ714](#)

REFERENCES [Which information is available about a wildcard?](#)

---

## Which information is available about a wildcard?

### *The upper and the lower bound.*

Wildcards can have an upper or a lower bound. Consequently, a wildcard represented reflectively by an object of type `java.lang.reflect.Wildcard` supports retrieval of the bound.

For illustration, let us revisit the wildcards from the previous FAQ entry [Practicalities.FAQ714](#), namely the wildcards that appear in the method signature

```
<T extends Object & Comparable<? super T>> T Collections.max(Collection<? extends T>).
```

Say, we retrieved a presentation of the wild `"? super T"` as described in the previous FAQ entry [Practicalities.FAQ714](#). Then we can obtain its upper bound by calling the methods `Wildcard.getLowerBounds()` and `Wildcard.getUpperBounds()`.

Example (of retrieving a wildcard's bound):

```
Method method = Collections.class.getMethod("max",Collection.class);
TypeVariable<Method> typeParameter = method.getTypeParameters()[0];
ParameterizedType bound = (ParameterizedType)typeParameter.getBounds()[1];
WildcardType wildcard = (WildcardType)bound.getActualTypeArguments()[0];
System.out.println("WILDCARD: "+wildcard);

Type[] lowerBounds = wildcard.getLowerBounds();
System.out.print("lower bound: ");
if (lowerBounds != null && lowerBounds.length > 0) {
    for (Type t : lowerBounds)
        System.out.println("\t" + t);
}
else {
    System.out.println("\t" + "<none>");
}

Type[] upperBounds = wildcard.getUpperBounds();
System.out.print("upper bound: ");
if (upperBounds != null && upperBounds.length > 0) {
    for (Type t : upperBounds)
        System.out.println("\t" + t);
}
else {
    System.out.println("\t" + "<none>");
}
```

---

```
WILDCARD: ? super T
lower bound: T
upper bound: class java.lang.Object
```

Interestingly, we can retrieve upper *and* lower bounds although a wildcard can have at most one bound - either an upper bound or a lower bound, but never both.

The wildcard "`? super T`" has a lower bound, but no upper bound. Yet the `getUpperBounds()` method returns an upper bound, namely `Object`, which makes sense because `Object` can be seen as the default upper bound of every wildcard.

Conversely, the wildcard "`? extends T`" has an upper bound, but no lower bound. The `getLowerBounds()` method returns a zero-length array in that case.

This is illustrated by the wildcard in the method's parameter type `Collection<? extends T>`. Say, we retrieved a presentation of the wild "`? extends T`" as described in the previous FAQ entry [Practicalities.FAQ714](#). Then we can try out which bounds the methods `Wildcard.getLowerBounds()` and `Wildcard.getUpperBounds()` return.

Example (of retrieving a wildcard's bound):

```
Method method = Collections.class.getMethod("max",Collection.class);
ParameterizedType methodParameterType = (ParameterizedType)method.getGenericParameterTypes()[0];
WildcardType wildcard = (WildcardType)methodParameterType.getActualTypeArguments()[0];
System.out.println("WILDCARD: "+wildcard);

Type[] lowerBounds = wildcard.getLowerBounds();
System.out.print("lower bound: ");
if (lowerBounds != null && lowerBounds.length > 0) {
    for (Type t : lowerBounds)
        System.out.println("\t" + t);
}
else {
    System.out.println("\t" + "<none>");
}
Type[] upperBounds = wildcard.getUpperBounds();
System.out.print("upper bound: ");
if (upperBounds != null && upperBounds.length > 0) {
    for (Type t : upperBounds)
        System.out.println("\t" + t);
}
else {
    System.out.println("\t" + "<none>");
}
```

---

```
WILDCARD: ? extends T
lower bound: <none>
upper bound: T
```

LINK TO THIS

[Practicalities.FAQ715](#)

REFERENCES

[What is a wildcard type?](#)  
[java.lang.reflect.Wildcard](#)

# Technicalities - Under The Hood Of The Compiler

© Copyright 2003-2022 by Angelika Langer. All Rights Reserved.

## Compiler Messages

- [What is an "unchecked" warning?](#)
- [How can I disable or enable unchecked warnings?](#)
- [What is the -Xlint:unchecked compiler option?](#)
- [What is the `SuppressWarnings` annotation?](#)
- [How can I avoid "unchecked cast" warnings?](#)
- [Is it possible to eliminate all "unchecked" warnings?](#)
- [Why do I get an "unchecked" warning although there is no type information missing?](#)

## Heap Pollution

- [What is heap pollution?](#)
- [When does heap pollution occur?](#)

## Type Erasure

- [How does the compiler translate Java generics?](#)
- [What is type erasure?](#)
- [What is reification?](#)
- [What is a bridge method?](#)
- [Under which circumstances is a bridge method generated?](#)
- [Why does the compiler add casts when it translates generics?](#)
- [How does type erasure work when a type parameter has several bounds?](#)
- [What is a reifiable type?](#)
- [What is the type erasure of a parameterized type?](#)
- [What is the type erasure of a type parameter?](#)
- [What is the type erasure of a generic method?](#)
- [Is generic code faster or slower than non-generic code?](#)
- [How do I compile generics for use with JDK  \$\leq\$  1.4?](#)

## Type System

- [How do parameterized types fit into the Java type system?](#)
- [How does the raw type relate to instantiations of the corresponding generic type?](#)
- [How do instantiations of a generic type relate to instantiations of other generic types that have the same type argument?](#)
- [How do unbounded wildcard instantiations of a generic type relate to other instantiations of the same generic type?](#)
- [How do wildcard instantiations with an upper bound relate to other instantiations of the same generic type?](#)
- [How do wildcard instantiations with a lower bound relate to other instantiations of the same generic type?](#)
- [Which super-subtype relationships exist among instantiations of generic types?](#)
- [Which super-subset relationships exist among wildcards?](#)

- [Does "extends" always mean "inheritance"?](#)

## Exception Handling

- [Can I use parameterized types in exception handling?](#)
- [Why are generic exception and error types illegal?](#)
- [Can I use a type parameter in exception handling?](#)
- [Can I use a type parameter in a catch clause?](#)
- [Can I use a type parameter in in a throws clause?](#)
- [Can I throw an object whose type is a type parameter?](#)

## Static Context

- [How do I refer to static members of a parameterized type?](#)
- [How do I refer to a \(non-static\) inner class of a parameterized type?](#)
- [How do I refer to an interface type nested into a parameterized type?](#)
- [How do I refer to an enum type nested into a parameterized type?](#)
- [Can I import a particular instantiations of a generic type?](#)
- [Why are generic enum types illegal?](#)

## Type Argument Inference

- [What is type argument inference?](#)
- [Is there a correspondence between type inference for method invocation and type inference for instance creation?](#)
- [What is the "diamond" operator?](#)
- [What is type argument inference for instance creation expressions?](#)
- [Why does the type inference for an instance creation expression fail?](#)
- [What is type argument inference for generic methods?](#)
- [What is explicit type argument specification?](#)
- [Can I use a wildcard as the explicit type argument of a generic method?](#)
- [What happens if a type parameter does not appear in the method parameter list?](#)
- [Why doesn't type argument inference fail when I provide inconsistent method arguments?](#)
- [Why do temporary variables matter in case of invocation of generic methods?](#)

## Wildcard Capture

- [What is the capture of a wildcard?](#)
- [What is the capture of an unbounded wildcard compatible to?](#)
- [Is the capture of a bounded wildcard compatible to the bound?](#)

## Wildcard Instantiations

- [Which methods and fields are accessible/inaccessible through a reference variable of a wildcard parameterized type?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

- [Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)
- [Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)
- [Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)
- [In a wildcard parameterized type, can I read and write fields whose type is the type parameter?](#)
- [Is it really impossible to create an object whose type is a wildcard parameterized type?](#)

### Cast and instanceof

- [Which types can or must not appear as target type in an instanceof expression?](#)

### Overloading and Overriding

- [What is method overriding?](#)
- [What is method overloading?](#)
- [What is the @Override annotation?](#)
- [What is a method signature?](#)
- [What is a subsignature?](#)
- [What are override-equivalent signatures?](#)
- [When does a method override its supertype's method?](#)
- [What are covariant-return types?](#)
- [What are substitutable return types?](#)
- [Can a method of a non-generic subtype override a method of a generic supertype?](#)
- [Can a method of a generic subtype override a method of a generic supertype?](#)
- [Can a generic method override a generic one?](#)
- [Can a non-generic method override a generic one?](#)
- [Can a generic method override a non-generic one?](#)
- [What is overload resolution?](#)

## **Under The Hood Of The Compiler**

### **Compiler Messages**

**What is an "unchecked" warning?**



*A warning by which the compiler indicates that it cannot ensure type safety.*

The term "unchecked" warning is misleading. It does not mean that the warning is unchecked in any way. The term "unchecked" refers to the fact that the compiler and the runtime system do not have enough type information to perform all type checks that would be necessary to ensure type safety. In this sense, certain operations are "unchecked".

The most common source of "unchecked" warnings is the use of raw types. "unchecked" warnings are issued when an object is accessed through a raw type variable, because the raw type does not provide enough type information to perform all necessary type checks.

Example (of unchecked warning in conjunction with raw types):

```
TreeSet set = new TreeSet();
set.add("abc");           // unchecked warning
set.remove("abc");
```

---

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type java.util.TreeSet
    set.add("abc");
        ^
```

When the `add` method is invoked the compiler does not know whether it is safe to add a `String` object to the collection. If the `TreeSet` is a collection that contains `Strings` (or a supertype thereof), then it would be safe. But from the type information provided by the raw type `TreeSet` the compiler cannot tell. Hence the call is potentially unsafe and an "unchecked" warning is issued.

"unchecked" warnings are also reported when the compiler finds a cast whose target type is either a parameterized type or a type parameter.

Example (of an unchecked warning in conjunction with a cast to a parameterized type or type variable):

```
class Wrapper<T> {
    private T wrapped;
    public Wrapper(T arg) {wrapped = arg;}
    ...
    public Wrapper<T> clone() {
        Wrapper<T> clon = null;
        try {
            clon = (Wrapper<T>)super.clone(); // unchecked warning
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
        try {
            Class<?> clzz = this.wrapped.getClass();
            Method meth = clzz.getMethod("clone", new Class[0]);
            Object dupl = meth.invoke(this.wrapped, new Object[0]);
```

```
        clon.wrapped = (T)dupl; // unchecked warning
    } catch (Exception e) {}
    return clon;
}
}
```

---

```
warning: [unchecked] unchecked cast
found   : java.lang.Object
required: Wrapper<T>
        clon = (Wrapper<T>)super.clone();
                ^
warning: [unchecked] unchecked cast
found   : java.lang.Object
required: T
        clon.wrapped = (T)dupl;
                        ^
```

A cast whose target type is either a (concrete or bounded wildcard) parameterized type or a type parameter is unsafe, if a dynamic type check at runtime is involved. At runtime, only the type erasure is available, not the exact static type that is visible in the source code. As a result, the runtime part of the cast is performed based on the type erasure, not on the exact static type.

In the example, the cast to `Wrapper<T>` would check whether the object returned from `super.clone` is a `Wrapper`, not whether it is a wrapper with a particular type of members. Similarly, the casts to the type parameter `T` are cast to type `Object` at runtime, and probably optimized away altogether. Due to type erasure, the runtime system is unable to perform more useful type checks at runtime.

In a way, the source code is misleading, because it suggests that a cast to the respective target type is performed, while in fact the dynamic part of the cast only checks against the type erasure of the target type. The "unchecked" warning is issued to draw the programmer's attention to this mismatch between the static and dynamic aspect of the cast.

**LINK TO THIS**                    [Technicalities.FAQ001](#)

**REFERENCES**

- [What does type-safety mean?](#)
- [How can I disable or enable unchecked warnings?](#)
- [What is the raw type?](#)
- [Can I use a raw type like any other type?](#)
- [Can I cast to a parameterized type?](#)
- [Can I cast to the type that the type parameter stands for?](#)

---

## How can I disable or enable "unchecked" warnings?

*Via the compiler options `-Xlint:unchecked` and `-Xlint:-unchecked` and via the standard annotation `@SuppressWarnings("unchecked")`.*

The compiler option `-Xlint:-unchecked` disables *all* unchecked warnings that would occur in a compilation.

The annotation `@SuppressWarnings("unchecked")` suppresses all warnings for the annotated part of the program.

Note, in the first release of Java 5.0 the `SuppressWarnings` annotation is not yet supported.

LINK TO THIS [Technicalities.FAQ002](#)

REFERENCES [What is the `-Xlint:unchecked` compiler option?](#)  
[What is the `SuppressWarnings` annotation?](#)

---

## What is the `-Xlint:unchecked` compiler option?

*The compiler option `-Xlint:unchecked` enables "unchecked" warnings, the option `-Xlint:-unchecked` disables all unchecked warnings.*

"unchecked" warnings are by default disabled. If you compile a program with no particular compiler options then the compiler will not report any "unchecked" warnings. If the compiler finds source code for which it would want to report an "unchecked" warning it only gives a general hint. You will find the following note at the end of the list of all other errors and warnings:

```
Note: util/Wrapper.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

If you want to see the "unchecked" warnings you must start the compiler with the `-Xlint:unchecked` option.

Example (of globally enabling unchecked warnings):

```
javac -Xlint:unchecked util/Wrapper.java
```

The option `-Xlint:unchecked` enables the "unchecked" warnings. The "unchecked" warnings are also enabled when you use the `-Xlint:all` option.

The option `-Xlint:-unchecked` disables the "unchecked" warnings. This is useful to suppress all "unchecked" warnings, while other types of warnings remain enabled.

Example (of globally disabling unchecked warnings):

```
javac -g -source 1.5 -Xlint:all -Xlint:-unchecked util/Wrapper.java
```

In this example, using `-Xlint:all` all warnings (such as "unchecked", "deprecated", "fallthrough", etc.) are enabled and subsequently the "unchecked" warnings are disabled using `-Xlint:-unchecked`. As a result all warnings except "unchecked" warnings will be reported.

LINK TO THIS [Technicalities.FAQ003](#)

REFERENCES [What is an "unchecked" warning?](#)  
[What is the `SuppressWarnings` annotation?](#)

---

## What is the `SuppressWarnings` annotation?

*A standard annotation that suppresses warnings for the annotated part of the program.*

The compiler supports a number of standard annotations (see package [java.lang.annotation](#)). Among them is the `SuppressWarnings` annotation. It contains a list of warning labels. If a definition in the source code is annotated with the `SuppressWarnings` annotation, then all warnings, whose labels appear in the annotation's list of warning labels, are suppressed for the annotated definition or any of its parts.

The `SuppressWarnings` annotation can be used to suppress any type of labelled warning. In particular we can use the annotation to suppress "unchecked" warnings.

Example (of suppressing unchecked warnings):

```
@SuppressWarnings("unchecked")
class Wrapper<T> {
    private T wrapped;
    public Wrapper(T arg) {wrapped = arg;}
    ...
    public Wrapper<T> clone() {
        Wrapper<T> clon = null;
        try {
            clon = (Wrapper<T>)super.clone(); // unchecked warning suppressed
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
        try {
            Class<?> clzz = this.wrapped.getClass();
            Method meth = clzz.getMethod("clone", new Class[0]);
            Object dupl = meth.invoke(this.wrapped, new Object[0]);
            clon.wrapped = (T)dupl; // unchecked warning suppressed
        } catch (Exception e) {}
        return clon;
    }
}
```

This example would usually raise 2 "unchecked" warnings. Since we annotated the entire class, all unchecked warnings raised anywhere in the class implementation are suppressed.

We can suppress several types of annotations at a time. In this case we must specify a list of warning labels.

Example (of suppressing several types of warnings):

```
@SuppressWarnings(value={"unchecked","deprecation"})
public static void someMethod() {
    ...
    TreeSet set = new TreeSet();
    set.add(new Date(104,8,11));    // unchecked and deprecation warning suppressed
}
```

```
} ...
```

This example would usually raise 2 warnings when the call to method `add` is compiled:

```
warning: [deprecation] Date(int,int,int) in java.util.Date has been deprecated
    set.add(new Date(104,8,11));
           ^
warning: [unchecked] unchecked call to add(E) as a member of the raw type java.util.TreeSet
    set.add(new Date(104,8,11));
```

The annotation preceding the enclosing method suppresses all unchecked and deprecation warnings anywhere in the method implementation.

Annotations can *not* be attached to statements, expressions, or blocks, only to program entities with a definition like types, variables, etc.

Example (of illegal placement of annotation):

```
public static void someMethod() {
    ...
    TreeSet set = new TreeSet();
    @SuppressWarnings(value={"unchecked"}) // error
    set.add(new Date(104,8,11));
    ...
}
```

Annotations can be attached to the definition of packages, classes, interfaces, fields, methods, parameters, constructors, local variables, enum types, enum constants, and annotation types. An annotated package declaration must go into a file named `package-info.java` in the directory that represents the package.

Note, in release of Java 5.0 the `SuppressWarnings` annotation is not yet supported by all compilers. Sun's compiler will support it in release 6.0.

LINK TO THIS

[Technicalities.FAQ004](#)

REFERENCES

[What is an "unchecked" warning?](#)  
[How can I disable or enable unchecked warnings?](#)

---

## How can I avoid "unchecked cast" warnings?

*By using an unbounded wildcard parameterized type as target type of a cast expression.*

Occasionally, we would like to cast to a parameterized type, just to discover that the compiler flags it with an "unchecked" warning. As we are interested in warning-free compilation of our source code, we would like to avoid this warning. Use of an unbounded wildcard parameterized type instead of a concrete or a bounded wildcard parameterized type would help avoid the warning.

A typical example is the implementation of methods such as the `equals` method, that take `Object` reference and where a cast down to the actual type must be performed.

Example (not recommended):

```
class Wrapper<T> {
    private T wrapped;
    ...
    public boolean equals(Object other) {
        ...
        Wrapper<T> otherWrapper = (Wrapper<T>)other; // warning; unchecked cast
        return (this.wrapped.equals(otherWrapper.wrapped));
    }
}
```

When we replace the cast to `Wrapper<T>` by a cast to `Wrapper<?>` the warning disappears, because unbounded wildcard parameterized types are permitted as target type of a cast without any warnings.

Example (implementation of equals):

```
class Wrapper<T> {
    private T wrapped;
    ...
    public boolean equals(Object other) {
        ...
        Wrapper<?> otherWrapper = (Wrapper<?>)other;
        return (this.wrapped.equals(otherWrapper.wrapped));
    }
}
```

Note, this technique works in this example only because we need no write access to the fields of the object referred to through the wildcard parameterized type and we need not invoke any methods. Remember, use of the object that a wildcard reference variable refers to is restricted. In other situations use of a wildcard parameterized type might not be a viable solution, because full access to the referenced object is needed.

LINK TO THIS

[Technicalities.FAQ005](#)

REFERENCES

[Can I cast to a parameterized type?](#)

[What is an "unchecked" warning?](#)

[How can I disable or enable unchecked warnings?](#)

[How do I best implement the equals method of a generic type?](#)

---

## Is it possible to eliminate all "unchecked" warnings?

*Almost.*

"Unchecked" warnings stem either from using generic types in their raw form or from casts whose target type is a type parameter or a concrete or bounded wildcard parameterized type. If you refrain from both using raw types and the critical casts you can theoretically eliminate all "unchecked" warnings. Whether this is doable

in practice depends on the circumstances.

## Raw types.

When source code is compiled for use in Java 5.0 that was developed before Java 5.0 and uses classes that are generic in Java 5.0, then "unchecked" warnings are inevitable. For instance, if "legacy" code uses types such as `List`, which used to be a regular (non-generic) types before Java 5.0, but are generic in Java 5.0, all these uses of `List` are considered uses of a raw type in Java 5.0. Use of the raw types will lead to "unchecked" warnings. If you want to eliminate the "unchecked" warnings you must re-engineer the "legacy" code and replace all raw uses of `List` with appropriate instantiations of `List` such as `List<String>`, `List<Object>`, `List<?>`, etc. All "unchecked" warnings can be eliminated this way.

In source code developed for Java 5.0 you can prevent "unchecked" warnings in the first place by never using raw types. Always provide type arguments when you use a generic type. There are no situations in which you are forced to use a raw type. In case of doubt, when you feel you have no idea which type argument would be appropriate, try the unbounded wildcard `"?"`.

In essence, "unchecked" warnings due to use of raw types can be eliminated if you have access to legacy code and are willing to re-engineer it.

## Casts.

"Unchecked" warnings as a result of cast expressions can be eliminated by eliminating the offensive casts. Eliminating such casts is almost always possible. There are, however, a few situations where a cast to a type parameter or a concrete or bounded wildcard parameterized type cannot be avoided.

These are typically situations where a method returns a supertype reference to an object of a more specific type. The classic example is the `clone` method; it returns an `Object` reference to an object of the type on which it was invoked. In order to recover the returned object's actual type a cast is necessary. If the cloned object is of a parameterized type, then the target type of the cast is an instantiation of that parameterized type, and an "unchecked" warning is inevitable. The `clone` method is just one example that leads to unavoidable "unchecked" warnings. Invocation of methods via reflection has similar effects because the return value of a reflectively invoked method is returned via an `Object` reference. It is likely that you will find further examples of unavoidable "unchecked" casts in practice. For a detailed discussion of an example see [Technicalities.FAQ502](#), which explains the implementation of a `clone` method for a generic class.

In sum, there are situations in which you cannot eliminate "unchecked" warnings due to a cast expression.

### LINK TO THIS

[Technicalities.FAQ006](#)

### REFERENCES

[What is an "unchecked" warning?](#)  
[How do I best implement the clone method of a generic type?](#)

---

## Why do I get an "unchecked" warning although there is no type information missing?

*Because the compiler performs all type checks based on the type erasure when you use a raw type.*

Usually the compiler issues an "unchecked" warning in order to alert you to some type-safety hazard that the compiler cannot prevent because there is not enough type information available. One of these situations is the invocation of methods of a raw type.

Example (of unchecked warning in conjunction with raw types):

```
class TreeSet<E> {
    booleanadd(E o){ ... }
}
TreeSet set = new TreeSet();
set.add("abc");           // unchecked warning
```

---

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type TreeSet
        set.add("abc");
           ^
```

When the `add` method is invoked the compiler does not know whether it is safe to add a `String` object to the collection because the raw type `TreeSet` does not provide any information regarding the type of the contained elements.

Curiously, an unchecked warning is also issued in situations where there is enough type information available.

Example (of a spurious unchecked warning in conjunction with raw types):

```
class SomeType<T> {
    public List<String> getList() { ... }
}
SomeTyperaw = new SomeType();
List<String> listString = raw.getList(); // unchecked warning
```

---

```
warning: [unchecked] unchecked conversion
found   : List
required: List<String>
        List<String> listString = raw.getList();
           ^
```

In this example, there is no type information missing. The `getList` method is declared to return a `List<String>` and this is so even in the raw type because the method does not depend on the enclosing class's type parameter. Yet the compiler complains.

The reason is that the compiler computes the type erasure of a generic type when it finds an occurrence of the raw type in the source code. Type erasure does not only elide all occurrences of the type parameter `T`, but also elides the type argument of the `getList` method's return type. After type erasure, the `getList` method returns just a `List` and no longer a `List<String>`. All subsequent type checks are performed based on the type erasure; hence the "unchecked" warning.

The "unchecked" warning can easily be avoided by refraining from use of the raw type. `SomeType` is a generic type and should always be used with a type argument. In general, the use of raw types will inevitably result in "unchecked" warnings; some of the warnings may be spurious, but most of them are justified.

---

Note, that no spurious warning is issued when the method in question is a static method.

Example (of invoking a static method of a raw type):

```
class SomeType<T> {
    public static List<String> getList() { ... }
```



```
}  
SomeType raw = new SomeType();  
List<String> listString = raw.getList(); // fine
```

**LINK TO THIS**

[Technicalities.FAQ007](#)

**REFERENCES**

[What is an "unchecked" warning?](#)

[Should I prefer parameterized types over raw types?](#)

[Why shouldn't I mix parameterized and raw types, if I feel like it?](#)

---

## Heap Pollution

### What is heap pollution?

*A situation where a variable of a parameterized type refers to an object that is not of that parameterized type.*

It can happen that a variable of a parameterized type such as `List<String>` refers to an object that is not of that parameterized type.

Example (of heap pollution):

```
List ln = new ArrayList<Number>();  
List<String> ls =ln; // unchecked warning  
String s = ls.get(0); // ClassCastException
```

After the assignment of the reference variable `ln` to the reference variable `ls`, the `List<String>` variable will point to a `List<Number>` object. Such a situation is called *heap pollution* and is usually indicated by an unchecked warning. A polluted heap is likely to lead to an unexpected `ClassCastException` at runtime. In the example above, it will lead to a `ClassCastException`, when a object is retrieved from the `List<String>` and assigned to a `String` variable, because the object is a `Number`, not a `String`.

**LINK TO THIS**

[Technicalities.FAQ050](#)

**REFERENCES**

[What is an "unchecked" warning?](#)

[When does heap pollution occur?](#)

---

### When does heap pollution occur?

*As a result of mixing raw and parameterized type, unwise casting, and separate compilation.*

Heap pollution occurs in three situations:

- mixing raw types and parameterized types
- performing unchecked casts
- separate compilation of translation units

With the exception of separate compilation, the compiler will always issue an unchecked warning to draw your attention to the potential heap pollution. If you co-compile your code without warnings then no heap pollution can ever occur.

---

### *Raw Types*

*Heap pollution can occur when raw types and parameterized types are mixed and a raw type variable is assigned to a parameterized type variable. Note, that heap pollution does not necessarily occur, even if the compiler issues an unchecked warning.*

*Example (of mixing raw and parameterized types):*

```
List ln = new ArrayList<Number>();
List ls = new LinkedList<String>();

List<String> list;
list = ln; // unchecked warning + heap pollution
list = ls; // unchecked warning + NO heap pollution
```

*The first assignment leads to heap pollution, because the List<String> variable would then point to a List<Number>. The second assignment does not result in heap pollution, because the raw type variable on the right-hand side of the assignment refers to a List<String>, which matches the parameterized type on the left-hand side of the assignment.*

*Mixing raw and parameterized types should be avoided, if possible. It cannot be avoided when non-generic legacy code is combined with modern generic code. But otherwise, the mix is bad programming style.*

---

### *Unchecked Casts*

*Unwise casting can lead to all kinds of unhealthy situations. In particular, it can lead to heap pollution.*

*Example (of cast to parameterized type polluting the heap):*

```
List<? extends Number> ln = new ArrayList<Long>();
List<Short> ls = (List<Short>) ln; // unchecked warning + heap pollution
List<Long> ll = (List<Long>) ln; // unchecked warning + NO heap pollution
```

*The compiler permits the two casts in the example above, because List<? extends Number> is a supertype of the types List<Short> and List<Long>. The casts*

are similar to casts from supertype `Object` to subtype `Short` or `Long`. The key difference is that the correctness of a cast to a non-parameterized type can be ensured at runtime and will promptly lead to `ClassCastException`, while a cast to a parameterized type cannot be ensured at runtime because of type erasure and might result in heap pollution.

Casts with a parameterized target type can lead to heap pollution, and so do casts to type variables.

Example (of cast to type variable polluting the heap):

```
<S,T> S convert(T arg) {
    return (S)arg; // unchecked warning
}
Number n = convert(new Long(5L)); // fine
String s = convert(new Long(5L)); // ClassCastException
```

In this example we do not cast to a parameterized type, but a type variable `S`. The compiler permits the cast because the cast could succeed, but there is no way to ensure success of the cast at runtime.

Casts, whose target type is a parameterized type or a type variable, should be avoided, if possible.

---

#### Separate Compilation

Another situation, in which heap pollution can occur is separate compilation of translation units.

Example (initial implementation):

```
file FileCrawler.java:
final class FileCrawler {
    ...
    public List<String> getFileNames() {
        List<String> list = new LinkedList<String>();
        ...
        return list;
    }
}
file Test.java:
final class Test {
    public static void main(String[] args) {
        FileCrawler crawler = new FileCrawler("http:\\www.goofy.com");
        List<String> files = crawler.getFileNames();
        System.out.println(files.get(0));
    }
}
```

The program compiles and runs fine. Now, let's assume that we modify the `FileCrawler` implementation. Instead of returning a `List<String>` we return a `List<StringBuilder>`. Note, the other class is not changed at all.

Example (after modification and co-compilation):

```
file FileCrawler.java:
final class FileCrawler {
    ...
    public List<StringBuilder> getFileNames() {
```

```

        List<StringBuilder> list = new LinkedList<StringBuilder>();
        ...
        return list;
    }
}
file Test.java:
final class Test {
    public static void main(String[] args) {
        FileCrawler crawler = new FileCrawler("http:\\www.goofy.com");
        List<String> files = crawler.getFileNames(); // error
        System.out.println(files.get(0));
    }
}

```

When we co-compile both translation units, the compiler would report an error in the unmodified file `Test.java`, because the return type of the `getNames()` method does no longer match the expected type `List<String>`.

If we compile separately, that is, only compile the modified file `Test.java`, then no error would be reported. This is because the class, in which the error occurs, has not been re-compiled. When the program is executed, a `ClassCastException` will occur.

Example (after modification and separate compilation):

```

file FileCrawler.java:
final class FileCrawler {
    ...
    public List<StringBuilder> getFileNames() {
        List<StringBuilder> list = new LinkedList<StringBuilder>();
        ...
        return list;
    }
}
file Test.java:
final class Test {
    public static void main(String[] args) {
        FileCrawler crawler = new FileCrawler("http:\\www.goofy.com");
        List<String> files = crawler.getFileNames(); // fine
        System.out.println(files.get(0)); // ClassCastException
    }
}

```

This is another example of heap pollution. The compiler, since it does not see the entire program, but only a part of it, cannot detect the error. Co-compilation avoids this problem and enables the compiler to detect and report the error.

Separate compilation in general is hazardous, independently of generics. If you provide a method that first returns a `String` and later you change it to return a `StringBuilder`, without re-compiling all parts of the program that use the method, you end up in a similarly disastrous situation. The crux is the incompatible change of the modified method. Either you can make sure that the modified part is co-compiled with all parts that use it or you must not introduce any incompatible changes such as changes in semantics of types or signatures of methods.

LINK TO THIS [Technicalities.FAQ051](#)

#### REFERENCES

- [What is heap pollution?](#)
- [What is an "unchecked" warning?](#)
- [How can I avoid "unchecked cast" warnings?](#)
- [Is it possible to eliminate all "unchecked" warnings?](#)

# Type Erasure

## How does the compiler translate Java generics?

*By creating one unique byte code representation of each generic type (or method) and mapping all instantiations of the generic type (or method) to this unique representation.*

The Java compiler is responsible for translating Java source code that contains definitions and usages of generic types and methods into Java byte code that the virtual machine can interpret. How does that translation work?

A compiler that must translate a generic type or method (in any language, not just Java) has in principle two choices:

- *Code specialization.* The compiler generates a new representation for every instantiation of a generic type or method. For instance, the compiler would generate code for a list of integers and additional, different code for a list of strings, a list of dates, a list of buffers, and so on.

*Code sharing.* The compiler generates code for only one representation of a generic type or method and maps all the instantiations of the generic type or method to the unique representation, performing type checks and type conversions where needed.

Code specialization is the approach that C++ takes for its templates:

The C++ compiler generates executable code for every instantiation of a template. The downside of code specialization of generic types is its potential for code bloat. A list of integers and a list of strings would be represented in the executable code as two different types. Note that code bloat is not inevitable in C++ and can generally be avoided by an experienced programmer.

Code specialization is particularly wasteful in cases where the elements in a collection are references (or pointers), because all references (or pointers) are of the same size and internally have the same representation. There is no need for generation of mostly identical code for a list of references to integers and a list of references to strings. Both lists could internally be represented by a list of references to any type of object. The compiler just has to add a couple of casts whenever these references are passed in and out of the generic type or method. Since in Java most types are reference types, it deems natural that Java chooses code sharing as its technique for translation of generic types and methods.

The Java compiler applies the code sharing technique and creates one unique byte code representation of each generic type (or method). The various instantiations of the generic type (or method) are mapped onto this unique representation by a technique that is called *type erasure*.

LINK TO THIS [Technicalities.FAQ100](#)

REFERENCES [What is type erasure?](#)

---

## What is type erasure?

*A process that maps a parameterized type (or method) to its unique byte code representation by eliding type parameters and arguments.*

The compiler generates only one byte code representation of a generic type or method and maps all the instantiations of the generic type or method to the unique

representation. This mapping is performed by type erasure. The essence of type erasure is the removal of all information that is related to type parameters and type arguments. In addition, the compiler adds type checks and type conversions where needed and inserts synthetic bridge methods if necessary. It is important to understand type erasure because certain effects related to Java generics are difficult to understand without a proper understanding of the translation process.

The type erasure process can be imagined as a translation from generic Java source code back into regular Java code. In reality the compiler is more efficient and translates directly to Java byte code. But the byte code created is equivalent to the non-generic Java code you will be seeing in the subsequent examples.

The steps performed during type erasure include:

#### *Eliding type parameters.*

When the compiler finds the definition of a generic type or method, it removes all occurrences of the type parameters and replaces them by their leftmost bound, or type `Object` if no bound had been specified.

#### *Eliding type arguments.*

When the compiler finds a parameterized type, i.e. an instantiation of a generic type, then it removes the type arguments. For instance, the types `List<String>`, `Set<Long>`, and `Map<String, ?>` are translated to `List`, `Set` and `Map` respectively.

Example (before type erasure):

```
interface Comparable<A> {
    public int compareTo(A that);
}
final class NumericValue implements Comparable<NumericValue> {
    private byte value;
    public NumericValue(byte value) { this.value = value; }
    public byte getValue() { return value; }
    public int compareTo(NumericValue that) { return this.value - that.value; }
}
class Collections {
    public static <A extends Comparable<A>>A max(Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
final class Test {
    public static void main (String[ ] args) {
        LinkedList<NumericValue> numberList = new LinkedList<NumericValue>();
        numberList.add(new NumericValue((byte)0));
        numberList.add(new NumericValue((byte)1));
        NumericValue y = Collections.max(numberList);
    }
}
```

```
}
```

Type parameters are green and type arguments are blue. During type erasure the type arguments are discarded and the type parameters are replaced by their leftmost bound.

Example (after type erasure):

```
interface Comparable {
    public int compareTo(Object that);
}
final class NumericValue implements Comparable {
    private byte value;
    public NumericValue(byte value) { this.value = value; }
    public byte getValue() { return value; }
    public int compareTo(NumericValue that) { return this.value - that.value; }
    public int compareTo(Object that) { return this.compareTo((NumericValue)that); }
}
class Collections {
    public static Comparable max(Collection xs) {
        Iterator xi = xs.iterator();
        Comparable w = (Comparable) xi.next();
        while (xi.hasNext()) {
            Comparable x = (Comparable) xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
final class Test {
    public static void main (String[ ] args) {
        LinkedList numberList = new LinkedList();
        numberList.add(new NumericValue((byte)0));
        numberList.add(new NumericValue((byte)1));
        NumericValue y = (NumericValue) Collections.max(numberList);
    }
}
```

The generic `Comparable` interface is translated to a non-generic interface and the unbounded type parameter `A` is replaced by type `Object`.

The `NumericValue` class implements the non-generic `Comparable` interface after type erasure, and the compiler adds a so-called *bridge method*. The bridge method is needed so that class `NumericValue` remains a class that implements the `Comparable` interface after type erasure.

The generic method `max` is translated to a non-generic method and the bounded type parameter `A` is replaced by its leftmost bound, namely `Comparable`. The parameterized interface `Iterator<A>` is translated to the raw type `Iterator` and the compiler adds a cast whenever an element is retrieved from the raw type `Iterator`.

The uses of the parameterized type `LinkedList<NumericValue>` and the generic `max` method in the `main` method are translated to uses of the non-generic type and method and, again, the compiler must add a cast.

LINK TO THIS [Technicalities.FAQ101](#)

REFERENCES [What is a bridge method?](#)  
[Why does the compiler add casts when it translates generics?](#)  
[How does type erasure work when a type parameter has several bounds?](#)

---

## What is reification?

*Representing type parameters and arguments of generic types and methods at runtime. Reification is the opposite of type erasure.*

In Java, type parameters and type arguments are elided when the compiler performs type erasure. A side effect of type erasure is that the virtual machine has no information regarding type parameters and type arguments. The JVM cannot tell the difference between a `List<String>` and a `List<Date>`.

In other languages, like for instance C#, type parameters and type arguments of generics types and methods do have a runtime representation. This representation allows the runtime system to perform certain checks and operations based on type arguments. In such a language the runtime system can tell the difference between a `List<String>` and a `List<Date>`.

LINK TO THIS [Technicalities.FAQ101A](#)

REFERENCES [What is type erasure?](#)  
[What is a reifiable type?](#)

---

## What is a bridge method?

*A synthetic method that the compiler generates in the course of type erasure. It is sometimes needed when a type extends or implements a parameterized class or interface.*

The compiler insert bridge methods in subtypes of parameterized supertypes to ensure that subtyping works as expected.

Example (before type erasure):

```
interface Comparable<A> {
    public int compareTo(A that);
}
final class NumericValue implements Comparable<NumericValue> {
    private byte value;
```



```

public NumericValue(byte value) { this.value = value; }
public byte getValue() { return value; }
public int compareTo(NumericValue that) { return this.value - that.value; }
}

```

In the example, class `NumericValue` implements interface `Comparable<NumericValue>` and must therefore override the superinterface's `compareTo` method. The method takes a `NumericValue` as an argument. In the process of type erasure, the compiler translates the parameterized `Comparable<A>` interface to its type erased counterpart `Comparable`. The type erasure changes the signature of the interface's `compareTo` method. After type erasure the method takes an `Object` as an argument.

Example (after type erasure):

```

interface Comparable {
    public int compareTo(Object that);
}
final class NumericValue implements Comparable {
    private byte value;
    public NumericValue(byte value) { this.value = value; }
    public byte getValue() { return value; }
    public int compareTo(NumericValue that) { return this.value - that.value; }
    public int compareTo(Object that) { return this.compareTo((NumericValue)that); }
}

```

After this translation, method `NumericValue.compareTo(NumericValue)` is no longer an implementation of the interface's `compareTo` method. The type erased `Comparable` interface requires a `compareTo` method with argument type `Object`, not `NumericValue`. This is a side effect of type erasure: the two methods (in the interface and the implementing class) have identical signatures before type erasure and different signatures after type erasure.

In order to achieve that class `NumericValue` remains a class that correctly implements the `Comparable` interface, the compiler adds a bridge method to the class. The bridge method has the same signature as the interface's method after type erasure, because that's the method that must be implemented. The bridge method delegates to the original methods in the implementing class.

The existence of the bridge method does not mean that objects of arbitrary types can be passed as arguments to the `compareTo` method in `NumericValue`. The bridge method is an implementation detail and the compiler makes sure that it normally cannot be invoked.

Example (illegal attempt to invoke bridge method):

```

NumericValue value = new NumericValue((byte)0);
value.compareTo(value); // fine
value.compareTo("abc"); // error

```

The compiler does not invoke the bridge method when an object of a type other than `NumericValue` is passed to the `compareTo` method. Instead it rejects the call with an error message, saying that the `compareTo` method expects a `NumericValue` as an argument and other types of arguments are not permitted.

You can, however, invoke the synthetic bridge message using reflection. But, if you provide an argument of a type other than `NumericValue`, the method will fail with a `ClassCastException` thanks of the cast in the implementation of the bridge method.

Example (failed attempt to invoke bridge method via reflection):

```
int reflectiveCompareTo(NumericValue value, Object other)
    throws NoSuchMethodException, IllegalAccessException, InvocationTargetException
{
    Method meth = NumericValue.class.getMethod("compareTo", new Class[]{Object.class});
    return (Integer)meth.invoke(value, new Object[]{other});
}
NumericValue value = new NumericValue((byte)0);
reflectiveCompareTo(value, value); // fine
reflectiveCompareTo(value, "abc"); // ClassCastException
```

The cast to type `NumericValue` in the bridge method fails with a `ClassCastException` when an argument of a type other than `NumericValue` is passed to the bridge method. This was it is guaranteed that a bridge method, even when it is called, will fail for unexpected argument types.

LINK TO THIS [Technicalities.FAQ102](#)

REFERENCES [What is type erasure?](#)  
[Under which circumstances is a bridge method generated?](#)

---

## Under which circumstances is a bridge method generated?

*When a type extends or implements a parameterized class or interface and type erasure changes the signature of any inherited method.*

Bridge methods are necessary when a class implements a parameterized interface or extends a parameterized superclass and type erasure changes the argument type of any of the inherited non-static methods.

Below is an example of a class that extends a parameterized superclass.

Example (before type erasure):

```
class Superclass<T extends Bound> {
    public void m1(T arg) { ... }
    public T m2() { ... }
}
class Subclass extends Superclass<SubTypeOfBound> {
    public void m1(SubTypeOfBound arg) { ... }
    public SubTypeOfBound m2() { ... }
}
```

Example (after type erasure):

```
class Superclass {
    void m1(Bound arg) { ... }
    Bound m2() { ... }
}
```

```

}
class Subclass extends Superclass {
    public void m1(SubTypeOfBound arg) { ... }
    public void m1(Bound arg) { m1((SubTypeOfBound)arg); }
    public SubTypeOfBound m2() { ... }
    public Bound m2() { return m2(); }
}

```

Type erasure changes the signature of the superclass's methods. The subclass's methods are no longer overriding versions of the superclass's method after type erasure. In order to make overriding work the compiler adds bridge methods.

The compiler must add bridge methods even if the subclass does not override the inherited methods.

Example (before type erasure):

```

class Superclass<T extends Bound> {
    public void m1(T arg) { ... }
    public T m2() { ... }
}
class AnotherSubclass extends Superclass<SubTypeOfBound> {
}

```

Example (after type erasure):

```

class Superclass {
    void m1(Bound arg) { ... }
    Bound m2() { ... }
}
class AnotherSubclass extends Superclass {
    public void m1(Bound arg) { super.m1((SubTypeOfBound)arg); }
    public Bound m2() { return super.m2(); }
}

```

The subclass is derived from a particular instantiation of the superclass and therefore inherits the methods with a particular signature. After type erasure the signature of the superclass's methods are different from the signatures that the subclass is supposed to have inherited. The compiler adds bridge methods, so that the subclass has the expected inherited methods.

No bridge method is needed when type erasure does not change the signature of any of the methods of the parameterized supertype. Also, no bridge method is needed if the signatures of methods in the sub- and supertype change in the same way. This can occur when the subtype is generic itself.

Example (before type erasure):

```

interface Callable<V> {
    public V call();
}
class Task<T> implements Callable<T> {
    public T call() { ... }
}

```

```
}
```

Example (after type erasure):

```
interface Callable {
    public Object call();
}
class Task implements Callable {
    public Object call() { ... }
}
```

The return type of the `call` method changes during type erasure in the interface and the implementing class. After type erasure the two methods have the same signature so that the subclass's method implements the interface's method without a bridge method.

However, it does not suffice that the subclass is generic. The key is that the method signatures must not match after type erasure. Otherwise, we again need a bridge method.

Example (before type erasure):

```
interface Copyable<V> extends Cloneable {
    public V copy();
}
class Triple<T extends Copyable<T>> implements Copyable<Triple<T>> {
    public Triple<T> copy() { ... }
}
```

Example (after type erasure):

```
interface Copyable extends Cloneable {
    public Object copy();
}
class Triple implements Copyable {
    public Triple copy() { ... }
    public Object copy() { return copy(); }
}
```

The method signatures change to `Object copy()` in the interface and `Triple copy()` in the subclass. As a result, the compiler adds a bridge method.

LINK TO THIS [Technicalities.FAQ103](#)

REFERENCES [What is type erasure?](#)

---

## Why does the compiler add casts when it translates generics?

*Because the return type of methods of a parameterized type might change as a side effect of type erasure.*

During type erasure the compiler replaces type parameters by the leftmost bound, or type `Object` if no bound was specified. This means that methods whose return type is the type parameter would return a reference that is either the leftmost bound or `Object`, instead of the more specific type that was specified in the parameterized type and that the caller expects. A cast is need from the leftmost bound or `Object` down to the more specific type..

Example (before type erasure):

```
public class Pair<X,Y> {
    private X first;
    private Y second;
    public Pair(X x, Y y) {
        first = x;
        second = y;
    }
    public X getFirst() { return first; }
    public Y getSecond() { return second; }
    public void setFirst(X x) { first = x; }
    public void setSecond(Y y) { second = y; }
}

final class Test {
    public static void main(String[] args) {
        Pair<String,Long> pair = new Pair<String,Long>("limit", 10000L);
        String s = pair.getFirst();
        Long l = pair.getSecond();
        Object o = pair.getSecond();
    }
}
```

Example (after type erasure):

```
public class Pair {
    private Object first;
    private Object second;
    public Pair(Object x, Object y) {
        first = x;
        second = y;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object x) { first = x; }
    public void setSecond(Object y) { second = y; }
}

final class Test {
    public static void main(String[] args) {
        Pair pair = new Pair("limit", 10000L);
        String s= (String) pair.getFirst();
    }
}
```

```

    Long l = (Long) pair.getSeond();
    Object o = pair.getSecond();
}

```

After type erasure the methods `getFirst` and `getSecond` of type `Pair` both have the return type `Object`. Since the declared static type of the pair in our test case is `Pair<String,Long>` the caller of `getFirst` and `getSecond` expects a `String` and a `Long` as the return value. Without a cast this would not work and in order to make it work the compiler adds the necessary casts from `Object` to `String` and `Long` respectively.

The inserted casts cannot fail at runtime with a `ClassCastException` because the compiler already made sure at compile-time that both fields are references to objects of the expected type. The compiler would issue an error method if arguments of types other than `String` or `Long` had been passed to the constructor or the `set` methods. Hence it is guaranteed that these casts cannot fail.

In general, casts silently added by the compiler are guaranteed not to raise a `ClassCastException` if the program was compiled without warnings. This is the type-safety guarantee.

Implicit casts are inserted when methods are invoked whose *return type* changed during type erasure. Invocation of methods whose *argument type* changed during type erasure do not require insertion of any casts. For instance, after type erasure the `setFirst` and `setSecond` methods of class `Pair` take `Object` arguments. Invoking them with arguments of a more specific type such as `String` and `Long` is possible without the need for any casts.

LINK TO THIS [Technicalities.FAQ104](#)

REFERENCES [What is type erasure?](#)  
[What does type-safety mean?](#)

## How does type erasure work when a type parameter has several bounds?

*The compiler adds casts as needed.*

In the process of type erasure the compiler replaces type parameters by their leftmost bound, or type `Object` if no bound was specified. How does that work if a type parameter has several bounds?

Example (before type erasure):

```

interface Runnable {
    void run();
}
interface Callable<V> {
    V call();
}
class X<T extends Callable<Long> & Runnable> {
    private T task1, task2;
    ...
    public void do() {

```

```
    task1.run();
    Long result = task2.call();
}
}
```

Example (after type erasure):

```
interface Runnable {
    void run();
}
interface Callable {
    Object call();
}
class X {
    private Callable task1, task2;
    ...
    public void do() {
        ((Runnable)task1).run();
        Long result = (Long) task2.call();
    }
}
```

The type parameter `T` is replaced by the bound `Callable`, which means that both fields are held as references of type `Callable`. Methods of the leftmost bound (which is `Callable` in our example) can be called directly. For invocation of methods of the other bounds (`Runnable` in our example) the compiler adds a cast to the respective bound type, so that the methods are accessible. The inserted cast cannot fail at runtime with a `ClassCastException` because the compiler already made sure at compile-time that both fields are references to objects of a type that is within both bounds.

In general, casts silently added by the compiler are guaranteed not to raise a `ClassCastException` if the program was compiled without warnings. This is the type-safety guarantee.

LINK TO THIS [Technicalities.FAQ105](#)

REFERENCES [What does type-safety mean?](#)

---

## What is a reifiable type?

*A type whose type information is fully available at runtime, that is, a type that does not lose information in the course of type erasure.*

As a side effect of type erasure, some type information that is present in the source code is no longer available at runtime. For instance, parameterized types are translated to their corresponding raw type in a process called *type erasure* and lose the information regarding their type arguments.

For example, types such as `List<String>` or `Pair<? extends Number, ? extends Number>` are available to and used by the compiler in their exact form, including the type argument information. After type erasure, the virtual machine has only the raw types `List` and `Pair` available, which means that part of the type information is lost.

In contrast, non-parameterized types such as `java.util.Date` or `java.lang.Thread.State` are not affected by type erasure. Their type information remains exact, because they do not have type arguments.

Among the instantiations of a generic type only the unbounded wildcard instantiations, such as `Map<?,?>` or `Pair<?,?>`, are unaffected by type erasure. They do lose their type arguments, but since all type arguments are unbounded wildcards, no information is lost.

Types that do NOT lose any information during type erasure are called *reifiable types*. The term reifiable stems from *reification*. Reification means that type parameters and type arguments of generic types and methods are available at runtime. Java does not have such a runtime representation for type arguments because of type erasure. Consequently, the reifiable types in Java are only those types for which reification does not make a difference, that is, the types that do not need any runtime representation of type arguments.

The following types are reifiable:

- primitive types
- non-generic (or non-parameterized) reference types
- unbounded wildcard instantiations
- raw types
- arrays of any of the above

The non-reifiable types, which lose type information as a side effect of type erasure, are:

- instantiations of a generic type with at least one concrete type argument
- instantiations of a generic type with at least one bounded wildcard as type argument

Reifiable types are permitted in some places where non-reifiable types are disallowed. Reifiable types are permitted (and non-reifiable types are prohibited):

- as type in an `instanceof` expression
- as component type of an array

#### LINK TO THIS

[Technicalities.FAQ106](#)

#### REFERENCES

[What is type erasure?](#)

[What is reification?](#)

[What is an unbounded wildcard parameterized type?](#)

[What is the raw type?](#)

[Which types can or must not appear as target type in an instanceof expression?](#)

[Can I create an array whose component type is a concrete parameterized type?](#)

[Can I create an array whose component type is a wildcard parameterized type?](#)

[Why is it allowed to create an array whose component type is an unbounded wildcard parameterized type?](#)

---

## What is the type erasure of a parameterized type?

*The type without any type arguments.*



The erasure of a parameterized type is the type without any type arguments (i.e. the raw type). This definition extends to arrays and nested types.

Examples:

<i>parameterized type</i>	<i>type erasure</i>
List<String>	List
Map.Entry<String,Long>	Map.Entry
Pair<Long,Long>[]	Pair[]
Comparable<? super Number>	Comparable

The type erasure of a non-parameterized type is the type itself.

LINK TO THIS [Technicalities.FAQ107](#)

REFERENCES [What is the raw type?](#)

---

## What is the type erasure of a type parameter?

*The type erasure of its leftmost bound, or type `Object` if no bound was specified.*

The type erasure of a type parameter is the erasure of its leftmost bound. The type erasure of an unbounded type parameter is type `Object`.

Examples:

<i>type parameters</i>	<i>type erasure</i>
<T>	Object
<T extends Number>	Number
<T extends Comparable<T>>	Comparable
<T extends Cloneable & Comparable<T>>	Cloneable
<T extends Object & Comparable<T>>	Object
<S, T extends S>	Object, Object

LINK TO THIS [Technicalities.FAQ108](#)

REFERENCES [What is a bounded type parameter?](#)

---

## What is the type erasure of a generic method?

*A method with the same name and the types of all method parameters replaced by their respective type erasures.*

The erasure of a method signature is a signature consisting of the same name and the erasures of all the formal method parameter types.

Examples:

<i>parameterized method</i>	<i>type erasure</i>
<code>Iterator&lt;E&gt; iterator()</code>	<code>Iterator iterator()</code>
<code>&lt;T&gt; T[] toArray(T[] a)</code>	<code>Object[] toArray(Object[] a)</code>
<code>&lt;U&gt; AtomicLongFieldUpdater&lt;U&gt; newUpdater(Class&lt;U&gt; tclass, String fieldName)</code>	<code>AtomicLongFieldUpdater newUpdater(Class tclass, String fieldName)</code>

LINK TO THIS [Technicalities.FAQ109](#)

REFERENCES [What is type erasure?](#)  
[What is the type erasure of a parameterized type?](#)  
[What is the type erasure of a type parameter?](#)

---

## Is generic code faster or slower than non-generic code?

*There is no perceivable difference.*

Some programmers, especially those with a C++ background, expect that generic code should perform much faster than non-generic code, because this is one observable benefit of using templates in C++. Other programmers assume that the synthetic bridge methods and implicit casts inserted by the compiler in the process of type erasure would degrade the runtime performance of generic code. Which one is true?

The short answer is: it is likely that one will find neither a substantial difference in runtime performance nor any consistent trend. However, this has not yet been verified by any benchmarks I know of. Nevertheless, let us take a look at the various overlapping effects that might explain such a statement.

*Implicit casts.*

The casts added by the compiler are exactly the casts that would appear in non-generic code. Hence the implicit casts do not add any overhead.

Example (generic code):

```
List<String> list = new List<String>();  
list.add("abc");  
String s = list.get(0);
```

Example (non-generic code):

```
List list = new LinkedList();  
list.add("abc");  
String s = (String)list.get(0);
```

Example (after type erasure):

```
List list = new LinkedList();
list.add("abc");
String s = (String)list.get(0);
```

The non-generic code is exactly what the compiler generates in the process of type erasure, hence there is no difference in performance.

### *Bridge methods.*

The compiler adds bridge methods. These synthetic methods cause an additional method invocation at runtime, they are represented in the byte code and increase its volume, and they add to the memory footprint of the program.

Example (generic code):

```
final class Byte implements Comparable<Byte> {
    private byte value;
    public Byte(byte value) {
        this.value = value;
    }
    public byte byteValue() { return value; }
    public int compareTo(Byte that) {
        return this.value - that.value;
    }
}
```

Example (non-generic code):

```
final class Byte implements Comparable {
    private byte value;
    public Byte(byte value) {
        this.value = value;
    }
    public byte byteValue() { return value; }
    public int compareTo(Object that) {
        return this.value - ((Byte)that).value;
    }
}
```

Example (after type erasure):

```
final class Byte implements Comparable {
    private byte value;
    public Byte(byte value) {
        this.value = value;
    }
    public byte byteValue() { return value; }
    public int compareTo(Byte that) {
        return this.value - that.value;
    }
    public int compareTo(Object that) {
        return this.compareTo((Byte)that);
    }
}
```

It is likely that there is a slight performance penalty for the bridge method that affects runtime execution and class loading. However, only new (i.e. 5.0) source code is affected. If we compile legacy (i.e. 1.4-compatible) source code, there are no additional bridge methods and the byte code should be identical, more or less, to the way it was before. Most likely the slight performance penalty is compensated for by improvements in Hotspot.

### *Runtime type information.*

Static information about type parameters and their bounds is made available via reflection. This runtime type information adds to the size of the byte code and the memory footprint, because the information must be loaded into memory at runtime. Again, this only affects new (i.e. 5.0) source code. On the other hand, there are some enhancements to reflection that apply even to existing language features, and those do require slightly larger class files, too. At the same time, the representation of runtime type information has been improved. For example, there is now an access bit for "synthetic" rather than a class file attribute, and class literals now generate only a single instruction. These things often balance out. For any particular program you might notice a very slight degradation in startup time due to slightly larger class files, or you might find improved running time because of shorter code sequences. Yet it is unlikely that one will find any large or consistent trends.

### *Compilation time.*

Compiler performance might decrease because translating generic source code is more work than translating non-generic source code. Just think of all the static type checks the compiler must perform for generic types and methods. On the other hand, the performance of a compiler is often more dominated by its implementation techniques rather than the features of the language being compiled. Again, it is unlikely that one will find any perceivable or measurable trends.

**LINK TO THIS**                      [Technicalities.FAQ110](#)

**REFERENCES**                      [How does the compiler translate Java generics?](#)  
[What is type erasure?](#)  
[What is a bridge method?](#)  
[Why does the compiler add casts when it translates generics?](#)

---

## **How do I compile generics for use with JDK <= 1.4?**

### *Use a tool like Retroweaver.*

Retroweaver is a Java bytecode weaver that enables you to take advantage of the new 1.5 language features in your source code, while still retaining compability with 1.4 virtual machines. Retroweaver rewrites class files so that they are compatible with older virtual machines. Check out <http://sourceforge.net/projects/retroweaver>.

**LINK TO THIS**                      [Technicalities.FAQ111](#)

**REFERENCES**                      [How does the compiler translate Java generics?](#)  
[Retroweaver Download Page](#)

---

## Type System

### How do parameterized types fit into the Java type system?

*Instantiations of generic types have certain super-subtype relationship among each other and have a type relationship to their respective raw type. These type relationships are relevant for method invocation, assignment and casts.*

*Relevance of type relationships and type conversion rules in practice.*

The type system of a programming language determines which types are convertible to which other types. These conversion rules have an impact on various areas of a programming language. One area where conversion rules and type relationships play role is casts and `instanceof` expressions. Other area is assignment compatibility and method invocation, where argument and return value passing relies on convertibility of the involved types.

The type conversion rules determine which casts are accepted and which ones are rejected. For example, the types `String` and `Integer` have no relationship and for this reason the compiler rejects the attempt of a cast from `String` to `Integer`, or vice versa. In contrast, the types `Number` and `Integer` have a super-subtype relationship; `Integer` is a subtype of `Number` and `Number` is a supertype of `Integer`. Thanks to this relationship, the compiler accepts the cast from `Number` to `Integer`, or vice versa. The cast from `Integer` to `Number` is not even necessary, because the conversion from a subtype to a supertype is considered an implicit type conversion, which need not be expressed explicitly in terms of a cast; this conversion is automatically performed by the compiler whenever necessary. The same rules apply to `instanceof` expressions.

The conversion rules define which types are assignment compatible. Using the examples from above, we see that a `String` cannot be assigned to an `Integer` variable, or vice versa, due to the lack of a type relationship. In contrast, an `Integer` can be assigned to a `Number` variable, but not vice versa. A side effect of the super-subtype relationship is that we can assign a subtype object to a supertype variable, without an explicit cast anywhere. This is the so-called *widening reference conversion*; it is an implicit conversion that the compiler performs automatically whenever it is needed. The converse, namely assignment of a supertype object to a subtype variable, is not permitted. This is because the so-called *narrowing reference conversion* is not an implicit conversion. It can only be triggered by an explicit cast.

The rules for assignment compatibility also define which objects can be passed to which method. An argument can be passed to a method if its type is assignment compatible to the declared type of the method parameter. For instance, we cannot pass an `Integer` to a method that asks for `String`, but we can pass an `Integer` to a method that asks for a `Number`. The same rules apply to the return value of a method.

*Super-subtype relationships of parameterized types.*

In order to understand how objects of parameterized types can be used in assignments, method invocations and casts, we need an understanding of the relationship that parameterized types have among each other and with non-parameterized types. And we need to know the related conversion rules.

We already mentioned *super-subtype relationships* and the related *narrowing* and *widening reference conversions*. They exist since Java was invented, that is, among non-generic types. The super-subtype relationship has been extended to include parameterized types. In the Java 5.0 type system super-subtype relationships and the related narrowing/widening reference conversions exist among parameterized types, too. We will explain the details in separate FAQ entries. Here are some initial examples to get a first impression of the impact that type relationships and conversion rules have on method invocation.

Consider a method whose declared parameter type is a wildcard parameterized type. A wildcard parameterized type acts as supertype of all members of the type family that the wildcard type denotes.

Example (of widening reference conversion from concrete instantiation to wildcard instantiation):

```
void printAll(LinkedList<? extends Number> c) { ... }

LinkedList<Long> l = new LinkedList<Long>();
...
printAll(l); // widening reference conversion
```

We can pass a `List<Long>` as an argument to the `printAll` method that asks for a `LinkedList<? extends Number>`. This is permitted thanks to the super-subtype relationship between a wildcard instantiation and a concrete instantiation. `LinkedList<Long>` is a member of the type family denoted by `LinkedList<? extends Number>`, and as such a `LinkedList<Long>` is a subtype of `LinkedList<? extends Number>`. The compiler automatically performs a widening conversion from subtype to supertype and thus allows that a `LinkedList<Long>` can be supplied as argument to the `printAll` method that asks for a `LinkedList<? extends Number>`.

Note that this super-subtype relationship between a wildcard instantiation and a member of the type family that the wildcard denotes is different from inheritance. Inheritance implies a super-subtype relationship as well, but it is a special case of the more general super-subtype relationship that involves wildcard instantiations.

We know inheritance relationships from non-generic Java. It is the relationship between a superclass and its derived subclasses, or a super-interface and its sub-interfaces, or the relationship between an interface and its implementing classes. Equivalent inheritance relationships exist among instantiations of different generic types. The prerequisite is that the instantiations must have the same type arguments. Note that this situation differs from the super-subtype relationship mentioned above, where we discussed the relationship between wildcard instantiations and concrete instantiations of the *same* generic type, whereas we now talk of the relationship between instantiations of *different* generic types with identical type arguments.

Example (of widening reference conversion from one concrete parameterized type to another concrete parameterized type):

```
void printAll(Collection<Long> c) { ... }

LinkedList<Long> l = new LinkedList<Long>();
...
printAll(l); // widening reference conversion
```

The raw types `Collection` and `LinkedList` have a super-subtype relationship; `Collection` is a supertype of `LinkedList`. This super-subtype relationship among the raw types is extended to the parameterized types, provided the type arguments are identical: `Collection<Long>` is a supertype of `LinkedList<Long>`, `Collection<String>` is a supertype of `LinkedList<String>`, and so on.

It is common that programmers believe that the super-subtype relationship among type arguments would extend into the respective parameterized type. This is not true. Concrete instantiations of the same generic type for different type arguments have no type relationship. For instance, `Number` is a supertype of `Integer`, but `List<Number>` is not a supertype of `List<Integer>`. A type relationship among different instantiations of the same generic type exists only among wildcard instantiations and concrete instantiations, but never among concrete instantiations.

Example (of illegal attempt to convert between different concrete instantiations of the same generic type):

```
void printAll(LinkedList<Number> c) { ... }
```

```
LinkedList<Long> l = new LinkedList<Long>();
...
printAll(l); // error; no conversion
```

Due to the lack of a type relationship between `LinkedList<Number>` and `LinkedList<Long>` the compiler cannot convert the `LinkedList<Long>` to a `LinkedList<Number>` and the method call is rejected with an error message.

### *Unchecked conversion of parameterized types.*

With the advent of parameterized types a novel category of type relationship was added to the Java type system: the relationship between a parameterized type and the corresponding raw type. The conversion from a parameterized type to the corresponding raw type is a widening reference conversion like the conversion from a subtype to the supertype. It is an implicit conversion. An example of such a conversion is the conversion from a parameterized type such as `List<String>` or `List<? extends Number>` to the raw type `List`. The counterpart, namely the conversion from the raw type to an instantiation of the respective generic type, is the so-called *unchecked conversion*. It is an automatic conversion, too, but the compiler reports an "unchecked conversion" warning. Details are explained in separate FAQ entries. Here are some initial examples to get a first impression of usefulness of unchecked conversions. They are mainly permitted for compatibility between generic and non-generic source code.

Below is an example of a method whose declared parameter type is a raw type. The method might be a pre-Java-5.0 method that was defined before generic and parameterized types had been available in Java. For this reason it declares `List` as the argument type. Now, in Java 5.0, `List` is a raw type.

Example (of a widening reference conversion from parameterized type to raw type):

```
void printAll(List c) { ... }

List<String> l = new LinkedList<String>();
...
printAll(l); // widening reference conversion
```

Source code such as the one above is an example of a fairly common situation, where non-generic legacy code meets generic Java 5.0 code. The `printAll` method is an example of legacy code that was developed before Java 5.0 and uses raw types. If more recently developed parts of the program use instantiations of the generic type `List`, then we end up passing an instantiation such as `List<String>` to the `printAll` method that declared the raw type `List` as its parameter type. Thanks to the type relationship between the raw type and the parameterized type, the method call is permitted. It involves an automatic widening reference conversion from the parameterized type to the raw type.

Below is an example of the conversion in the opposite direction. We consider a method has a declared parameter type that is a parameterized type. We pass a raw type argument to the method and rely on an unchecked conversion to make it work.

Example (of an unchecked conversion from raw type to parameterized type):

```
void printAll(List<Long> c) { ... }

List l = new LinkedList();
...
printAll(l); // unchecked conversion
```

Like the previous example, this kind of code is common in situation where generic and non-generic code are mixed.

The subsequent FAQ entries discuss details of the various type relationships and conversions among raw types, concrete parameterized types, bounded and unbounded wildcard parameterized types.

LINK TO THIS

[Technicalities.FAQ201](#)

REFERENCES

[What is the raw type?](#)

[What is a wildcard parameterized type?](#)

[Which super-subtype relationships exist among instantiations of generic types?](#)

[How does the raw type relate to instantiations of the corresponding generic type?](#)

[How do instantiations of a generic type relate to instantiations of other generic types that have the same type argument?](#)

[How do unbounded wildcard instantiations of a generic type relate to other instantiations of the same generic type?](#)

[How do wildcard instantiations with an upper bound relate to other instantiations of the same generic type?](#)

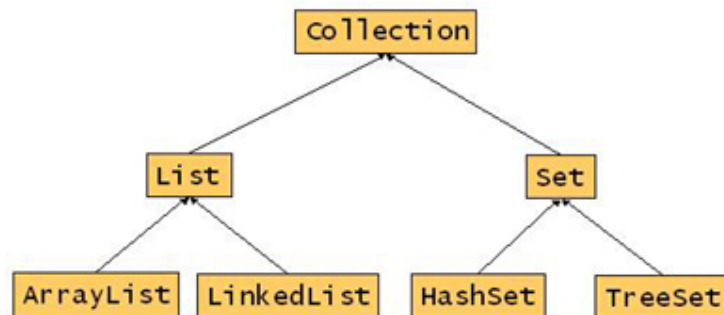
[How do wildcard instantiations with a lower bound relate to other instantiations of the same generic type?](#)

---

## How does the raw type relate to instantiations of the corresponding generic type?

*The raw type is the supertype of all instantiations of the corresponding generic type.*

The raw types have the regular supertype-subtype relationship with other raw types. For illustration we use the collection classes and interfaces from the JDK (see package [java.util](#)).



In addition, the raw types are supertypes of all concrete and all wildcard instantiations of the generic type. For instance, the raw type `Collection` is a supertype of all instantiations of the generic type `Collection`.





Regarding conversions, the usual reference widening conversion from subtype to supertype is allowed. That is, every instantiation of a generic type can be converted to the corresponding raw type. The reverse is permitted, too, for reasons of compatibility between generic and non-generic types. It is the so-called unchecked conversion and is accompanied by an "unchecked" warning. In detail, the conversion from a raw type to a concrete or bounded wildcard instantiation of the corresponding generic type leads to a warning. The conversion from the raw type to the unbounded wildcard instantiation is warning-free.

LINK TO THIS

[Technicalities.FAQ202](#)

REFERENCES

[How do parameterized types fit into the Java type system?](#)

[What is the raw type?](#)

[What is a concrete parameterized type?](#)

[What is a wildcard parameterized type?](#)

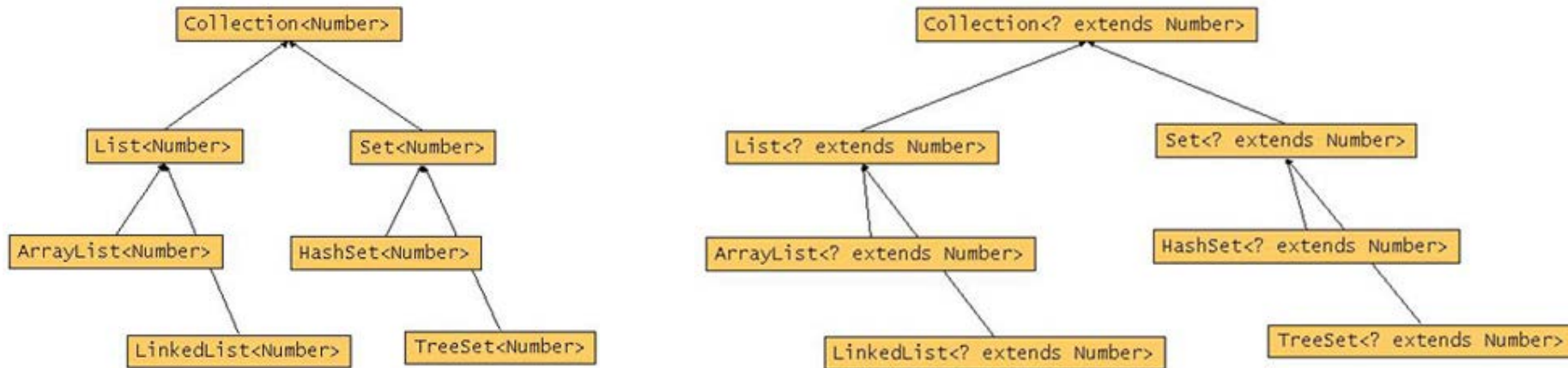
[What is the unbounded wildcard parameterized type?](#)

[Which super-subtype relationships exist among instantiations of generic types?](#)

## How do instantiations of a generic type relate to instantiations of other generic types that have the same type argument?

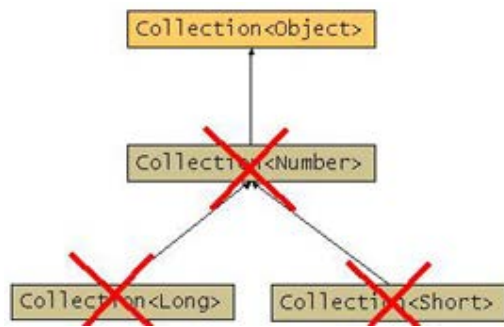
*An instantiation of a generic type is the supertype of all instantiations of generic subtypes that have the same type argument.*

Instantiations of the generic types have supertype-subtype relationships with concrete instantiations of generic subtypes provided they all have the exact same type arguments. The example below uses the JDK collection types (see package [java.util](#)) for illustration.



The diagram illustrates the super-subtype relationship among instantiations that have the same type argument. The type argument can be a concrete type, but also a bounded or unbounded wildcard. For instance, `Collection<Number>` is the supertype of `List<Number>` and `LinkedList<Number>`, `Collection<? extends Number>` is the supertype of `List<? extends Number>` and `LinkedList<? extends Number>`, `Collection<?>` is the supertype of `List<?>` and `LinkedList<?>`, and so on.

Type relationships to other concrete instantiations do not exist. In particular, the supertype-subtype relationship among the type arguments does not extend to the instantiations. For example, `Collection<Number>` is NOT a supertype of `Collection<Long>`.



Regarding conversions, the usual reference widening conversion from subtype to supertype is allowed. The reverse is not permitted.

**LINK TO THIS**

[Technicalities.FAQ203](#)

**REFERENCES**

[How do parameterized types fit into the Java type system?](#)

[What is the raw type?](#)

[What is a concrete parameterized type?](#)

[What is a wildcard parameterized type?](#)

[What is the unbounded wildcard parameterized type?](#)

[Which super-subtype relationships exist among instantiations of generic types?](#)

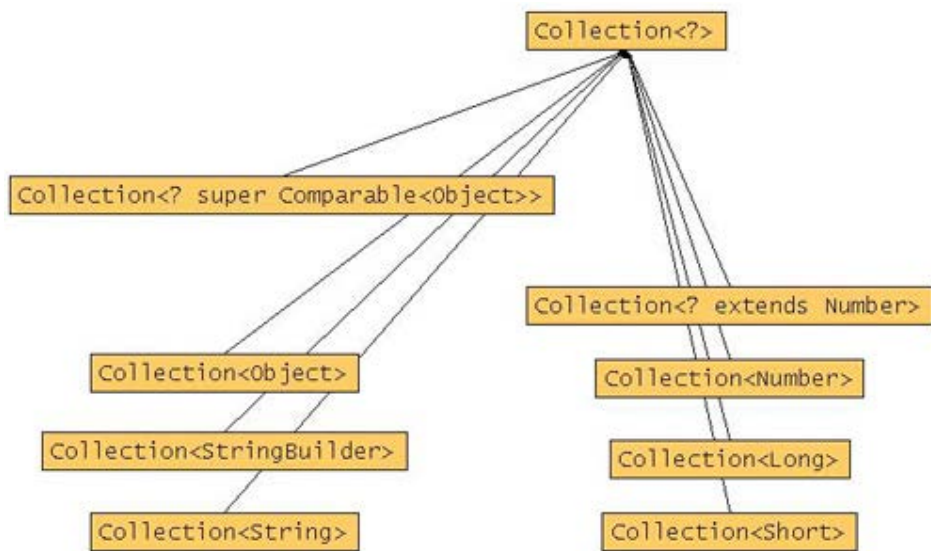
---

## How do unbounded wildcard instantiations of a generic type relate to other instantiations of the same generic type?

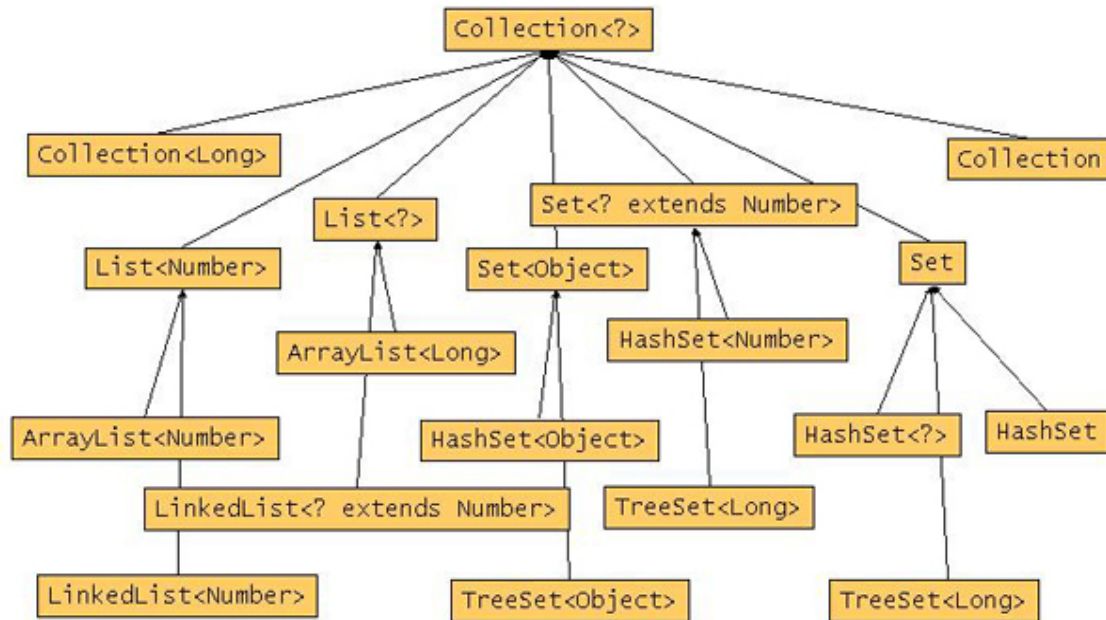
*An unbounded wildcard instantiation is the supertype of all instantiations of the generic type.*

The unbounded wildcard instantiation of a generic type is supertype of all concrete and all wildcard instantiations of the same generic type.

For instance, the unbounded wildcard instantiation `Collection<?>` is supertype of all instantiations of the generic type `Collection`. The example below uses the JDK collection types (see package [java.util](#)) for illustration.



At the same, an unbounded wildcard instantiation is supertype of all unbounded instantiations of any generic subtypes. For instance, the unbounded wildcard instantiation `Collection<?>` is supertype of `List<?>`, `LinkedList<?>`, and so on. Both type relationships combined, an unbounded wildcard instantiation is supertype of all instantiations of the same generic type and of all instantiations of all its generic subtypes.



Regarding conversions, the usual reference widening conversion from subtype to supertype is allowed. The reverse is not permitted.

There is one special rule for unbounded wildcard instantiations: the conversion from a raw type to the unbounded wildcard instantiation is not an "unchecked" conversion, that is, the conversion from `Collection` to `Collection<?>` does not lead to an "unchecked" warning. This is different for concrete and bounded instantiations, where the conversion from the raw type to the concrete and bounded wildcard instantiation leads to an "unchecked" warning.

LINK TO THIS [Technicalities.FAQ204](#)

REFERENCES

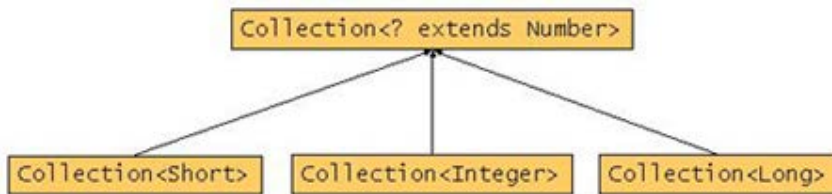
- [How do parameterized types fit into the Java type system?](#)
- [Which super-subtype relationships exist among instantiations of parameterized types?](#)
- [What is the raw type?](#)
- [What is a concrete parameterized type?](#)
- [What is a wildcard parameterized type?](#)
- [What is the unbounded wildcard parameterized type?](#)
- [What is a wildcard?](#)
- [What is an unbounded wildcard?](#)
- [What is a bounded wildcard?](#)

---

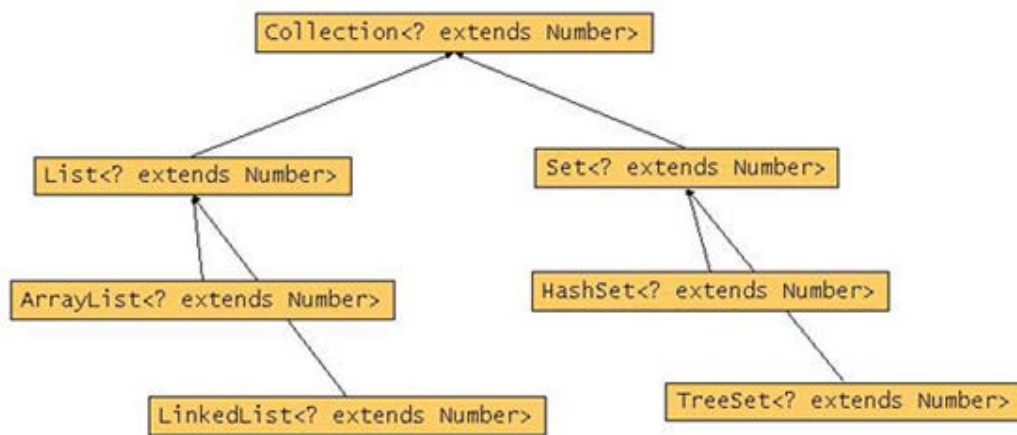
## How do wildcard instantiations with an upper bound relate to other instantiations of the same generic type?

***A wildcard instantiation with an upper bound is supertype of all instantiations of the same generic type where the type argument is a subtype of the upper bound.***

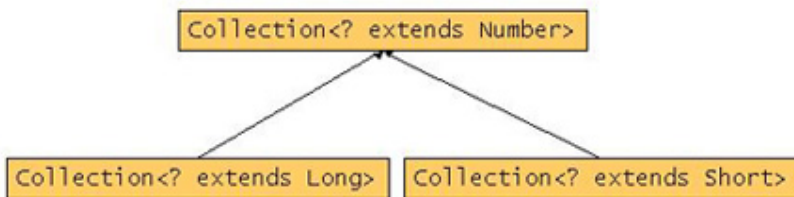
An upper bound wildcard instantiation is supertype of all concrete instantiations with type arguments that are subtypes of the upper bound, the upper bound itself being included. For instance, `Collection<? extends Number>` is supertype of `Collection<Number>`, `Collection<Long>`, `Collection<Short>`, etc., because `Number`, `Long`, and `Short` are subtypes (or same type) of `Number`. The underlying idea is: a subtype of the upper bound (e.g. `Long`) belongs to the family of types that the wildcard (e.g. `? extends Number`) stands for and in this case the wildcard instantiation (e.g. `Collection<? extends Number>`) is a supertype of the concrete instantiation on the subtype of the upper bound (e.g. `Collection<Long>`).



At the same time, a wildcard instantiation with an upper bound is supertype of all generic subtypes that are instantiated on the same upper bound wildcard. For instance, `Collection<? extends Number>` is supertype of `Set<? extends Number>` and `ArrayList<? extends Number>`.



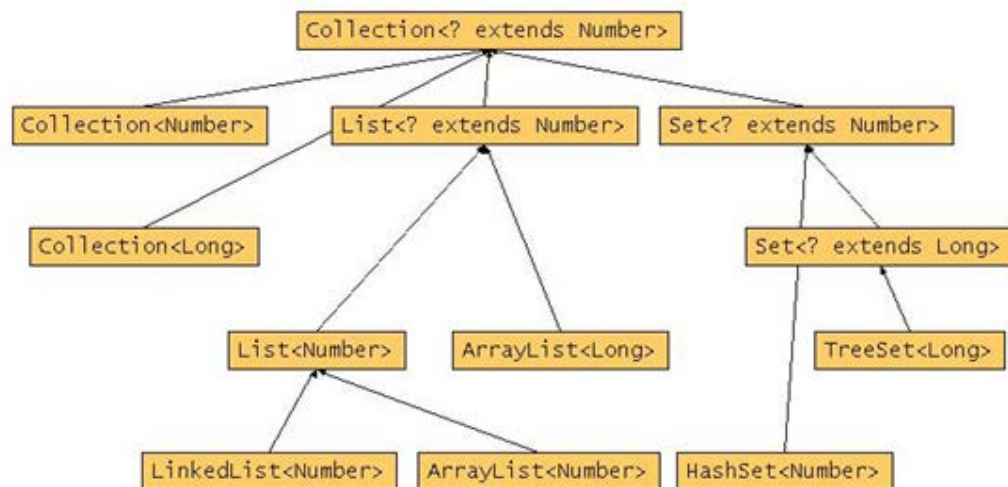
The upper bound wildcard instantiation is also supertype of other upper bound wildcard instantiation with an upper bound that is a subtype of the own upper bound. For instance, `Collection<? extends Number>` is supertype of `Collection<? extends Long>` and `Collection<? extends Short>`, because `Long` and `Short` are subtypes of `Number`.



Similarly, `Collection<? extends Comparable<?>>` is supertype of `Collection<? extends Number>` and `Collection<? extends Delayed>`, because `Number` is a subtype of `Comparable<Number>`, which is a subtype of `Comparable<?>`, and `Delayed` (see [java.util.concurrent.Delayed](http://java.util.concurrent.Delayed)) is a subtype of `Comparable<Delayed>`, which is a subtype of `Comparable<?>`. The idea is that if the upper bound of one wildcard is a supertype of the upper bound of another wildcard then the type family with the supertype bound includes the type family with the subtype bound. If one family of types (e.g. `? extends Comparable<?>`) includes the other (e.g. `? extends Number>` and `? extends Delayed`) then the wildcard instantiation on the larger family (e.g. `Collection<? extends Comparable<?>>`) is supertype of the wildcard instantiation of the included family (e.g. `Collection<? extends Number>`).

All these type relationships combined make a bounded wildcard instantiation supertype of a quite a number of instantiations of the same generic type and subtypes thereof.





Regarding conversions, the usual reference widening conversion from subtype to supertype is allowed. The reverse is not permitted.

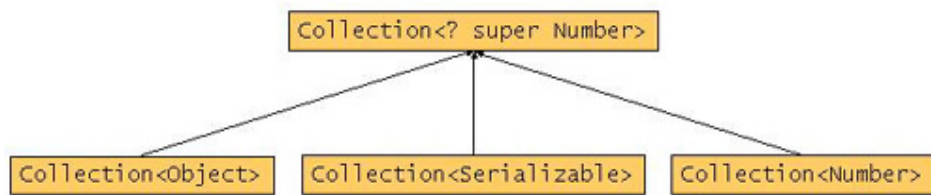
LINK TO THIS [Technicalities.FAQ205](#)

- REFERENCES
- [How do parameterized types fit into the Java type system?](#)
  - [Which super-subtype relationships exist among instantiations of parameterized types?](#)
  - [What is the raw type?](#)
  - [What is a concrete parameterized type?](#)
  - [What is a wildcard parameterized type?](#)
  - [What is the unbounded wildcard parameterized type?](#)
  - [What is a wildcard?](#)
  - [What is an unbounded wildcard?](#)
  - [What is a bounded wildcard?](#)

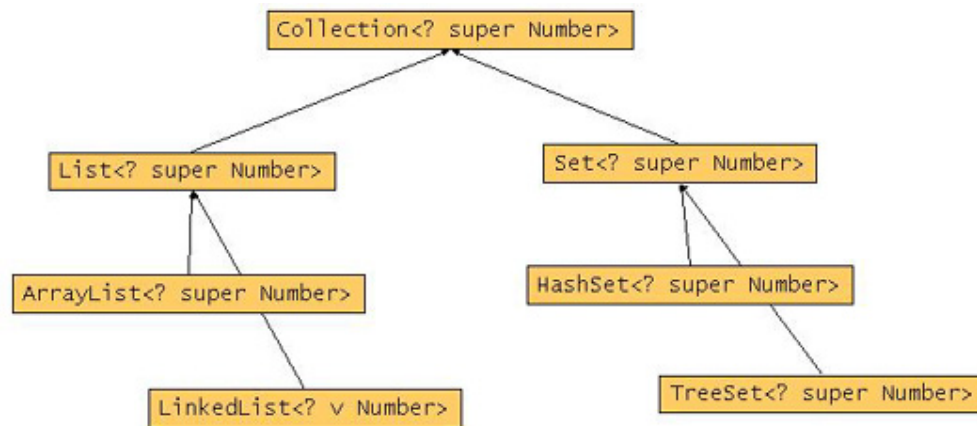
## How do wildcard instantiations with a lower bound relate to other instantiations of the same generic type?

*A wildcard instantiation with a lower bound is supertype of all instantiations of the generic type where the type argument is a supertype of the lower bound.*

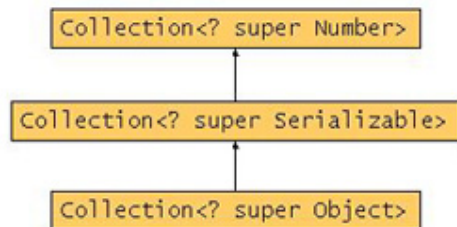
A wildcard instantiation with a lower bound is supertype of all concrete instantiation with type arguments that are supertypes of the lower bound, the lower bound itself included. For instance, `Collection<? super Number>` is supertype of `Collection<Number>`, `Collection<Serializable>`, and `Collection<Object>`, because `Number`, `Serializable` and `Object` are supertypes (or same type) of `Number`. The underlying idea is: a supertype of the lower bound (e.g. `Object`) belongs to the family of types that the wildcard (e.g. `? super Number`) stands for and in this case the wildcard instantiation (e.g. `Collection<? super Number>`) is a supertype of the concrete instantiation on the supertype of the lower bound (e.g. `Collection<Object>`).



At the same time, a wildcard instantiation with a lower bound is supertype of parameterized subtypes that are instantiated on the same lower bound wildcard. For instance, `Collection<? super Number>` is supertype of `Set<? super Number>` and `ArrayList<? super Number>`.



The lower bound wildcard instantiation is also supertype of other lower bound wildcard instantiation with a lower bound bound that is a supertype of the own lower bound. For instance, `Collection<? super Number>` is supertype of `Collection<? super Serializable>`, because `Serializable` is a supertype of `Number`. The idea is that if the lower bound of one wildcard is a subtype of the lower bound of another wildcard then the type family with the subtype bound includes the type family with the supertype bound. If one family of types (e.g. `? super Number`) includes the other (e.g. `? super Serializable`) then the wildcard instantiation on the larger family (e.g. `Collection<? super Number>`) is supertype of the wildcard instantiation of the included family (e.g. `Collection<? super Serializable>`).



Regarding conversions, the usual reference widening conversion from subtype to supertype is allowed. The reverse is not permitted.

LINK TO THIS [Technicalities.FAQ206](#)

REFERENCES [How do parameterized types fit into the Java type system?](#)  
[Which super-subtype relationships exist among instantiations of parameterized types?](#)



[What is the raw type?](#)

[What is a concrete parameterized type?](#)

[What is a wildcard parameterized type?](#)

[What is the unbounded wildcard parameterized type?](#)

[What is a wildcard?](#)

[What is an unbounded wildcard?](#)

[What is a bounded wildcard?](#)

---

## Which super-subtype relationships exist among instantiations of generic types?

*This is fairly complicated and the type relationships are best determined setting up tables as explained in this FAQ entry.*

Super-subtype relationships among instantiations of generic types are determined by two orthogonal aspects.

On the one hand, there is the inheritance relationship between a supertype and a subtype. This is the usual notion of inheritance as we know it from non-generic Java. For instance, the interface `Collection` is a supertype of the interface `List`. This inheritance relationship is extended in analogy to instantiations of generic types, i.e. to parameterized types. The prerequisite is that the instantiations must have identical type arguments. An example is the supertype `Collection<Long>` and its subtype `List<Long>`. The rule is: as long as the type arguments are identical, the inheritance relationship among generic types leads to a super-subtype relationship among corresponding parameterized types.

On the other hand, there is a relationship based on the type arguments. The prerequisite is that at least one of the involved type arguments is a wildcard. For example, `Collection<? extends Number>` is a supertype of `Collection<Long>`, because the type `Long` is a member of the type family that the wildcard "`? extends Number`" denotes.

This kind of type relationship also exists between two wildcard instantiations of the same generic type with different wildcards as type arguments. The prerequisite is that the type family denoted by one wildcard is a superset of the type family denoted by the other wildcard. For example, `Collection<?>` is a supertype of `Collection<? extends Number>`, because the family of types denoted by the wildcard "?" is a superset of the family of types denoted by the wildcard "`? extends Number`". The super-subset relationship among the type arguments leads to a super-subtype relationship among corresponding instantiations of the same parameterized type. The type relationship mentioned above, between a wildcard instantiation and a concrete instantiation of the same generic type, is a special case of this rule; you just interpret the concrete type argument as a type family with only one member, namely the concrete type itself.

Both effects - the super-subtype relationship due to inheritance and the super-subtype relationship due to type arguments - are combined and lead to a two-dimensional super-subtype relationship table. The tables below use examples for illustration.

The vertical axis of the table lists parameterized types according to their inheritance relationship, starting with the supertype on the top to a subtype on the bottom. The horizontal axis lists type arguments according to their super-subset relationship of the type families they denote, starting with the largest type set on the lefthand side to the smallest type set on the righthand side.

→  
superset => subset

	?	? extends Serializable	? extends Number	Long
Collection	Collection<?>	Collection<? extends Serializable>	Collection<? extends Number>	Collection<Long>

$\downarrow$ supertype $\Leftarrow$ subtype $\Leftarrow$ subtype	<b>List</b>	List<?>	<b>List&lt;? extends Serializable&gt;</b>	List<? extends Number>	List<Long>
	<b>ArrayList</b>	ArrayList<?>	ArrayList<? extends Serializable>	ArrayList<? extends Number>	ArrayList<Long>

If you pick a certain entry in the table, say List<? extends Serializable>, then the subtable to the bottom and to the right contains all subtypes of the entry.

Below is another example that involves lower bound wildcards. Again, the horizontal axis lists type arguments according to their super-subset relationship of the type families they denote, starting with largest set of types denoted by the unbounded wildcard "?" over type families of decreasing size to a type set consisting of one concrete type. The difficulty with lower bound wildcards is that the super-subset relationship of the type families denoted by lower bound wildcards is slightly counter-intuitive to determine. Details are discussed in a separate FAQ entry.

		$\rightarrow$ superset $\Rightarrow$ subset			
$\downarrow$ supertype $\Leftarrow$ subtype $\Leftarrow$ subtype		?	? super Long	? super Number	Object
	<b>Collection</b>	Collection<?>	Collection<? super Long>	Collection<? super Number>	Collection<Object>
	<b>List</b>	List<?>	<b>List&lt;? super Long&gt;</b>	List<? super Number>	List<Object>
	<b>ArrayList</b>	ArrayList<?>	ArrayList<? super Long>	ArrayList<? super Number>	ArrayList<Object>

If you pick a certain entry in the table, say List<? super Long>, then the subtable to the bottom and to the right contains all subtypes of the entry and you would find information such as: ArrayList<? super Number> is a subtype of List<? super Long>.

### Generic Types With More Than One Type Parameter

We have been setting up tables to determine the super-subtype relationships among different instantiations of different, yet related parameterized types. These tables were two-dimensional because we took into account inheritance on the one hand and various values for a type argument on the other hand. A similar technique can be applied to parameterized types with more than one type parameter.

The example below uses a generic class Pair with two type parameters. The vertical axis lists the first type arguments order by their super-subset relationship from the largest type set to the smallest type set. The horizontal axis does the same for the second type argument.

		$\rightarrow$ superset $\Rightarrow$ subset			
		?	? super Long	? super Number	Object

?	Pair<?,?>	Pair<?,? super Long>	Pair<?,? super Number>	Pair<?,Object>
? extends Serializable	Pair<? extends Serializable,?>	<b>Pair&lt;? extends Serializable,? super Long&gt;</b>	Pair<? extends Serializable,? super Number>	Pair<? extends Serializable,Object>
? extends Number	Pair<? extends Number,?>	Pair<? extends Number,? super Long>	Pair<? extends Number,? super Number>	Pair<? extends Number,Object>
Long	Pair<Long,?>	Pair<Long,? super Long>	Pair<Long,? super Number>	Pair<Long,Object>

Tables with more than two dimensions can be set up in analogy. The key point is that you list generic types from supertype to subtype and type arguments from superset to subset. For this purpose you need to interpret type arguments as sets of types and determine their super-subtype relationships. Note that the latter can be rather counter-intuitive in case of multi-level wildcards involving lower bounds. Details are discussed in a separate FAQ entry.

LINK TO THIS

[Technicalities.FAQ207](#)

REFERENCES

[How do parameterized types fit into the Java type system?](#)

[Which super-subset relationships exist among wildcards?](#)

[What is the raw type?](#)

[What is a concrete parameterized type?](#)

[What is a wildcard parameterized type?](#)

[What is the unbounded wildcard parameterized type?](#)

## Which super-subset relationships exist among wildcards?


*A super-subtype relationship between upper bounds leads to a super-subset relationship between the resulting upper bound wildcards, and vice versa for a lower bounds.*

A wildcard denotes a family (or set) of types. These type sets can have super-subset relationships, namely when one type set includes the other type set. The super-subset relationship among wildcards determines the super-subtype relationships of instantiations of parameterized types using these wildcards as type arguments. The super-subtype relationships among instantiations of parameterized types play an important role in method invocation, assignments, type conversions, casts and type checks (as was discussed in a previous FAQ entry). For this reason, we need to know how to determine the super-subset relationships among the type sets denoted by wildcards.


Here are the rules:

- The unbounded wildcard "?" denotes the set of all types and is the superset of all sets denoted by bounded wildcards.
- A wildcard with an upper bound A denotes a superset of another wildcard with an upper bound B, if A is a supertype of B.
- A wildcard with a lower bound A denotes a superset of another wildcard with a lower bound B, if A is a subtype of B.
- A concrete type denotes a set with only one element, namely the type itself.

upper bound wildcards    lower bound wildcards

	? ? extends SuperType ? extends SubType <b>concrete type*</b> *either SubType or a subtype thereof	? ? super SubType ? super SuperType <b>concrete type*</b> *either SuperType or a supertype thereof
--	--	--

Here are some examples illustrating the rules:

	upper bound wildcards	lower bound wildcards
	? ? extends Serializable ? extends Number Long	? ? super Long ? super Number Serializable

The wildcard "?" denotes the largest possible type set, namely the set of all types. It is the superset of all type sets.

Upper bound wildcards are fairly easy to understand, compared to lower bound wildcards. The wildcard "? extends Serializable" denotes the family of types that are subtypes of Serializable, the type Serializable itself being included. Naturally, this type family is a subset of the type set denoted by the wildcard "?".

The wildcard "? extends Number" denotes the family of types that are subtypes of Number, the type Number itself being included. Since Number is a subtype of Serializable, the type family "? extends Number" is a subset of the type family "? extends Serializable". In other words, the super-subtype relationship between the upper bounds leads to a super-subset relationship between the resulting type sets.

The concrete type Long denotes a single-member type set, which is a subset of the type family "? extends Number" because Long is a subtype of Number.

Among the lower bound wildcards, the wildcard "? super Long" denotes the family of types that are supertypes of Long, the type Long itself being included. The wildcard "? super Number" denotes the family of types that are supertypes of Number, the type Number itself being included. Number is a supertype of Long, and for this reason the type family "? super Number" is smaller than the type family "? super Long"; the latter includes type Long as member, while the former excludes it. In other words, the super-subtype relationship between the lower bounds leads to the opposite relationship between the resulting type sets, namely a sub-superset relationship.

### Multi-Level Wildcards

Matters are more complicated when it comes to multi-level wildcards. In principle, the rules outlined above are extended to multi-level wildcards in analogy. However, the resulting super-subset relationships tend to be everything but intuitive to understand, so that in practice you will probably want to refrain from overly complex multi-level wildcards. Nonetheless, we discuss in the following the rules for super-subset relationships among two-level wildcards.

For multi-level wildcards we apply the same rules as before. The only difference is that the bounds are wildcards instead of concrete types. As a result we do not consider type relationships among the bounds, but set relationships.

- A wildcard with an upper bound A denotes a superset of another wildcard with an upper bound B, if A denotes a superset of B.
- A wildcard with a lower bound A denotes a superset of another wildcard with a lower bound B, if A denotes a subset of B.

The bounds A and B are wildcards and we can look up their super-subset relationships in the tables above, which leads us to the following tables for multi-level wildcards:

	upper-upper bound wildcards	upper-lower bound wildcards
$\downarrow$ <small>isqns &lt;= pscndms</small>	?	?
	? extends ParType<?>	? extends ParType<?>
	? extends ParType<? extends SuperType>	? extends ParType<? super SubType>
	? extends ParType<? extends SubType>	? extends ParType<? super SuperType>
	? extends ParType<concrete type*> *either SubType or a subtype thereof	? extends ParType<concrete type*> *either SuperType or a supertype thereof
	concrete type*<concrete type**> *either ParType or a subtype thereof **either SubType or a subtype thereof	concrete type*<concrete type**> *either ParType or a subtype thereof **either SuperType or a supertype thereof
	lower-upper bound wildcards	lower-lower bound wildcards
$\downarrow$ <small>isqns &lt;= pscndms</small>	?	?
	? super ParType<concrete type*> *either SubType or a subtype thereof	? super ParType<concrete type*> *either SuperType or a supertype thereof
	? super ParType<? extends SubType>	? super ParType<? super SuperType>
	? super ParType<? extends SuperType>	? super ParType<? super SubType>
	? super ParType<?>	? super ParType<?>
	concrete type*<?> *either ParType or a supertype thereof	concrete type*<?> *either ParType or a supertype thereof

The rules look fairly complex, but basically it is a recursive process. For instance: the type set denoted by "? extendsParType<? extends SuperType>" is a superset of the type set denoted by "? extends ParType<? extends SubType>" because the upper bound "? extends SuperType" is a superset of the upper bound "? extends SubType", and this is because the inner upper bound SuperType is a supertype of the inner upper bound SubType. In this recursive way, the super-subset relationships of multi-level wildcards can be made plausible. Here are some concrete examples:

Examples:

	upper-upper bound wildcards	upper-lower bound wildcards
$\downarrow$ <small>is &lt;= pscndms</small>	?	?
	? extends List<?>	? extends List<?>
	? extends List<? extends Serializable>	? extends List<? super Long>
	? extends List<? extends Number>	? extends List<? super Number>

	? extends List<Long> ArrayList<Long>		? extends List<Serializable> LinkedList<Serializable>
	<b>lower-upper bound wildcards</b>		<b>lower-lower bound wildcards</b>
	? ? super List<Long> ? super List<? extends Number> ? super List<? extends Serializable> ? super List<?> Collection<?>		? ? super List<Object> ? super List<? super Serializable> ? super List<? super Number> ? super List<?> Collection<?>

**LINK TO THIS**

[Technicalities.FAQ208](#)

**REFERENCES**

- [How do parameterized types fit into the Java type system?](#)
- [Which super-subtype relationships exist among instantiations of parameterized types?](#)
- [What is the raw type?](#)
- [What is a concrete instantiation?](#)
- [What is a wildcard instantiation?](#)
- [What is the unbounded wildcard instantiation?](#)

## Does "extends" always mean "inheritance"?

*No.*

`extends` is an overloaded keyword in the Java programming language. It has different meanings, depending on the context in which it appears. The `extends` keyword can appear in four different locations:

- in the definition of a class
- in the definition of an interface
- in the definition of type parameter bounds
- in the definition of wildcard bounds

Below we discuss each of these cases in detail. Basically it boils down to the observation that `extends` refers to more general super-subtype relationship, of which inheritance is a special case. In conjunction with class and interface definitions `extends` means inheritance. In conjunction with type parameter bounds and wildcard bounds `extends` it means either inheritance, or identity, or member of a type family denoted by a wildcard instantiation.

### *Definition of classes and interfaces*

`extends`

Example ( in the definition of a class):

```
public class Quadruple<T> extends Triple<T> {
    private T fth;
    public Triple(T t1, T t2, T t3, T t4) {
        super(t1, t2, t3);
        fth = t4;
    }
    ...
}
```

This is an example where `extends` means inheritance. We define a subclass `Quadruple` that is derived from a superclass `Triple`. This kind of inheritance exists between non-generic and generic classes. It leads to a super-subtype relationship between the two types. In case of generic types it leads to a super-subtype relationship between instantiations of the two types that have identical type arguments. For instance, `Triple<Long>` is a subtype of `Triple<Long>`, and `Quadruple<? extends Number>` is a subtype of `Triple<? extends Number>`.

Example (`extends` in the definition of an interface):

```
public interface SortedSet<E> extends Set<E> {
    ...
}
```

This is another example where `extends` means inheritance. This time the inheritance relationship exists between two interfaces instead of two classes. We have the super-subtype relationships as for classes.

### *Definition of type parameter bounds*

Example (`extends` in the definition of type parameter bounds):

```
public class Caller<V, T extends Callable<V>> {
    public Caller(T task) {
        FutureTask<V> future = new FutureTask<V>(task);
        Thread thread = new Thread(future);
        thread.setDaemon(false);
        thread.start();
        try { System.out.println ("result: " + future.get()); }
        catch (Exception e) { e.printStackTrace(); }
    }
}
```

In this example `extends` does not mean inheritance. The `extends` keyword is here used to define the bounds of a type parameter. A common misunderstanding is that the type argument that later replaces the type parameter in an instantiation must inherit from the bounds. This is often the case, but it is not the only option.

In our example, we can supply a sub-interface of `Callable<V>` as a type argument, that is, an interface that extends the bound. But we can also supply a class type that implements the bound. In other words, `extends` in conjunction with type parameter bounds does not strictly mean inheritance, but it also includes the `implements`-relationship that exists between classes and interfaces.

In conjunction with type parameter bounds the `extends` keyword refers to an even broader notion of subtyping. It includes relationships that cannot be expressed in terms of `extends` or `implements` as we know them from non-generic Java. Consider the following example.

Example (`extends` in the definition of a type parameter bound that is a final class):

```
public class SomeClass<T extends String> {  
    ...  
}
```

In this example the bound is a final class. Final classes cannot be inherited from. The only permitted type argument is type `String`, which is not a type that inherits from the bound; it is the bound itself. When applied to a bound that is a final class, `extends` does not mean inheritance; it means *identity*.

Let us consider another example that demonstrates that `extends` in conjunction with type parameter bounds means more than inheritance.

Example (`extends` in the definition of a type parameter bound that is a wildcard parameterized type):

```
public class SomeClass<T extends Collection<?>> {  
    ...  
}
```

In this example the bound is a wildcard instantiation of the `Collection` interface. Wildcard parameterized types cannot be inherited from. In this case `extends` does not mean inheritance either. It refers to the super-subtype relationship that exists among wildcard parameterized types and concrete parameterized type.

Conceivable type arguments would be concrete parameterized type, such as `Collection<String>` or `List<Long>`, but also other wildcard parameterized types, such as `Collection<? extends Number>`. These type arguments do not inherit from the bound, but they are members of the type family that the bound denotes. When applied to a bound that is a wildcard parameterized type, `extends` does not mean inheritance; it means *member of the type family* denoted by the wildcard parameterized type.

### *Definition of wildcard bounds*

Example (`extends` in the definition of a wildcard bound):

```
List<? extends Number> ref = new ArrayList<Number>();
```

The meaning of `extends` in conjunction with wildcard bounds is similar to the meaning of `extends` in conjunction with type parameter bounds. It does NOT mean that the unknown type that the wildcard stands for (the so-called *wildcard capture*) must inherit from the wildcard bound. It can, but it does not have to. The capture can equally well be the bound itself and `extends` would mean identity instead of inheritance.

If the bound is a wildcard parameterized type, then `extends` refers to the subtype relationship that exists among wildcard parameterized types and concrete parameterized types.

Example (`extends` in the definition of a wildcard bound):

```
Triple<? extends Collection<?>> q = new Triple<List<? extends Number>>();
```

In this example the wildcard capture cannot be a type that inherits from the bound, because wildcard parameterized types such as `Collection<?>` cannot be inherited from. Instead the wildcard capture must be a member of the type family denoted by `Collection<?>`, such as `Collection<String>`, `List<Number>`,



List<?> Or List<? extends Number>. Again, extends does not mean inheritance; it means *member of the type family* denoted by the wildcard parameterized type.

LINK TO THIS [Technicalities.FAQ209](#)

REFERENCES [How do parameterized types fit into the Java type system?](#)  
[Which super-subset relationships exist among wildcards?](#)  
[What is the capture of a wildcard?](#)

---

## Exception Handling

### Can I use generic or parameterized types in exception handling?

*No. Exception and error types must not be generic.*

It is illegal to define generic types that are directly or indirectly derived from class `Throwable`. Consequently, no instantiations of generic type appear anywhere in exception handling.

LINK TO THIS [Technicalities.FAQ301](#)

REFERENCES [Why are generic exception and error types illegal?](#)

---

### Why are generic exception and error types illegal?

*Because the virtual machine cannot distinguish between different instantiations of a generic exception type.*

A generic class must not directly or indirectly be derived from class `Throwable`, which means that generic exception or error types are not allowed. Imagine if they were allowed ...

Example (of illegal generic exception type):

```
class IllegalArgumentException<T> extends Exception { // illegal
    private T info;
    public IllegalArgumentException(T arg) { info = arg; }
    public T getInfo() { return info; }
}
```

We might then want to catch instantiations of this (illegal) generic exception type.

Example (of illegal use of illegal parameterized exception type):

```
void method_1() {
    try { method_2(); }
    catch (IllegalArgumentException<String> e) { ... } // illegal
    catch (IllegalArgumentException<Long> e) { ... } // illegal
    catch (Throwable e) { ... }
}
```

Taking into account that generic Java source code is translated to Java byte code by type erasure, it should be clear that the method's `catch` clauses do not make any sense. Both parameterized exception types have the same runtime type and the mechanism for catching exceptions is a runtime mechanism performed by the virtual machine based on the non-exact runtime types. The JVM has no chance to distinguish between different instantiations of the same generic (exception) type. For this reason, generic exception and error types are pointless in Java and banned from the language. (Note that generic exception and error types are not pointless per se, but in the context of Java generics with type erasure they are nonsensical.)

Other problems occur when we define methods that throw instantiations of an (illegal) generic exception type.

Example (of illegal use of illegal parameterized exception type):

```
void method_1()
    throws IllegalArgumentException<String>, IllegalArgumentException<Long> { // illegal
    ... do something ...
    throw new IllegalArgumentException<String>("argument missing");
    ... do something else ...
    throw new IllegalArgumentException<Long>(timeout);
}
```

Again, after type erasure, both parameterized exception types have the same runtime type and the method's `throws` clause is nonsensical. The method could at best throw the raw type, which means that it must not create and throw any instantiations of the (illegal) generic type either. Another reason to disallow generic exception and error types.

LINK TO THIS [Technicalities.FAQ302](#)

REFERENCES [What is type erasure?](#)

---

## Can I use a type parameter in exception handling?

*It depends.*

Type parameters can appear in `throws` clauses, but not in `catch` clauses.

LINK TO THIS [Technicalities.FAQ303](#)

REFERENCES [Can I use a type parameter in a catch clause?](#)  
[Can I use a type parameter in in a throws clause?](#)

## Can I use a type parameter in a catch clause?

*No.*

Using a type parameter in a `catch` clause is nonsensical because of the translation by type erasure.

Example (before type erasure):

```
<E extends Exception> void someMethod() {
    try { &hellip; do something that might raise an exception &hellip;
    } catch (E e) { &hellip; do something special for this particular exception type &hellip;
    } catch (IllegalStateException e) { &hellip; handle illegal state &hellip;
    } catch (Exception e) { &hellip; handle all remaining exceptions &hellip;
    }
}
```

Example (after type erasure):

```
void someMethod() {
    try { &hellip; do something that might raise an exception &hellip;
    } catch (Exception e) { &hellip; do something special for this particular exception type &hellip;
    } catch (IllegalStateException e) { &hellip; handle illegal state &hellip;
    } catch (Exception e) { &hellip; handle all remaining exceptions &hellip;
    }
}
```

After type erasure the `catch` clause would boil down to a `catch` clause using the type parameter's bound. This is because type parameters do not have a runtime type representation of themselves. Type parameters are replaced by their leftmost bound in the course of translation by type erasure. In our example the `catch` clause for the unknown exception type `E` is translated by type erasure to a `catch` clause for type `Exception`, which is the bound of the type parameter `E`. This `catch` clause for type `Exception` precedes further `catch` clauses for other exception types, and renders them pointless.

In other words, there never is a `catch` clause for the particular unknown exception type that the type argument stands for. Instead of catching a particular exception type we end up catching the bound of the unknown exception type, which changes the meaning of the sequence of `catch` clauses substantially and is almost always undesired. For this reason, the use of type parameters in `catch` clauses is illegal.

### LINK TO THIS

[Technicalities.FAQ304](#)

### REFERENCES

[Can I use a type parameter in in a throws clause?](#)

[Can I throw an object whose type is a type parameter?](#)

---

## Can I use a type parameter in in a throws clause?

**Yes.**

Using a type parameter in a `throws` clause is permitted.

Example (before type erasure):

```
public interface Action<E extends Exception> {
    void run() throws E;
}
public final class Executor {
    public static <E extends Exception>
        void execute(Action<E> action) throws E {
        &hellip;
        action.run();
        &hellip;
    }
}
public final class Test {
    private static class TestAction implements Action<java.io.FileNotFoundException> {
        public void run() throws java.io.FileNotFoundException {
            &hellip;
            throw new java.io.FileNotFoundException();
            &hellip;
        }
    }
    public static void main(String[] args) {
        try {
            Executor.execute(new TestAction());
        } catch (java.io.FileNotFoundException f) { &hellip; }
    }
}
```

In this example we see a generic interface `Action` whose type parameter `E` is the exception type that its `run` method throws. This is perfectly reasonable, because `throws` clauses are a compile-time feature and the lack of a runtime type representation of the type parameter is not needed. At runtime, a particular exception type will have replaced the type parameter `E`, so that we would throw and catch an object of a concrete exception type. In our example we instantiate the `Action` interface using the `FileNotFoundException` exception type as a type argument, so that a `FileNotFoundException` exception is raised and caught.

Even after type erasure the code snippet above still captures the intent.

Example (after type erasure):

```
public interface Action {
    void run() throws Exception;
}
public final class Executor {
    public static void execute(Action action) throws Exception {
        &hellip;
    }
}
```

```

        action.run();
        &hellip;
    }
}
public final class Test {
    private static class TestAction implements Action {
        public void run() throws java.io.FileNotFoundException {
            &hellip;
            throw new java.io.FileNotFoundException();
            &hellip;
        }
    }
    public static void main(String[] args) {
        try {
            Executor.execute(new TestAction());
        } catch (java.io.FileNotFoundException f) { &hellip; }
    }
}

```

LINK TO THIS

[Technicalities.FAQ305](#)

REFERENCES

[Can I use a type parameter in a catch clause?](#)

[Can I throw an object whose type is a type parameter?](#)

---

## Can I throw an object whose type is a type parameter?

*In principle, yes, but in practice, not really.*

We can declare methods that throw an exception (or error) of unknown type.

Example (of method with type parameter in `throws` clause):

```

interface Task<E extends Exception> {
    void run() throws E;
}

```

Can such a method throw an object of the unknown exception (or error) type? The method has in principle 3 ways of raising such an exception (or error):

- create a new exception (or error) and throw it
- catch the exception (or error) of an invoked operation and re-throw it
- propagate the exception (or error) of an invoked operation

As we must not create objects of an unknown type the first possibility is not an option. Since type parameters must not appear in catch clauses, we cannot catch

and therefore not re-throw an exception of an unknown type. At best, we can propagate an exception of unknown type that is raised by any of the invoked operations.

Example (throwing an object of unknown type):

```
final class Cleanup<E extends Exception, T extends Task<E>> {
    public void cleanup(T task) throws E {
        task.run();
    }
}
final class DisconnectTask implements Task<IllegalAccessException> {
    public void run() throws IllegalAccessException {
        ...
        throw new IllegalAccessException();
        ...
    }
}
class Test {
    public static void main(String[] args) {
        Cleanup<IllegalAccessException, DisconnectTask> cleaner
            = new Cleanup<IllegalAccessException, DisconnectTask>();
        try { cleaner.cleanup(new DisconnectTask()); }
        catch (IllegalAccessException e) { e.printStackTrace(); }
    }
}
```

LINK TO THIS

[Technicalities.FAQ306](#)

REFERENCES

[Can I create an object whose type is a type parameter?](#)

[Can I use a type parameter in a catch clause?](#)

[Can I use a type parameter in a throws clause?](#)

---

## Static Context

**How do I refer to static members of a parameterized type?**

*Using the raw type as the scope qualifier, instead of the any instantiation of the generic type.*

If you refer to a static member of a generic type, the static member name must be preceded by the name of the enclosing scope, such as `EnclosingType.StaticMember`. In case of a generic enclosing type the question is: which instantiation of the generic type can or must be used as the scope qualifier?

The rule is that no instantiation can be used. The scope is qualified using the raw type. This is because there is only one instance of a static member per type.

Example (of a generic type with static members):

```
public final class Counted<T> {
    public static final int MAX = 1024;
    public static class BeyondThresholdException extends Exception {}

    private static int count;
    public static int getCount() { return count; }

    private final T value;
    public Counted(T arg) throws BeyondThresholdException {
        value = arg;
        count++;
        if (count >= 1024) throw new BeyondThresholdException();
    }
    public void finalize() { count--; }
    public T getValue() { return value; }
}
```

---

```
int m = Counted.MAX;           // ok
int k = Counted<Long>.MAX;     // error
int n = Counted<?>.MAX;       // error

try {
    Counted<?>[] array = null;
    array[0] = new Counted<Long>(10L);
    array[1] = new Counted<String>("abc");
}
catch (Counted.BeyondThresholdException e) {
    e.printStackTrace();
}

System.out.println(Counted.getCount()); // ok
System.out.println(Counted<Long>.getCount()); // error
System.out.println(Counted<?>.getCount()); // error
```

In the example above, the generic class `Counted` has a static member `MAX` and there is only one unique `Counted.MAX`, regardless of the number of objects of type `Counted` and regardless of the number of instantiations of the generic type `Counted` that may be used somewhere in the program. Referring to `MAX` as `Counted<String>.MAX`, `Counted<Long>.MAX`, `Counted<?>.MAX`, etc. would be misleading, because it suggests that there were several manifestations of the `MAX`

member, which is not true. There is only one `Counted.MAX`, and it must be referred to using the raw type `Counted` as the scope qualifier. The same is true for other static members such as static methods and static nested types. The example illustrates that we must refer to the static method as `Counted.getCount` and to the static nested exception class as `Counted.BeyondThresholdException`.

In sum, it is illegal to refer to a static member using an instantiation of the generic enclosing type. This is true for all categories of static members, including static fields, static methods, and static nested types, because each of these members exists only once per type.

LINK TO THIS [Technicalities.FAQ351](#)

#### REFERENCES

[Is there one instances of a static field per instantiation of a generic type?](#)  
[Why can't I use a type parameter in any static context of the generic class?](#)  
[How do I refer to an interface type nested into a generic type?](#)  
[How do I refer to an enum type nested into a generic type?](#)  
[How do I refer to a \(non-static\) inner class of a generic type?](#)  
[Can I import a particular instantiation of generic type?](#)

---

## How do I refer to a (non-static) inner class of a generic type?

*Using an instantiation of the enclosing generic type as the scope qualifier, instead of the raw type.*

Static nested types are referred to using the raw form of the enclosing generic type as the scope qualifier. This is different for inner classes.

Like a static nested type, an inner class exists only once, in the sense that there is only one `.class` file that represents the inner class. Different from a static nested type, an inner class depends on the type argument of its outer class type. This is because each object of an inner class type has a hidden reference to an object of the outer class type. The type of this hidden reference is an instantiation of the generic enclosing type. As a result, the inner class type is not independent of the enclosing class's type arguments.

Example (of an inner class nested into a generic outer class):

```
class Sequence<E> {
    private E[] theSequence;
    private int idx;

    // static classes
    public static class NoMoreElementsException extends Exception {}
    public static class NoElementsException extends Exception {}

    // (non-static) inner class
    public class Iterator {
        boolean hasNext() {
            return (theSequence != null && idx < theSequence.length);
        }
        E getNext() throws NoElementsException,
            NoMoreElementsException,
            java.lang.IllegalStateException {
```



```

        if (theSequence == null)
            throw new NoElementsException();
        if (idx < 0)
            throw new java.lang.IllegalStateException();
        if (idx >= theSequence.length)
            throw new NoMoreElementsException();
        else
            return theSequence[idx++];
    }
}
public Iterator getIterator() {
    return this.new Iterator();
}
}

class Test {
    private static <T> void print(Sequence<T> seq)
        throws Sequence.NoElementsException,
            Sequence.NoMoreElementsException {
        Sequence<T>.Iterator iter = seq.new Iterator();
        while (iter.hasNext())
            System.out.println(iter.getNext());
    }
    public static void main(String[] args) {
        try {
            Sequence<String> seq1 = new Sequence<String>();
            ... fill the sequence ...
            print(seq1);
            Sequence<Long> seq2 = new Sequence<Long>();
            ... fill the sequence ...
            print(seq2);
        } catch (Exception e) { e.printStackTrace(); }
    }
}
}

```

In the example above, the inner class `Iterator` depends on the outer class's type parameter `E`: the type parameter `E` is the return type of the iterator's `getNext` method and the inner class access the outer class's array of elements of type `E`. For this reason, the iterator type is referred to as `Sequence<T>.Iterator` (using an instantiation), instead of just `Sequence.Iterator` (using the raw type). This does not imply, that a scope qualification using the raw type is illegal; it is permitted and it means that the outer object being referred to by the inner object is of the raw type instead of a more specific type.

In contrast, the nested exception types are *static* classes and do *not* depend on the outer class's type parameter. (Static nested types never depend on their enclosing type's type parameters, because the type parameters must not appear in any static context of a generic class. Details are explained in a separate FAQ entry.) Static nested classes must be referred to using the raw type as the scope qualifier, that is, as `Sequence.NoElementsException`; use of an instantiation as the scope, such as `Sequence<T>.NoElementsException`, is illegal.

---

## How do I refer to an interface type nested into a generic type?

*Using the raw type as the scope qualifier, instead of the any instantiation of the generic type.*

Nested interfaces are implicitly static. This is sometimes confusing because the interface looks like it were a non-static member of its enclosing class, while in fact it is static. As a static member of a generic type it must be referred to using the raw form of the enclosing type as the scope qualifier. Using an instantiation as the scope qualifier is illegal and rejected by the compiler.

Example (of a nested interface):

```
class Controller<E extends Executor> {
    private E executor;
    public Controller(E e) { executor = e; }
    ...
    public interface Command { void doIt(Runnable task); }

    public Command command() {
        return new Command() {
            public void doIt(Runnable task) { executor.execute(task); }
        };
    }
}
class Test
    public static void test(Controller<ThreadPoolExecutor> c) {
        Controller<ExecutorService> controller
            = new Controller<ExecutorService>(Executors.newCachedThreadPool());
        Controller.Command command = controller.command();
        ...
    }
}
```

The `Command` interface is nested into the generic `Controller` class. The compiler does not allow that we refer to the nested interface using any instantiation of the enclosing generic class as the scope qualifier. Instead of saying `Controller<ExecutorService>.Command` we must say `Controller.Command`.

Below is another example of a nested interface taken from the `java.util` package. The generic `Map` interface has a nested `Entry` interface.

Example (of a nested interface taken from package `java.util`):

```
public interface Map<K,V> {
    public interface Entry<K,V> {
```

```

    public K getKey();
    public V getValue();
    ...
}
public Set<Map.Entry<K, V>> entrySet();
...
}

```

The source code above is an excerpt from the JDK source code. Note that the nested interface `Entry` is generic itself and has its own type parameters, which are independent of the outer interface's type parameters. The fact that the type parameters of inner and outer interface have the same names, namely `K` and `V`, is perhaps confusing, but perfectly legal. The inner interface's type parameters `K` and `V` are visible only inside the inner interface and have nothing to do with the outer interface's type parameters `K` and `V`.

When the inner interface `Entry` is used it must be referred to using the raw type `Map` as the scope qualifier, that is, as `Map.Entry<String,Long>` for instance. A qualification such as `Map<String,Long>.Entry<String,Long>` is illegal.

**LINK TO THIS**                    [Technicalities.FAQ353](#)

**REFERENCES**                    [Why can't I use a type parameter in any static context of the generic class?](#)  
[How do I refer to static members of a parameterized type?](#)  
[Can I import a particular instantiation of parameterized type?](#)

## How do I refer to an enum type nested into a generic type?

*Using the raw type as the scope qualifier, instead of the any instantiation of the generic type.*

Nested enum types are implicitly static. This is sometimes confusing because the enum type looks like it were a non-static member of its enclosing class, while in fact it is static. As a static member of a generic type it must be referred to using the raw form of the enclosing type as the scope qualifier. Using an instantiation as the scope qualifier is illegal and rejected by the compiler.

Example (of a nested enum type):

```

class Controller<E extends Executor> {
    private State state;
    ...
    public enum State { VALID, INVALID; }
    public State getState() { return state; }
}
class Test
public static void test(Controller<ThreadPoolExecutor> c) {
    Controller<ExecutorService> controller
        = new Controller<ExecutorService>(Executors.newCachedThreadPool());
    Controller.State state = controller.getState();
    switch (state) {

```

```
        case INVALID: ... ;
        case VALID: ... ;
    }
    ...
}
```

The enum type `State` is nested into the generic `Controller` class. The compiler does not allow that we refer to the nested interface using any instantiation of the enclosing generic class as the scope qualifier. Instead of saying `Controller<ExecutorService>.State` we must say `Controller.State`. The same applies to the enum constants; they are referred to as `Controller.State.VALID` and `Controller.State.INVALID`.

LINK TO THIS

[Technicalities.FAQ354](#)

REFERENCES

[How do I refer to static members of a parameterized type?](#)

[Can I import a particular parameterized type?](#)

---

## Can I import a particular parameterized type?

*No.*

In an `import` statement we must not use parameterized types; only raw types are permitted. This applies to regular and `static import` statements.

Example (of a generic type with static members):

```
package com.sap.util;

public final class Counted<T> {
    public static final int MAX = 1024;
    public static class BeyondThresholdException extends Exception {}

    private static int count;
    public static int getCount() { return count; }

    private final T value;
    public Counted(T arg) throws BeyondThresholdException {
        value = arg;
        count++;
        if (count >= 1024) throw new BeyondThresholdException();
    }
    public void finalize() { count--; }
    public T getValue() { return value; }
}
```

---

```
import com.sap.util.*;           // ok
```

```
import com.sap.util.Counted; // ok
import com.sap.util.Counted<String>; // error

import static com.sap.Counted.*; // ok
import static com.sap.Counted<String>.*; // error
import static com.sap.Counted.BeyondThresholdException; // ok
import static com.sap.Counted<String>.BeyondThresholdException; // error
```

LINK TO THIS

[Technicalities.FAQ355](#)

REFERENCES

[How do I refer to static members of a parameterized type?](#)

---

## Why are generic enum types illegal?

*Because they do not make sense in Java.*

An enum type is similar to a class type of which only a limited number of instances, namely the enum values, exist. The enum values are static fields of the enum type. The key question is: of which type would the static enum values be if the enum type were allowed to be parameterized?

Example (of an illegal generic enum type):

```
public enum Tag<T> { // illegal, but assume we could do this
    good, bad;

    private T attribute;
    public void setAttribute(T arg) { attribute = arg; }
    public T getAttribute() { return attribute; }
}
```

This enum type would be translated to a class that roughly looks like this:

```
public class Tag<T> extends Enum<Tag<T>> {
    public static final Tag<??> good;
    public static final Tag<??> bad;
    private static final Tag $VALUES[];
    private T attribute;

    private Tag(String s, int i) { super(s, i); }

    static {
        good = new Tag("good", 0);
        bad = new Tag("bad", 1);
        $VALUES = (new Tag[] { good, bad });
    }

    public void setAttribute(T arg) { attribute = arg; }
}
```

```
    public T getAttribute() { return attribute; }  
}
```

The static enum values cannot be of type `Tag<T>` because type parameters such a `T` must not appear in any static context. Should they be of the raw type `Tag` then? In this case the private `attribute` field would be of type `Object`, the invocation of the `setAttribute` method would be flagged an "unchecked call" and the `getAttribute` method would only return `Object`. The entire parameterization would be pointless then.

On the other hand, if we wanted that the type of the enum values is a particular instantiation of the generic enum type, how would we tell the compiler? There is no syntax for specifying the type of an enum value.

Also, when we refer to the enum values we must qualify their name by the name of their defining class, that is, `Tag.good` and `Tag.bad`. Although `Tag` is a parameterized type, we cannot say `Tag<String>.good` or `Tag<Long>.bad`. This is because static members of a generic type must be referred to via the raw type name. In other words, we would not even be capable of expressing that we intend to refer to an enum value belonging to a particular instantiation of the generic enum type.

No matter how we put it: generic enum types do not make sense.

LINK TO THIS [Technicalities.FAQ356](#)

REFERENCES [Why can't I use a type parameter in any static context of the generic class?](#)  
[How do I refer to static members of a parameterized type?](#)

---

## Type Argument Inference

### What is type argument inference?

*The automatic deduction of the type arguments at compile time.*

Type inference happens when the compiler can deduce the type arguments of a generic type or method from context information. In this case the type arguments need not be explicitly specified.

There are two situations in which type inference is attempted:

- *when an object of a generic type is created, and*
- *when a generic method is invoked.*

Example (of automatic type inference on instance creation):

```

class ArrayList<E> {
    ...
}
List<String> list1 = new ArrayList<String>(); // type parameter specified    => E:=String
List<String> list2 = new ArrayList<>();      // type parameter inferred        => E:=String
List<String> list3 = new ArrayList();       // type parameter omitted         => using raw type

```

Class `ArrayList<E>` is a generic class. Usually you must specify the type parameter whenever you use the generic type (unless you want to use the raw type). In an instance creation expression you can omit the type parameter and replace it by empty angle brackets. When the compiler sees the empty angle brackets in the `new`-expression it takes a look at the lefthand side of the assignment in which the `new`-expression appears. From the static type of the lefthand side variable the compiler infers the type parameter of the generic type of the newly created object. In the example above the compiler concludes that the type variable `E` must be replaced by the type `String`. If you neither specify the type parameter no use the empty angle brackets you refer to the raw type.

[Note: Type inference for `new`-expressions was introduced in Java 7 and did not exist in Java 5 and 6.]

Example (of automatic type inference on method invocation):

```

class Collections {
    ...
    public static <T> void copy(List<? super T> dest, List<? extends T> src) { ... }
    ...
}

List<String> src = new ArrayList<>();
List<Object> dst = new ArrayList<>();

Collections.<CharSequence>copy(dst,src); // type parameter specified    => T:=CharSequence
Collections.<>copy(dst,src);             // error: illegal syntax
Collections.copy(dst,src);              // type parameter inferred        => T:=String

```

The `copy()` method is a generic method. When the generic method is invoked without explicit specification of the type argument then the compiler takes a look at the arguments that are provided for the method call. From their static types the compiler infers the type parameter of the generic `copy()` method. In the example above the compiler concludes that the type variable `T` must be replaced by the type `String`. Different from type inference in conjunction with `new`-expressions empty angle bracket are not permitted; the brackets must be omitted entirely.

LINK TO THIS [Technicalities.FAQ400](#)

#### REFERENCES

- [What is type argument inference for generic methods?](#)
- [What is type argument inference for instance creation expressions?](#)
- [What is the diamond operator?](#)
- [Is there a correspondence between type inference for method invocation and type inference for instance creation?](#)

**Is there a correspondence between type inference for method invocation and type inference for instance creation?**

*Yes, the instance creation expression for a generic type is treated like invocation of a generic creator method.*

The rules for type inference for method invocation and type inference for instance creation are the same. Consider a generic class with a constructor:

```
class SomeClass<T> {
    SomeClass(T arg) { ... }
}
```

The creation of an object of this type can involve type inference, e.g. in this example:

```
SomeClass<Long> ref = new SomeClass<>(0L);
```

The type inference for the instance creation is performed in the same way as for a static creator method. If the class had the following creator method:

```
class SomeClass<T> {
    SomeClass(T arg) { ... }
    static <E> SomeClass<E> make(E arg) { return new SomeClass<E>(arg); }
}
```

then the type inference for the invocation of the creator method would yield the same result as type inference for the instance creation. That is, the following leads to equivalent type inference:

```
SomeClass<Long> ref = new SomeClass<>(0L);
SomeClass<Long> ref = SomeClass.make(0L);
```

LINK TO THIS

[Technicalities.FAQ400A](#)

REFERENCES

[What is type argument inference for instance creation expressions?](#)

[What is type argument inference for generic methods?](#)

---

## What is the "diamond" operator?

*It denotes the empty angle brackets that are used for type inference in new-expression.*

Since Java 7 the compiler can infer the type parameters of a parameterized type in a `new`-expression. In order to trigger the type inference the so-called *diamond operator* is used. Below is an example.

Example (of diamond operator):

```
List<String> list1 = new ArrayList<String>();
List<String> list2 = new ArrayList<>();
```

The empty angle brackets `<>` are called the *diamond operator*. The diamond operator is not really an operator in the sense of the syntax specification of the Java programming language. Rather it is just an empty type parameter specification. The empty brackets are needed in order to distinguish between the raw type `ArrayList` and the incomplete type `ArrayList<>`.

LINK TO THIS

[Technicalities.FAQ400B](#)



## What is type argument inference for instance creation expressions?

### *The automatic deduction of the type arguments in a new-expression.*

When an object of a parameterized type is created using a `new`-expression (also called an *instance creation expression*) then the compiler can infer part of the type information of the object to be created. More specifically, the compiler can deduce the type parameters of the parameterized type if it is incomplete. In order to trigger the automatic type inference the so-called *diamond operator* is used (see [Technicalities.FAQ400A](#)). [Note: Type inference for `new`-expressions is available since Java 7.]

Type inference takes into account the context in which the `new`-expression appears and what the `new`-expression looks like. The type inference process works in two separate steps:

- First, the compiler takes a look at the static types of the constructor arguments in the `new`-expression.
- Second, if any of the missing type parameters cannot be resolved from the constructor arguments, then the compiler uses information from the context in which the `new`-expression appears.

In Java 7, the only permitted inference context is an *assignment context*. If the `new`-expression is the right-hand side of an assignment, then the compiler deduces the missing type parameter information from the lefthand side of the assignment, if possible.

Since Java 8, an additional inference context is permitted, namely the *method invocation context*. If the `new`-expression is the argument to a method invocation, then the compiler deduces the missing type parameter information from the method's declared argument type, if possible.

Examples (of type inference for an instance creation expression without constructor arguments):

```
List<String> list1 = new ArrayList<String>();
List<String> list2 = new ArrayList<>();           // type inference

Map<Callable<String>, List<Future<String>>> tasks1
= new HashMap<Callable<String>, List<Future<String>>>();
Map<Callable<String>, List<Future<String>>> tasks2
= new HashMap<>();                             // type inference
```

In the examples above no constructor arguments are specified. As a result, the compiler cannot deduce any type information from the constructor arguments. Instead it takes a look at the static type of the expression on the lefthand side of the assignment and infers the missing type parameters from there.

Examples (of improved type inference in Java 8):

```
List<String> list1 = new ArrayList<>();           // fine since Java 7
List<String> list2 = Collections.synchronizedList(new ArrayList<>()); // error in Java 7; fine in Java 8
```

---

error: incompatible types

```
List<String> list3 = Collections.synchronizedList(new ArrayList<>());
                                     ^
required: List<String>
found:    List<Object>
```

In both examples type inference is needed, because no constructor arguments are specified. The first `new`-expression appears in an assignment context and the compiler infers the missing type parameter `String` from the left-hand side of the assignment. The second `new`-expression appears in a method invocation context. Before Java 8, this yields a compile-time error. Since Java 8, the compiler infers the missing type parameter `String` from the declared type of the argument of the `synchronizedList` method. (Actually, it is a little more complicated. The `synchronizedList` method is a generic method and the compiler must first infer the generic method's type parameter before it knows the method's declared argument type. The generic `synchronizedList` method appears in an assignment context, from which the compiler deduces that the type parameter for `synchronizedList` must be `String`. Its declared argument type then is `List<String>` and from this information the compiler infers that the new `ArrayList` must be an `ArrayList<String>`.)

The inference is different if constructor arguments are provided. In such a context the compiler takes a look at the static types of the constructor arguments and ignores the lefthand side of the assignment.

Examples (of type inference for an instance creation expression with constructor arguments):

```
Set<Long> s1 = new HashSet<>();
Set<Long> s2 = new HashSet<>(Arrays.asList(0L,0L));
Set<Number> s3 = new HashSet<>(Arrays.asList(0L,0L)); // error in Java 7; fine in Java 8
Set<Number> s4 = new HashSet<Number>(Arrays.asList(0L,0L));
```

---

```
error: incompatible types
      Set<Number> s3 = new HashSet<>(Arrays.asList(0L,0L));
                                     ^
required: Set<Number>
found:    HashSet<Long>
```

- The first `new`-expression does not have constructor arguments; the missing type parameter is inferred from the lefthand side of the assignment. The lefthand side is of type `Set<Long>` and compiler concludes that the missing type parameter must be `Long`.
- The second `new`-expression has constructor arguments; the missing type parameter is inferred from the constructor argument. The constructor argument is of type `List<Long>` and the compiler again concludes that the missing type parameter must be `Long`. Note, the lefthand side of the assignment is ignored because the constructor argument provides enough information for the type inference to complete successfully.
- The third `new`-expression demonstrates that the lefthand side of the assignment is indeed ignored (in Java 7). The compiler again infers from the constructors argument, i.e., the result of the `asList` method, that the missing type parameter for the new `HashSet` must be `Long`. This leads to a type mismatch and an according error message. The compiler does not conclude that the missing type parameter should be `Number` because it ignores the lefthand side of the assignment. In Java 8, the type inference was modified and improved. Since then, compiler infers `Number` as the type parameter form the new `HashSet` on the right-hand side of the compiler and from that deduces `Number` as the type parameter for the `asList` method. In Java 8, this compiles just fine.
- The fourth `new`-expression does not rely on type inference and simply specifies the intended type parameter explicitly.

There is yet a different result of type inference if the context of the `new`-expression has neither constructor arguments nor a lefthand side of an assignment. Here is an example.

Examples (of type inference for an instance creation expression in a method invocation context):

```
void method(Set<Long> arg) { ... }

method(new HashSet<>()); // error in Java 7; fine in Java 8
method(new HashSet<>(Arrays.asList(0L,1L,2L)));
```

---

```
error: method method in class TypeInference cannot be applied to given types
  method(new HashSet<>());
    ^
required: Set<Long>
found: HashSet<Object>
```

The `new`-expressions appears as the argument of a method invocation. In the first invocation there is neither a constructor argument nor a lefthand side of an assignment. For lack of more specific information the compiler in Java 7 concludes that the missing type parameter must be `Object`. This leads to a type mismatch and an according error message. This changed with Java 8. The method invocation context is now a permitted type inference context. The compiler concludes from the declared argument type of `method` that the `new HashSet` must be a `HashSet<Long>`.

The second invocation is fine because the compiler can infer the missing type parameter from the constructor argument.

LINK TO THIS [Technicalities.FAQ400C](https://www.baeldung.com/java-7-type-inference-fail)

REFERENCES [What is the diamond operator?](https://www.baeldung.com/java-7-type-inference-fail)  
[Why does the type inference for an instance creation expression fail?](https://www.baeldung.com/java-7-type-inference-fail)

---

## Why does the type inference for an instance creation expression fail?

*Usually because there is not enough context information.*

Occasionally, the type inference for instance creation expressions yields results that are surprising at first sight and might lead to unexpected compile-time errors. Here is an example.

The result of type inference can be surprising for a `new`-expression that appears in a context other than an assignment.

Example (of surprising type inference):

```
String s = new ArrayList<>().iterator().next(); // error
```

---

```
error: incompatible types: Object cannot be converted to String
  String s = new ArrayList<>().iterator().next();
```

In the example above an error message is issued because the `new`-expression `new ArrayList<>()` does not have constructor arguments and it neither appears on the right-hand side of an assignment nor as the argument of a method invocation. Instead, it appears in a chain of method calls. Such a chain is not a valid type inference context.

Hence the compiler has no information for type inference and concludes that the missing type parameter of the `ArrayList` is type `Object`. The iterator's `next` method then returns an `Object` instead of the expected `String` and the compiler accordingly reports a type mismatch.

**Note:** Type inference generally fails for anonymous inner classes.

Here is an example:

```
Callable<Long> task = new Callable <> () { // error
    public Long call() { return 0L; }
};
```

---

```
error: Callable<Long> task = new Callable <> () {
                                   ^
    reason: cannot use '<>' with anonymous inner classes
```

#### LINK TO THIS

[Technicalities.FAQ400D](#)

#### REFERENCES

[What is type argument inference for generic methods?](#)

[What is type argument inference for instance creation expressions?](#)

[Is there a correspondence between type inference for method invocation and type inference for instance creation?](#)

[Why do temporary variables matter in case of invocation of generic methods?](#)

---

## What is type argument inference for generic methods?

*The automatic deduction of the type arguments of a generic method at compile time.*

A generic method can be invoked in two ways:

- *Explicit type argument specification.* The type arguments are explicitly specified in a type argument list.
- *Automatic type argument inference.* The method is invoked like regular non-generic methods, that is, without specification of the type arguments. In this case the compiler automatically infers the type arguments from the invocation context.

Example (of automatic type inference):

```

class Collections {
    public static <A extends Comparable<? super A>> A max (Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
final class Test {
    public static void main (String[ ] args) {
        LinkedList<Long> list = new LinkedList<Long>();
        list.add(0L);
        list.add(1L);
        Long y = Collections.max(list);
    }
}

```

In this example, the `max` method is invoked like a regular method and the compiler automatically infers the type argument from the type of the method argument. In our example the compiler finds that the formal method parameter is `Collection<A>` and that the actual method argument is of type `LinkedList<Long>`. From this information the compiler concludes that `A` must be replaced by `Long`, which yields an applicable `max` method with the signature `Long max(Collection<Long>)`.

LINK TO THIS [Technicalities.FAQ401](#)

REFERENCES

- [What is a parameterized or generic method?](#)
- [What is explicit type argument specification?](#)
- [What happens if a type parameter does not appear in the method parameter list?](#)
- [Why doesn't type argument inference fail when I provide inconsistent method arguments?](#)

## What is explicit type argument specification?

### *Providing a type argument list when the method is invoked.*

A generic method can be invoked with or without an explicit type argument specification. If you do not want to rely on the compiler's automatic type argument inference process you can specify the type arguments explicitly.

Example (of a generic method and its invocation):

```

public class Utilities {
    public static <T extends Comparable> T max(T arg1, T arg2) {
        return (arg1.compareTo(arg2)>0)?arg1:arg2;
    }
}

```

```

}
public class Test {
    public static void main(String[] args) {
        System.out.println(Utilities.<String>max("abc", "xyz"));
    }
}

```

The `max` method can be invoked as `Utilities.<String>max("abc", "xyz")`, where the type argument is explicitly provided, or as `Utilities.max("abc", "xyz")`, in which case the compiler automatically infers the type argument.

The syntax for explicit type argument specification requires that the type argument list precedes the method name, like in `Utilities.<String>max`. Note the difference to parameterized types, where the type argument list follows the type name, like in `List<String>`.

There is a little syntax quirk in the explicit type argument specification: it is required that the type argument(s) is preceded by either a type (for a static method like the one in the example), or the object on which the method is invoked (for a non-static method), or `super` (in case that a superclass's method is invoked). This is even required when methods are invoked in the scope of the same class. In the scope of the class we usually omit the type or object on which the method is invoked; we simply say `method()` rather than `this.method()`. This shorthand is not permitted when a type argument list precedes the method name. We must not say `<String>method()`, instead we must say `this.<String>method()`.

Example:

```

class SomeClass {
    private static <T> void parameterizedClassMethod()    { ... }
    private static    void regularClassMethod()         { ... }
    private          <T> void parameterizedInstanceMethod() { ... }
    private          void regularInstanceMethod()       { ... }

    public void anotherMethod() {
        ...
        regularClassMethod();           // fine
        SomeClass.regularClassMethod(); // fine
        regularInstanceMethod();       // fine
        this.regularInstanceMethod();  // fine

        <String>parameterizedClassMethod(); // error
        SomeClass.<String>parameterizedClassMethod(); // fine
        <String>parameterizedMethod(); // error
        this.<String>parameterizedMethod(); // fine
        ...
    }
}

```

LINK TO THIS

[Technicalities.FAQ402](#)

REFERENCES

[What is a parameterized or generic method?](#)

[How do I invoke a generic method?](#)

## Can I use a wildcard as the explicit type argument of a generic method?

*No, wildcards are not permitted as explicit type arguments of a generic method.*

When a generic method is invoked, then we usually rely on the compiler to infer the type argument of the generic method. In rare cases, we specify the type argument explicitly. Here is an example where explicit type argument specification would make sense.

Example (of generic methods):

```
class Factory {
    private static Class< ? > type;

    public static <T> void setComponentType(Class< ? extends T> token) {
        type = token;
    }
    public static <T> List<T> make(int size) {
        Object array = Array.newInstance(type, size);
        return Arrays.asList((T[])array);    // warning: unchecked cast
    }
}
```

This class has two factory methods that create a list which is backed by an array. Such a list has a fixed size and is type-safe in the sense that it rejects elements of an undesired type, just like an array would reject elements of a type that is not compatible to the array's component type.

Example (of using the factory methods):

```
public static void main(String[] args) {
    Factory.setComponentType(String.class);

    List<String> stringList;
    stringList = Factory.make(10);
    stringList.set(0, "Hello");
    stringList.set(1, new Date());    // expected error: illegal argument type

    List<Date> objectList;
    dateList = Factory.make(10);    // compiles although it should not !!!
    dateList.set(1, new Date());    // run-time error: ArrayStoreException
}
```

The factory is configured (using the `setComponentType()` method) to create lists that are backed by an array of strings. When we invoke a factory method then the compiler performs type inference and tries to figure out what the best type argument for invocation of the generic `make()` method would be. Since the type parameter does not appear in the method's arguments list the compiler infers the type argument from the context in which the return value is used.

As long as we assign the result of invoking the `make()` method to a reference of type `List<String>`, all is fine. The compiler infers `T:=String` when the `make()` method is invoked. We can place strings into the list, but no other type of object, because the reference of type `List<String>` will not permit it.

If we try assigning the result of invoking the `make()` method to a reference of type `List<Date>`, it will compile as well. The compiler will again infer the type argument from the usage of the return value; this time it infers `T:=Date`. However, when we attempt placing a date into the list, which is backed by a string array internally, an `ArrayStoreException` will be raised. Basically, what we created is a list that will always raise `ArrayStoreExceptions`. This undesired situation is a side effect of the "unchecked cast" warning in the implementation of the `make()` method.

In order to prevent the undesired situation, the `make()` method is best invoked with an explicitly specified type argument.

Example (of explicit type argument specification):

```
public static void main(String[] args) {
    Factory.setComponentType(String.class);

    List<String> stringList;
    stringList = Factory.<String>make(10);
    stringList.set(0,"Hello");

    List<Date> objectList;
    dateList = Factory.<String>make(10); // does no longer compile
    dateList.set(1,new Date()); // run-time error: ArrayStoreException
}
```

Basically, the example shows a situation in which explicit type argument specification serves a purpose and is helpful.

In the example we have been using a concrete type when we specified the type argument explicitly, which raises the question: can we also use wildcards as explicit type arguments? The answer is: No, wildcards are not permitted as explicit type arguments of parameterized method.

Example (of a wildcard as explicitly specified type arguments of generic methods):

```
List<? super String> list1 = Factory.make(10); // fine
List<? super String> list2 = Factory.<String>make(10); // fine
List<? super String> list3 = Factory.<? super String>make(10); // error
```

The Java syntax simply does not allow wildcards in the location where explicit type arguments appear and you might see funny compiler message ranging from "wildcard is not allowed in this location" to less helpful statements such as "illegal start of type", "illegal start of expression", "( expected", "; expected" or the like.

LINK TO THIS

[Technicalities.FAQ402A](#)

REFERENCES

[What is type argument inference?](#)

[What is explicit type argument specification?](#)

[What happens if a type parameter does not appear in the method parameter list?](#)

---

**What happens if a type parameter does not appear in the method parameter list?**



### *The compiler tries to infer the type argument from the calling context.*

If the type parameter does not appear in the types of the method arguments, then the compiler cannot infer the type arguments by examining the types of the actual method arguments. If the type parameter appears in the method's return type, then the compiler takes a look at the context in which the return value is used. If the method call appears as the righthand side operand of an assignment, then the compiler tries to infer the method's type arguments from the static type of the lefthand side operand of the assignment.

Example (for inference from assignment context):

```
public final class Utilities {
    ...
    public static <T> HashSet<T> create(int size) {
        return new HashSet<T>(size);
    }
}
public final class Test
    public static void main(String[] args) {
        HashSet<Integer> hi = Utilities.create(10);
    }
}
```

The `create` method is generic and the type parameter `T` does not appear in the method parameter list; it appears in the method's return type `HashSet<T>` though. The result of the method is assigned to a variable of type `HashSet<Integer>` so that the compiler infers that the type argument of the `create` method must be `T:=Integer`. The compiler is even smart enough to infer the type argument of the `create` method if the method result is assigned to a variable of a supertype of `HashSet<Integer>`, such as `Collection<Integer>`.

The invocation of a generic method might appear as the argument of another method invocation. Such an invocation context was not considered for type inference before Java 8. This changed with the improved type inference in Java 8.

In Java 5, 6, and 7, the compiler does not try to perform any special type inference when the invocation of a generic method appears in a method invocation context. Instead the compiler handles the method call as though it would appear in no context. No context means that the compiler performs the type inference algorithm as though the method result was assigned to a variable of type `Object`.

Example (for inference from a method invocation context):

```
public final class Utilities {
    ...
    public static <T> HashSet<T> create(int size) {
        return new HashSet<T>(size);
    }
    public static void print(HashSet<String> h) {
        for (String s : h) System.out.println(s);
    }
}
public final class Test
    public static void main(String[] args) {
```

```
Utilities.print(Utilities.create(10));    // error in Java 5,6,7; fine in Java 8
}
}
```

---

```
error: print(java.util.HashSet<java.lang.String>) cannot be applied to (java.util.HashSet<java.lang.Object>)
    Utilities.print(Utilities.create(10));
                   ^
```

In Java 5, 6, and 7, the compiler treats the call `Utilities.print(Utilities.create(10))` like it were an assignment `Object o = Utilities.create(10)`. A lefthand side of type `Object` does not provide any particular type information so that the compiler cannot really infer anything. If no specific type can be inferred then the compiler chooses `Object` as the type argument. With type `Object` as the type argument the `create` method returns a `HashSet<Object>`, which is incompatible to a `HashSet<String>` and leads to the error message displayed above.

In Java 8, the compiler considers that the `print` method needs an argument of type `HashSet<String>` and figures out that the type parameter for the `create` method must be `String` then.

The key difference is that previously the method invocation context was treated like no context for type inference and in Java 8 it a valid type inference context, from which the compiler retrieves information for the type deduction process.

If the type argument inference does not lead to the desired result or if we want to disable the automatic inference, we can explicitly specify the type arguments.

Example (for explicit type argument specification):

```
public final class Utilities {
    ...
    public static <T> HashSet<T> create(int size) {
        return new HashSet<T>(size);
    }
    public static void print(HashSet<String> h) {
        for (String s : h) System.out.println(s);
    }
}
public final class Test
    public static void main(String[] args) {
        Utilities.print(Utilities.<String>create(10));
    }
}
```

#### LINK TO THIS

[Technicalities.FAQ403](#)

#### REFERENCES

[What is a parameterized or generic method?](#)

[What is type argument inference?](#)

[What is explicit type argument specification?](#)

[Why doesn't type argument inference fail when I provide inconsistent method arguments?](#)

[Why do temporary variables matter in case of invocation of generic methods?](#)

---

## Why doesn't type argument inference fail when I provide inconsistent method arguments?

*Because the "inconsistent" arguments might make sense to the compiler.*

Occasionally the compiler infers a type where we might expect that no type can be inferred.

Example (of surprising type argument inference):

```
public final class Utilities {
    ...
    public static <T> void fill(T[] array, T elem) {
        for (int i=0; i<array.length; ++i) { array[i] = elem; }
    }
}

public final class Test {
    public static void main(String[] args) {
        Utilities.fill(new String[5], new String("XYZ")); // T:=String
        Utilities.fill(new String[5], new Integer(100)); // T:=Object&Serializable&Comparable
    }
}
```

This is the example of a method whose type argument appears in several method arguments. Quite obviously the intent is that the component type of the array should match the type of the second argument. For this reason we might expect that a method invocation such as `Utilities.fill(new String[5], new Integer(100))` would fail, because the argument types `String[]` and `Integer` are inconsistent.

However, the compiler does not reject this method call. Instead it performs type inference and infers the common supertypes of `String` and `Integer` as type argument. To be precise the compiler infers

```
T := Object & Serializable & Comparable<? extends Object&Serializable&Comparable<?>>
```

which is a synthetic type construct used internally by the compiler. It denotes the set of supertypes of `String` and `Integer`.

Whether the result of this successful type inference is desired or not depends on the circumstances. In this example the source code compiles, but the method invocation in question will fail at runtime with an `ArrayStoreException`, because the method would try to store integers in an array of strings.

In Java 5, 6, and 7 it was possible to prevent the perhaps undesired type argument inference by a minor modification of the `fill` method.

Example (modified, in Java 5, 6, and 7):

```
public final class Utilities {
    ...
    public static <T, S extends T> void fill(T[] array, S elem) {
        for (int i=0; i<array.length; ++i) { array[i] = elem; }
    }
}
```

```

public final class Test {
    public static void main(String[] args) {
        Utilities.fill(new String[5], new String("XYZ")); // T:=String and S:=String
        Utilities.fill(new String[5], new Integer(100)); // T:=String and S:=Integer => error (in Java 5,6,7)
    }
}

```

In Java 5, 6, and 7 the compiler inferred both type arguments separately as `String` and `Integer`. When it checked the bounds it found that `s` is not within bounds because `Integer` is not a subtype of `String` and the call was rejected.

Since Java 8 the compiler does no longer issue an error message and you will observe the same behavior as in the initial example without the additional type variable "S extends T". The fact that the type inference process in Java 5, 6, and 7 could not find a common super type and issued an error message was just an accident. The glitch was fixed in Java 8 and nowadays the compiler exploits the covariance of arrays in this example, too.

Example (same as above, but in Java 8):

```

public final class Utilities {
    ...
    public static <T, S extends T> void fill(T[] array, S elem) {
        for (int i=0; i<array.length; ++i) { array[i] = elem; }
    }
}

public final class Test {
    public static void main(String[] args) {
        Utilities.fill(new String[5], new String("XYZ")); // T:=Object&Serializable&Comparable and S:=String => fine
        Utilities.fill(new String[5], new Integer(100)); // T:=Object&Serializable&Comparable and S:=Integer => fine (in Java 8)
    }
}

```

#### LINK TO THIS

[Technicalities.FAQ404](#)

#### REFERENCES

[What is a parameterized or generic method?](#)  
[What is type argument inference?](#)  
[What is explicit type argument specification?](#)  
[What happens if a type parameter does not appear in the method parameter list?](#)  
[Why do temporary variables matter in case of invocation of generic methods?](#)

---

## Why do temporary variables matter in case of invocation of generic methods?

*Because of the automatic type argument inference.*

Usually, it does not make a difference whether we use the result of a method call for invocation of the next method, like in

```
f(x).g();
```

or whether we store the result first in a temporary variable and then pass the temporary variable to the next method, like in

```
tmp=f(x);  
tmp.g();
```

The effect is usually the same. However, when the methods are generic methods, it may well make a difference.

Example:

```
public final class Utilities {  
    ...  
    public static <T> HashSet<T> create(int size) {  
        return new HashSet<T>(size);  
    }  
}
```

Let us consider a chained method call where the result of the `create` method is used for calling the `iterator` method. We can perform the chained method call in two steps using a temporary variable:

```
HashSet<Integer> tmp = Utilities.create(5);  
Iterator<Integer> iter = tmp.iterator();
```

Or we can perform it one step:

```
Iterator<Integer> iter = Utilities.create(5).iterator();
```

If the methods were non-generic methods the result of both invocation techniques would be the same. Not so in our example, where the first method is generic and the compiler must infer the type parameter from the context. Obviously, the context is different in a chained call compared to the use of temporaries.

In the two-step invocation the result of the `create` method appears in an assignment, and assignment is a context that the compiler considers for type inference. In the one-step invocation the result of the `create` method is used to invoke another method, and method chains are not considered for type inference.

Here is what the compiler infers for the two-step call:

```
HashSet<Integer> tmp = Utilities.create(5);           // T:=Integer  
Iterator<Integer> iter = tmp.iterator();           // T:=Integer
```

For inference of the type argument of the `create` method the compiler takes a look at the type of the left-hand side of the assignment, namely the type of the temporary variable. It is of type `HashSet<Integer>` and the compiler infers `T:=Integer`.

The one-step call, in contrast, does not compile because the type argument inference works differently in this case:

```
Iterator<Integer> iter = Utilities.create(5).iterator();
```

---

```
error: incompatible types
    Iterator<Integer> iter = Utilities.create(5).iterator();
                                                    ^
required: Iterator<Integer>
found:    Iterator<Object>
```

For inference of the type argument of the `create` method the compiler does not consider any context information, because it neither appears in an assignment nor a method invocation context, but in a method chain instead. It treats the invocation of `create` as though the result of `create` were assigned to a variable of type `Object`, which does not provide any type information to deduce anything from. For this reason the compiler infers `T:=Object`, which yields an instantiation of the `create` method with the signature `HashSet<Object> create()`.

Equipped with this information the compiler finds that the `iterator` method return an `Iterator<Object>`, which leads to the error message.

The example demonstrates, that in rare cases there can be a difference between a one-step nested method call such as `f(x).g()`; and a two-step call using a temporary variable such as `tmp=f(x); tmp.g()`;. The difference stems from the fact that an assignment context is considered for the type argument inference, while other situations are not.

LINK TO THIS [Technicalities.FAQ405](#)

REFERENCES

- [What is a parameterized or generic method?](#)
- [What is type argument inference?](#)
- [What is explicit type argument specification?](#)
- [What happens if a type parameter does not appear in the method parameter list?](#)
- [Why doesn't type argument inference fail when I provide inconsistent method arguments?](#)

---

## Wildcard Capture

### What is the capture of a wildcard?

*An anonymous type variable that represents the particular unknown type that the wildcard stands for. The compiler uses the capture internally for evaluation of expressions and the term "capture of ?" occasionally shows up in error message.*

A wildcard is compatible with all types from a set of types. For instance, all instantiations of the generic type `List`, such as `List<String>`, `List<Number>`, `List<Long>`, etc. can be assigned to a reference variable of type `List<?>`. Wildcards are typically used as argument or return types in method signatures. The goal and effect is that the method accepts arguments of a larger set of types, namely all types that belong to the type family that the wildcard denotes.

Example (of a wildcard in a method signature):

```
public static void reverse(List<?> list) {
```

```
    // ... the implementation ...  
}
```

The `reverse` method accepts arguments that are of a type that is an instantiation of the generic type `List`. In order to use the type argument `list` of type `List<?>` the compiler converts the wildcard instantiation to the so-called *capture* of the wildcard. The capture represents the particular albeit unknown type of the argument that is actually passed the method. This particular unknown type is, of course, a member of the type family that the wildcard denotes.

The wildcard capture can be imagined as an anonymous type variable that the compiler generates internally and uses as the static type of the method parameter. A type variable is like a type parameter of a generic type or method; it stands for a particular unknown type. Changing the method parameter's type from the instantiation using a wildcard to an instantiation using the capture is known as *capture conversion*. The translation of our method can be imagined as though the compiler had re-implemented the method to use a synthetic generic method that has the wildcard capture as its type parameter.

Example (pseudo code showing the wildcard capture):

```
private static <T_?001> void reverse_?001(List<T_?001> list) {  
    // ... the implementation ...  
}  
public static void reverse(List<?> list) {  
    reverse_?001(list);  
}
```

For the analysis and translation of the implementation of the `reverse` method the compiler will use `List<T_?001>` as the type of the method parameter. The synthetic type variable `T_?001` is used for all purposes where the method parameter's static type information is needed, like for instance, in type checks and for type inference.

For illustration, let us consider a conceivable implementation of the `reverse` method using a generic helper method `rev` for which the compiler must infer the type arguments from the wildcard.

Example (implementation of the `reverse` method):

```
private static <T> void rev(List<T> list) {  
    ListIterator<T> fwd = list.listIterator();  
    ListIterator<T> rev = list.listIterator(list.size());  
    for (int i = 0, mid = list.size() >> 1; i < mid; i++) {  
        T tmp = fwd.next();  
        fwd.set(rev.previous());  
        rev.set(tmp);  
    }  
}  
public static void reverse(List<?> list) {  
    rev(list);  
}
```

The compiler applies capture conversion and generates an anonymous type variable. In principle, the compiler translates the `reverse` method to something like this.

Example (pseudo code showing the wildcard capture):

```

private static <T> void rev(List<T> list) {
    ListIterator<T> fwd = list.listIterator();
    ListIterator<T> rev = list.listIterator(list.size());
    for (int i = 0, mid = list.size() >> 1; i < mid; i++) {
        T tmp = fwd.next();
        fwd.set(rev.previous());
        rev.set(tmp);
    }
}
private static <T_?001> void reverse_?001(List<T_?001> list) {
    rev(list);
}
public static void reverse(List<?> list) {
    reverse_?001(list);
}

```

Among other things, the compiler uses the wildcard capture `T_?001` for inference of the type argument `T` of the generic helper method `rev`. It infers `T:=T_?001` and therefore invokes the instantiation `<T_?001>rev` of method `rev`.

LINK TO THIS [Technicalities.FAQ501](#)

REFERENCES [What is a wildcard capture assignment-compatible to?](#)

## What is the capture of an unbounded wildcard compatible to?

*Nothing, except the unbounded wildcard itself.*

The capture of a wildcard is compatible to a corresponding wildcard, never to a concrete type.

Example (implementation of assignment of wildcard instantiation):

```

private static void method(List<?> list) {
    List<String> l11 = list; // error
    List<? extends String> l12 = list; // error
    List<? super String> l13 = list; // error
    List<?> l14 = list; // fine
}

```

---

```

error: incompatible types
found   : java.util.List<capture of ?>
required: java.util.List<java.lang.String>
    List<String> l11 = list;
                ^
error: incompatible types
found   : java.util.List<capture of ?>
required: java.util.List<? extends java.lang.String>

```



```
List<? extends String> l12 = list;
      ^
error: incompatible types
found   : java.util.List<capture of ?>
required: java.util.List<? super java.lang.String>
List<? super String> l13 = list;
      ^
```

The method takes an argument of type `List<?>`, which the compiler translates to type `List<capture of ?>`. The wildcard capture denotes a particular unknown type. From the static type information "capture of ?" the compiler cannot tell, whether capture stands for `String`; the capture could stand for any type. On this ground the compiler rejects the assignment of the method result of type `List<capture of ?>` to the variable of type `List<String>` in the example.

LINK TO THIS [Technicalities.FAQ502](#)

REFERENCES [What is the capture of a wildcard?](#)  
[What does type-safety mean?](#)

---

## Is the capture of a bounded wildcard compatible to the bound?

*No, not even if the bound is a final class.*

The capture of a wildcard is compatible to a corresponding wildcard, never to a concrete type. Correspondingly, the capture of a bounded wildcard is compatible solely to other wildcards, but never to the bound.

For illustration we use the `getClass` method, which is defined in class `Object` (see [java.lang.Object.getClass](#)). The result of the `getClass` method is of type `Class<? extends X>`, where `X` is the erasure of the static type of the expression on which `getClass` is called.

Example (with bounded wildcard):

```
Number n = new Integer(5);
Class<Number> c = n.getClass(); // error
```

---

```
error: incompatible types
found   : java.lang.Class<capture of ? extends java.lang.Number>
required: java.lang.Class<java.lang.Number>
Class<Number> c = n.getClass();
      ^
```

In our example the static type of the expression on which `getClass` is called is `Number`. Hence the return value is of type `Class<capture of ? extends Number>`. A variable of type `Class<capture of ? extends Number>` can refer to a `Class<Number>`, a `Class<Long>`, a `Class<Integer>`, etc. There's no guarantee that it actually refers to a `Class<Number>`, and indeed, in our example it refers to a `Class<Integer>`. The compiler rightly complains. How do we fix it?

Example (corrected):

```
Number n = new Integer(5);
```

```
Class<?>          c1 = n.getClass();
Class<? extends Number> c2 = n.getClass();
```

Since `getClass` returns a wildcard instantiation we must assign the result to a wildcard instantiation. `Class<?>` would be correct, but also the more specific type `Class<? extends Number>` would work.

Interestingly, the capture of a bounded wildcard, whose upper bound is a final class, is still incompatible to the bounds type, although the set of types that the bounded wildcard denotes contains only one type, namely the bounds type itself.

Example (using final class as bound):

```
String s = new String("abc");
Class<String>      c0 = s.getClass(); // error
Class<?>          c1 = s.getClass(); // fine
Class<? extends String> c2 = s.getClass(); // fine
```

---

```
error: incompatible types
found   : java.lang.Class<capture of ? extends java.lang.String>
required: java.lang.Class<java.lang.String>
    Class<String>          c0 = s.getClass();
                                ^
```

**LINK TO THIS**            [Technicalities.FAQ503](#)

**REFERENCES**            [What is the capture of a wildcard?](#)  
                          [What is a wildcard capture assignment-compatible to?](#)

---

## Wildcard Instantiations

**Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?**

*It depends on the kind of wildcard.*

Using an object through a reference variable of a wildcard parameterized type is restricted. Consider the following class:

Example ( of a parameterized class):

```

class Box<T> {
    private T t;
    public Box(T t) { this.t = t; }
    public void put(T t) { this.t = t; }
    public T take() { return t; }
    public boolean equalTo(Box<T> other) { return this.t.equals(other.t); }
    public Box<T> copy() { return new Box<T>(t); }
}

```

In a wildcard parameterized type such as `Box<?>` the type of the field and the argument and the return types of the methods would be unknown. It is like the field `t` would be of type `"?"` and the `put` method would take an argument of type `"?"` and so on. In this situation the compiler does not let us assign anything to the field or pass anything to the `put` method. The reason is that the compiler cannot make sure that the object that we are trying to assign to the field or pass as an argument to a method is of the expected type, since the expected type is unknown. Similar effects can be observed for methods such as like `equalTo` and `clone`, which have a parameterized argument or return type and the type parameter `T` appears as type argument of the parameterized argument or return type.

Below is a table that lists which uses of fields and methods are legal or illegal. It assumes a class like this:

```

class X<T> {
    private T t;

    public T m() { ... }
    public void m(T arg) { ... }

    public Y<T> f() { ... }
    public void f(Y<T> arg) { ... }

    public Y<? extends T> f() { ... }
    public void f(Y<? extends T> arg) { ... }

    public Y<? super T> f() { ... }
    public void f(Y<? super T> arg) { ... }
}

```

Examples and further explanations can be found in subsequent FAQ entries.

<b>X&lt;?&gt;</b>	<b>unbounded wildcard</b>			
	<b>legal</b>		<b>illegal</b>	
<b>fields</b>	<b>x = t</b>	We can <u>read</u> a field whose type is the type parameter. The field is accessible through a reference of type <code>Object</code> .	<b>t = x</b>	We cannot <u>assign</u> to a field whose type is the type parameter except: <code>null</code>
<b>methods</b>	<b>Tm()</b>	We can call methods that use the type parameter as the <u>return type</u> . The returned value is accessible through a reference of type <code>Object</code> .	<b>void m(T)</b>	We cannot call methods that use the type parameter as an <u>argument type</u> . except: <code>null</code>

	<b>Y&lt;T&gt; f()</b> <b>Y&lt;? extends T&gt; f()</b> <b>Y&lt;? super T&gt; f()</b>	We can call methods that use the type parameter as <u>type argument</u> in the return type or as an <u>upper</u> or <u>lower wildcard bound</u> in the <u>return type</u> . The returned value is accessible through the unbounded wildcard instantiation of the return type (i.e. Y<?>).	<b>void f(Y&lt;T&gt;)</b> <b>void f(Y&lt;? extends T&gt;)</b>	We cannot call methods that use the type parameter as <u>type argument</u> in an argument type or as an <u>upper wildcard bound</u> in an <u>argument type</u> . except: null
	<b>void f(Y&lt;? super T&gt;)</b>	We can call methods that use the type parameter as a <u>lower wildcard bound</u> in an <u>argument type</u> . The method argument must be either null or of type Y<Object> (the argument type instantiated for type Object).		
<b>X&lt;? extends B&gt;</b>	<b>wildcard with upper bound</b>			
	<b>legal</b>		<b>illegal</b>	
<b>fields</b>	<b>x = t</b>	We can <u>read</u> a field whose type is the type parameter. The field is accessible through a reference whose type is the upper bound.	<b>t = x</b>	We cannot <u>assign</u> to a field whose type is the type parameter. except: null
<b>methods</b>	<b>T m()</b>	We can call methods that use the type parameter as the <u>return type</u> . The returned value is accessible through a reference whose type is the upper bound.	<b>void m(T)</b>	We cannot call methods that use the type parameter as an <u>argument type</u> . except: null
	<b>Y&lt;T&gt; f()</b> <b>Y&lt;? extends T&gt; f()</b>	We can call methods that use the type parameter as <u>type argument</u> in the return type or as an <u>upper wildcard bound</u> in the <u>return type</u> . The returned value is accessible through the upper bound wildcard instantiation of the return type (i.e. Y<? extends B>).	<b>void f(Y&lt;T&gt;)</b> <b>void f(Y&lt;? extends T&gt;)</b>	We cannot call methods that use the type parameter as <u>type argument</u> in an argument type or as an <u>upper wildcard bound</u> in an <u>argument type</u> . except: null
	<b>Y&lt;? super T&gt; f()</b>	We can call methods that use the type parameter as an <u>lower wildcard bound</u> in the <u>return type</u> . The returned value is accessible through the unbounded wildcard instantiation of the return type (i.e. Y<?>).		
	<b>void f(Y&lt;? super T&gt;)</b>	We cannot call methods that use the type parameter as an <u>lower wildcard bound</u> in an <u>argument type</u> . The method argument must be either null or of a type that belongs to the family denoted by Y<? super B>.		
<b>X&lt;? super B&gt;</b>	<b>wildcard with lower bound</b>			
	<b>legal</b>		<b>illegal</b>	

fields	<b>t = x</b>	We can <u>assign</u> to a field whose type is the type parameter. The value to be assigned must be either <code>null</code> or of a type that is the lower bound or a subtype thereof.		
	<b>x = t</b>	We can <u>read</u> a field whose type is the type parameter. The field is accessible through a reference of type <code>Object</code> .		
methods	<b>void m(T)</b>	We can call methods that use the type parameter as an <u>argument type</u> . The method argument must be either <code>null</code> or of a type <code>B</code> (the lower bound).		
	<b>T m()</b>	We can call methods that use the type parameter as the <u>return type</u> . The returned value is accessible through a reference of type <code>Object</code> .		
	<b>Y&lt;T&gt; f()</b> <b>Y&lt;? super T&gt; f()</b>	We can call methods that use the type parameter as <u>type argument</u> in the return type or as a <u>lower wildcard bound</u> in the <u>return type</u> . The returned value is accessible through the lower bound wildcard instantiation of the return type (i.e. <code>Y&lt;? super B&gt;</code> ).	<b>void f(Y&lt;T&gt;)</b>	We cannot call methods that use the type parameter as type argument in an <u>argument type</u> , except: <code>null</code>
	<b>Y&lt;? extends T&gt; f()</b>	We can call methods that use the type parameter as an <u>upper wildcard bound</u> in the <u>return type</u> . The returned value is accessible through the unbounded wildcard instantiation of the return type (i.e. <code>Y&lt;?&gt;</code> ).		
	<b>void f(Y&lt;? extends T&gt;)</b>	We can call methods that use the type parameter as an <u>upper wildcard bound</u> in an <u>argument type</u> . The method argument must be either <code>null</code> or of a type that belong to the family denoted by <code>Y&lt;? extends B&gt;</code> .		
	<b>void f(Y&lt;? super T&gt;)</b>	We can call methods that use the type parameter as a <u>lower wildcard bound</u> in an <u>argument type</u> . The method argument must be either <code>null</code> or of type <code>Y&lt;Object&gt;</code> (the argument type instantiated for type <code>Object</code> ).		

## REFERENCES

[What is a wildcard instantiation?](#)

[What is a wildcard?](#)

[What is an unbounded wildcard?](#)

[What is a bounded wildcard?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[In a wildcard parameterized type, can I read and write fields whose type is the type parameter?](#)

---

## Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?

*We cannot call methods through an unbounded wildcard parameterized type that take arguments of the "unknown" type. But we can call methods that return objects of the "unknown" type.*

Example:

```
class Box<T> {
    private T t;

    public Box(T t) { this.t = t; }
    public void put(T t) { this.t = t; }
    public T take() { return t; }

    public boolean contains(T t) { return this.t == t; }
    public String toString() { return "Box[" + t.toString() + "]; }
}
class Test {
    public static void main(String[] args) {
        Box<?> box = new Box<String>("abc");

        box.put("xyz");    // error
        box.put(null);    // ok

        box.contains("abc"); // error
        box.toString(); // ok

        String s = box.take(); // error
        Object o = box.take(); // ok
    }
}
```

We cannot call the `put` method of the `Box` type through a reference variable of type `Box<?>`, because the method takes an argument of the unknown type that the wildcard stands for. From the type information `Box<?>` the compiler does not know whether the object we are passing to the method is compatible with the actual

object contained in the box. If the `Box<?>` would be referring to a `Box<Long>`, then it would clearly violate the type guarantees if we could put a string into the box that is supposed to contain a long value. The only argument that is accepted is the `null` reference, because it has no type.

The same reasoning applies to *all* methods that take an argument of the "unknown" type, even if the method does not even modify the box, like the `contains` method. It just takes an object of the "unknown" type and compares it. If a string is passed to the `contains` method of a box that contains a long value, it simply returns `false`. No harm is done. Yet the invocation is illegal if performed through a reference variable of type `Box<?>`. [Defining the `contains` method as taking an argument of type `Object`, instead of `T`, would avoid this effect. In this case the `contains` method would not take an object of "unknown", but an object of "any" type, and it would be permitted to invoke it through a reference variable of type `Box<?>`.]

We can freely invoke any methods that neither take nor return objects of the "unknown" type.

The example demonstrates that methods returning an object of the unknown type can be called and return an object of unknown type, which can be assigned to a reference variable of type `Object`, but not to a reference variable of a more specific type.

LINK TO THIS [Technicalities.FAQ602](#)

#### REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[In a wildcard instantiation, can I read and write fields whose type is the type parameter?](#)

[What is a wildcard instantiation?](#)

[What is a wildcard?](#)

[What is an unbounded wildcard?](#)

[What is a bounded wildcard?](#)

---

## Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?

*We cannot call methods through an unbounded wildcard parameterized type that take arguments of the "unknown" type. But we can call methods that return objects of the "unknown" type.*

The rules for an upper bound wildcard parameterized type are the same as for an unbounded wildcard parameterized type. The only difference is that a returned object of "unknown" type is known to be compatible to the upper bound.

Example:

```
class Box<T> {
    private T t;

    public Box(T t) { this.t = t; }
    public void put(T t) { this.t = t; }
    public T take() { return t; }
```

```

    public boolean contains(T t) { return this.t == t; }
    public String toString() { return "Box["+t.toString()+"]"; }
}
class Test {
    public static void main(String[] args) {
        Box<? extends Number> box = new Box<Long>(0L);

        box.put(1L);        // error
        box.put(null);     // ok

        box.contains(0L);  // error
        box.toString();    // ok

        Long l = box.take(); // error
        Number n = box.take(); // ok
    }
}

```

The returned object of "unknown" type is known to be compatible to the upper bound. Hence we can assign the result of `take` to a reference variable of type `Number` in our example.

#### LINK TO THIS

[Technicalities.FAQ603](#)

#### REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[In a wildcard instantiation, can I read and write fields whose type is the type parameter?](#)

[What is a wildcard instantiation?](#)

[What is a wildcard?](#)

[What is an unbounded wildcard?](#)

[What is a bounded wildcard?](#)

## Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?

*We can call methods through an unbounded wildcard parameterized type that take arguments of the "unknown" type. But we cannot call methods that return objects of the "unknown" type.*

Compared to the rules for upper bound wildcard parameterized types the rules for wildcard parameterized types with a lower bound wildcard are the other way round.



We can call methods through an unbounded wildcard parameterized type that take arguments of the "unknown" type. But we cannot call methods that return objects of the "unknown" type.

Example:

```
class Box<T> {
    private T t;

    public Box(T t) { this.t = t; }
    public void put(T t) { this.t = t;}
    public T take() { return t; }

    public boolean contains(T t) { return this.t == t; }
    public String toString() { return "Box["+t.toString()+"]"; }
}
class Test {
    public static void main(String[] args) {
        Box<? super Long> box = new Box<Number>(0L);
        Number number = new Integer(1);

        box.put(1L);          // ok
        box.put(null);       // ok
        box.put(number);     // error

        box.contains(0L);    // ok
        box.toString();      // ok

        Long l = box.take(); // error
        Number n = box.take(); // error
        Object o = box.take(); // ok
    }
}
```

Methods that take an argument of the "unknown" type can be invoked with either `null` or an argument whose type is the lower bound or a subtype thereof. That is, we can pass a `Long` to method `take` through the reference of type `Box<? super Long>`. But we cannot pass a `Number` as an argument, because the compiler does not know whether the `Box<? super Long>` refers to a `Box<Number>` or perhaps to a `Box<Comparable<Long>>`, in which case a `Number` were unacceptable, because it is not comparable.

Methods that return a value of the "unknown" type can be invoked, but only if no assumptions are made regarding the type of the returned object and it is treated like an `Object`.

LINK TO THIS

[Technicalities.FAQ604](#)

REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[In a wildcard instantiation, can I read and write fields whose type is the type parameter?](#)

[What is a wildcard instantiation?](#)

[What is a wildcard?](#)

[What is an unbounded wildcard?](#)

[What is a bounded wildcard?](#)

---

## Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?

*We cannot call methods that use the type parameter as type argument in an argument type. We can call methods that use the type parameter as type argument in the return type; the returned value is accessible through a wildcard instantiation of the return type.*

In a wildcard parameterized type, we cannot call methods that use the type parameter as type argument in an argument type in an argument type. But, we can call methods that use the type parameter as type argument in the return type; the returned value is accessible through a wildcard instantiation of the return type. The wildcard instantiation of the return type corresponds to the wildcards instantiation that was used for the method invocation, e.g. a method of `Box<? extends Number>` would return a `ReturnType<? extends Number>` and a method of `Box<? super Number>` would return a `ReturnType<? super Number>`.

*Unbounded wildcard parameterized type.*

Example (access through unbounded wildcard):

```
class Box<T> {
    private T t;
    ...
    public boolean equalTo(Box<T> other) { return this.t.equals(other.t); }
    public Box<T> copy() { return new Box<T>(t); }
}
class Test {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<String>("abc");
        Box<?> unknownBox = stringBox;

        boolean equal = true;
        equal = unknownBox.equalTo(unknownBox); // error
        equal = unknownBox.equalTo(stringBox); // error

        Box<?> box1 = unknownBox.copy(); // ok
        Box<String> box2 = unknownBox.copy(); // error
    }
}
```

---

```

error: equalTo(Box<capture of ?>) in Box<capture of ?> cannot be applied to (Box<capture of ?>)
    equal = unknownBox.equalTo(unknownBox);
                        ^
error: equalTo(Box<capture of ?>) in Box<capture of ?> cannot be applied to (Box<java.lang.String>)
    equal = unknownBox.equalTo(stringBox);
                        ^
error: incompatible types
found   : Box<capture of ?>
required: Box<java.lang.String>
    Box<String> box2 = unknownBox.copy();
                        ^

```

We cannot call the `equalTo` method of the `Box` type through a reference variable of type `Box<?>`, because the compiler does not know which type of object is expected as an argument of the `equalTo` method. In our example the reference variable `unknownBox` refers to a `Box<String>` and hence only a `Box<String>` would be acceptable as an argument of the `equalTo` method. But from the type information `Box<?>` the compiler does not know which type of box would be acceptable as an argument. For this reason the compiler rejects all attempts to invoke the `equalTo` method.

Invocation of the `copy` method is permitted. It returns an object of an unknown instantiation of the `Box` type to which we can refer through a variable of type `Box<?>`. More specific information about the instantiation of the `Box` type is not available and for this reason the assignment to a reference variable of an instantiation different from `Box<?>` fails.

In the example, the parameterized argument and return type happens to be the enclosing type. This is just by chance. The rules explained above apply in the same way to unrelated parameterized argument and return types. For instance, if the `Box` class had methods taking or returning a `Comparable<T>` the same rules would apply: methods that take arguments of type `Comparable<T>` cannot be called and methods that return a `Comparable<T>` can be called and the result can be assigned to a `Comparable<?>`.

#### *A note on the error messages and the term "capture of":*

The compiler uses the term "capture of ?" when it refers to the unknown type that the wildcard stands for. More on the wildcard capture can be found in [Technicalities.FAQ501](#). The compiler uses the capture of a wildcard in order to denote the signatures of methods in wildcard instantiations.

The signature of a method in a wildcard parameterized type is determined by replacing all occurrences of the type parameter by the capture of the parameterized type's wildcard. For instance, the method `equalTo(Box<T>)` in `Box<?>` has the signature `equalTo(Box<capture of ?>)`. The same method in `Box<? extends Number>` has the signature `equalTo(Box<capture of ? extends Number>)`. The method `takeContentFrom(Box<? extends T>)` in `Box<?>` has the signature `takeContentFrom(Box<? extends capture of ?>)` and the same method in `Box<? super Number>` has the signature `takeContentFrom(Box<? extends capture of ? super Number>)`.

What these "capture of capture" things mean is explained below in the discussion of various examples. Just to give you an idea, the term "`? extends capture of ?`" refers to a subtype ("`? extends ...`") of an unknown type ("`capture of ?`"), and the term "`? extends capture of ? super Number`" refers to subtype ("`? extends ...`") of an unknown type ("`capture of ? ...`") that is a supertype of `Number` ("`? super Number`").

#### *Bounded wildcard parameterized types.*

The example above used a reference variable of the unbounded wildcard type `Box<?>`. If we use a bounded wildcard type such as `Box<? extends Number>` or `Box<? super Number>` the same rules apply.

Example (access through bounded wildcard):

```

class Box<T> {
    private T t;
    ...
    public boolean equalTo(Box<T> other) { return this.t.equals(other.t); }
    public Box<T> copy() { return new Box<T>(t); }
}
class Test {
    public static void main(String[] args) {
        Box<Number> numberBox = new Box<Number>(0L);
        Box<? extends Number> unknownBox = numberBox;

        boolean equal = true;
        equal = unknownBox.equalTo(unknownBox); // error
        equal = unknownBox.equalTo(numberBox); // error

        Box<?> box1 = unknownBox.copy(); // ok
        Box<? extends Number> box2 = unknownBox.copy(); // ok
        Box<Number> box3 = unknownBox.copy(); // error
    }
}

```

---

```

error: equalTo(Box<capture of ? extends java.lang.Number>)
in Box<capture of ? extends java.lang.Number>
cannot be applied to (Box<capture of ? extends java.lang.Number>)
    equal = unknownBox.equalTo(unknownBox);
                          ^
error: equalTo(Box<capture of ? extends java.lang.Number>)
in Box<capture of ? extends java.lang.Number>
cannot be applied to (Box<java.lang.Number>)
    equal = unknownBox.equalTo(numberBox);
                          ^
error: incompatible types
found   : Box<capture of ? extends java.lang.Number>
required: Box<java.lang.Number>
    Box<Number> box3 = unknownBox.copy();
                          ^

```

The `equalTo` method cannot be called through a reference variable of type `Box<? extends Number>` because the argument type is still an unknown type. Thanks to the upper bound the compiler knows that the expected argument type must be an instantiation of `Box` for a type argument that is a subtype of `Number`, but the compiler still does not know which instantiation exactly. Hence, the compiler cannot make sure that the right type of argument is provided for the method invocation and rejects the method invocation. The effect is exactly the same as for `Box<?>`, and likewise for `Box<? super Number>`.

The `copy` method can be called and in this case the result can be assigned to a reference variable of the more specific type `Box<? extends Number>`, instead of just `Box<?>`. When invoked on a `Box<? super Number>` the result would be assignable to a `Box<? super Number>`.

## REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[In a wildcard instantiation, can I read and write fields whose type is the type parameter?](#)

[What is a wildcard instantiation?](#)

[What is a wildcard?](#)

[What is the capture of a wildcard?](#)

---

## Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?

*We cannot call methods that use the type parameter as an upper wildcard bound in an argument type, with an exception for access through a lower bound wildcard parameterized type. We can call methods that use the type parameter as the upper wildcard bound in the return type; the returned value is accessible through the unbounded wildcard instantiation of the return type.*

In an upper bound wildcard parameterized type, we cannot call methods that use the type parameter as an upper wildcard bound in an argument type. This holds for access through unbounded and upper bound wildcard parameterized types. Access through a lower bound wildcard parameterized type is possible for certain argument types.

We can call methods that use the type parameter as the upper wildcard bound in the return type; the returned value is accessible through the unbounded wildcard instantiation of the return type. This holds for access through unbounded and lower bound wildcard parameterized types. Access through an upper bound wildcard parameterized type yields a more specific return type, namely a return type that corresponds to the upper bound wildcard instantiation that was used for the method invocation, e.g. a method of `Box<? extends Number>` would return a `ReturnType<? extends Number>`.

*Unbounded wildcard parameterized type.*

Example (access through unbounded wildcard):

```
class Box<T> {
    private T t;
    ...
    public void takeContentFrom(Box<? extends T> box) { t = box.t; }
    public Class<? extends T> getContentType() { ... }
}
class Test {
    public static void main(String[] args) {
        Box<Number> numberBox = new Box<Number>(5L);
        Box<?> unknownBox = numberBox;
```

```

unknownBox.takeContentFrom(numberBox); // error
unknownBox.takeContentFrom(unknownBox); // error

Class<Number> type0 = unknownBox.getContentTypes(); // error
Class<? extends Number> type1 = unknownBox.getContentTypes(); // error
Class<?> type2 = unknownBox.getContentTypes(); // ok
}
}

```

---

```

error: takeContentFrom(Box<? extends capture of ?>) in Box<capture of ?>
cannot be applied to (Box<java.lang.Number>)
    unknownBox.takeContentFrom(numberBox);
                ^
error: takeContentFrom(Box<? extends capture of ?>) in Box<capture of ?>
cannot be applied to (Box<capture of ?>)
    unknownBox.takeContentFrom(unknownBox);
                ^
error: incompatible types
found   : java.lang.Class<capture of ? extends capture of ?>
required: java.lang.Class<java.lang.Number>
    Class<Number> type0 = unknownBox.getContentTypes();
                                ^
error: incompatible types
found   : java.lang.Class<capture of ? extends capture of ?>
required: java.lang.Class<? extends java.lang.Number>
    Class<? extends Number> type1 = unknownBox.getContentTypes();
                                ^

```

We cannot call the `takeContentFrom` through a reference variable of type `Box<?>`, because the compiler does not know which type of object is expected as an argument of the `takeContentFrom` method. From the error message you can tell what the compiler find, namely a method with the signature `takeContentFrom(Box<? extends capture of ?>)`. The term `Box<? extends capture of ?>` stands for an instantiation of `Box` with a type argument of an unknown type that is a subtype of another unknown type. In essence, the argument type is unknown and the compiler has no chance to perform any type checks to make sure the correct type of argument is passed to the method call. And hence the invocation is illegal.

Invocation of the `getContentTypes` method is permitted. The return value is of a type that is an unknown instantiation of the `Class` type to which we can refer through a variable of type `Class<?>`. More specific information about the instantiation of the `Class` type is not available and for this reason the assignment to instantiations such as `Class<? extends Number> as Class<Number>` fails.

*Upper bound wildcard parameterized type.*

Let us see what the situation is like when we use a bounded wildcard parameterized type instead of the unbounded one.

Example (access through upper bound wildcard):

```

class Box<T> {
    private T t;
    ...
    public void takeContentFrom(Box<? extends T> box) { t = box.t; }
    public Class<? extends T> getContentTypeInfo() { ... }
}
class Test {
    public static void main(String[] args) {
        Box<Number> numberBox = new Box<Number>(5L);
        Box<? extends Number> unknownBox = numberBox;

        unknownBox.takeContentFrom(numberBox); // error
        unknownBox.takeContentFrom(unknownBox); // error

        Class<Number> type0 = unknownBox.getContentTypeInfo(); // error
        Class<? extends Number> type1 = unknownBox.getContentTypeInfo(); // ok
    }
}

```

---

```

error: takeContentFrom(Box<? extends capture of ? extends java.lang.Number>)
in Box<capture of ? extends java.lang.Number>
cannot be applied to (Box<java.lang.Number>)
    unknownBox.takeContentFrom(numberBox);
                ^
error: takeContentFrom(Box<? extends capture of ? extends java.lang.Number>)
in Box<capture of ? extends java.lang.Number>
cannot be applied to (Box<capture of ? extends java.lang.Number>)
    unknownBox.takeContentFrom(unknownBox);
                ^
error: incompatible types
found   : java.lang.Class<capture of ? extends capture of ? extends java.lang.Number>
required: java.lang.Class<java.lang.Number>
    Class<Number> type1 = unknownBox.getContentTypeInfo();
                                ^

```

In an upper bound wildcard parameterized type such as `Box<? extends Number>` the behavior is the same as in an unbounded wildcard parameterized type. The invocation of `takeFromContent` is rejected because the argument type in `unknown`. The argument type is `Box<? extends capture of ? extends Number>` which is an instantiation of `Box` for an unknown subtype of an unknown subtype of `Number`.

Invocation of the `getContentTypeInfo` method is permitted. The return type is more specific than for the unbounded wildcard case; we can refer to the result through a variable of type `Class<? extends Number>`. This is because the return type is `Class<capture of ? extends capture of ? extends Number>`, which is an unknown subtype of an unknown subtype of `Number`, that is, a subtype of `Number`.

*Lower bound wildcard parameterized type.*

Last but not least, the invocation through a lower bound wildcard.

Example (access through lower bound wildcard):

```
class Box<T> {
    private T t;
    ...
    public void takeContentFrom(Box<? extends T> box) { t = box.t; }
    public Class<? extends T> getContent() { ... }
}
class Test {
    public static void main(String[] args) {
        Box<Long> longBox = new Box<Long>(0L);
        Box<Number> numberBox = new Box<Number>(0L);
        Box<Object> objectBox = new Box<Object>(5L);
        Box<? extends Number> unknownNumberBox = numberBox;
        Box<? super Number> unknownBox = numberBox;

        unknownBox.takeContentFrom(longBox); // ok
        unknownBox.takeContentFrom(numberBox); // ok
        unknownBox.takeContentFrom(objectBox); // error
        unknownBox.takeContentFrom(unknownNumberBox); // ok
        unknownBox.takeContentFrom(unknownBox); // error

        Class<Number> type1 = unknownBox.getContent(); // error
        Class<? extends Number> type2 = unknownBox.getContent(); // error
        Class<? super Number> type3 = unknownBox.getContent(); // error
        Class<?> type4 = unknownBox.getContent(); // ok
    }
}
```

---

```
error: takeContentFrom(Box<? extends capture of ? super java.lang.Number>)
```

```
in Box<capture of ? super java.lang.Number>
cannot be applied to (Box<java.lang.Object>)
```

```
    unknownBox.takeContentFrom(objectBox);
```

```
    ^
```

```
error: takeContentFrom(Box<? extends capture of ? super java.lang.Number>)
```

```
in Box<capture of ? super java.lang.Number>
cannot be applied to (Box<capture of ? super java.lang.Number>)
```

```
    unknownBox.takeContentFrom(unknownBox);
```

```
    ^
```

```
error: incompatible types
```

```
found   : java.lang.Class<capture of ? extends capture of ? super java.lang.Number>
```

```
required: java.lang.Class<java.lang.Number>
```

```
    Class<Number> type1 = unknownBox.getContent();
```



```

error: incompatible types
found   : java.lang.Class<capture of ? extends capture of ? super java.lang.Number>
required: java.lang.Class<? extends java.lang.Number>
    Class<? extends Number> type2 = unknownBox.getContentType();
                                             ^
error: incompatible types
found   : java.lang.Class<capture of ? extends capture of ? super java.lang.Number>
required: java.lang.Class<? super java.lang.Number>
    Class<? super Number> type3 = unknownBox.getContentType();
                                             ^

```

The key difference for lower bound wildcards is that the `takeContentFrom` method can be called for certain argument types, namely for those types that are members of the type family denoted by `Box<? extends Number>` in our example. It is basically as though the `takeContentFrom` method in `Box<? super Number>` had the signature `takeContentFrom(Box<? extends Number>)`. Why is this? The compiler determines the signature of the `takeContentFrom` method in `Box<? super Number>` as `takeContentFrom(Box<? extends capture of ? super java.lang.Number>)`. Now, what does "`? extends capture of ? super java.lang.Number`" mean? It is an unknown subtype of an unknown supertype of `Number`. No matter what this unknown supertype of `Number` may be, the subtypes of `Number` would conform to this description.

Invocation of the `getContentType` method is permitted, as expected. Perhaps surprising is that fact that the return type is not `Class<? super Number>`, but only `Class<?>`. This is because the return type is `Class<capture of ? extends capture of ? super Number>`, which is an unknown subtype of an unknown supertype of `Number`. Just imagine the unknown supertype of `Number` were `Object`, then it could be any type. Hence we know nothing about the return type except that it is an instantiation of class `Class` and `Class<?>` correctly describes it.

LINK TO THIS [Technicalities.FAQ606](#)

#### REFERENCES

- [Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)
- [Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)
- [Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized typ?](#)
- [Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)
- [Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)
- [In a wildcard instantiation, can I read and write fields whose type is the type parameter?](#)
- [What is a wildcard instantiation?](#)
- [What is a wildcard?](#)
- [What is the capture of a wildcard?](#)

---

## Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?

*We can call methods that use the type parameter as a lower wildcard bound in the return type; the returned value is accessible through a wildcard instantiation of the return type (which one depends on the wildcard parameterized type being used). We can call methods that use the type parameter as a lower wildcard*

*bound in an argument type; the argument type is restricted depending on the wildcard being use.*

In a lower bound wildcard parameterized type, we can call methods that use the type parameter as a lower wildcard bound in the return type. The returned value is accessible through a wildcard instantiation of the return type. In an unbounded wildcard parameterized type and an upper bound wildcard parameterized type the return type is the unbounded wildcard instantiation of the return type. In a lower bound wildcard parameterized type the return type is more specific, namely the lower bound wildcard instantiation of the return type. That is, a method in class `Box` that return `Comparable<? super T>` would return `Comparable<?>` in `Box<?>` and `Box<? extends Number>` and would return `Comparable<? super Number>` in `Box<? super Number>`.

This matches the return types of methods with an upper bound wildcard; a method in class `Box` that return `Comparable<? extends T>` would return `Comparable<?>` in `Box<?>` and `Box<? super Number>` and would return `Comparable<? extends Number>` in `Box<? extends Number>`. The reasoning for lower bound wildcard parameterized types is the exactly the same as for upper bound wildcard parameterized types. The more specific return type in a lower bound wildcard parameterized type stems from the fact that the return type of a method that returns `Comparable<? super T>` in the instantiation `Box<? super Number>` would be `Comparable<capture of ? super capture of ? super Number>`, which boils down to `Comparable<? super Number>`.

More interesting is the invocation of methods that take arguments with lower wildcard bounds. We can call methods that use the type parameter as a lower wildcard bound in an argument type. In an unbounded wildcard parameterized type and a lower bound wildcard parameterized type the only permitted argument type is the argument type instantiated for type `Object`. That is, a method in class `Box` that takes an argument of type `Comparable<? super Number>` would in the parameterized types `Box<?>` and `Box<? super Number>` accept arguments of `Comparable<Object>`. Note, this is different from methods that use the type parameter as a upper wildcard bound in an argument type; they cannot be invoked at all. The reason for this permitted argument type is that such a method would have the signature `method(Comparable<? super capture of ?>)` in an unbounded parameterized type such as `Box<?>`, and `"? super capture of ?"` denotes an unknown supertype of an unknown type. The ultimate supertype of all types is `Object`, hence `Comparable<Object>` is permitted as an argument type. And likewise in `Box<? super Number>`, where the signature would involve `"? super capture of ? super Number"`, and again `Object` is the only type that would fit.

In an upper bound wildcard parameterized type with upper bound `Bound` the permitted arguments types are the types that belong to the family denoted by `ArgumentType<? super Bound>`. That is, a method in class `Box` that takes an argument of type `Comparable<? super Number>` would in the instantiation `Box<? extends Number>` accept arguments from the type family `Comparable<? super Number>`. Note, this is similar to a method with an upper bound argument type in an lower bound parameterized type; e.g. a method in class `Box` that takes an argument of type `Comparable<? extends Number>` would in the parameterized type `Box<? super Number>` accept arguments from the type family `Comparable<? extends Number>`. The reason for the permitted argument types is that the compiler determines the signature of such a method in `Box<? extends Number>` as `method(Box<? super capture of ? extends Number>)`. `" super capture of ? extends Number"` means unknown supertype of an unknown subtype of `Number`, in other words all supertypes of `Number`.

*Unbounded wildcard parameterized type.*

Example (access through unbounded wildcard):

```
class Box<T> {
    private T t;
    ...
    public int compareTo(Comparable<? super T> other) { return other.compareTo(t); }
    public Box<? super T> copy() { return new Box<T>(t); }
}
class Test {
    public static void main(String[] args) {
        Box<Number> numberBox = new Box<Number>(5L);
        Box<?> unknownBox = numberBox;
    }
}
```

```

Comparable<?> comparableToUnknown = new Integer(1);
Comparable<Object> comparableToObject = ...;
Comparable<? super Number> comparableToNumber = comparableToObject;

int compared = 0;
compared = unknownBox.compareTo(comparableToUnknown); // error
compared = unknownBox.compareTo(comparableToObject); // ok
compared = unknownBox.compareTo(comparableToNumber); // error

Box<?>          box1 = unknownBox.copy(); // ok
Box<? extends Number> box2 = unknownBox.copy(); // error
Box<? super Number> box3 = unknownBox.copy(); // error
}
}

```

---

```

error: compareTo(java.lang.Comparable<? super capture of ?>) in Box<capture of ?> cannot be applied to
(java.lang.Comparable<capture of ?>)

```

```

    compared = unknownBox.compareTo(comparableToUnknown);

```

```

    ^

```

```

error: compareTo(java.lang.Comparable<? super capture of ?>) in Box<capture of ?> cannot be applied to
(java.lang.Comparable<capture of ? super java.lang.Number>)

```

```

    compared = unknownBox.compareTo(comparableToNumber);

```

```

    ^

```

```

error: incompatible types

```

```

found   : Box<capture of ? super capture of ?>

```

```

required: Box<? extends java.lang.Number>

```

```

    Box<? extends Number> box2 = unknownBox.copy();

```

```

    ^

```

```

error: incompatible types

```

```

found   : Box<capture of ? super capture of ?>

```

```

required: Box<? super java.lang.Number>

```

```

    Box<? super Number> box3 = unknownBox.copy();

```

```

    ^

```

The example shows that method `compareTo` can only be invoked for arguments of type `Comparable<Object>` and that the return type of method `copy` is `Comparable<?>`.

*Upper bound wildcard parameterized type.*

Example (access through upper bound wildcard):

```

class Box<T> {
    private T t;
    ...
    public int compareTo(Comparable<? super T> other) { return other.compareTo(t); }
}

```

```

    public Box<? super T> copy() { return new Box<T>(t); }
}
class Test {
    public static void main(String[] args) {
        Box<Number> numberBox = new Box<Number>(5L);
        Box<? extends Number> unknownBox = numberBox;

        Comparable<?> comparableToUnknown = new Integer(1);
        Comparable<Object> comparableToObject = ...;
        Comparable<? super Number> comparableToNumber = comparableToObject;

        int compared = 0;
        compared = unknownBox.compareTo(comparableToUnknown); // error
        compared = unknownBox.compareTo(comparableToObject); // ok
        compared = unknownBox.compareTo(comparableToNumber); // ok

        Box<?>          box1 = unknownBox.copy(); // ok
        Box<? extends Number> box2 = unknownBox.copy(); // error
        Box<? super Number>  box3 = unknownBox.copy(); // error
    }
}

```

---

```

error: compareTo(java.lang.Comparable<? super capture of ? extends java.lang.Number>) in Box<capture of ? extends
java.lang.Number> cannot be applied to (java.lang.Comparable<capture of ?>)

```

```

    compared = unknownBox.compareTo(comparableToUnknown);
                                ^

```

```

error: incompatible types

```

```

found   : Box<capture of ? super capture of ? extends java.lang.Number>
required: Box<? extends java.lang.Number>

```

```

    Box<? extends Number> box2 = unknownBox.copy();
                                ^

```

```

error: incompatible types

```

```

found   : Box<capture of ? super capture of ? extends java.lang.Number>
required: Box<? super java.lang.Number>

```

```

    Box<? super Number> box3 = unknownBox.copy();
                                ^

```

The example shows that method `compareTo` can only be invoked for arguments from the type family denoted by `Comparable<? super Number>` and that the return type of method `copy` is `Comparable<?>`.

*Lower bound wildcard parameterized type.*

Example (access through lower bound wildcard):

```

class Box<T> {
    private T t;

```

```

...
public int compareTo(Comparable<? super T> other) { return other.compareTo(t); }
public Box<? super T> copy() { return new Box<T>(t); }
}
class Test {
public static void main(String[] args) {
    Box<Number> numberBox = new Box<Number>(5L);
    Box<? super Number> unknownBox = numberBox;

    Comparable<?> comparableToUnknown = new Integer(1);
    Comparable<Object> comparableToObject = ...;
    Comparable<? super Number> comparableToNumber = comparableToObject;

    int compared = 0;
    compared = unknownBox.compareTo(comparableToUnknown); // error
    compared = unknownBox.compareTo(comparableToObject); // ok
    compared = unknownBox.compareTo(comparableToNumber); // error

    Box<?>          box1 = unknownBox.copy(); // ok
    Box<? extends Number> box2 = unknownBox.copy(); // error
    Box<? super Number> box3 = unknownBox.copy(); // ok
}
}

```

---

```

error: compareTo(java.lang.Comparable<? super capture of ? super java.lang.Number>) in Box<capture of ? super
java.lang.Number> cannot be applied to (java.lang.Comparable<capture of ?>)

```

```

    compared = unknownBox.compareTo(comparableToUnknown);

```

```

    ^

```

```

error: compareTo(java.lang.Comparable<? super capture of ? super java.lang.Number>) in Box<capture of ? super
java.lang.Number> cannot be applied to (java.lang.Comparable<capture of ? super java.lang.Number>)

```

```

    compared = unknownBox.compareTo(comparableToNumber);

```

```

    ^

```

```

error: incompatible types

```

```

found   : Box<capture of ? super capture of ? super java.lang.Number>
required: Box<? extends java.lang.Number>

```

```

    Box<? extends Number> box3 = unknownBox.copy();

```

```

    ^

```

The example shows that method `compareTo` can only be invoked for arguments of type `Comparable<Object>` and that the return type of method `copy` is `Comparable<? super Number>`.

#### LINK TO THIS

[Technicalities.FAQ607](#)

#### REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[In a wildcard parameterized type, can I read and write fields whose type is the type parameter?](#)

[What is a wildcard instantiation?](#)

[What is a wildcard?](#)

[What is the capture of a wildcard?](#)

---

## In a wildcard parameterized type, can I read and write fields whose type is the type parameter?

*It depends on the kind of wildcard.*

In a wildcard parameterized type a field whose type is the type parameter of the enclosing generic class is of unknown type. Depending on the wildcard (whether it is unbounded or has an upper or lower bound) different kinds of access are permitted or disallowed.

---

*Unbounded wildcards.*

Assignment to a field of the "unknown" type is rejected because the compiler cannot judge whether the object to be assigned is an acceptable one. Read access to a field of the "unknown" type is permitted, but the field has no specific type and must be referred to through a reference of type `Object`.

Example:

```
class Box<T> {
    public T t; // public just for sake of demonstration
    ...
}
class Test {
    public static void main(String[] args) {
        Box<?> box = new Box<String>("abc");

        box.t = "xyz";    // error
        box.t = null;    // ok

        String s = box.t; // error
        Object o = box.t; // ok
    }
}
```

---

*Wildcards with an upper bound.*

The same rules apply to wildcard parameterized type with an upper bound wildcard.

Example:

```
class Box<T> {
    public T t; // public just for sake of demonstration
    ...
}
class Test {
    public static void main(String[] args) {
        Box<? extends Number> box = new Box<Long>(0L);

        box.t = 1L; // error
        box.t = null; // ok

        Number n = box.t; // ok
        Object o = box.t; // ok
    }
}
```

The only difference that a field of "unknown" type is known to be compatible to the upper bound. Hence we can assign the field to a reference variable of type `Number` in our example.

---

*Wildcards with a lower bound.*

The rules for wildcard parameterized types with a lower bound wildcard are the other way round. We can assign `null` or a value whose type is the lower bound or a subtype thereof, but we cannot assign a value that is of a supertype of the lower bound. And we must not make any assumptions regarding the type of the field; it must be treated like an `Object`.

Example:

```
class Box<T> {
    public T t; // public just for sake of demonstration
    ...
}
class Test {
    public static void main(String[] args) {
        Box<? super Long> box = new Box<Number>(0L);
        Number number = new Integer(1);

        box.t = 1L; // ok
        box.t = null; // ok
        box.t = number; // error

        Long l = box.t; // error
        Number n = box.t; // error
        Object o = box.t; // ok
    }
}
```

LINK TO THIS

[Technicalities.FAQ608](#)

REFERENCES

[Which methods and fields are accessible/inaccessible through a reference variable of a wildcard type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an unbounded wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in an upper bound wildcard parameterized type?](#)

[Which methods that use the type parameter in the argument or return type are accessible in a lower bound wildcard parameterized type?](#)

[Which methods that use the type parameter as type argument of a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as upper wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[Which methods that use the type parameter as lower wildcard bound in a parameterized argument or return type are accessible in a wildcard parameterized type?](#)

[In a wildcard parameterized type, can I read and write fields whose type is the type parameter?](#)

[What is a wildcard instantiation?](#)

[What is a wildcard?](#)

[What is an unbounded wildcard?](#)

[What is a bounded wildcard?](#)

---

## Is it really impossible to create an object whose type is a wildcard parameterized type?

*It is actually illegal and the compiler tries to prevent it, but there's a workaround.*

The creation of objects of a wildcard parameterized type is discouraged: it is illegal that a wildcard parameterized type appears in a new expression.

Example (of illegal creation of objects of a wildcard parameterized type):

```
class Sequence<E> {
    public Sequence() {...}
    public Sequence(int size) {...}
    public <F extends E> Sequence(F arg) {...}
    public <F extends E> Sequence(F[] a) {...}
    public Sequence(Sequence<? extends E> s) {...}
}

Sequence<?> seq1 = new Sequence<?>(); // error
Sequence<?> seq2 = new Sequence<?>(10); // error

Sequence<String> s = new Sequence<String>();
... populate the sequence ...

Sequence<? super String> seq3 = new Sequence<? super String>("xyz"); // error
Sequence<? super String> seq4 = new Sequence<? super String>(new String[] {"abc", "ABC"}); // error
Sequence<? super String> seq5 = new Sequence<? super String>(s); // error
```

The compiler rejects all attempts to create an object of wildcard types such as `Sequence<?>` or `Sequence<? super String>`.

The compiler's effort to prevent the creation of objects of a wildcard parameterized type can easily be circumvented. It is unlikely that you will ever want to create an object of a wildcard parameterized type, but should you ever need one, here's the workaround.

Instead of directly creating the object of the wildcard parameterized type via a new expression you define a generic factory method that creates the object.



Example (of creation of objects of a wildcard parameterized type; workaround):

```
class Factory {
    public interface Dummy<T> {}
    public final static Dummy<?> unboundedDummy = null;
    public final static Dummy<? super String> superStringDummy = null;

    public static <T> Sequence<T> make(Dummy<T> arg) {
        return new Sequence<T>();
    }
    public static <T> Sequence<T> make(Dummy<T> arg, int size) {
        return new Sequence<T>(size);
    }
    public static <T,S extends T> Sequence<T> make(Dummy<T> d, S arg) {
        return new Sequence<T>(arg);
    }
    public static <T,S extends T> Sequence<T> make(Dummy<T> d, S[] array) {
        return new Sequence<T>(array);
    }
    public static <T> Sequence<T> make(Dummy<T> d, Sequence<? extends T> seq) {
        return new Sequence<T>(seq);
    }
}

Sequence<?> seq1 = Factory.make(Factory.unboundedDummy); // fine
Sequence<?> seq2 = Factory.make(Factory.unboundedDummy,10); // fine

Sequence<String> s = new Sequence<String>();
... populate the sequence ...
Sequence<? super String> seq3 = Factory.make(Factory.superStringDummy,"xyz"); // fine
Sequence<? super String> seq4 = Factory.make(Factory.superStringDummy,new String[] {"abc","ABC"}); // fine
Sequence<? super String> seq5 = Factory.make(Factory.superStringDummy,s); // fine
```

The trick is defining a generic factory method that takes a dummy argument for each constructor that you had otherwise used. This dummy argument must be a of a generic type and is needed so that the compiler can infer the type argument when the factory method is invoked. When the method is invoked with a dummy argument that is of a wildcard type, then the compiler infers that the factory method's type argument is that particular wildcard and consequently the factory method creates an object of the corresponding wildcard parameterized type.

In the example the `make` method is first invoked twice with a dummy argument of the unbounded wildcard type `Dummy<?>`. The type argument `T` is inferred as `T:=?` (or more precisely `T:=capture of ?`) and hence the `make` method creates an object of type `Sequence<?>`. The subsequent three invocations use a dummy argument of the unbounded wildcard type `Dummy<? super String>`. Consequently, the compiler infers `T` as `T:=? super String` (or more precisely `T:=capture of ? super String`). In addition, the compiler infers `s` as `S:=String` and checks that `s` is a subtype of `T`.

As mentioned above, it is unlikely you will ever need this workaround.

## REFERENCES

[What is a wildcard parameterized type?](#)

[Can I create an object whose type is a wildcard parameterized type?](#)

[What is type argument inference?](#)

---

## Cast and instanceof

### Which types can or must not appear as target type in an instanceof expression?

*Only reifiable types are permitted.*

It is a compile-time error if the reference type mentioned after the `instanceof` operator does not denote a reifiable type. In other words, concrete and bounded wildcard parameterized types are NOT permitted in an `instanceof` expression.

Examples:

```
Object o = new LinkedList<Long>();

System.out.println (o instanceof List);
System.out.println (o instanceof List<?>);
System.out.println (o instanceof List<Long>);           // error
System.out.println (o instanceof List<? extends Number>); // error
System.out.println (o instanceof List<? super Number>); // error
```

The reason for disallowing non-reifiable types (i.e., instantiations of a generic type with at least one type arguments that is a concrete type or a bounded wildcard) in `instanceof` expression is that these parameterized types do not have an exact runtime type representation. Their dynamic type after type erasure is just the raw type. The evaluation of an `instanceof` expression could at best check whether the object in question is an instance of the raw type. In the example, the expression `(o instanceof List<Long>)` would check whether `o` is an instance of type `List`, which is a check different from what the source code suggests. In order to avoid confusion, the non-reifiable types are prohibited in `instanceof` expression.

Only the reifiable types (i.e., the raw type and the unbounded wildcard parameterized type) are permitted in an `instanceof` expression. The reifiable types do not lose any type information during translation by type erasure. For this reason, the `instanceof` check makes sense and is allowed.

## LINK TO THIS

[Technicalities.FAQ701](#)

## REFERENCES

[What is a reifiable type?](#)

[What is an unbounded wildcard instantiation?](#)

---

## Overloading and Overriding

---

## What is method overriding?

*When a subtype redefines a method that was inherited from a supertype.*

Overriding is what we do when we derive subtypes from supertypes and specialize the subtype's behaviour by means of implementing in the subtype a specialized version of a method inherited from the supertype. Overriding is one of the key concepts of object-oriented programming.

Example (of method overriding):

```
class Super {
    public Type method(Type arg) { ... }
}
class Sub extends Super {
    public Type method(Type arg) { ... } // overrides Super.method
}
```

The purpose of overriding is polymorphic method dispatch, that is, when the method is invoked through a reference of the supertype and that reference refers to an instance of the subtype then the subtype version of the method is invoked.

Example (of polymorphic method dispatch):

```
Super ref1 = new Super();
Super ref2 = new Sub();

ref1.method(new Type()); // calls Super.method
ref2.method(new Type()); // calls Sub.method
```

When the overridden method is an abstract method or when the supertype is an interface, then it is said that the overriding method *implements* the supertype's method declaration.

Example (of an overriding method that implements a supertype's method):

```
interface Callable<V> {
    V call();
}
class Task implements Callable<Long> {
    public Long call() { ... } // implements Callable<Long>.call
}
```

LINK TO THIS

[Technicalities.FAQ801](#)

REFERENCES

[What is method overriding?](#)

[What is method overloading?](#)

[What is the @Override annotation?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

[When does a method override its supertype's method?](#)

[Can a method of a non-generic subtype override a method of a generic supertype?](#)

[Can a method of a generic subtype override a method of a generic supertype?](#)

[Can a generic method override a generic one?](#)

[Can a non-generic method override a generic one?](#)

[Can a generic method override a non-generic one?](#)

[Why doesn't method overriding work as I expect it?](#)

---

## What is method overloading?

*When a class has two methods with the same name, but signatures that are not override-equivalent.*

Method overloading happens among methods defined in the same type, when the methods have the same name and differ in the number or type of arguments.

Example (of method overloading):

```
class Container {
    public void remove(int index) { ... }
    public void remove(Object o) { ... } // overloads
}
```

The overloading methods need not necessarily be defined in the same type, but can also be inherited from a supertype.

Example (of method overloading):

```
class Super {
    public void method(String arg) { ... }
    public void method(StringBuffer arg) { ... } // overloads
}
class Sub extends Super {
    public void method(StringBuilder arg) { ... } // overloads
}
```

In this example, all three versions of the method overload each other because they have the same name but different argument types. When the method is invoked the compiler picks the best match from the three available candidates. This process is called *overload resolution*.

Example (of overload resolution):

```
Sub ref = new Sub();

ref.method("ABC"); // calls Super.method(String)
```

```
ref.method(new StringBuilder("ABC")); // calls Sub.method(StringBuilder)
```

The return type is irrelevant for overloading, that is, overloaded methods can have different return types. It is an error if the difference is only in the return type.

Example (of method overloading):

```
class Container {
    public boolean remove(Object o) { ... }
    public Object  remove(int index) { ... } // overloads
    public void    remove(int index) { ... } // error: already defined
}
```

The first two methods overload each other, because they have the same name and different arguments type. The last two methods are in conflict, because they have the same name and the same argument type and differ only in the return type. The general rule is that overloaded methods must have the same name but different method *signatures*, and the return type is not part of a method signature. A more precise definition of method signature is given in FAQ entry [FAQ810](#).

Generic methods can be overloaded as well.

Example (of overloaded generic methods):

```
class SomeClass {
    public <T>          void method(T arg) { ... }
    public <T extends Number> void method(T arg) { ... } // overloads
    public             void method(Long arg) { ... } // overloads
}
```

These three methods have different signatures. Even the first and the second method differ, because they have different argument types, namely a type parameter with bound `Object` and a type parameter with bound with bound `Number`. This is because the type parameters of a generic method including the type parameter bounds are part of the method signature.

Method overloading is sometimes confused with method *overriding*. Overriding happens among a method in a subtype and a method with the same name inherited from a supertype, if the signatures of those methods are *override-equivalent*. This means that methods in the same class never override each other; they can only overload each other. But methods in a subtype can either override or overload a supertype's method with the same name. It is sometimes difficult to tell the difference between overloading and overriding. The rule is: the subtype method overloads the supertype method if it is not override-equivalent. Override-equivalence is explained in more detail in FAQ entry [FAQ812](#).

Example (of method overloading vs. overriding):

```
class Super {
    public void method(String arg) { ... }
}
class Sub extends Super {
    public void method(String arg) { ... } // overrides
    public void method(StringBuilder arg) { ... } // overloads
}
```

The first method in the subtype has an override-equivalent signature, basically because it has the same signature as the supertype method. The second method has a different argument type and therefore is not override-equivalent and hence overloads the supertype method.

The rules for overriding are more complex than sketched out above; they are discussed in detail in the subsequent FAQ entries. In particular, overriding does not only require override-equivalent signatures, but additionally a compatible return type and a compatible `throws` clause. No such rules apply to overloading.

Example (of overloaded methods with different return types and `throws` clauses):

```
class SomeClass {
    public void    method(String arg) { ... }
    public void    method(StringBuffer arg) throws IllegalArgumentException { ... } // overloads
    public boolean method(StringBuilder arg) { ... } // overloads
}
```

There is one additional rule, that applies to methods in general: methods declared in the same type (or inherited from a supertype) must not have the same erasure. This is because generics are translated by type erasure. Two methods with the same erasure would have identical signatures in the byte code and the virtual machine could not distinguish between them. This is prohibited and the compiler issues an error method if two method have the same erasure.

Example (of conflicting erasures):

```
class SomeClass {
    public <T extends Number> void method(T arg) { ... }
    public                          void method(Number arg) { ... } // error: already defined
}
```

Before type erasure the two methods have different signatures, namely `<T1_extends_Number>method(T1_extends_Number)` and `method(Number)`. After type erasure they both have the signature `method(Number)` and consequently the compiler reports an error.

LINK TO THIS [Technicalities.FAQ802](#)

REFERENCES [What is overload resolution?](#)  
[Why doesn't method overloading work as I expect it?](#)

---

## What is the `@Override` annotation?

*An annotation that you can attach to a method so that the compiler issues an error if the annotated method does not override any supertype method.*

Overriding and overloading are often confused. If you are in doubt, you can annotate a subtype's method with the standard annotation [`java.lang.Override`](#). The compiler then checks whether the annotated method overrides a supertype method. If not, it reports an error.

Example (of using the `@Override` annotation):

```
class Super {
    public <T> void method(T arg) { ... }
}
class Sub extends Super {
    @Override public <T extends Number> void method(T arg) { ... } // error: name clash
}
```

```
}
```

```
error: method does not override a method from its superclass
    @Override public <T extends Number> void method(T arg) {
        ^
```

#### LINK TO THIS

[Technicalities.FAQ803](#)

#### REFERENCES

[What is method overriding?](#)  
[What is method overloading?](#)

## What is a method signature?

*An identification of a method that consist of the method's name and its arguments types.*

A method *signature* consists of the method's name and its arguments types. This is different from a method *descriptor*, which is the signature plus the return type. In case of generic methods, the type parameters are part of the signature. Method signatures play a role in overloading and overriding.

Examples:

<i>Declaration</i>	<i>Signature</i>
<code>void method(String arg)</code>	<code>method(String)</code>
<code>int method(String arg)</code>	<code>method(String)</code>
<code>String method(String arg1, int arg2) throws Exception</code>	<code>method(String,int)</code>
<code>void method(List&lt;? extends Number&gt; list)</code>	<code>method(List&lt;?_extends_Number&gt;)</code>
<code>T method(T arg)</code> *) where T is enclosing class's type parameter	<code>method(\$T1_extends_Object)</code>
<code>void add(K key, V val)</code> *) with type parameters K and V from enclosing class	<code>add(\$T1_extends_Object,\$T2_extends_Object)</code>
<code>&lt;T&gt; T method(T arg)</code>	<code>&lt;\$T1_extends_Object&gt;method(\$T1_extends_Object)</code>
<code>&lt;E&gt; int method(E arg)</code>	<code>&lt;\$T1_extends_Object&gt;method(\$T1_extends_Object)</code>
<code>&lt;T extends Number&gt; T method(T arg)</code>	<code>&lt;\$T1_extends_Number&gt;method(\$T1_extends_Number)</code>
<code>&lt;A extends I&amp;J, B extends J&amp;I&gt; void method(A a, B b)</code>	<code>&lt;\$T1_extends_I&amp;J,\$T2_extends_I&amp;J&gt;method(\$T1_extends_I&amp;J,\$T2_extends_I&amp;J)</code> <code>&lt;\$T1_extends_Object&gt;method</code>

<code>&lt;T&gt; void method(List&lt;T&gt; list)</code>	<code>(List&lt;\$T1_extends_Object&gt;)</code>
<code>&lt;S extends Number&gt; void method(List&lt;S&gt; list)</code>	<code>&lt;\$T1_extends_Number&gt;method (List&lt;\$T1_extends_Number&gt;)</code>
<code>&lt;E&gt; void sort(List&lt;? extends E&gt; list)</code>	<code>&lt;\$T1_extends_Object&gt;sort (List&lt;? extends \$T1_extends_Object&gt;)</code>
<code>&lt;E extends Comparable&lt;Number&gt;&gt; void sort(List&lt;E&gt; list)</code>	<code>&lt;\$T1_extends_Comparable&lt;Number&gt;&gt;sort (List&lt;\$T1_extends_Comparable&lt;Number&gt;&gt;)</code>
<code>&lt;E extends Comparable&lt;E&gt;&gt; void sort(List&lt;E&gt; list)</code>	<code>&lt;\$T1_extends_Comparable&lt;\$T1_extends_Comparable&gt;&gt;sort (List&lt;\$T1_extends_Comparable&lt;\$T1_extends_Comparable&gt;&gt;)</code>
<code>&lt;E extends Comparable&lt;? super K&gt;&gt; void sort(List&lt;E&gt; l)</code>	<code>&lt;\$T1_extends_Comparable&lt;?_super_\$T2_extends_Object&gt;&gt;sort (List&lt;\$T1_extends_Comparable&lt;\$T2_extends_Object&gt;&gt;)</code>

\*)  
where K is enclosing class's type parameter

The terms `$Tn_extends_Bound1&...&Boundn` stand for synthetic type variables. These synthetic names are used in lieu of the original names because the name of a type variable is irrelevant for the signature. For instance, the methods `<T> void method(T arg)` and `<S> void method(S arg)` have the same signature; they are a method named `method` with one unbounded type parameter and one method parameter whose type is the unbounded type parameter. Basically, the compiler renames all type variables of a generic method and uses synthetic names instead of the actual names.

The terms `$Bound1&...&Boundn` denote the type variable's bounds. For the purpose of a signature, the order of the bounds is irrelevant, that is, "`A extends I&J`" is equivalent to "`B extends J&I`". Just imagine the compiler would order the bounds alphabetically.

**LINK TO THIS**

[Technicalities.FAQ810](#)

**REFERENCES**

[What is method overriding?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

[When does a method override its supertype's method?](#)

## What is a subsignature?

*A method signature that is identical to another method signature, or identical to the erasure of another method signature.*

Subsignatures are relevant in conjunction with overriding. A prerequisite for overriding is that the subtype method's signature is a subsignature of the supertype method's signature.

A signature is a *subsignature* of another signature if:

- the two signatures are identical, or
- the signature is identical to the erasure of the other signature.

Example (subsignatures):

```
class Super {
    public void setName(String arg) { ... }
```



```

    public void setNames(Collection<String>c) { ... }
    public void printCollection(Collection<?>c) { ... }
}
class Sub extends Super {
    public void setName(String arg) { ... }           // subsignature
    public void setNames(Collection c) { ... }       // subsignature
    public void printCollection(Collection c) { ... } // subsignature
}

```

In the example, the subclass version of the `setName` method is a subsignature of the superclass version of the method, because it has the same signature. The subclass versions of the `setNames` and `printCollection` methods are subsignatures of the superclass versions of the methods, because their signatures are identical to the erasures of the superclass's methods.

Note, that in the case of the `setNames` method an unchecked warning is issued. The reasons for this warning are discussed in FAQ entry [FAQ812](#). The unchecked warning is issued whenever an argument type of the superclass method is a parameterized type and the corresponding argument type in the subclass method is a raw type, unless the parameterized type is the unbounded wildcard instantiation of the generic type.

LINK TO THIS [Technicalities.FAQ811](#)

#### REFERENCES

[What is method overriding?](#)  
[What is a method signature?](#)  
[What are override-equivalent signatures?](#)  
[When does a method override its supertype's method?](#)

## What are override-equivalent signatures?

*Two method signatures where one signature is a subsignature of the other.*

Override-equivalent signatures are relevant in conjunction with overriding and overloading. We talk of *override-equivalent* signatures when a subtype method's signature is a subsignature of a supertype method's signature.

The term *subsignature* was explained in FAQ entry [FAQ811](#): A signature is a *subsignature* of another signature if the signature is identical to the other signature or identical to the erasure of the other signature.

Override-equivalence is a prerequisite of method overriding.

- We talk of *overriding* if the signature of a subtype's method is override-equivalent to (synonym to: "is a subsignature of") a supertype's method.
- We talk of *overloading* if a class has two methods with the same name, but signatures that are *not* override-equivalent, that is, neither is a subsignature of the other.

Example (overloading and overriding):

```

class Super {

```

```

    public void setName(String arg) { ... }
}
class Sub extends Super {
    public void setName(StringBuilder arg) { ... } // overloads
    public void setName(String arg) { ... } // overrides
}

```

In the example, the two `setName` methods in the subclass have the same name, but different signatures: one method takes a `String` argument, the other takes a `StringBuilder` argument. As the two signatures are different (and neither is the subsignature of the other), these two methods are not override-equivalent and hence the *overload*.

In contrast, the two versions of the `setName` method taking a `String` argument in the super- and subclass *override*. This is because they have identical and therefore override-equivalent signatures.

In the example above, the erasure of the methods is irrelevant, because none of the methods has a generic argument type. Let us consider an examples that involves generic types.

Example (overloading and overriding):

```

class Super {
    public void printCollection(Collection<?> c) { ... }
}
class Sub extends Super {
    public void printCollection(List<?> c) { ... } // overloads
    public void printCollection(Collection c) { ... } // overrides
}

```

The two `printCollection` methods in the subclass have the same name, but different signatures: one method takes a `List<?>` argument, the other takes a `Collection` argument. As the two signatures are different (and neither is the subsignature of the other), these two methods *overload*.

In contrast, the two versions of the `printCollection` method taking a `Collection<?>` argument in the superclass and a `Collection` argument in the subclass *override* although the methods do not have identical signatures. But the signatures are almost identical: the subclass signature is the erasure of the superclass signature.

Note, that the converse is not permitted. If the superclass method had a signature that is identical to the erasure of a subclass method, the compiler would issue an error message. This is neither overloading nor overriding, but just a name clash.

Example (neither overloading nor overriding):

```

class Super {
    public void printCollection(Collection c) { ... }
}
class Sub extends Super {
    public void printCollection(Collection<?> c) { ... } // error
}

```

---

```

error: name clash: printCollection(List<?>) in Sub and printCollection(List) in Super
have the same erasure, yet neither overrides the other

```

```
class Sub extends Super {  
  ^
```

The notion of override-equivalent signatures is slightly more complex among generic methods. See [FAQ820](#) and subsequent entries for further details and examples.

#### LINK TO THIS

[Technicalities.FAQ812](#)

#### REFERENCES

[What is method overriding?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[When does a method override its supertype's method?](#)

[Can a method of a non-generic subtype override a method of a generic supertype?](#)

[Can a method of a generic subtype override a method of a generic supertype?](#)

[Can a generic method override a generic one?](#)

[Can a non-generic method override a generic one?](#)

[Can a generic method override a non-generic one?](#)

---

## When does a method override its supertype's method?

*When the subtype's method has a signature that is a subsignature of the supertype's method and the two methods have compatible return types and throws clauses.*

A subtype's method overrides a supertype's method if the subtype's method has a signature that is:

- identical to the supertype's method's signature, or
- identical to the erasure of the supertype's method's signature.

It is said that the subtype method has a *subsignature* or that the two signatures are *override-equivalent*.

---

### Subsignatures

First an example where the subtype's method has the same signature as the supertype's method.

Example (of overriding methods with identical signatures):

```
class Super {  
  public void method(String arg) { ... }  
}  
class Sub extends Super {  
  public void method(String arg) { ... } // overrides  
}
```

Both methods have the same signature, namely `method(String)`, and for this reason the subclass's method overrides the superclass's method.

Second, an example where the subtype's method has a signature whose erasure is identical to the signature of the supertype's method.

Example (of overriding methods with generics involved):

```
class Super {
    public void printCollection(Collection<?> c) { ... }
}
class Sub extends Super {
    public void printCollection(Collection c) { ... } // overrides
}
```

The erasures of the methods are the same, namely `printCollection(Collection)`. In this situation, the erased subtype method is considered an overriding version of the non-erased supertype method.

This kind of overriding is permitted in order to allow that legacy supertypes can be re-engineered and generified without affecting any existing subtypes of the legacy supertype. Imagine the supertype had originally not used any generics. In that original situation, the signatures of supertype and subtype method had been identical. After generification of the supertype the signatures are different. Without the rule that the subtype method can be the erasure of the supertype method all subtypes would also need to be re-engineered and generified. Fortunately, the additional rule renders this re-engineering effort unnecessary and the generification of the supertype does not affect overriding methods in subtypes.

---

There are additional rules for overriding. When a subtype's method overrides a supertype's method, then:

- the subtype method's return type must be substitutable for the supertype method's return type, and
- the `throws` clauses of both methods must not be in conflict.

### *Substitutable Return Types*

Let us first consider some examples of substitutable and incompatible return types before we look into conflicting `throws` clauses.

Example (incompatible return types):

```
class Super {
    public void method(String arg) { ... }
}
class Sub extends Super {
    public int method(String arg) { ... } // error
}
```

---

```
error: method(String) in Sub cannot override method(String) in Super;
attempting to use incompatible return type
found   : int
required: void
```

```
public int method(String arg) { ... }
      ^
```

Both methods have the same signature, but non-substitutable return types. A subtype method's return type  $R_{\text{Sub}}$  is *substitutable* for a supertype method's return type  $R_{\text{Super}}$  if:

- both type are `void`
- both types are the same primitive type
- $R_{\text{Sub}}$  is identical to  $R_{\text{Super}}$
- $R_{\text{Sub}}$  is a subtype of  $R_{\text{Super}}$
- $R_{\text{Sub}}$  is a raw type that is identical to the erasure of  $R_{\text{Super}}$
- $R_{\text{Sub}}$  is a raw type that is a subtype of the erasure of  $R_{\text{Super}}$

The last two situations lead to an unchecked warning unless  $R_{\text{Super}}$  is an unbounded wildcard instantiation.

Example (substitutable return types):

```
class Super {
    public Super      copy() { ... }
    public List<String> getList() { ... }
    public Map        getLinks() { ... }
}
class Sub extends Super {
    public Sub        copy() { ... } // overrides
    public ArrayList<String> getList() { ... } // overrides
    public Map<String,File> getLinks() { ... } // overrides
}
```

In these examples, the subtype method's return type is a subtype of the supertype method's return type. The technical term for this kind of substitutability is *covariant return types*.

Note, that in addition to the regular super-/subtype relationship, an instantiation of a generic type is also considered a subtype of the corresponding raw type. The `getLinks` method in the code snippet above illustrates this kind of substitutable return types: the supertype method returns the raw type `Map` and the subtype method returns the instantiation `Map<String,File>`.

The converse is also permitted, namely a supertype version of a method that returns an instantiation and a subtype version that return the corresponding raw type. But this kind of overriding is not type-safe unless the supertype method's return type is an unbounded wildcard instantiation, as illustrated in the code sample below.

Example (substitutable return types):

```
class Super {
    public Class<?> getType() { ... }
    public Map<String,File> getLinks() { ... }
}
class Sub extends Super {
```

```
public Class getType() { ... } // overrides
public Map getLinks() { ... } // overrides with unchecked warning
}
```

---

```
warning: getLinks() in Sub overrides getLinks() in Super;
return type requires unchecked conversion
found   : java.util.Map
required: java.util.Map<java.lang.String,java.io.File>
    public Map getLinks() { ... }
           ^
```

### *Conflicting throws Clauses*

The supertype and the overriding subtype method must not have conflicting `throws` clauses.

Example (conflicting `throws` clauses):

```
class Super {
    public void method() { ... }
}
class Sub extends Super {
    public void method() throws UserException { ... } // error
}
}
```

---

```
error: method() in Sub cannot override method() in Super;
overridden method does not throw UserException
    public void method() throws UserException { ... }
           ^
```

Both methods have the same signature, but conflicting `throws` clauses. The overriding subtype method may not be declared to throw more checked exceptions than the overridden supertype method. The `throws` clause of the supertype method must at least contain a supertype of each exception type in the `throws` clause of the subtype method.

Example (compatible `throws` clauses):

```
class Super {
    public void method() throws IOException, InterruptedException { ... }
}
class Sub extends Super {
    public void method() throws FileNotFoundException { ... } // overrides
}
}
```

LINK TO THIS

[Technicalities.FAQ813](#)

REFERENCES

[What is method overriding?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

[What are covariant return types?](#)

[What are substitutable return types?](#)

[Can a method of a non-generic subtype override a method of a generic supertype?](#)

[Can a method of a generic subtype override a method of a generic supertype?](#)

[Can a generic method override a generic one?](#)

[Can a non-generic method override a generic one?](#)

[Can a generic method override a non-generic one?](#)

[Why doesn't method overriding work as I expect it?](#)

---

## What are covariant return types?

*Return types of overriding methods where the supertype method's return type is a supertype of the subtype method's return type.*

When a method defined in a subtype overrides a method defined in a supertype then there are certain rules for the return types of those methods. Either the return types are identical, or the subtype method's return type is a raw type that is identical to the supertype method's return type. Java allows one exception from this rule: the subtype method's return type is allowed to be a subtype of the supertype method's return type, or the subtype method's return type is a raw type that is identical to a subtype of the supertype method's return type.

Example (substitutable return types):

```
class Super {
    public Super      copy() { ... }
    public List<String> getList() { ... }
    public Map        getLinks() { ... }
}
class Sub extends Super {
    public Sub        copy() { ... } // overrides
    public ArrayList<String> getList() { ... } // overrides
    public Map<String,File> getLinks() { ... } // overrides
}
```

In these examples, the subtype method's return type is a subtype of the supertype method's return type. Note, that an instantiation of a generic type is considered a subtype of the corresponding raw type. The technical term for this kind of return type substitutability is *covariant return types*.

### LINK TO THIS

[Technicalities.FAQ814](#)

### REFERENCES

[What is method overriding?](#)

[What is method overloading?](#)

[What is the @Override annotation?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

## What are substitutable return types?

*The return type of an overriding subclass method must be substitutable for the superclass method's return type.*

Substitutable return types are required when a method overrides another method. In that situation the subtype method's return type must be substitutable for the supertype method's return type.

A subtype method's return type  $R_{\text{Sub}}$  is considered *substitutable* for a supertype method's return type  $R_{\text{Super}}$  if:

- both type are `void`
- both types are the same primitive type
- $R_{\text{Sub}}$  is identical to  $R_{\text{Super}}$
- $R_{\text{Sub}}$  is a subtype of  $R_{\text{Super}}$
- $R_{\text{Sub}}$  is a raw type that is identical to the erasure of  $R_{\text{Super}}$
- $R_{\text{Sub}}$  is a raw type that is a subtype of the erasure of  $R_{\text{Super}}$

The last two situations lead to an unchecked warning unless  $R_{\text{Super}}$  is an unbounded wildcard instantiation.

Let's take a look at a couple of examples. In the following code snippet the overriding method has the same return type as its superclass method.

Example (plain substitutable return types):

```
class Super {
    public void      setName(String s) { ... }
    public boolean   equals(Object other) { ... }
    public String    toString() { ... }
}
class Sub extends Super {
    public void      setName(String s) { ... } // overrides
    public boolean   equals(Object other) { ... } // overrides
    public String    toString() { ... } // overrides
}
```

A little more exciting are the cases in which the subtype method's return type is a subtype of the supertype method's return type. In this case we talk of *covariant* return types.

Example (covariant substitutable return types):

```
class Super {
    public Super     copy() { ... }
    public           getNames() { ... }
```



```

        List<String>
    public Map        getLinks() { ... }
}
class Sub extends Super {
    public Sub        copy() { ... }    // overrides
    public ArrayList<String> getNames() { ... } // overrides
    public Map<String,File> getLinks() { ... } // overrides
}

```

Worth mentioning is that an instantiation of a generic type is considered a subtype of the corresponding raw type in this context. The `getLinks` method illustrates this case. The supertype method returns the raw type `Map` and the overriding subtype method returns the more specific type `Map<String,File>`.

The converse is permitted, too, but is generally not type-safe. The safe situation is when the subtype's method returns a raw type and the supertype's method returns the wildcard instantiation of the corresponding generic type or of a generic supertype.

Example (raw substitutable return types):

```

class Super {
    public Class<?>    getContentType() { ... }
    public List<?>    asList() { ... }
}
class Sub extends Super {
    public Class        getContentType() { ... } // overrides
    public LinkedList   asList() { ... }        // overrides
}

```

The `getContentType` method in the supertype returns the wildcard instantiation `Class<?>` and the subtype method's return type `Class`, the raw type, is considered a substitutable return type. Note, that the subtype method's return type can additionally be covariant, that is, a raw type that is a subtype of the erasure of the supertype method's return type. This is illustrated by the `asList` method.

When the supertype method's return type is an instantiation different from a wildcard instantiation, then the substitution by the raw type in the subclass method is not type-safe and the compiler issues a warning.

Example (raw substitutable return types):

```

class Super {
    public Class<Super> getThisType() { ... }
    public List<String> asStrings() { ... }
}
class Sub extends Super {
    public Class        getThisType() { ... } // overrides with unchecked warning
    public LinkedList   asStrings() { ... } // overrides with unchecked warning
}

```

The type-safety problem is that the superclass method `getThisType` for instance promises that a type token of type `Class<Super>` is returned, while the subclass method only returns a type token of the raw type `Class`, which may or may not be compatible with `Class<Super>`. The compiler cannot prevent that the subtype method returns an inappropriate type token that is actually of type `Class<Alien>`.

## LINK TO THIS

[Technicalities.FAQ815](#)

## REFERENCES

[What is method overriding?](#)

[What is method overloading?](#)

[What is the @Override annotation?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

[When does a method override its supertype's method?](#)

[What are covariant-return types?](#)

[Can a method of a non-generic subtype override a method of a generic supertype?](#)

[Can a method of a generic subtype override a method of a generic supertype?](#)

[Can a generic method override a generic one?](#)

[Can a non-generic method override a generic one?](#)

[Can a generic method override a non-generic one?](#)

[What is overload resolution?](#)

How does overload resolution work when generic methods are involved?

[Why doesn't method overloading work as I expect it?](#)

[Why doesn't method overriding work as I expect it?](#)

---

## Can a method of a non-generic subtype override a method of a generic supertype?

*Yes.*

This FAQ entry discusses whether and when methods of a regular, *non-generic* subtype override methods inherited from a *generic* supertype. The question comes up because overriding requires that the method signatures are override-equivalent. If the supertype is generic then its method signatures might involve a type parameter. At the same time, the subtype is non-generic and its method signatures will *not* involve type parameters. Hence the signatures are different, which raises the question whether overriding is possible when the supertype is generic and the subtype is not.

The answer is: Yes, it is possible. A non-generic type can extend or implement a concrete instantiation of a generic supertype and then its methods can override methods from the generic supertype. Here is an example:

Example (of a non-generic subtype with overriding methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
class Sub extends Super<Number> {
    ...
    public void set(Number arg) { ... } // overrides
```

```
    public Number get() { ... }    // overrides
}
```

The subtype methods override the corresponding supertype methods. The subtype derives from a certain instantiation of the supertype, namely `Super<Number>` in the example. For this reason, all signatures in the subtype, where `T` is replaced by `Number`, are override-equivalent signatures.

Let us consider some deviations from this straight-forward case. What if we defined the following methods in the subclass?

Example (of a non-generic subtype with redefined supertype methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    ...
}
class Sub extends Super<Number> {
    ...
    public void set(Object arg) { ... } // error: same erasure
    public void set(Long arg) { ... }  // overloads
}
```

The first `set` method in the subclass is rejected because it has the same erasure as its superclass's version of the `set` method, namely `void set(Object)`. The second `set` method in the subclass is not override-equivalent, because its signature is truly different from the supertype method's signature; they have a different argument type. For this reason it overloads the `set` method, instead of overriding it.

Let us consider the `get` methods. What if we defined the following methods in the subclass?

Example (of a non-generic subtype with redefined supertype methods):

```
class Super<T> {
    ...
    public T get() { ... }
    ...
}
class Sub extends Super<Number> {
    ...
    public Object get() { ... } // error: incompatible return type
    public Long get() { ... }  // overrides
}
```

The `Sub.get()` method that returns `Object` is recognized as a potentially overriding method, but with incompatible return type, and is therefore rejected with an error message.

The `Sub.get()` method that returns `Long` has an override-equivalent signature, because its return type is a subtype of the supertype method's return type (covariant return type).

Naturally, the two `get` methods cannot coexist in the same class anyway, because their signatures are the same and they only differ in the return type.

## REFERENCES

[What is method overriding?](#)  
[What is method overloading?](#)  
[What is the @Override annotation?](#)  
[What is a method signature?](#)  
[What is a subsignature?](#)  
[What are override-equivalent signatures?](#)  
[When does a method override its supertype's method?](#)  
[What are covariant-return types?](#)  
[What are substitutable return types?](#)  
[Can a method of a generic subtype override a method of a generic supertype?](#)  
[Why doesn't method overriding work as I expect it?](#)

---

## Can a method of a generic subtype override a method of a generic supertype?

*Yes, but make sure you do not inadvertently overload instead of override.*

This FAQ entry discusses whether and when methods of a *generic* subtype can override methods of a *generic* supertype. Method signatures in the sub- and the supertype may involve type parameters of the respective enclosing class. What is the required relationship between type variables in sub- and supertype methods in order to allow for method overriding?

The answer is: Yes, methods of a generic subtype can override methods of a generic supertype, but it is not always trivial to get it right and it is common that mistakes are made and that overriding is confused with overloading.

Let us start with a simple example:

Example (of a generic subtype with override-equivalent methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
class Sub<S> extends Super<S> {
    ...
    public void set(S arg) { ... } // overrides
    public S get() { ... } // overrides
}
```

The subtype methods override the corresponding supertype methods. The generic subtype derives from a certain instantiation of the supertype, namely `Super<S>` in the example, where `s` is the subtype's type parameter. Since the names of type parameters are irrelevant for the method signatures, the corresponding methods in super- and subtype have identical signatures, namely `set($T1_extends_Object)` and `get()` and identical return types.

The remainder of this FAQ entry discusses slightly more complex overriding situation for further illustration of the principle.

---

Here is another example of successful overriding.

Example (of a generic subtype with override-equivalent methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
class Sub<A,B> extends Super<A> {
    ...
    public void set(A arg) { ... } // overrides
    public A get() { ... } // overrides
}
```

The signatures and the return types are identical - a classical example of overriding. The second type parameter `B` simply does not appear in the overriding methods' signatures and is therefore irrelevant for overriding.

However, if we slightly change the subtype, our attempt to override the inherited methods goes wrong. We declare the subtype methods using the type parameter `B` although we derive from the supertype instantiation on type parameter `A`.

Example (of a generic subtype without override-equivalent methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
class Sub<A,B> extends Super<A> {
    ...
    public void set(B arg) { ... } // error: same erasure
    public B get() { ... } // error: incompatible return type
}
```

---

error: name clash: set(B) in Sub<A,B> and set(T) in Super<A> have the same erasure,  
yet neither overrides the other

```
class Sub<A,B> extends Super<A> {
    ^
error: get() in Sub cannot override get() in Super;
attempting to use incompatible return type
found   : B
required: A
    public B get() { ... }
```

The `set` methods have signatures that are no longer override-equivalent, namely `set($T1_extends_Object)` in the supertype and `set($T2_extends_Object)` in the subtype. This is because the compiler distinguishes between different type parameters. At the same time, both signatures have the identical erasures, namely `set(Object)`, and the compiler rejects the subtype method `set` with an error message because no two methods in the same type may have identical signatures.

The `get` methods have identical signatures, but incompatible return types. Again, because the compiler distinguishes between different type parameters. For this reason the subtype method `get` is rejected with an error message.

Let us modify the subtype a second type and see what happens now. The modification is that the second type parameter is bounded by the first type parameter.

Example (of a generic subtype without override-equivalent methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
class Sub<A,B extends A> extends Super<A> {
    ...
    public void set(B arg) { ... } // error: same erasure
    public B get() { ... } // overrides
}
```

---

error: name clash: set(B) in Sub<A,B> and set(T) in Super<A> have the same erasure,  
yet neither overrides the other

```
class Sub<A,B extends A> extends Super<A> {
    ^
```

The `set` methods still have signatures that are not override-equivalent, namely `set($T1_extends_Object)` in the supertype and `set($T2_extends_$T1)` in the subtype. And both signatures still have the same erasure, namely `set(Object)`. Again, the compiler rejects the subtype method `set` with an error message. The `get` methods have identical signatures and this time compatible return types, because the type parameter `B` is a subtype of the type parameter `A`. For this reason the subtype method `get` overrides the supertype method.

---

Let us consider a situation where the subtype's type parameter has different bounds than the supertype's type parameter.

Example (of a generic subtype with override-equivalent methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
```

```

class Sub<N extends Number> extends Super<N> {
    ...
    public void set(N arg) { ... } // overrides
    public N get() { ... } // overrides
}

```

The signatures and the return types of the methods are identical in super and subclass, namely `set($T1_extends_Number)` with return type `void` and `get()` with return type `$T1_extends_Number`.

Let us change the subtype declaration slightly; we declare subclass methods so that they use the bound in the method signatures instead of the type parameter .

Example (of a generic subtype with different yet override-equivalent methods):

```

class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
class Sub<N extends Number> extends Super<N> {
    ...
    public void set(Number arg) { ... } // overrides
    public Number get() { ... } // error: incompatible return type
}

```

The `set` methods now have different signatures, namely `set($T1_extends_Number)` in the supertype and `set(Number)` in the subtype. Normally, this would be overloading, but in this case with type parameters involved the signatures are considered override-equivalent. This is because the supertype's `set` method takes an argument of type `$T1_extends_Number`, which is a placeholder for a subtype of `Number`. The subtype's `set` method takes an argument of type `Number`, which means that any argument passed to the `set` method via a supertype reference is acceptable to the subtype version of the method.

Consider the example of a concrete parameterization of our super- and subtype in order to understand why it makes sense that the subtype version of the `set` method is an overriding rather than an overloading version of the supertype method, despite of the different signature. For illustration we use `Sub<Integer>` and its supertype `Super<Integer>`. The two methods involved in overriding are `Super<Integer>.set(Integer)` and `Sub<Integer>.set(Number)`. When a supertype reference refers to a subtype object and we invoke the `set` method through the supertype reference, then we see `super<Integer>.set(Integer)`, but actually invoke the overriding `sub<Integer>.set(Number)`. Is it type-safe?

Example (of calling the overridden `set` method):

```

Super<Integer> ref = new Sub<Integer>();
ref.set(10); // calls Sub<Integer>.set(Number)

```

All we can pass to `Super<Integer>.set(Integer)` are objects of type `Integer`; they are handed over to `Sub<Integer>.set(Number)`, which happily accepts the `Integer` objects because they are subtypes of `Number`. And the same is true for *all* parameterizations of type `Sub`. because the declared argument type of the supertype method is always a subtype of the declared argument type of the subtype method. Hence it is type-safe that the subtype version of the `set` method is considered an overriding version of the supertype's `set` method.

The opposite is true for the `get` methods. They have identical signatures, but incompatible return types, namely `$T1_extends_Number` in the supertype and `Number` in the subtype. If we invoked the subtype's `set` method via a supertype reference then we would expect a return value of type `$T1_extends_Number`, which is a

placeholder for a subtype of `Number`, while in fact the subtype method would return a `Number` reference, which can refer to any arbitrary subtype of `Number`, not necessarily the one we are prepared to receive. Hence considering the subtype version of the `get` method an overriding version of the supertype's `get` method would not be type-safe and is therefore rejected as an error.

In contrast, the following fairly similar example leads to an overloading situation.

Example (of a generic subtype without override-equivalent methods):

```
class Super<T> {
    ...
    public void set(T arg) { ... }
    public T get() { ... }
}
class Sub<N extends Number> extends Super<Number> {
    ...
    public void set(N arg) { ... } // overloads
    public N get() { ... } // overrides
}
```

The `set` methods again have different yet similar signatures. This time it is the other way round: we have `set(Number)` in the supertype and `set($T1_extends_Number)` in the subtype. This is overloading rather than overriding, because the supertype method has a declared argument type that is a supertype of the subtype method's declared argument type. Overriding would not be type-safe in this situation.

To understand it let us use the same concrete parameterization as above, namely `Sub<Integer>` and its supertype `Super<Number>`. The two methods in question are `Super<Number>.set(Number)` and `Sub<Integer>.set(Integer)`. When a supertype reference refers to a subtype object and we invoke the `set` method through the supertype reference, then we see `super<Number>.set(Number)`, but would actually invoke `sub<Integer>.set(Integer)`, if this were overriding. This is not type-safe because we could pass an object of type `Long` to `Super<Number>.set(Number)`, but the subtype method `Sub<Integer>.set(Integer)` cannot take it.

The `get` methods are not problematic in this example. They have the identical signatures and covariant return type, namely `Number` in the supertype and `$T1_extends_Number` in the subtype. Hence, we have overriding for the `get` methods.

Overloading leads to confusing and/or ambiguous method invocations. Consider for example the instantiation `Sub<Number>`. In this instantiation we have `Super<Number>.set(Number)` in the supertype and overloaded version `Sub<Number>.set(Number)` in the subtype, both methods have the same signature.

Example (of calling the overloaded `set` method):

```
Integer integer = 10;
Number number = integer;

Sub<Number> ref = new Sub<Number>();
ref.set(integer); // error: ambiguous
ref.set(number); // error: ambiguous
```

---

```
error: reference to set is ambiguous,
both method set(T) in Super<Number> and method set(N) in Sub<Number> match
ref.set(integer);
```



```
    ^
error: reference to set is ambiguous,
both method set(T) in Super<Number> and method set(N) in Sub<Number> match
    ref.set(number);
    ^
```

The point to take notice of is that the `set` method in `Sub<Number>` does not override the `set` method inherited from `Super<Number>` although in this particular instantiation both methods have the same argument type.

This is because the decision whether a subtype method overrides or overloads a supertype method is made by the compiler when it compiles the generic subtype. At that point in time there is no knowledge regarding the concrete type by which the type parameter `N` might later be replaced. Based on the declaration of the generic subtype the two `set` methods have different signatures, namely `set(Number)` in the supertype and `set($T1_extends_Number)` in the generic subtype. In a certain instantiation of the subtype, namely in `Sub<Number>`, the type parameter `N` (or `$T1_extends_Number`) might be replaced by the concrete type `Number`. As a result both `set` methods of `Sub<Number>` suddenly have the same arguments type. But that does not change the fact that the two methods still have different signatures and therefore overload rather than override each other.

#### LINK TO THIS

[Technicalities.FAQ821](#)

#### REFERENCES

[What is method overriding?](#)

[What is method overloading?](#)

[What is the @Override annotation?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

[When does a method override its supertype's method?](#)

[What are covariant-return types?](#)

[What are substitutable return types?](#)

[Can a method of a non-generic subtype override a method of a generic supertype?](#)

[Why doesn't method overriding work as I expect it?](#)

---

## Can a generic method override a generic one?

*Yes.*

This FAQ entry discusses the relationship between generic methods in super- and subtypes with respect to overriding and overloading.

Say, we have a supertype with generic methods and we intend to override the generic methods in a subtype. Which subtype methods are considered overriding versions of the the generic supertype methods? FAQ entry [FAQ823](#) discusses which non-generic methods in a subtype override a generic method in a supertype. In this FAQ entry we explore which *generic* subtype methods override generic methods in a supertype.

Example (of generic subtype methods overriding generic supertype methods):

```
class Super {
```

```

    public <T> void set(T arg) { ... }
    public <T> T get() { ... }
}
class Sub extends Super {
    public <S> void set(S arg) { ... } // overrides
    public <S> S get() { ... } // overrides
}

```

In this example the subtype methods are generic methods and have the same signatures as the supertype methods, namely `<T>void set(T arg)` and `<T>T get()`. Note, that the names of the type parameters of the generic methods differ in the super- and the subtype. This, however, does not affect the signatures because the names of type parameters are irrelevant for the signature.

If the methods have type parameters *with different bounds*, then they do not override, because the methods have signatures that are not override-equivalent. Remember, the type parameter bounds are part of a generic method's signature.

Example (of generic subtype methods overloading generic supertype methods; not recommended):

```

class Super {
    public <T> void set(T arg) { ... }
    public <T> T get() { ... }
}
class Sub extends Super {
    public <S extends Number> void set(S arg) { ... } // overloads
    public <S extends Number> S get() { ... } // overloads
}

```

In this example the subtype methods have the signatures `<S extends Number>void set(S arg)` and `<S extends Number>S get()`, while the supertype methods have the signatures `<T>void set(T arg)` and `<T>T get()`. The signatures are clearly different and there is no chance that the subtype methods can ever override the supertype methods. The resulting overload situation makes for "interesting" effects when the methods are invoked; most of the time you will see the compiler complaining about ambiguous method calls. For this reason complex overloading situations like the one above are generally best avoided.

Remember that the order of the type parameter bounds does not affect the signature of a generic method. For this reason, the methods in the following example do override, although they have different type parameter bounds.

Example (of generic subtype methods overriding generic supertype methods):

```

class Super {
    public <T extends Comparable<T> & Serializable> void set(T arg) { ... }
    public <T extends Comparable<T> & Serializable> T get() { ... }
}
class Sub extends Super {
    public <S extends Serializable & Comparable<S>> void set(S arg) { ... } // overrides
    public <S extends Serializable & Comparable<S>> S get() { ... } // overrides
}

```

The respective methods have identical signatures because the order of the type parameter bounds is irrelevant.

**LINK TO THIS**

[Technicalities.FAQ822](#)

**REFERENCES**

[What is method overriding?](#)  
[What is method overloading?](#)  
[What is the @Override annotation?](#)  
[What is a method signature?](#)  
[What is a subsignature?](#)  
[What is overload resolution?](#)  
[Can a generic method override a non-generic one?](#)  
[What is overload resolution?](#)  
[Why doesn't method overloading work as I expect it?](#)

---

## Can a non-generic method override a generic one?

*Yes.*

This FAQ entry discusses the relationship between generic and non-generic methods with respect to overloading.

Say, we have a supertype with generic methods and we intend to override the generic methods in a subtype. Which subtype methods are considered overriding versions of the the generic supertype methods? Before we discuss non-generic subtype methods that override generic supertype methods, let us consider the more obvious case of a subtype with generic methods that override generic methods from the supertype.

Example (of generic subtype methods overriding generic supertype methods):

```
class Super {
    public <T> void set(T arg) { ... }
    public <T> T get() { ... }
}
class Sub extends Super {
    public <S> void set(S arg) { ... } // overrides
    public <S> S get() { ... } // overrides
}
```

In this example the subtype methods are generic methods and have the same signatures as the supertype methods, namely `<S>void set(S arg)` and `<S>S get()`. Note, that the names of the type parameters of the generic methods differ in the super- and the subtype. This, however, does not affect the signatures because the names of type parameters are irrelevant for the signature.

Now, let us explore an example where non-generic subtype methods override generic supertype methods. Non-generic subtype methods are considered overriding versions of the generic supertype methods if the signatures' erasures are identical.

Example (of non-generic subtype methods overriding generic supertype methods):

```

class Super {
    public <T> void set(T arg) { ... }
    public <T> T get() { ... }
}
class Sub extends Super {
    public void set(Object arg) { ... } // overrides
    public Object get() { ... } // overrides with unchecked warning
}

```

---

```

warning: get() in Sub overrides <T>get() in Super;
return type requires unchecked conversion
found   : Object
required: T
    public Object get() {
           ^

```

Here the subtype methods have signatures, namely `set(Object)` and `get()`, that are identical to the erasures of the supertype methods. These type-erased signatures are considered override-equivalent.

There is one blemish in the case of the `get` method: we receive an unchecked warning because the return types are not really compatible. The return type of the subtype method `get` is `Object`, the return type of the supertype method `get` is an unbounded type parameter. The subtype method's return type is neither identical to the supertype method's return type nor is it a subtype thereof; in both situations the compiler would happily accept the return types as compatible. Instead, the subtype method's return type `Object` is convertible to the supertype method's return type by means of an unchecked conversion. An unchecked warning indicates that a type check is necessary that neither the compiler nor the virtual machine can perform. In other words, the unchecked operation is not type-safe. In case of the convertible return types someone would have to make sure that the subtype method's return value is type-compatible to the supertype method's return type, but nobody except the programmer can ensure this.

---

The main purpose of allowing that methods with erased signatures override methods with generic signatures is backward compatibility. Imagine both classes had initially been non-generic legacy classes. In this initial situation both classes had had methods that take and return `Object` reference.

Example (of legacy classes):

```

class Super {
    public void set(Object arg) { ... }
    public Object get() { ... }
}
class Sub extends Super {
    public void set(Object arg) { ... } // overrides
    public Object get() { ... } // overrides
}

```

Later we decide to re-engineer the superclass and turn it into a class with generic methods.

Example (same as before, but after generification of the supertype):

```

class Super {
    public <T> void set(T arg) { ... }
    public <T> T get() { ... }
}
class Sub extends Super {
    public void set(Object arg) { ... } // overrides ???
    public Object get() { ... } // overrides ???
}

```

Now the question is: do the subtype methods still override the generified supertype methods? The answer is: yes. This is because subtype methods whose signatures are identical to the erasures of the supertype methods overload the supertype methods. That is, we can generify the supertype without affecting the legacy subtypes.

---

At last, let us discuss a counter-example where the non-generic subtype methods do *not* override the generic supertype methods.

Example (of non-generic subtype method overloading, instead of overriding, generic supertype methods; not recommended):

```

class Super {
    public <T> void set(T arg) { ... }
    public <T> T get() { ... }
}
class Sub extends Super {
    public void set(String arg) { ... } // overloads
    public String get() { ... } // overrides with unchecked warning
}

```

---

```

warning: get() in Sub overrides <T>get() in Super;
return type requires unchecked conversion
found   : String
required: T
    public String get() {
           ^

```

Here the subtype method `set` has the signature `set(String)` and this signature is not override-equivalent to the supertype method's signature `<T>set(T)`. This is because the subtype method's signature is different from both the supertype method's signature and the erasure thereof. As the subtype method `set` does not override, it overloads.

The subtype method `get` on the other hand overrides the supertype method `get`. This is because the subtype method's signature is identical to the erasure of the supertype method's signature. The return types are not compatible, but only convertible by unchecked conversion; the subtype method's return type `String` is a subtype of the erasure of the supertype method's return type. That is, the subtype method's return type is acceptable as a result of the combination of unchecked conversion and covariance.

The entire subclass is an example of poor design. The combination of overloading and overriding of corresponding methods is confusing at best.

## REFERENCES

[What is an unchecked warning?](#)  
[What is method overriding?](#)  
[What is method overloading?](#)  
[What is the @Override annotation?](#)  
[What is a method signature?](#)  
[What is overload resolution?](#)  
[Can a generic method override a generic one?](#)  
[Can a generic method override a non-generic one?](#)  
[Why doesn't method overloading work as I expect it?](#)

---

## Can a generic method override a non-generic one?

*No.*

This FAQ entry discusses whether a generic method in a subtype can override a non-generic method inherited from a supertype. We have seen in [FAQ823](#) that the converse is possible, namely that a regular, non-generic method in a subtype can override a generic method inherited from a supertype. It raises the question whether the same is true if the roles are flipped and a generic method in a subtype attempts to override a non-generic method from a supertype.

The answer is: No, generic methods in a subtype *cannot* override non-generic methods in a supertype. Generic subtype methods can only overload non-generic supertype methods.

Let us consider a superclass with non-generic methods and let us explore why generic methods in subtypes do not override the non-generic supertype methods.

Example (of a supertype with non-generic methods and a subtype with generic methods):

```
class Super {
    public void set(Object arg) { ... }
    public Object get() { ... }
}
class Sub extends Super {
    public <S> void set(S arg) { ... } // error
    public <S> S get() { ... } // error
}
```

---

```
error: name clash: <S>set(S) in Sub and set(Object) in Super have the same erasure,
yet neither overrides the other
```

```
class Sub extends Super {
^
```

```
error: name clash: <S>get() in Sub and get() in Super have the same erasure,
yet neither overrides the other
```

```
class Sub extends Super {
^
```

Although the erasures of the subtype methods have signatures that are identical to the supertype methods' signatures the respective signatures are not override-equivalent. The override-equivalence only holds when the subtype method's signature is identical to the erasure of the supertype method's signature, but not vice versa. Consequently, the subtype methods overload, instead of override, the supertype methods. Moreover, the fact that the erasures of corresponding methods in super- and subtype are identical is in conflict with the rule that a class (the subclass in this case) must not have methods with identical erasures. As a result the compiler issues error messages.

Let us consider a slightly different subtype, where again overloading is mixed up with overriding.

Example (of a supertype with non-generic methods and a subtype with generic methods; not recommended):

```
class Super {
    public void set(Object arg) { ... }
    public Object get() { ... }
}
class Sub extends Super {
    public <S extends Number> void set(S arg) { ... } // overloads
    public <S extends Number> S get() { ... } // overloads
}
```

This subclass compiles, but it does not show an example of overriding, but instead of overloading. This is because the supertype methods have the signatures `set(Object)` and `get()`, while the subtype methods have the signatures `<S extends Number>set(S)` and `<S extends Number>get()`. In this example, not even the erasures of the signatures are identical, so that there is not the slightest chance that the subtype methods could override the supertype methods. Instead, the subtype methods overload the supertype methods. What happens when we invoke the overloaded methods?

Example (of invoking the overloaded methods):

```
Sub sub = new Sub();
Super sup = sub;

sup.set("abc"); // calls Super.set(Object)
sup.set(0); // calls Super.set(Object)
Object o = sup.get(); // calls Super.get()

sub.set("abc"); // calls Super.set(Object)
sub.set(0); // calls Sub.<Integer>set(Integer)
Integer i = sub.get(); // error: ambiguous
```

---

```
error: reference to get is ambiguous,
both method get() in Super and method <S>get() in Sub match
Integer i = sub.get();
                ^
```

The invocation through a supertype reference leads to calls to the supertype versions of the overloaded method, regardless of the dynamic type of the referenced object. This is the behavior that is expected of overloaded methods.

The invocation through a subtype reference leads to a call to the subtype versions of the overloaded `set` method, provided the argument is of a subtype of `Number`, and to invocation of the supertype version of the method otherwise. Invocation of the overloaded `get` method through a subtype reference leads to an error message.

Let us see how that happens.

When the `set` method is invoked with a `String` argument, then the compiler finds only one viable method, namely the non-generic `set(Object)` method from the supertype. The generic subtype method is not a candidate for this invocation because `String` is no subtype of `Number` and hence the possible type argument `String` is not within bounds. Consequently, the supertype method is called.

When the `set` method is invoked with a `Integer` argument, then the compiler finds two candidates for the invocation of the `set` method: the non-generic `set(Object)` method from the supertype and the parametrization `<Integer>set(Integer)` inferred from the generic subtype. Note, the compiler performs type argument inference before it looks for the best match among the candidate methods. Since the parametrization is the better match, the subtype method is invoked.

For the invocation of the `get` method, the compiler finds two candidates: `get()` from the supertype and `<T1_extends_Number>get()` from the subtype. Neither of the two signatures is more specific and thus the compiler reports an error. Note, the compiler does not consider the return type when it resolves the overloading and in particular does not perform type inference based on a generic method's return type. In our example it means that the compiler does *not* infer from the calling context that the instantiation `<Integer>get()` would be a viable candidate method and then picks it as the better match. Only the signature, and never the return type, of a method are relevant for overload resolution.

The overload resolution is slightly different, when explicit instantiations are invoked.

Example (of invoking the overloaded methods with explicitly specified type arguments):

```
Sub sub = new Sub();
Super sup = sub;

Integer n = sub.<Integer>get(); // calls Sub.<Integer>set(Integer)
sub.<Integer>set(0);           // calls Sub.<Integer>get()
```

In this situation the candidate set contains only the respective generic subtype method, because the supertype methods are not generic and for this reason cannot be invoked with explicitly specified type arguments.

As the examples demonstrate, there is no way that a generic subtype method can override a non-generic supertype method. Instead, generic subtype methods overload non-generic supertype methods. Such overloading should generally be avoided because it is confusing and hard to understand.

**LINK TO THIS**            [Technicalities.FAQ824](#)

**REFERENCES**

- [What is type argument inference?](#)
- [What is method overriding?](#)
- [What is method overloading?](#)
- [What is the @Override annotation?](#)
- [What is a method signature?](#)
- [What is overload resolution?](#)
- [Can a generic method override a generic one?](#)
- [Can a non-generic method override a generic one?](#)
- [Can a generic method override a non-generic one?](#)
- [What is overload resolution?](#)
- [Why doesn't method overloading work as I expect it?](#)

---



## What is overload resolution?

*The process of determining the best match from a set of overloaded methods.*

When a type has several overloaded methods, then the compiler must decide which method to call when it finds a method invocation expression. The process of picking the best match from the set of candidate methods is called *overload-resolution*.

Example (of method overloading):

```
class Super {
    public void method(String arg) { ... }
    public void method(StringBuffer arg) { ... } // overloads
}
class Sub extends Super {
    public void method(StringBuilder arg) { ... } // overloads
}
```

In this example, all three versions of the method overload each other because they have the same name but different argument types. When the method is invoked the compiler picks the best match from the three available candidates. The compiler takes a look at the type of the argument provided to the method call and tries to find a method in the candidate set that accepts this particular type of argument.

Example (of overload resolution):

```
Sub ref = new Sub();

ref.method("ABC"); // calls Super.method(String)
ref.method(new StringBuilder("ABC")); // calls Sub.method(StringBuilder)
```

In the first method call a `String` is provided and consequently the compiler invokes the method that takes a `String` argument. When a `StringBuilder` is provided the method with the `StringBuilder` argument type is the best match. In both cases there is an exact match. This is not always so.

When there is no exact match the compiler considers various conversions, among them the conversion from a subtype to a supertype (called reference-widening), conversions among primitive types (e.g. `int` to `long`), autoboxing and unboxing (e.g. `int` to `Integer`), and the conversion from a parameterized type to the raw type (called unchecked conversion). By and large, overload resolution is a complex process and can lead to surprising results, in the sense that the compiler picks and invokes a method that nobody had expected would be called.

This sounds harmless, but can be a serious problem. For instance, when an overloading method is added to an existing class, then this additional candidate can change the result of overload resolution and thereby inadvertently change the effect of method invocations in an unrelated part of the program. This is usually undesired and the resulting bugs are difficult to track down, because symptom and source of the problem are mostly unrelated.

As a general rule, overloading should be used sparingly and judiciously.

### LINK TO THIS

[Technicalities.FAQ830](#)

### REFERENCES

[What is method overloading?](#)

[What is the @Override annotation?](#)

[What is a method signature?](#)

[What is a subsignature?](#)

[What are override-equivalent signatures?](#)

[Why doesn't method overriding work as I expect it?](#)

---

[CONTENT](#) [PREVIOUS](#) [NEXT](#) [INDEX](#)

# More Information on Java Generics

© Copyright 2004-2014 by Angelika Langer. All Rights Reserved.

[Where can I find a generics tutorial?](#)

[Where can I find a specification of the Java generics language features?](#)

[Which books cover Java generics?](#)

[What webpages are devoted to Java generics?](#)

---

## Where can I find a generics tutorial?

**TUTORIAL:** <http://docs.oracle.com/javase/tutorial/extra/generics/index.html>  
and <http://docs.oracle.com/javase/tutorial/java/generics/>

A tutorial for Java Generics written by Gilad Bracha was published in February 2004 and later extended.

**LINK TO THIS** [Information.FAQ002](#)

**REFERENCES**

---

## Where can I find a specification of the Java generics language features?

**SPECIFICATION:** <http://jcp.org/aboutJava/communityprocess/review/jsr014/>

The public review draft of the Java Generics specification has been put together by the [JSR014 specification group](#). Sadly, it's outdated (August 2001) and no revision has been published here. Later versions of the specification were available as part of the prototype releases of the compiler. However, there is no final release of the specification.

**SPECIFICATION:** <http://java.sun.com/docs/books/jls/>

The final specification is part of the 3<sup>rd</sup> edition of the Java Language Specification (JLS 3), which has been published in April 2005 and is available at the URL above.

**LINK TO THIS** [Information.FAQ003](#)

**REFERENCES**

---

## Which books cover Java generics?

*None that I know of. A couple of books give an introduction and overview, but non covers Java generics in depth.*

There is a book that covers generics and the collections framework.

### **Generics and Collections in Java 5**

Maurice Naftalin and Philip Wadler  
O'Reilly & Associates, November 2006

Some books are devoted to all the new features in Java 5.0, including Java generics.

## Java 1.5 Tiger

Brett McLaughlin and David Flanagan  
O'Reilly & Associates, June 2004

Some Java textbooks have been updated to include Java generics. Examples are:

### **Core Java, Volume I - Fundamentals, 7th Edition**

Cay Horstmann and Gary Cornell  
Prentice Hall, August 2004

### **Thinking in Java, 4th Edition**

Bruce Eckel  
Prentice Hall PTR, February 2006

LINK TO THIS [Information.FAQ004](#)

REFERENCES

---

## What webpages are devoted to Java generics?

LINKS: [http://en.wikipedia.org/wiki/Generics\\_in\\_Java](http://en.wikipedia.org/wiki/Generics_in_Java)

The Wikipedia entry provides some rough explanations and a couple of links to other websites related to Java Generics.

LINK TO THIS [Information.FAQ005](#)

REFERENCES

---

[CONTENT](#) [PREVIOUS](#) [NEXT](#) [INDEX](#)

# Glossary

© Copyright 2004-2014 by Angelika Langer. All Rights Reserved.

In this glossary links of the form [XXX.FAQnnn](#) refer to an entry in this [FAQ](#), link of the form [JLS n.n.n](#) refer to a paragraph in [JLS3](#), the Java Language Specification, 3rd Edition, and links of the form [J2SE API package.class.method](#) refer to an entry in the Java platform libraries' [API](#) documentation.

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

---

## A

---

## B

### bounded type parameter

A [type parameter](#) with one or more bounds. The [type parameter bounds](#) restrict the set of types that can be used as [type arguments](#) and give access to the methods defined by the bounds.

Example:

```
public class TreeMap<Key extends Comparable<Key>,Data>{
    private static class Entry<K,V> { ... }
    ...
    private Entry<Key,Data> getEntry(Key key) {
        ...key.compareTo(p.key) ...
    }
}
```

see also: [TypeParameters.FAQ002](#), JLS 4.4

### bounded wildcard

A [wildcard](#) with either an upper or a lower bound.

Example:

```
public class Collections {
    public static <T> void copy
        (List<? super T> dest, List<? extends T> src) { // bounded wildcard
        parameterized types
            for (int i=0; i<src.size(); i++)
                dest.set(i,src.get(i));
        }
}
```

see also: [TypeArguments.FAQ103](#), JLS 4.5.1

### bridge method

A synthetic method that the compiler generates in the course of [type erasure](#). It is sometimes needed when a type extends or implements a parameterized class or interface.

Example (before type erasure):

```
interface Comparable<A> {
    public int compareTo(A that);
}
final class NumericValue implements Comparable<NumericValue> {
    ...
}
```

```
    public int compareTo(NumericValue that) { return this.value - that.value; }  
}
```

Example (after type erasure):

```
interface Comparable {  
    public int compareTo(Object that);  
}  
final class NumericValue implements Comparable {  
    ...  
    public int compareTo(NumericValue that) { return this.value - that.value;  
    }  
    public int compareTo(Object that) { return  
this.compareTo((NumericValue)that); } // bridge method  
}
```

see also: [TechnicalDetails.FAQ102](#)

---

## C

### checked collection

A view to a regular collection that performs a runtime type check each time an element is inserted.

Example:

```
List<String> stringList  
    = Collections.checkedList(new ArrayList<String>(),String.class);
```

see also: [ProgrammingIdioms.FAQ004](#), [J2SE API java.util.Collection.checkedCollection](#)

### class literal

A literal of type `class`. A class literal is an expression consisting of the name of a class, interface, array, or primitive type, or the pseudo-type `void`, followed by the suffix `".class"`.

Example:

```
if (s.getClass() == String.class) ...
```

see also: JLS 15.8.2

### code sharing

A translation technique for generics where the compiler generates code for only one representation of a generic type or method and maps all the instantiations of the generic type or method to the unique representation, performing type checks and type conversions where needed.

see also: [TechnicalDetails.FAQ100](#)

### code specialization

A translation technique for generics where the compiler generates code for only one representation of a generic type or method and maps all the instantiations of the generic type or method to the unique representation, performing type checks and type conversions where needed.

see also: [TechnicalDetails.FAQ100](#)

### concrete instantiation

see: [concrete parameterized type](#)

### concrete parameterized type

An instantiation of a [generic type](#) where all [type arguments](#) are concrete types rather than [wildcards](#).

Examples:

```
List<String>  
Map<String,Date>
```

but not:

```
List<? extends Number>  
Map<String,?>
```

Synonyms: concrete instantiation

see also: [ParameterizedTypes.FAQ101](#)

---

## D

### diamond operator

The empty angle brackets <> that trigger type inference in new-expressions.

Example:

```
List<String> list = new ArrayList<>();
```

see also: [TechnicalDetails.FAQ400A](#)

---

## E

### enum type

A reference type that defines a finite number of enum constants, where each enum constant defines an instance of the enum type.

Example:

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER }
```

see also: JLS 8.9

### explicit type argument specification

Providing a type argument list when a generic method is invoked rather than relying on type inference.

Example:

```
public class Utilities {  
    public static <T extends Comparable> T max(T arg1, T arg2) { ... }  
}  
public class Test {  
    public static void main(String[] args) {  
        System.out.println(Utilities.<String>max("abc", "xyz"));  
    }  
}
```

Similarly for the type argument of a generic type; it can be explicitly specified, which is the norm, or deduced via type inference.

```
List<String> list1 = new ArrayList<String>();
```

see also: [TechnicalDetails.FAQ402](#), JLS15.12

---

## F

---

## G

### generic type

A class or interface with one or more type parameters.

Examples:

```
class List<E> { ... }
interface Comparable<T> { ... }
interface Map<K,V> { ... }
```

Synonyms: parameterized type

Most authors use the term parameterized type as a synonym for an instantiation or an invocation of a generic type. Some (few) authors use the term parameterized type as a synonym for generic type.

see also: [GenericTypes.FAQ001](#), JLS 8.1.2

### generic method

A method with one or more type parameters.

Examples:

```
<T> void set(T arg) { ... }
<T> T[] toArray(T[] array) { ... }
```

see also: [GenericMethods.FAQ001](#), JLS 8.4.4

### generification

Converting a legacy (non-generic) class or interface to be a generic one.

Examples: Classes (e.g., `List`) of the Collections framework in Java 1.4 have been generified (e.g., `List<T>`) in Java 5.0.

see also: [generic type](#)

---

## H

### heap pollution

A situation where a variable of a parameterized type refers to an object that is not of that parameterized type.

Examples:

```
List ln = new ArrayList<Number>();
List<String> ls =ln; // unchecked warning + heap pollution

List<? extends Number> ln = new ArrayList<Long>();
List<Short> ls = (List<Short>) ln; // unchecked warning + heap pollution
```

see also: [TechnicalDetails.FAQ050](#), JLS 4.12.2.1

---



# I

## instantiated method

A method created from a generic method by providing an actual type argument per formal type parameter. It is the result of either type argument inference or explicit type argument specification.

Example:

```
List<Number> list = new ArrayList<Number>();  
Object[] numbers2 = list.toArray(new Long[0]); // calls method  
<Long>toArray  
Object[] numbers1 = list.<Number>toArray(new Long[0]); // calls method  
<Number>toArray
```

In line 1 an instantiated method with the signature `Long[] toArray(Long[])` is created from the generic method `<T> T[] toArray(T[] a)` by replacing the formal type parameter `T` with the inferred actual type argument `Long`.

In line 2 an instantiated method with the signature `Number[] toArray(Number[])` is created from the generic method `<T> T[] toArray(T[] a)` by replacing the formal type parameter `T` with the explicitly-specified actual type argument `Number`.

see also: [TechnicalDetails.FAQ401](#), [TechnicalDetails.FAQ402](#)

## instantiated type

A type created from a generic type by providing an actual type argument per formal type parameter.

Examples:

```
List<String> (created from the generic type List<E>)  
Map<String,Date> (created from the generic type Map<K,V>)
```

Synonyms: instantiation, parameterized type, invocation

Most authors use the terms instantiation (short for: instantiated type) or parameterized type. The JLS and other documents by Sun Microsystems use the terms invocation and parameterized type.

see also: [GenericTypes.FAQ001](#), JLS 8.1.2

## invocation

A type created from a generic type by providing an actual type argument per formal type parameter. Used to denote both the process of creating a type from a generic type or method as well as the result of that process.

Examples:

```
List<String> (created from the generic type List<E>)  
Map<String,Date> (created from the generic type Map<K,V>)
```

Synonyms: parameterized type, instantiation, instantiated type

The JLS and other documents by Sun Microsystems use the term invocation for the process and result of creating a type from a generic type by replacing formal type parameters by actual type arguments. The JLS uses the term parameterized type as a synonym for the resulting type. Most authors use the term instantiation. This FAQ prefers the terms parameterized type and instantiation over invocation.

Other Meanings:

The term invocation is also used in conjunction with methods and means *calling a method*.

see also: [GenericTypes.FAQ001](#), JLS 8.1.2

## instantiation

A type created from a generic type or a generic method by providing an actual type argument per formal type parameter. Used to denote both the process of creating a type or method from a generic type or method as well as the result of that process.

Examples:

```
List<String>                (created from the generic type List<E>)  
Map<String,Date>          (created from the generic type Map<K,V>)  
Collections.<Number>toArray (created from the generic method <T> T[])  
Collections.toArray(T[])
```

Synonyms: parameterized type, invocation

Most authors use the term instantiation. The JLS and other documents by Sun Microsystems use the term invocation for the process and result of creating a type from a generic type by replacing formal type parameters by actual type arguments. The JLS uses the term parameterized type as a synonym for the resulting type. The JLS has no term for the result of creating a method from a generic method.

Other Meanings:

The term instantiation is also used to denote the *creation of an instance* (or object) of a type.

see also: [GenericTypes.FAQ001](#), JLS 8.1.2

---

## J

---

## K

---

## L

### lower bound

see: [lower wildcard bound](#)

### lower wildcard bound

A reference type that is used to further describe a wildcard; it denotes the family of types that are supertypes of the lower bound.

Example:

```
Comparable<? super Long>
```

see also: [TypeArguments.FAQ201](#), JLS 4.5.1

---

## M

---

## N

---

## O

---

# P

## parameterized type

A type created from a generic type by providing an actual type argument per formal type parameter.

Examples:

```
List<String>      (created from the generic type List<E>)  
Map<String,Date> (created from the generic type Map<K,V>)
```

Synonyms: invocation, instantiation, instantiated type

One of the most blurred terms in the area of generics. The JLS uses the term parameterized type for the result of replacing formal type parameters by actual type arguments; it is a synonym for an invocation of a generic type.

Other authors use the term instantiation or instantiated type instead. Few authors equate parameterized type with generic type.

This FAQ favors the terms parameterized type and instantiation of a generic type.

see also: [GenericTypes.FAQ001](#), JLS 8.1.2

---

# Q

---

# R

## raw type

A (non-generic) type created from a generic type by omitting the type parameters ("de-generification").

Examples:

```
List (created from the generic type List<E>)  
Map  (created from the generic type Map<K,V>)
```

see also: [GenericTypes.FAQ201](#), JLS 4.8

## reification

Representing type parameters and arguments of generic types and methods at runtime. Reification is the opposite of type erasure.

see also: [TechnicalDetails.FAQ101A](#)

## reifiable type

A type whose type information is fully available at runtime, that is, a type that does not lose information in the course of type erasure.

Examples:

```
int  
Number  
List<?>  
List
```

```
Pair<?,?>[]
```

but not:

```
List<String>  
List<? extends Number>
```

see also: [TechnicalDetails.FAQ106](#), JLS 4.7

---

## S

### SuppressWarnings annotation

A standard annotation that suppresses warnings for the annotated part (e.g., method, field, class, ... etc.) of the program.

Example:

```
@SuppressWarnings(value={"unchecked","deprecation"})  
public static void someMethod() {  
    ...  
    TreeSet set = new TreeSet();  
    set.add(new Date(104,8,11)); // unchecked and deprecation warning  
    suppressed  
    ...  
}
```

see also: [TechnicalDetails.FAQ004](#), JLS 9.6.1.5

---

## T

### type argument

A reference type or a wildcard that is used for instantiation / invocation of a generic type or a reference type used for instantiation / invocation of a generic method.

Example:

```
List<?> list = new LinkedList<String>();
```

see also: [TypeArguments.FAQ001](#), JLS 4.5.1

### type inference

The automatic deduction of the type arguments of a generic method or generic type.

Example:

```
class Collections {  
    public static <A extends Comparable<? super A>> A max (Collection<A> xs)  
    { ... }  
}  
  
LinkedList<Long> list = new LinkedList<>(); // 1  
Long y = Collections.max(list); // 2
```

In line 1 the compiler infers the missing type argument (`Long`) from the lefthand side of the assignment; in line 2 it infers that the method type argument must be `Long` from the method argument's type.

see also: [TechnicalDetails.html.FAQ400](#), [TechnicalDetails.html.FAQ400A](#), [TechnicalDetails.html.FAQ401](#), JLS 15.12.2.7

## type erasure

The process that maps generic types and generic methods and their instantiations / invocations to their unique bytecode representation by eliding type parameters and type arguments.

Example:

```
before type erasure: public static <A extends Comparable<? super A>> A max
(Collection<A> xs) { ... }
```

```
after type erasure: public static Comparable max (Collection xs) { ... }
```

see also: [TechnicalDetails.FAQ101](#), JLS 4.6

## type parameter

A place holder for a type argument. Each type parameter is replaced by a type argument when a generic type or generic method is instantiated / invoked.

Example:

```
interface Comparable<E> {
    int compareTo(E other);
}
```

Synonyms: type variable

The Language Specification uses the terms type parameter and type variable as synonyms. Other authors sometimes use the term *generic parameter* as a synonym for type parameter.

see also: [TypeParameters.FAQ001](#), JLS 4.4

## type parameter bound

A reference type that is used to further describe a type parameter. It restricts the set of types that can be used as type arguments and gives access to the non-static methods that it defines.

Examples:

```
class Enum<E extends Enum<E>> { ... }
<T extends Comparable<? super T>> void sort(List<T> list) { ... }
```

see also: [TypeParameters.FAQ101](#), JLS 4.4

## type safety

A program is considered type-safe if the entire program compiles without errors and warnings and does not raise any unexpected `ClassCastException`s at runtime.

see also: [Fundamentals.FAQ004](#)

## type variable

A place holder for a type argument. Each type parameter is replaced by a type argument when a parameterized type or an instantiated method are created.

Example:

```
interface Comparable<E> {
    int compareTo(E other);
}
```

Synonyms: [type parameter](#)

The Java Language Specification uses the terms [type parameter](#) and [type variable](#) as synonyms.  
see also: [TypeParameters.FAQ001](#), JLS 4.4

---

## U

### unbounded wildcard

A [wildcard](#) without a bound. Basically it is the "?" wildcard.

Example:

```
Pair<?,String>
```

see also: [TypeArguments.FAQ102](#), JLS 4.5.1

### unbounded wildcard instantiation

see: [unbounded wildcard parameterized type](#)

### unbounded wildcard parameterized type

A [parameterized type](#) in which all type arguments are [unbounded wildcards](#).

Examples:

```
Pair<?,?>  
Map<?,?>
```

but not:

```
Pair<? extends Number,? extends Number>  
Map<String,?>
```

Synonyms: [unbounded wildcard instantiation](#)

see also: [GenericTypes.FAQ302](#)

### unchecked warning

A warning by which the compiler indicates that it cannot ensure [type safety](#).

Example:

```
TreeSet set = new TreeSet();  
set.add("abc");           // unchecked warning
```

---

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type  
java.util.TreeSet  
    set.add("abc");  
        ^
```

see also: [TechnicalDetails.FAQ001](#)

### upper bound

see: [upper wildcard bound](#)

### upper wildcard bound

A reference type that is used to further describe a wildcard; it denotes the family of types that are subtypes of the upper bound.

Examples:

```
List<? extends String>
List<? extends Thread.State>
List<? extends int[]>
List<? extends Callable<String>>
List<? extends Comparable<? super Long>>
```

see also: [TypeArguments.FAQ201](#), JLS 4.5.1

---

## V

### varargs warning

A warning by which the compiler indicates that methods with a variable argument list can lead to [heap pollution](#).

Example:

```
public static <E> void addAll(List<E> list, E... array) { // varargs warning
    for (E element : array) list.add(element);
}
```

---

```
warning: [varargs] Possible heap pollution from parameterized vararg type E
public static <E> void addAll(List<E> list, E... array) {
                                                ^
```

see also: [Practicalities.FAQ300A](#)

---

## W

### wildcard

A syntactic construct that denotes a family of types.

Examples:

```
?
? extends Number
? super Number
```

see also: [TypeArguments.FAQ101](#), JLS 4.5.1

### wildcard bound

A reference type that is used to further describe the family of types denoted by a wildcard.

Examples:

```
List<? super String>
List<? extends int[]>
List<? extends Callable<String>>
List<? extends >
```

`Comparable<? super Long>`

see also: [TypeArguments.FAQ201](#), JLS 4.5.1

## wildcard capture

An anonymous type parameter that represents the particular unknown type that the wildcard stands for. The compiler uses the capture internally for evaluation of expressions, and the term "capture of ?" occasionally shows up in error messages.

see also: [TechnicalDetails.FAQ501](#)

## wildcard instantiation

see: wildcard parameterized type

## wildcard parameterized type

A parameterized type where at least one type argument is a wildcard (as opposed to a concrete type).

Examples:

```
Collection<?>
List<? extends Number>
Comparator<? super String>
Pair<String, ?>
```

see also: [GenericTypes.FAQ301](#), JLS 4.5.1

---

# X

---

# Y

---

# Z

---

[CONTENT](#) [PREVIOUS](#)



# Index

© Copyright 2004-2022 by Angelika Langer. All Rights Reserved.

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

---

## A

actual type:

in reflection, see [reflective actual type](#)

annotation:

Override, see [Override annotation](#)

SuppressWarnings, see [SuppressWarnings annotation](#)

array:

component type, see [array component type](#)

[generic creation](#)

reference variable, see [array reference variable](#)

array reference variable:

[to array of concrete parameterized type](#)

[to array of bounded wildcard parameterized type](#)

[to array of unbounded wildcard parameterized type](#)

array component type:

[concrete parameterized type](#)

[wildcard parameterized type](#)

[type parameter/variable](#)

[unbounded wildcard parameterized type](#)

---

## B

bound:

, see [lower bound](#)

of type parameter, see [type parameter bounds](#)

of wildcard, see [wildcard bound](#)

upper, see [upper bound](#)

bounded:

[type parameter](#)

[wildcard](#)

bridge method:

performance penalty, see [performance](#)

[purpose](#)

[what is ...?](#)

---

## C

cast:

[compiler-generated](#)  
performance penalty, see [performance](#)  
[target type](#)  
[to parameterized type](#)  
[to type parameter](#)

[checked collection](#)

Class :  
[type parameter](#)

circular generic type, see [self-referential generic type](#)

class literal:

[of parameterized type](#)  
[of wildcard parameterized type](#)  
[of type parameter / variable](#)  
[what is ...?](#)

clone :  
[of generic type](#)

code sharing, see [compilation](#)

code specialization, see [compilation](#)

collection:

[Collection<Object>](#)  
[Collection<?>](#)  
[checked](#)  
generic, see [generic collection](#)  
mixed, see [collection of objects of different types](#)  
[of objects of different types](#)  
parameterized, see [parameterized collection](#)

compatibility:  
[binary](#)

compilation of generics  
[how does the compiler translate generics](#)

compiler-generated:  
bridge method, see [bridge method](#)  
[cast](#)

concrete:  
instantiation, see [concrete parameterized type](#)  
parameterized type, see [concrete parameterized type](#)  
concrete parameterized type:

[array of](#)  
[class literal](#)  
[compatibility](#)  
[of generic type](#)  
[use of](#)  
[vs. unbounded wildcard instantiation](#)  
[vs. wildcard instantiation](#)  
[what is ...?](#)

covariant return type

[vs. covariant return type and overriding](#)  
[what is ...?](#)

---

## D

declared type:  
in reflection, see [reflective declared type](#)

derivation:  
[from wildcard parameterized type](#)  
[from type parameter](#)

descriptor  
[vs. signature](#)

diamond operator  
[what is ...?](#)

dynamic type  
in reflection, see [reflective actual type](#)

---

## E

enum type:  
[decryption of Enum<E extends Enum<E>>](#)  
nested into generic type, see [nested enum type](#)  
[generic](#)

equals :  
[of generic / parameterized type](#)

erasure, see [type erasure](#)

exception handling:  
[parameterized types](#)

[explicit type argument specification](#)

extends keyword  
[meaning of](#)  
see also [upper bound](#)

---

## F

features:  
see [language features](#)

---

## G

generic:

array, see [java.lang.reflect.GenericArrayType](#)  
code, see [generic code](#)  
collection, see [generic collection](#)  
[exceptions](#)  
interface, see [generic interface](#)  
method, see [generic method](#)  
type, see [generic type](#)

[GenericArrayType](#), see [java.lang.reflect.GenericArrayType](#)

generic code:  
[mix with legacy code](#)

generic collection  
[vs. non-generic collection](#)

generic interface:  
[as bound of a type parameter](#)  
[implementing different instantiations](#)  
[in class hierarchy](#)  
[name collision](#)

generic method:  
[explicit type argument specification](#)  
[definition](#)  
[invocation](#)  
reflection, see [reflective generic method](#)  
[type argument inference](#)  
[vs. regular method](#)  
[what is ...?](#)

generic type:  
[as type parameter bound](#)  
[cast to](#)  
[definition](#)  
[enum type](#)  
[exception handling](#)  
[instantiation](#)  
[nested interface](#)  
reflection, see [reflective generic type](#)  
[runtime type of](#)  
static member, see [static member of generic type](#)  
[super-subtype relationship](#)  
[type system](#)  
[vs. raw type](#)  
[vs. regular type](#)  
[what is ...?](#)

generics, see [Java generics](#)

generification:  
[of legacy classes](#)

[getThis trick](#)

[heap pollution](#)

---

## I

import:

- [generic type](#)
- [parameterized type](#)

improved type inference, see [type argument inference](#)

inference, see [type argument inference](#)

instanceof:

- [target type](#)

instantiation:

- [concrete instantiation](#)
- [of generic method](#)
- [of generic type](#)
- [super-subtype relationship](#)
- [type relationship among instantiations of different generic types](#)
- [type relationship to lower bound wildcard instantiation](#)
- [type relationship to raw type](#)
- [type relationship to unbounded wildcard instantiation](#)
- [type relationship to upper bound wildcard instantiation](#)
- [wildcard instantiation](#)
- [vs. raw type](#)

interface:

- nested into generic type, see [nested interface](#)
- generic, see [generic interface](#)

---

## J

Java generics:

- [benefit](#)
- [purpose](#)
- [translation by type erasure](#)
- [what are ...?](#)

[java.lang.reflect.GenericArrayType](#)

see also: [JavaDoc](#)

[java.lang.reflect.ParameterizedType](#)

see also: [JavaDoc](#)

[java.lang.reflect.Type](#)

- [subtypes](#)

see also: [JavaDoc](#)

[java.lang.reflect.TypeVariable](#)

see also: [JavaDoc](#)

[java.lang.reflect.WildcardType](#)

see also: [JavaDoc](#)

---

## K

---

## L

language features:

- [bridge method](#)
- [concrete instantiation](#)
- [explicit type argument specification](#)
- [instantiation of generic method](#)
- [instantiation of generic type](#)
- [overview](#)
- [generic method](#)
- [generic type](#)
- [parameterized type](#)
- [raw type](#)
- [type argument](#)
- [type argument inference](#)
- [type erasure](#)
- [type parameter](#)
- [type parameter bound](#)
- [wildcard](#)
- [wildcard bound](#)
- [wildcard capture](#)
- [wildcard instantiation](#)

legacy code:

- [generify](#)
  - [mix with generic code](#)
  - re-engineer, see [generify](#)

lower bound:

- [what is ...?](#)
- [on wildcard](#)
- wildcard instantiation, see [lower bound wildcard instantiation](#)
- [on type parameter](#)
- [difference on wildcard and on type parameter](#)

lower bound wildcard instantiation:

- [type relationship to other instantiations of the same type](#)

---

## M

method:

- bridge, see [bridge method](#)
- descriptor, see [descriptor](#)
- generic, see [generic method](#)
- parameterized, see [generic method](#)
- signature, see [signature](#)
- synthetic, see [bridge method](#)
- [with wildcard return type](#)
- [with wildcard argument type](#)
- [with multi-level wildcard argument type](#)

with varargs, see [variable argument list](#)

multi-level wildcard  
[as argument type of a method](#)  
[what is ...?](#)

---

## N

nested enum type (with parameterized enclosing type):  
[scope name](#)

nested interface (with parameterized enclosing type):  
[scope name](#)  
[access to enclosing type parameters](#)

non-reifiable type:  
[and arrays](#)  
[and varargs](#)  
[vs. reifiable type](#)

---

## O

object:  
[generic creation](#)  
[of concrete parameterized type](#)  
[of wildcard parameterized type](#)  
[of type parameter/variable](#)  
[of unbounded wildcard parameterized type](#)

overloading:  
[problems with](#)  
[what is ...?](#)

overload resolution:  
[and type variables](#)  
[what is ...?](#)

[Override annotation](#)

override-equivalent signature:  
[and overloading](#)  
[and overriding](#)  
[vs. subsignature](#)  
[what is ...?](#)

overriding:  
[generic sub-method + generic super-method](#)  
[generic sub-method - non-generic super-method](#)  
[generic subtype + generic supertype](#)  
[non-generic sub-method + generic super-method](#)  
[non-generic subtype + generic supertype](#)  
[problems with](#)  
[what is ...?](#)

---

# P

parameterized:

collection, see [parameterized collection](#)

interface, see [parameterized interface](#)

method, see [generic method](#)

type, see [parameterized type](#)

parameterized collection

[vs. non-parameterized collection](#)

parameterized interface:

[as bound of a type parameter](#)

[implementing different instantiations](#)

[in class hierarchy](#)

[name collision](#)

ParameterizedType, see [java.lang.reflect.ParameterizedType](#)

parameterized type:

[as type parameter bound](#)

[cast to](#)

[definition](#)

[enum type](#)

[exception handling](#)

[instantiation](#)

[nested interface](#)

reflection, see [reflective parameterized type](#)

[runtime type of](#)

static member, see [static member of parameterized type](#)

[super-subtype relationship](#)

[type system](#)

[vs. raw type](#)

[vs. regular type](#)

[what is ...?](#)

[performance](#)

polluted heap:

see [heap pollution](#)

polymorphic method dispatch:

[and overriding](#)

primitive types:

[as type arguments](#)

---

# Q

---

# R

raw type:

[purpose](#)

[type relationship to instantiation](#)

[use of](#)



[vs. parameterized type](#)  
[vs. unbounded wildcard instantiation](#)  
[what is ...?](#)

recursive

bound, see [self-referential generic type](#)  
generic type, see [self-referential generic type](#)  
type parameter, see [self-referential generic type](#)

reifiable type:

[vs. non-reifiable type](#)  
[what is ...?](#)

[reification](#)

reflection:

[and generics](#)

reflective ...

[actual type](#)

[declared type](#)

dynamic type, see [reflective actual type](#)  
generic method, see [reflective generic method](#)  
generic type, see [reflective generic type](#)  
parameterized type, see [reflective parameterized type](#)  
static type, see [reflective declared type](#)  
type parameter, see [reflective type parameter](#)  
wildcard type, see [reflective wildcard type](#)

[reflective generic method](#)

[reflective generic type](#)

[difference](#) to reflective parameterized type

[reflective parameterized type](#)

[difference](#) to reflective generic type

[reflective type parameter](#)

[reflective wildcard type](#)

return type:

covariant, see [covariant return type](#)  
substitutable, see [substitutable return type](#)

runtime type:

[class literal](#)  
[information](#)  
[of parameterized type](#)

---

## S

safety, see [type safety](#)

[self-referential generic type](#)

signature

[and overloading](#)  
[and overriding](#)  
override-equivalent, see [override-equivalent signature](#)  
subsignature, see [subsignature](#)  
[what is ...?](#)

static:

member, see [static member of generic type](#)  
type, see [static type in reflection](#)

static type

in reflection, see [reflective declared type](#)

static member of generic type

[how many instances?](#)  
[name of](#)  
[type parameter as type of](#)

subsignature

[and overloading](#)  
[and overriding](#)  
[vs. override-equivalent signature](#)  
[what is ...?](#)

substitutable return type:

[vs. covariant return type](#)  
[and overriding](#)  
[what is ...?](#)

super keyword

see [lower bound](#)

super-subset

[relationship among type families denoted by wildcards](#)

super-subtype

[relationship among instantiations of generic types](#)

[SuppressWarnings annotation](#)

---

## T

throws clause:

and overriding, see [conflicting](#)  
[and type parameter](#)

[conflicting](#)

Type, see [java.lang.reflect.Type](#)

type:

argument, see [type argument](#)  
check, see [type check](#)  
generic, see [generic type](#)  
parameter, see [type parameter](#)  
parameterized, see [parameterized type](#)  
[raw type](#)  
reifiable, see [reifiable type](#)  
[runtime type](#)

type argument:

- [and type parameter bounds](#)
- [explicit type argument specification](#)
- inference, see [type argument inference](#)
- [of generic method](#)
- [of parameterized type](#)
- [permitted types](#)
- [primitive types](#)
- [type parameters](#)
- [what is ...?](#)
- [wildcards](#)

type check:

- in checked collection, see [checked collection](#)
- in equals method, see [equals](#)
- [implicitly generated](#)
- [leading to unchecked warning](#)
- [target type is a parameterized type](#)
- target type is a type parameter:
  - [performed at compile time](#)
  - [performed at runtime time](#)
- see also, [cast](#)

type argument inference:

- [explicit type argument specification](#)
- [for generic methods](#)
- [for instance creation](#)
- [from context](#)
- [what is ...?](#)
- see also, [diamond operator](#)

type erasure:

- bridge method, see [bridge method](#)
- [compiler-generated cast](#)
- [of generic method](#)
- [of parameterized type](#)
- [of type parameter](#)
- reifiable type, see [reifiable type](#)
- [several bounds](#)
- [what is ...?](#)

type parameter:

- [array of](#)
- [as bound of another type parameter](#)
- [as part of its own bounds](#)
- [as supertype](#)
- [as target type of runtime type check](#)
- [as type argument](#)
- [bounded](#)
- bounds, see [type parameter bounds](#)
- [cast to](#)
- [class literal](#)
- [derive from](#)
- [in catch clause](#)
- [in exception handling](#)
- [in throw statement](#)

[in throws clause](#)  
[object of](#)  
[of an outer type](#)  
reflection, see [reflective type parameter](#)  
[scope of](#)  
[static context](#)  
[type-like use](#)  
[use of](#)  
[what is ...?](#)

type parameter bound  
[access to members](#)  
[and type arguments](#)  
[different instantiations of a same generic type](#)  
[extends clause](#)  
[permitted types](#)  
[vs. type wildcard bound](#)  
[what is ...?](#)

type relationship  
[inheritance](#)  
[super-subtype](#)  
see also [type relationship among](#)

type relationship among:  
[raw type and parameterized type](#)  
[instantiations of different generic types](#)  
[lower bound wildcard instantiations and other instantiations of the same generic type](#)  
[instantiations of generic type \(in general\)](#)  
[unbounded wildcard instantiations and other instantiations of the same generic type](#)  
[upper bound wildcard instantiations and other instantiations of the same generic type](#)

[type-safety](#)

type system, see [type relationship](#)

type token:  
[for generic creation of objects and arrays](#)  
[for dynamic retrieval of type arguments](#)

TypeVariable, see [java.lang.reflect.TypeVariable](#)

type variable, see [type parameter](#)

typing:  
[strong](#)  
[weak](#)

---

## U

unbounded  
[wildcard](#)  
wildcard instantiation, see [unbounded wildcard instantiation](#)

unbounded wildcard instantiation:  
[array of](#)  
[type relationship to other instantiations of the same generic type](#)

[vs. bounded wildcard instantiation](#)  
[vs. concrete wildcard instantiation](#)  
[vs. raw type](#)  
[what is ...?](#)

unchecked warning

[avoid](#)  
[disable](#)  
[eliminate](#)  
[enable](#)  
[spurious](#)  
[suppress](#)  
[what is ...?](#)

upper bound:

[wildcard](#)  
wildcard instantiation, see [upper bound wildcard instantiation](#)  
[what is ...?](#)

upper bound wildcard instantiation:

[type relationship to other instantiations of the same generic type](#)

---

## V

varargs, see [variable argument list](#) and [varargs warning](#)

varargs warning

[suppress](#)  
[what is ...?](#)

variable argument list:

[and non-reifiable types](#)

---

## W

warning:

unchecked, see [unchecked warning](#)

wildcard:

[as type argument](#)  
[bounded](#)  
capture, see [wildcard capture](#)  
[in method signatures](#)  
instantiation, see [wildcard parameterized type](#)  
multi-level, see [multi-level wildcard](#)  
parameterized type, see [wildcard parameterized type](#)  
reflection, see [reflective wildcard type](#)  
[super-subset relationship](#)  
type, see [wildcard parameterized type](#)  
[unbounded](#)  
[what is ...?](#)

wildcard bound

[extends clause](#)  
lower, see [lower bound](#)

[permitted types](#)  
[upper](#), see [upper bound](#)  
[vs. type parameter bound](#)  
[what is ...?](#)

wildcard capture:

[assignment-compatibility](#)  
[of bounded wildcard](#)  
[what is ...?](#)

wildcard parameterized type:

[access to fields and methods](#)  
[array of](#)  
[as argument type of a method](#)  
[as return type of a method](#)  
[as supertype](#)  
[class literal](#)  
[derive from](#)  
[in method signatures](#)  
[in new expression](#)  
[object of](#)  
[unbounded](#), see [unbounded wildcard instantiation](#)  
[use of](#)  
[vs. concrete instantiation](#)  
[vs. unbounded wildcard instantiation](#)  
[what is ...?](#)  
[with lower bound](#)

WildcardType, see [java.lang.reflect.WildcardType](#)

---

## X

[xlint compiler option](#)

---

## Y

---

## Z

---

I work as an independent freelance trainer and develop and conduct courses mainly in Europe and the USA. I am co-author of the authoritative book on "C++ Standard IOStreams and Locales" published at Addison Wesley. I wrote the column "Effective Standard Library" for the US magazines C++ Report and C/C++ Users Journal. Currently I am writing a column named "Effective Java" for the German magazine Java Magazin (formerly published in JavaSpektrum). I am a Java Champion and an observing member of the ISO/ANSI C++ standards committee. I am a regular speaker at conferences all over the world.

Currently my preferred fields of interest are training, coaching, and mentoring in the area of object-oriented software development in C++ and Java. I am most interested in development of explanatory material, including course material, multimedia training, books, and articles and I enjoy passing on my expertise in lectures, seminars, and workshops.

My area of expertise is advanced C++ and Java programming, concurrent programming, Enterprise Java, and many more. For further information go to my [CURRICULUM VITAE](#).



# Angelika Langer

## Trainer & Author & Mentor

Neumarkter Straße 86d  
D-81673 München  
Germany

[info@AngelikaLanger.com](mailto:info@AngelikaLanger.com)

---

I hereby request NOT to receive unsolicited commercial electronic mail. Failure to comply to this request will be prosecuted.

Gemäß § 28 BDSG widerspreche ich hiermit jeglicher Speicherung, Verwendung, Auswertung, Verknüpfung und Weitergabe der oben genannten personenbezogenen Daten zu kommerziellen Zwecken. Die Zusendung unverlangter Emails werblichen Inhalts an die oben genannte Email-Adresse wird hiermit ausdrücklich untersagt. Zuwiderhandlungen werden verfolgt.