rOOts 2003

Java Trends
JDK 1.5

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com

JDK 1.5

- release announced for end of 2003
- several new features in the language and the libraries
  - JSR 014 - generics
  - JSR 166 - concurrency utilities
  - JSR 201 - autoboxing, enum, ...

agenda

- generics
- concurrency utilities
- enum types
- autoboxing

Java Generics

- motivation for adding generic types and methods to Java:
  - higher expressiveness and improved type safety
  - make type parameters explicit and make type casts implicit
  - crucial for using libraries such as collections in a flexible, yet safe way

Java generics vs. C++ templates

- Java generics are said to be something like C++ templates.
  – common misconception
- Java generics have nearly nothing in common with C++ templates.
  – C++ templates is a Turing complete language.
  – Java generics is syntactic sugar that elides some casting.

agenda - generics

- overview
  – language changes: parameterized types and methods
  – library changes: parameterized collections & extended reflection
  – related language changes: covariant return types
  – type variables
  – translation to bytecode

terminology

- *parameterized type* or *method*
  - class / interface or method that has type parameters
- *type variable*
  - placeholder for a type, i.e. the type parameter

type variable

```
class Seq<E> implements List<E> {

  static boolean isSeq(Object x) {
    return x instanceof Seq;
  }

  static <T> boolean isSeq(List<T> x) {
    return x instanceof Seq<T>;
  }
  static boolean isSeqArray(Object x) {
    return x instanceof Seq[];
  }
}
```


paramterized types

- instantiations of parameterized types look like C++ templates

- examples:

```
Vector<String>
Seq<Seq<A>>
Seq<String>.Zipper<Integer>
Collection<Integer>
Pair<String,String>
```

- primitive types cannot be parameters
  - `Vector<int>` is illegal

benefit of parameterized types

- today: no information available about the type of the elements contained in a collection

cast might fail

```
void append(Vector v, char[] suffix) {
  for(int idx=0; idx<v.size(); ++idx) {
    StringBuffer buf = (StringBuffer) (v.get(idx));
    buf.append(suffix);
  }
}
```

- future: parameterized type provides more information and performs cast implicitly

cannot fail

```
void append(Vector<StringBuffer> v, char[] suffix) {
  for(int idx=0; idx<v.size(); ++idx) {
    StringBuffer buf = v.get(idx);
    buf.append(suffix);
  }
}
```


type variables

- definition of a parameterized class
  – type variables T1 and T2 act a parameters

```
class Pair <Type1,Type2> {
  private Type1 t1;
  private Type2 t2;
  ...
}
```

- type variable can have optional *bounds*
  – a bound consist of a class and/or several interfaces
  – if no bound is provided `Object` is assumed

```
class AssociativeArray <Key extends Comparable, Value> {
  ...
}
```

shared type identification

- all instantiations of a parameterized type have the same runtime type
  - type parameters are not maintained at runtime and do not show up in the byte code

```
Vector<String>  x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();

return x.getClass() == y.getClass();
```

true

raw types

- *raw type*: parameterized class without its parameters
  - variables of a raw type can be assigned from values of any of the type's parametric instances
  - reverse assignment permitted to enable interfacing with legacy code

```
Vector rawVector = new Vector();
Vector<string> stringVector = new Vector<String>();

rawVector = stringVector;

stringVector = rawVector;
```

fine

compiler warning:
assignment deprecated

raw types

- access to fields of a raw type

```
class Cell<Type> {
 private Type value;
 public Cell (Type v)    { value=v; }
 public Type get()       { return value; }
 public void set(Type v) { value=v; }
}
```

```
Cell rawCell = new Cell<String>("abc");
... rawCell.value ...;
... rawCell.get();

rawCell.set("def"); // deprecated
```

fine, value has type Object

compiler warning:
unchecked access to field

do we really benefit?

```
void append(Vector<StringBuffer> v, char[] suffix) {
  for(int idx=0; idx<v.size(); ++idx) {
    StringBuffer buf = v.get(idx);
    buf.append(suffix);
  }
}
```

- raw type can be assigned to instantiated type
  - creates compiler warning, but is permitted

```
Vector files = new Vector();
// fill with Strings, not StringBuffers !!!
Vector<StringBuffer> tmp = files;
append(tmp, ".txt");
```

implicit cast can fail

assignment of raw type permitted

parameterized methods

- method declarations can have a type parameter section like classes have

```
static <Elem> void swap(Elem[] a, int i, int j) {
  Elem temp = a[i]; a[i] = a[j]; a[j] = temp;
}
```

```
<Elem extends Comparable<Elem>> void sort(Elem[] a) {
  for (int i = 0; i < xs.length; i++)
  for (int j = 0; j < i; j++)
    if (a[j].compareTo(a[i]) < 0) <Elem>swap(a, i, j);
}
```

- constructors can be parameterized, too

parameter inference

- no special syntax for invocation
  – type parameters are inferred from arguments and calling context

```
Integer[] ints;
Strings[] strings;
…
swap(ints, 1, 3);    // infers Elem := Integer
sort(strings);      // infers Elem := String
```

- explicit specification of type parameters is allowed

```
<Integer>swap(ints, 1, 3);
<String>sort(ints);
```

agenda - generics

- overview
  - language changes: parameterized types and methods
  - library changes: parameterized collections & extended reflection
  - related language changes: covariant return types
  - type variables
  - translation to bytecode

parameterized collections

- collections from collection framework are parameterized
- examples:

```
public interface Set<E> extends Collection<E> {
 public boolean add(E e);
 public boolean contains(Object e);
 public Iterator<E> iterator();
 public <T> T[] toArray(T[] a);
 ...
}
public class TreeSet<E> extends AbstractSet<E> ...
{
 public TreeSet(SortedSet<E> s) ;
 public TreeSet(Comparator<E> c);
 ...
}
```

extended reflection

- additional information for parameterized types
- in class `Class`:

```
public Type getGenericSuperclass()
public Type[] getGenericInterfaces()
public ClassTypeVariable[] getTypeParameters()
```

- in class `Method` and class `Constructor`:

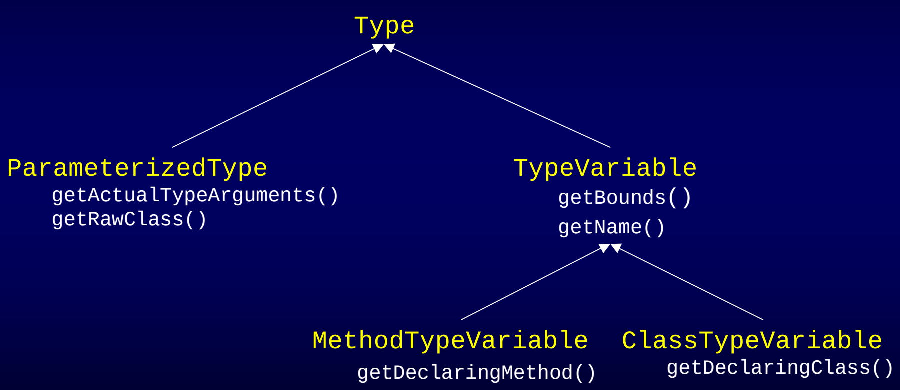
```
public Type getGenericReturnType()
public Type[] getGenericParameterTypes()
```

- in class `Field`:

```
public Type getGenericType()
```


extended reflection

- new hierarchy of interfaces



agenda - generics

- overview
  - language changes: parameterized types and methods
  - library changes: parameterized collections & extended reflection
  - related language changes: covariant return types
  - type variables
  - translation to bytecode

covariant return types

- overriding methods may have a result type that is a subtype of the result types of all methods it overrides
  - before generics, the result types had to be identical

overrides

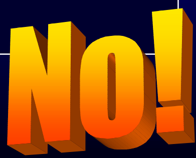
```
class Super implements Cloneable {
  Super copy() { return (Super)clone(); }
  ...
}
class Sub extends Super implements Cloneable {
  Sub copy() { return (Sub)clone(); }
  ...
}
```

no covariant argument types

- overriding methods must still have identical argument types

```
class Super implements Cloneable {
  Super clone() { ... }
  boolean equals(Super s) { ... }
  ...
}
class Sub extends Super implements Cloneable {
  Sub clone() { ... }
  boolean equals(Sub s) { ... }
  ...
}
```

does NOT override



agenda - generics

- overview
  – language changes: parameterized types and methods
  – library changes: parameterized collections & extended reflection
  – related language changes: covariant return types
  – type variables
  – translation to bytecode

several bounds

- a type parameter can have more than one bound

```
class X<T implements SuperClass & Interface1 & Interface2> {
    ...
}
```

- the erasures of all bounds must be pairwise different

```
class X<T extends Interface<A> & Interface<B>> {
  ...
}
```

error:
Interface cannot
be inherited
with different
arguments

- if no bound is given, `Object` is assumed

type variable vs. types

- type variables are not types
- type variables cannot be used
  - in static context
  - to create objects
  - for type checks via instanceof
  - as supertypes

type variables & static context

- scope of a type variable = all of the declared class
  - including the type parameter section itself
    - i.e. type variables can appear as parts of their own bounds
    - e.g. `<Elem extends Comparable<Elem>> void sort(Elem[] a)`
  - except any static members or initializers
    - no use for static data members
    - no use in static methods
    - interesting effects in conjunction with nested types

```
class X<T> {
         T t1;   // fine
  static T t2;   // illegal
}
```

```
class X<T> {
         T getT1(){...}    // fine
  static T getT2(){...}    // illegal
}
```


type variables & new expressions

- type variable cannot be used to create objects
  - can only declare reference variables

```
class Tuple <Elem> {
  private Elem p1, p2;

  public Tuple() {
    p1 = new Elem();   p2 = new Elem();
  }

  public Tuple(Elem a1, Elem a2) {
    p1 = new Elem(a1);   p2 = new Elem(a2);
  }
}
```

```
error: unexpected type
found: Elem
required: class
```

No!

handling reference

- parameterized classes can easily handle references
  - but value semantics are difficult

```
class Tuple <Elem> {
  private Elem p1, p2;

  public Tuple() {
    p1 = null;  p2 = null;
  }
  public Tuple(Elem a1, Elem a2)
{
    p1 = a1;  p2 = a2;
  }
  public Elem getFirst() {
    return p1;
  }
  public void setFirst(Elem a1) {
    p1 = a1;
  }
}
```


type variables & instanceof

- type variable cannot be used in a type check via `instanceof`

```
error: unexpected type
found: Elem
required: class
```

```
public class Tuple<Elem> {
  private Elem p1, p2;

  public <T extends Tuple> Tuple(T other) {
    if (other.p1 instanceof Elem)
      this.p1 = (Elem) other.p1;
    else
      this.p1 = null;
    ... same for p2 ...
  }
}
```

No!

type variables & casting

- type variable can be used in a cast
  - might yield a warning

warning:
unchecked cast to type Elem

```
public class Tuple<Elem> {
  private Elem p1, p2;

  public <T extends Tuple> Tuple(T other) {
    try {
      this.p1 = (Elem) other.p1;
    }
    catch (ClassClastException e) {
      this.p1 = null;
    }
    ... same for p2 ...
  }
}
```


type variables & casting

- cast is not guaranteed to fail at runtime
  - even if nonsensical

```
public <T extends Tuple> Tuple(T other) {
  try { this.p1 = (Elem) other.p1; }
  catch (ClassClastException e) { this.p1 = null; }
  ...
}
```

```
Tuple<String> pairOfAliens =
  new Tuple<String>("Dick","Doof");

Tuple<Exception> pairOfExceptions =
  new Tuple<Exception>(pairOfAliens);
```

cast should fail

No!

- cast should fail and trigger assignment of `null`
  - instead `String`s are stored in tuple of `Exception`s

avoid raw types

- alternative implementation of generic constructor
  - avoiding raw types is safer

```
public class Tuple<Elem> {
  private Elem p1, p2;

  public <E extends Elem> Tuple(Tuple<Elem> other) {
    this.p1 = other.p1;
    this.p2 = other.p2;
  }
}
```

```
Tuple<String> pairOfAliens =
  new Tuple<String>("Dick","Doof");

Tuple<Exception> pairOfExceptions =
  new Tuple<Exception>(pairOfAliens);
```

typesafe;
no cast needed

will not compile

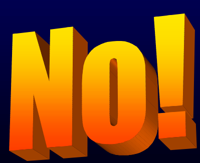
type variables & subclassing

- type variables cannot be subclassed from

```
class Outer<TypeVariable> {
  private class Inner extends TypeVariable {
    ...
  }
  ...
}
```

```
error: unexpected type
found: TypeVariable
required: class
```

- cannot build generic adapters
  - Type ⇨ Adapted<Type>



conclusion

- type variables are not types
  - can only be used as argument and return type of methods or for reference variables
  - are mapped to `Object` (or their leftmost bound)
- tremendous restrictions on variables of "unknown" type
  - stored and treated as `Object` references
  - no type information available
- surprising whenever two type variables are involved
  - like type parameters of a generic class and its generic method, or
  - type parameters of an outer and an inner class

agenda - generics

- overview
  - language changes: parameterized types and methods
  - library changes: parameterized collections & extended reflection
  - related language changes: covariant return types
  - type variables
  - translation to bytecode

translation to bytecode

- generics are translated to bytecode
  - unlike C++ templates, which are instantiated, i.e. further C++ classes and functions are generated, which are eventually translated to executable code
- process of translation of generics
  - erase all type parameters
  - map type variables to their bounds
  - insert casts as needed

translation of expressions

- casts are inserted where necessary
  - access to field whose type is a type parameter
  - invocation of method whose return type is a type parameter

field access example:

- erasure of `c.value` is `Object`
- `f()` returns `String`
- return statement translated to

```
return (String) c.value;
```

```
class Cell<A> {
   A value;
}
...
String f(Cell<String> c) {
   return c.value;
}
```

translation of methods

- method `T m(T`$_1$`, ..., T`$_n$`) throws E`$_1$`,...,E`$_m$ is translated to
  - a method with the same name
  - whose return type, argument types, and thrown types are the erasures of the corresponding types in the original method

- compile-time error
  - if different methods with identical names but different types are mapped to methods with the same type erasure

example - illegal methods

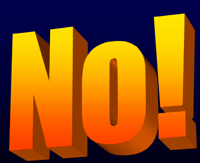
```
class C<A> {
  A id(A x) {...}
}
class D extends C<String> {
  Object id(Object x) {...}
}
```

error: same erasure

- class `D` has two methods with the same name and different signatures, but the same erasure:

  `Object id(Object)`
  - member of `D`

  `String id(String)`
  - inherited from `C<String>`
  - erasure: `Object id(Object)`



bridge methods

- a bridge method is generated
  - if a method `m` of a class or interface `C` is inherited in a subclass `D`

```
class C<A> {
  abstract A id(A x);
}
class D extends C<String> {
  String id(String x) { return x; }
}
```

is translated to:

```
class C {
  abstract Object id(Object x);
}
class D extends C {
  String id(String x) { return x; }
  Object id(Object x) { return id((String)x); }
}
```


example - bridge method

```
class C<A> {
  abstract A next();
}
class D extends C<String> {
  String next() { return ""; }
}
```

is translated to:

```
class C {
  abstract Object next();
}
class D extends C {
  String next_1() { return ""; }
  Object next_2() { return next_1(); }
}
```

Note, that the bridge method has the same signature as the original method.

example - covariant return types

- Same technique is used for overriding methods with covariant return types.

```
class C           { C dup() {...} }
class D extends C { D dup() {...} }
```

is translated to:

```
class C {
  C dup();
}
class D extends C {
  D dup1(){...}
  C dup2(){ return dup1(); }
}
```


agenda

- generics
- concurrency utilities
- enum types
- autoboxing

problem

Java threading primitives like

- synchronized blocks, and
- `Object.wait()`, `Object.notify()`

are

- too low-level for some application, and
- their overall functionality is too small for others.

scope

- standardize medium-level concurrency constructs
  - simplify application programming
  - avoid reinvention (and incompatibilities)
  - improve implementation quality and efficiency
- add minimal low-level support
  - overcome existing small design problems
  - avoid gratuitous incompatibilities with POSIX pthreads and RTSJ
  - Include only constructs ‘easy’ to add to common JVMs

concurrency utilities - overview

- locks
- condition variables
- queues
- synchronizers
- executors
- atomic variables
- timing
- concurrent collections
- uncaught exception handlers

agenda

- locks and semaphores
- conditions
- queues
- synchronizers
- executors

© Copyright 2003 by Angelika Langer & Klaus Kreft. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/7/2005 ,22:55

interface Lock

- package java.util.concurrent provides a Lock interface:

```
public interface Lock {

  // lock aquisition
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit granularity)
                                throws InterruptedException;

  // lock release
  public void unlock()

  ...
};
```


class ReentrantLock

- ReentrantLock is a class that implements Lock
- provides behavior similar to a mutex associated with an object
  - mutex is acquired and released implicitly
    - when passing in and out of a synchronized block
  - lock is locked and unlocked explicitly

```
synchronized(myObject) {   <---  myLock.lock()
   ...
}                          <---  myLock.unlock()
```

class `ReentrantLock` (cont.)

- upside:
  - `ReentrantLock` overcomes the limitation of synchronized blocks:
    - acquiring and releasing of locks not bound to block boundaries
      - e.g. hand-over-hand locking possible
    - waiting thread can be interrupted
    - waiting thread can timeout
- downside:
  - release of a `ReentrantLock` not enforced
    - use `finally`
      - to make sure that `unlock()` is also called in case of an exception

using a lock - example

```
Lock l = new RentrantLock();
l.lock();
try {
  // access the resource protected by this lock
} catch ( ... ) {
  // ensure consistency before releasing lock
} finally {
  l.unlock();
}
```

semaphore

conceptually, a semaphore maintains a set of permits

- `acquire()` blocks if necessary until a permit is available, and then takes it
- `release()` adds a permit, potentially releasing a blocking acquirer

the following classes are provided with JSR-166:

- `Semaphore`
  - no guarantees about the order in which threads acquire permits
- `FifoSemaphore`
  - FIFO order in which threads acquire permits

semaphore (cont.)

semaphores are used to restrict the number of threads that can access some resource

Binary Semaphore

- a semaphore that has at most one permit available
- it can serve as a mutual exclusive lock
  - similar to an instance of `ReentrantLock`
  - but with different lock policy:
    - not reentrant
    - the lock can be released by other threads if they have access to the semaphore,
      - i.e. semaphores have no ownerhip

read-write locks

JSR-166 provides the following interface:

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
};
```

classes implementing the interface vary in lock policy:

- preference: reader, writer, fifo, …
- lock upgrading and downgrading
- ownership of lock (e.g. writer owns, readers not)
- …

read-write locks (cont.)

- at the moment, details about supported policies and their combinations are still open
- read-write locks can significantly improve the performance of abstractions that are mostly read and rarely mutated

agenda

- locks
- conditions
- queues
- synchronizers
- executors

interface Condition

- java.util.concurrent provides a `Condition` interface:

```
public interface Condition {
  // waiting
  void awaitUninterruptibly();
  void await() throws InterruptedException;
  void awaitNanos(long t) throws InterruptedException;
  void awaitUntil(Date d) throws InterruptedException;
  // notifying
  void signal();
  void signalAll();
}
```

- offers more flexibility than the Java built-in condition that is associated with each object

interface `Condition` (cont.)

- advantage of `Conditions` over Java built-in conditions associated with each object:
  - flexible wait policy
  - more than one condition associated with one lock
    - allows more expressive implementations
      - i.e. closer to the logical solution
    - solves "nested monitor problem" in a convenient way
    - allows a programming style closer to POSIX pthreads

how to obtain a condition

- `Lock` interface provides a method:

  `Condition newCondition()`
  - creates condition bound to respective lock instance
- locks have a utility class `Locks`
  - contains static helper methods
  - (similar to `Collections`, `Arrays`, etc.)
- one of the helpers is:

  `Condition newConditionFor(Object o)`
  - creates condition bound to built-in mutex associated with object `o`

nested monitor problem

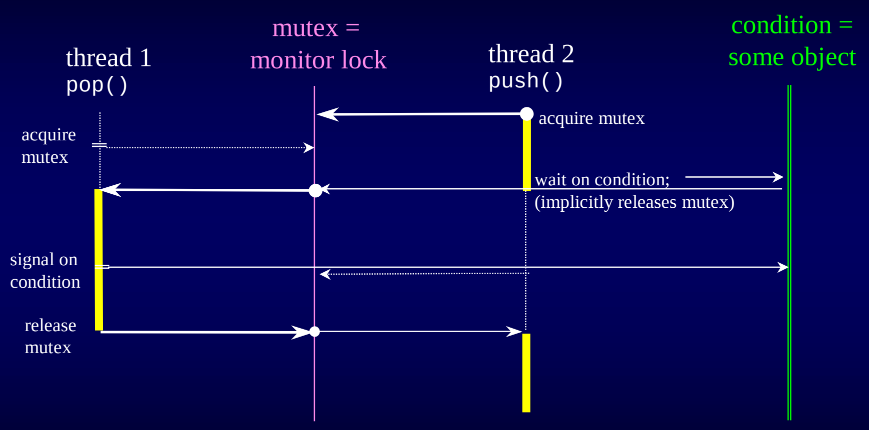
- built-in conditions are tied to objects
  - every object has a mutex and a condition that uses the mutex
    - mutex is implicitly used when methods/blocks are declared `synchronized`
    - condition is implicitly used when `wait()` / `notify()` are invoked
- intuitive approach for several logical conditions:
  - use a built-in condition for each logical condition
  - leads to "nested monitor problem"

several logical conditions - example

condition → `while (cnt == array.length)`
state change → `array[cnt++] = element;`
notification → `notifyAll();`
condition → `while (cnt == 0)`
notification → `notifyAll();`
state change → `return (array[--cnt]);`

```
public class blocking_int_stack {
 …
 public synchronized void push(int element) {
   while (cnt == array.length){
     try { wait(); }
     catch (InterruptedException e) { ... }
   }
   array[cnt++] = element;
   notifyAll();
 }
 public synchronized int pop() {
    while (cnt == 0) {
      try { wait(); }
      catch (InterruptedException e) { ... }
    }
    notifyAll();
    return (array[--cnt]);
  }
}
```

mechanics of wait / notify



using several built-in conditions - example

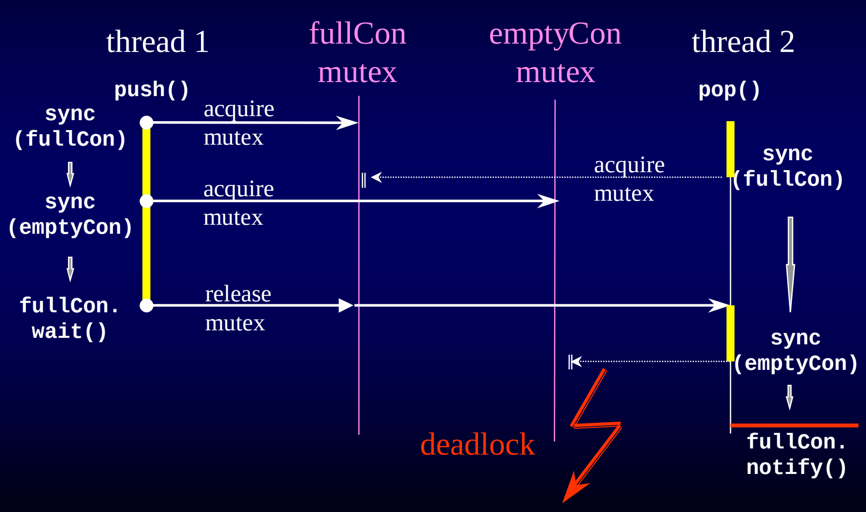
```
public class blocking_int_stack {
  private Object fullCon = new int[1];
  private Object emptyCon = new int[1];
  ...
  public void push(int element) {
    synchronized(fullCon) {
      synchronized(emptyCon) {
        while (cnt == size) {
         try { fullCon.wait(); }
         catch (InterruptedException e) { ... }
        }
        array[cnt++] = element;
        emptyCon.notify();
      }
    }
  }
```

several built-in conditions - example (cont.)

```
public class blocking_int_stack {
  private Object fullCon = new int[1];
  private Object emptyCon = new int[1];
  ...

  public int pop() {
    synchronized(fullCon) {
      synchronized(emptyCon) {
        while (cnt == 0) {
          try { emptyCon.wait(); }
          catch (InterruptedException e) { ... }
        }
        int tmp = array[--cnt];
        fullCon.notify();
        return (tmp);
      }
    }
  }
}
```


nested monitor problem



crux

- problem occurs due to acquisition of two locks
  - built-in monitor locks and conditions are not independent
    - both associated with same object
  - eliminate problem:
    - associate one mutex with both conditions
    - possible in general, but not with built-in conditions
    - use explicit conditions from `java.util.concurrent` package

avoid "nested monitor" with Conditions

- use two conditions associated with the one mutex of `this`
  - instead of two object-specific conditions associated with two object-specific mutexes

```
public class blocking_int_stack {
  private Condition fullCon  = Locks.newConditionFor(this);
  private Condition emptyCon = Locks.newConditionFor(this);
  ...
  public synchronized void push(int element) {
    while (cnt == size) {
      try { fullCon.await(); }
        catch (InterruptedException e) { ... }
    }
    array[cnt++] = element;
    emptyCon.signal();
  }
}
```

avoid "nested monitor" with Conditions (cont.)

```
public class blocking_int_stack {
  private Condition fullCon  = Locks.newConditionFor(this);
  private Condition emptyCon = Locks.newConditionFor(this);
  ...
  public synchronized int pop() {
   while (cnt == 0) {
     try { emptyCon.await(); }
     catch (InterruptedException e) { ... }
   }
   int tmp = array[--cnt];
   fullCon.signal();
   return (tmp);
  }
}
```


agenda

- locks
- conditions
- queues
- synchronizers
- executors

blocking queue interface

JSR-166 provides a queue interface:

```
public interface Queue<E> extends Collection<E> {
  // insertion
  boolean add(E e);
  boolean offer(E x);
  void put(E x);
  ...
  // removal
  E remove();
  E poll();
  E take();
  ...
}
```

- added to package `java.util.concurrency`
  - i.e. queues for inter-thread communication

blocking queue implementations

- various *blocking* queues:
  - `ArrayBlockingQueue`, bound, based on a fixed-size array
  - `LinkedBlockingQueue`, unbound, based on a linked list
  - `PriorityBlockingQueue`, unbound, arranges its elements like `PriorityQueue` from `java.util`
- `SynchronousQueue`, each put must wait for a take and vice versa
- `DelayQueue`, unbound, elements cannot be taken until delay time (specified in put) has been elapsed
- `LinkedQueue`, unbound, thread-save but non-blocking

using blocking queues - example

```
class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c = new Consumer(q);
    new Thread(p).start();
    new Thread(c).start();
  }
}
```


using blocking queues - example

```
class Producer implements Runnable {
  private final BlockingQueue queue;

  Producer(BlockingQueue q) { queue = q; }

  public void run() {
    try {
      while(true) { queue.put(produce()); }
    }
    catch (InterruptedException ex) {}
  }

  Object produce() { ... }
}
```

using blocking queues - example

```
class Consumer implements Runnable {
  private final BlockingQueue queue;

  Concumer(BlockingQueue q) { queue = q; }

  public void run() {
    try {
      while(true) { consume(queue.take()); }
    }
    catch (InterruptedException ex) {}
  }

  void consume(Object x) { ... }
}
```


agenda

- locks
- conditions
- queues
- synchronizers
- executors

synchronizer: Exchanger

class `Exchanger` provides a synchronization point at which two threads can exchange information:

```
public class Exchanger<E> {

  public Object exchange(E x)
                throws InterruptedException;

  public Object exchange(E x, long t, TimeUnit u)
                throws InterruptedException;
}
```


using exchangers - example

- use an Exchanger to swap buffers between threads
  - thread filling the buffer gets a freshly emptied one when it needs it,
  - handing off the filled one to the thread emptying the buffer

```
class FillAndEmpty {
  Exchanger<Buffer> exchanger = new Exchanger();
  Buffer initialEmptyBuffer = ... a made-up type ...;
  Buffer initialFullBuffer  = ... ;

  void start() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
  }
}
```

using exchangers - example

```
class FillAndEmpty {
 Exchanger<Buffer> exchanger = new Exchanger();

 class FillingLoop implements Runnable {
  public void run() {
    Buffer currentBuffer = initialEmptyBuffer;
    try {
      while (currentBuffer != null) {
        addToBuffer(currentBuffer);
        if (currentBuffer.full())
          currentBuffer = exchanger.exchange(currentBuffer);
      }
    }
    catch (InterruptedException ex) { }
  }
 }
}
```


using exchangers - example

```
class FillAndEmpty {
 Exchanger<Buffer> exchanger = new Exchanger();

 class EmptyingLoop implements Runnable {
   public void run() {
     Buffer currentBuffer = initialFullBuffer;
     try {
       while (currentBuffer != null) {
         takeFromBuffer(currentBuffer);
         if (currentBuffer.empty())
           currentBuffer = exchanger.exchange(currentBuffer);
       }
     }
     catch (InterruptedException ex) { }
   }
 }
}
```

agenda

- locks
- conditions
- queues
- synchronizers
- executors

executor interface

JSR-166 provides an executor interface:

```
public interface Executor {
  void execute(Runnable r);
  Future execute(Callable c, Object arg);
}
```

- an executor is a framework for executing Runnables
  - manages queueing and scheduling of tasks
  - creation and teardown of threads
    - execute in a newly created or an existing thread
    - execute sequentially or concurrently

Callable interface

executors use callables:

```
public interface Callable {
   Object call(Object arg) throws Exception;
}
```

- `Callable` is similar to `Runnable`
  - both are designed for classes whose instances are executed by a thread
  - a Runnable does not return a result and cannot throw a checked exception, but a Callable can

Future interface

execution returns a Future:

```
public interface Future {
   boolean isDone();
   Object get() throws InterruptedException,
                                 ExecutionException;
   Object get(long t, TimeUnit u) throws
            InterruptedException, ExecutionException;
}
```

- represents the result of an asynchronous computation
  - check if the computation is complete
  - retrieve the result of the computation

agenda

- generics
- concurrency utilities
- enum types
- autoboxing

enum type

```
public enum Season { winter, spring, summer, fall }
```

- design rationale:
  - compile-time type safety
  - performance comparable to `int` constants
  - type system provides a namespace for each enum type
    - you don't have to prefix each constant name
  - typesafe constants aren't compiled into clients
    - you can add, reorder or remove constants without recompiling clients
  - printed values are informative
  - enum constants can be used in collections, e.g. as HashMap keys
  - you can add arbitrary fields and methods to an enum class
  - an enum type can be made to implement arbitrary interfaces.

superclass Enum

- all enum types are derived from a predefined superclass

```
public abstract class Enum <T extends Enum<T>>
        implements Comparable<T>, Serializable {
    public final transient int    ordinal;
    public final          String name;
    protected Enum(String name, int ordinal);

    public abstract List<T> family();
    public final boolean equals(Object o);
    public final int hashCode();
    public String toString();
    public final int compareTo(T o);
    protected final Object clone()
         throws CloneNotSupportedException;
    protected final Object readResolve()
         throws ObjectStreamException;
}
```


synthetic fields

- each enum class has some automatically generated fields:
  - an immutable list containing the enum class's values

    `public static List<`*this enum class*`> VALUES;`

    `public final  List<`*this enum class*`> family();`
  - a static factory returning the enum constant to an enum identifier

    `public static <`*this enum class*`> valueOf(String name);`

additional fields and methods & use in switch

```
public enum Coin {
  penny(1), nickel(5), dime(10), quarter(25);

  private final int value;
  public Coin(int value) { this.value = value; }
  public int value() { return value; }
}
```

```
private enum CoinColor { copper, nickel, silver }
```

```
CoinColor color(Coin c) {
  if (c == null) throw new NullPointerException();
  switch(c) {
    case Coin.penny:   return CoinColor.copper;
    case Coin.nickel:  return CoinColor.nickel;
    case Coin.dime:    return CoinColor.silver;
    case Coin.quarter: return CoinColor.silver;
  }
  throw new AssertionError("Unknown coin: " + c);
}
```


methods per enum value

```
public abstract enum Operation {
  plus  {double eval(double x, double y) { return x + y; }},
  minus {double eval(double x, double y) { return x - y; }},
  times {double eval(double x, double y) { return x * y; }},
  div   {double eval(double x, double y) { return x / y; }};

  // Perform arithmetic operation represented by this constant
  abstract double eval(double x, double y);
}
```

```
void f(double x, double y) {
  for (Iterator<Operation> i = VALUES.iterator();
       i.hasNext(); ) {
    Operation op = i.next();
    System.out.println(x+" "+op+" "+y+" = "+op.eval(x, y));
  }
}
```

agenda

- generics
- concurrency utilities
- enum types
- autoboxing

autoboxing

- frequent need to explicitly convert data of primitive type to reference type
  - e.g. adding primitive data to collections
  - explicit conversions are verbose and clutter the code
- add autoboxing to the language
  - allow automatic conversion of data of primitive type to the corresponding wrapper type
- introduce a new conversion (*boxing conversion*)
  - used as part of assignment and method invocation
- no auto-unboxing proposed
  - automatic conversion from wrapper type to primitive type not supported

autoboxing - example

autoboxing →

future
```
Integer i(100);
int j = 0;
i = j;
```

past
```
Integer i(100);
int j = 0;
i = new Integer(j);
```

autoboxing →

future
```
void f(Integer arg)
{...}
f(5);
```

past
```
void f(Integer arg)
{...}
f(new Integer(5));
```

error:
no auto-unboxing →

future
```
void g(int arg)
{ ... }
Integer i(100);
g(i);
```

past
```
void g(int arg)
{ ... }
Integer i(100);
g(i.intValue());
```


references - generics

JCP: JSR 014 - Adding Generic Types to Java
http://www.jcp.org/en/jsr/detail?id=14

Draft Specification (April 27, 2001)
http://java.sun.com/aboutJava/communityprocess/review/jsr014/

Prototype compiler for Generics
http://developer.java.sun.com/developer/earlyAccess/adding_generics/

JCP: JSR 201 - Extending Java with Enumerations, Autoboxing, Enhanced for loops and Static Import
http://www.jcp.org/en/jsr/detail?id=201

references - concurrency utilities

JCP: JSR 166 - Concurrency Utilities
http://www.jcp.org/en/jsr/detail?id=166

Concurrency JSR-166 Interest Site
http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

Overview of package util.concurrent Release 1.3.2.
http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

authors

Angelika Langer
Training & Mentoring
Object-Oriented Software Development in C++ & Java
Munich, Germany
http: **//www.AngelikaLanger.com**

Klaus Kreft
Siemens Business Services, Munich, Germany
Email: **klaus.kreft@siemens.com**