Java 8

Lambdas & Streams

Angelika Langer

http://www.AngelikaLanger.com/

objective

- understand lambda expressions
- learn about method references
- explore the stream API
- get a feeling for its performance model

speaker's relationship to topic

- independent trainer / consultant / author
  - teaching C++ and Java for ~20 years
  - curriculum of a couple of challenging courses
  - JCP observer and Java champion since 2005
  - co-author of "Effective Java" column
  - author of Java Generics FAQ
  - author of Lambda Tutorial & Reference

agenda

- **lambda expressions**
- method references
- a first glance at streams
- intermediate vs. terminal operations
- stateful vs. stateless operations
- flatMap()
- collectors
- parallel streams
- internals of a collector

lambda expressions in Java

- *lambda expressions*
  - formerly known as *closures*

- concept from functional programming languages
  - anonymous method
    - ad-hoc implementation of functionality
  - “code-as-data”
    - pass functionality around (as parameter or return value)
  - superior to (anonymous) inner classes
    - concise syntax + less code + more readable + “more functional”

design goal

- *build better (JDK) libraries*
  - e.g. for easy parallelization on multi core platforms
- collections shall have parallel bulk operations
  - based on fork-join-framework (Java 7)
  - execute functionality on a collection in parallel
    - parallel streams
- separation between "*what* to do" & "*how* to do"
  - user => *what* functionality to apply
  - library => *how* to apply functionality
    (parallel/sequential, lazy/eager, out-of-order)

for-loop today

```
void checkBalance(List<Account> accList) {
   for (Account a : accList)
      if (a.balance() < threshold) a.alert();
}
```

- new **`for`**-loop style
  - actually an external **`iterator`** object is used:

```
Iterator iter = accList.iterator();
while (iter.hasNext()) {
  Account a = iter.next();
  if (a.balance() < threshold) a.alert();
}
```

- code is inherently serial
  - traversal logic is fixed
  - iterate from beginning to end

for-loop in Java 8

- collections provide a `forEach()` operation

```
void checkBalance(List<Account> accList) {
  accList.forEach( (Account a) -> {
      if (a.balance() < threshold) a.alert();
  } );
}
```

```
interface Iterable<T> ... {
     ...
  void forEach(Consumer<? super T> action);
     ...
}
```

```
interface Consumer<T> {
  void accept(T a);
} …
```

- `forEach()`'s iteration not inherently serial
  - traversal order defined by `forEach()`'s implementation
  - burden of parallelization put on library developer

what is a lambda?

```
Consumer<Account> block
    = (Account a) -> { if (a.balance() < threshold) a.alert(); };
```

- right side: lambda expression
- intuitively
  - a lambda is "something functional"
    - takes an `Account`
    - returns nothing (`void`)
    - throws no checked exception
    - has an implementation / body
  - kind of a *function type*: **`(Account)->void`**
- Java's type system does not have *function types*

functional interface = target type of a lambda

```
interface Consumer<T> { public void accept(T a); }

Consumer<Account> adder = a -> a.addInterest();
```

- lambdas are converted to *functional interfaces*
  - function interface ≈ interface with one abstract method
  - parameter type(s), return type, checked exception(s) must match
  - functional interface's name + method name are irrelevant
- conversion requires type inference
  - lambdas may only appear where target type can be inferred from enclosing context
  - e.g. variable declaration, assignment, method/constructor arguments, return statements, cast expression, …

lambda expressions & functional interfaces

- functional interfaces

```
interface Consumer<T> { void accept(T a); }
interface MyInterface { void applyToAccount(Account a); }
```

- conversions

```
Consumer<Account> block = a -> a.addInterest();

MyInterface mi = a -> a.addInterest();

mi = block;
```

error: types are not compatible

- problems

error: cannot infer target type

```
Object o1 =
            a -> a.addInterest();

Object o2 = (Consumer<Account>)
            a -> a.addInterest();
```

formal description

```
lambda = ArgList "->" Body

ArgList = Identifier
        | "(" Identifier [ "," Identifier ]* ")"
        | "(" Type Identifier [ "," Type Identifier ]* ")“

Body = Expression
     | "{" [ Statement ";" ]+ "}"
```

syntax samples

argument list

```
(int x, int y) -> { return x+y; }
        (x, y) -> { return x+y; }
            x  -> { return x+1; }

() -> { System.out.println(“I am a Runnable”); }
```

body

```
// single statement or list of statements
a -> {
       if (a.balance() < threshold) a.alert();
     }
// single expression
a -> (a.balance() < threshold) ? a.alert() : a.okay()
```

return type (is always inferred)

```
(Account a) -> { return a; }            // returns Account
()          ->          5               // returns int
```

local variable capture

- binding to local variables allowed
  - but local variable is implicitly final
  - same as with inner classes

```
void f() {
  int cnt = 16;
  Runnable r = () -> { System.out.print("  " + cnt); };
  pool.execute(r);
  cnt++;
}
```

error: cnt is read-only

binding to fields

- binding to fields allowed
  - fields are NOT implicitly final
  - same as with inner classes

```
class SomeClass {
  private int cnt = 16;
  private void f() {
    Runnable r = () -> { System.out.print("  " + cnt); };
    pool.execute(r);
    cnt++;                                    fine
  }
}
```

  - non-deterministic output (if executed repeatedly)

```
20 25 25 39 45 45 45 106 106 106 106 106 106 106 106 106 116 116 116
```

lexical scoping

- lambda body scoped in enclosing method
- effect on local variables:
  - capture works as shown before
  - no shadowing of lexical scope

lambda

```
int i = 16;
Runnable r = () -> { int i = 0;
                     System.out.println("i is: " + i); };
```

error

- different from inner classes
  - inner class body is a scope of its own

inner class

```
final int i = 16;
Runnable r = new Runnable() {
  public void run() { int i = 0;
                      System.out.println("i is: " + i); }
  };
```

fine

agenda

- lambda expressions
- **method references**
- a first glance at streams
- intermediate vs. terminal operations
- stateful vs. stateless operations
- flatMap()
- collectors
- parallel streams
- internals of a collector

method references

- a concise notation for certain lambdas

lambda expression:

```
accounts.forEach(a -> a.addInterest());
```

method reference:

```
accounts.forEach(Account::addInterest);
```

- advantage (over lambdas)
  - reuse existing method

- needs type inference context for target type
  - similar to lambda expressions

method references

various forms of method references ...

- static method: `Type::MethodName`
  - e.g. `System::currentTimeMillis`
- constructor: `Type::new`
  - e.g. `String::new`
- non-static method w/ unbound receiver: `Type::MethodName`
  - e.g. `String::length`
- non-static method w/ bound receiver: `Expr::Method`
  - e.g. `System.out::println`

reference to instance method

- situation
  - instance method needs an instance on which it can be invoked
  - called: *receiver*

- two possibilities
  - receiver is explicitly provided in an expression
    - called: *bound receiver*
  - receiver is implicitly provided from the context
    - called: *unbound receiver*

bound receiver

- syntax

```
expression "::" identifier
```

- example

```
List<String> stringList = ... ;
stringList.forEach(System.out::print);
```

$\underbrace{\texttt{System.out}}_{\texttt{receiver}}$

  – with lambda

```
stringList.forEach((String s) -> System.out.print(s));
```

unbound receiver

- syntax

```
type "::" identifier
```

- example

```
Stream<String> stringStream = ... ;
stringStream.sorted(String::compareToIgnoreCase);
```

  - with lambda

```
stringStream.sorted(
      (String s1, String s2) -> s1.compareToIgnoreCase(s2));
```

compare these situations

- example 1:

```
Stream<Person> psp = ... ;
psp.sorted(Person::compareByName);
```

  – with

```
class Person {
  public static int compareByName(Person a, Person b) { … }
}
```

- example 2:

```
Stream<String> stringStream = ... ;
stringStream.sorted(String::compareToIgnoreCase);
```

  – with

```
class String {
  public int compareToIgnoreClase(String str) { … }
}
```

note

- method references do not specify argument type(s)

- compiler infers from context
  - which overloaded version fits

```
List<String> stringList = ... ;
stringList.forEach(System.out::print);
```

→ `void print(String s)`

- resort to lambda expressions
  - if compiler fails or a different version should be used

wrap-up

- lambdas express functionality
  - invented to support new APIs in JDK 8
- lambdas are converted to functional interface types
  - needs type inference context
- lexical scoping
  - lambda is part of its enclosing scope
  - names have same meaning as in outer scope
- lambdas can access fields and (final) local variables
  - mutation is error-prone
- method references
  - even more concise than lambdas

agenda

- lambda expression
- method references
- **a first glance at streams**
- intermediate vs. terminal operations
- stateful vs. stateless operations
- flatMap()
- collectors
- parallel streams
- internals of a collector

bulk data operations for collections in Java 8

- extension of the JDK collections

- with …
  - functional view: sequence + operations
  - object-oriented view: collection + internal iteration
  - for-each/filter/map/reduce for Java

  - for-each
    apply a certain functionality to each element of the sequence

```
accounts.forEach(a -> a.addInterest());
```

bulk data operations (cont.)

- filter
  build a new sequence that is the result of a filter applied to each element in the original collection

```
accounts.filter(a -> a.balance() > 1_000_000);
```

- map
  build a new sequence, where each element is the result of a mapping from an element of the original sequence

```
accounts.map(a -> a.balance());
```

- reduce
  produce a single result from all elements of the sequence

```
accounts.map(a -> a.balance())
        .reduce(0, (b1, b2) -> b1+b2);
```

streams

- interface `java.util.stream.Stream<T>`
  - supports `forEach`, `filter`, `map`, `reduce`, and more
- two new methods in `java.util.Collection<T>`
  - `Stream<T> stream()`, sequential functionality
  - `Stream<T> parallelStream()`, parallel functionality

```
List<Account> accountCol = … ;

Stream<Account> accounts = accountCol.stream();

Stream<Account> millionaires =
                accounts.filter(a -> a.balance() > 1_000_000);
```

more about streams and their operations

- streams do not store their elements
  - not a collection, but created from a collection, array, …
  - view/adaptor of a data source (collection, array, …)

- streams provide functional operations

  `forEach`, `filter`, `map`, `reduce`, …
  - applied to elements of underlying data source

streams and their operations (cont.)

- actually applied functionality is two-folded
  - user-defined: functionality passed as parameter
  - framework method: stream operations

- separation between "*what* to do" & "*how* to do"
  - user => *what* functionality to apply
  - library => *how* to apply functionality
    (parallel/sequential, lazy/eager, out-of-order)

```
accounts.filter(a -> a.balance() > 1_000_000);

accounts.forEach(a -> a.addInterest());
```

parameters of stream operations …

… can be

- lambda expressions
- method references
- (inner classes)

- example: `forEach`

```
void forEach(Consumer<? super T> consumer);
```

```
public interface Consumer<T> {
    public void accept(T t);
}
```

```
accounts.forEach((Account a) -> { a.addInterest(); });
accounts.forEach(a -> a.addInterest());
accounts.forEach(Account::addInterest);
```

`Stream.map()` - possible

- `balance` is of primitive type `double`

```
public interface Stream<T> ... {
     ...
  <R> Stream<R> map(Function<? super T, ? extends R> mapper);
     ...
}
```

```
public interface Function<T, R> {
   public R apply(T t);
}
```

```
Stream<Double> balances = accounts.map(a -> a.balance());
```

  – triggers auto-boxing

Stream.mapToDouble() - preferred

- avoid auto-boxing

```
public interface Stream<T> ... {
     ...
  DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);
     ...
}
```

```
public interface ToDoubleFunction<T> {
   public double applyAsDouble(T t);
}
```

```
DoubleStream balances = accounts.mapToDouble(a -> a.balance());
```

primitive streams

- streams for elements with primitive type:
  `IntStream, LongStream, DoubleStream`

- reason: performance

- no stream types for `char` and `float`
  - use stream type of respective ‘bigger’ primitive type
    - `IntStream` for `char`, and `DoubleStream` for `float`
  - e.g. interface `CharSequence` contains:

```
IntStream chars();
```

how to obtain a stream ?

- `java.util.Collection<T>`
  - `Stream<T> stream()`, sequential functionality
  - `Stream<T> parallelStream()`, parallel functionality
- `java.util.Arrays`
  - `static <T> Stream<T> stream(T[] array)`
  - plus overloaded versions (primitive types, …)
- many more ...
- collections allow to obtain a parallel stream directly
  - in all other cases use stream's method: `parallel()`

```
Arrays.stream(accArray).parallel().forEach(Account::addInterest);
```

agenda

- lambda expression
- method references
- a first glance at streams
- **intermediate vs. terminal operations**
- stateful vs. stateless operations
- flatMap()
- collectors
- parallel streams
- internals of a collector

intermediate / terminal

- there are stream operations …
  - that produce a stream again: `filter()`, `map()`, …
    - intermediate (lazy)
  - that do something else: `forEach()`, `reduce()`, …
    - terminal (eager)

```
double sum = accountCol.stream()
                       .mapToDouble(a -> a.balance())
                       .reduce(0, (b1, b2) -> b1+b2);
```

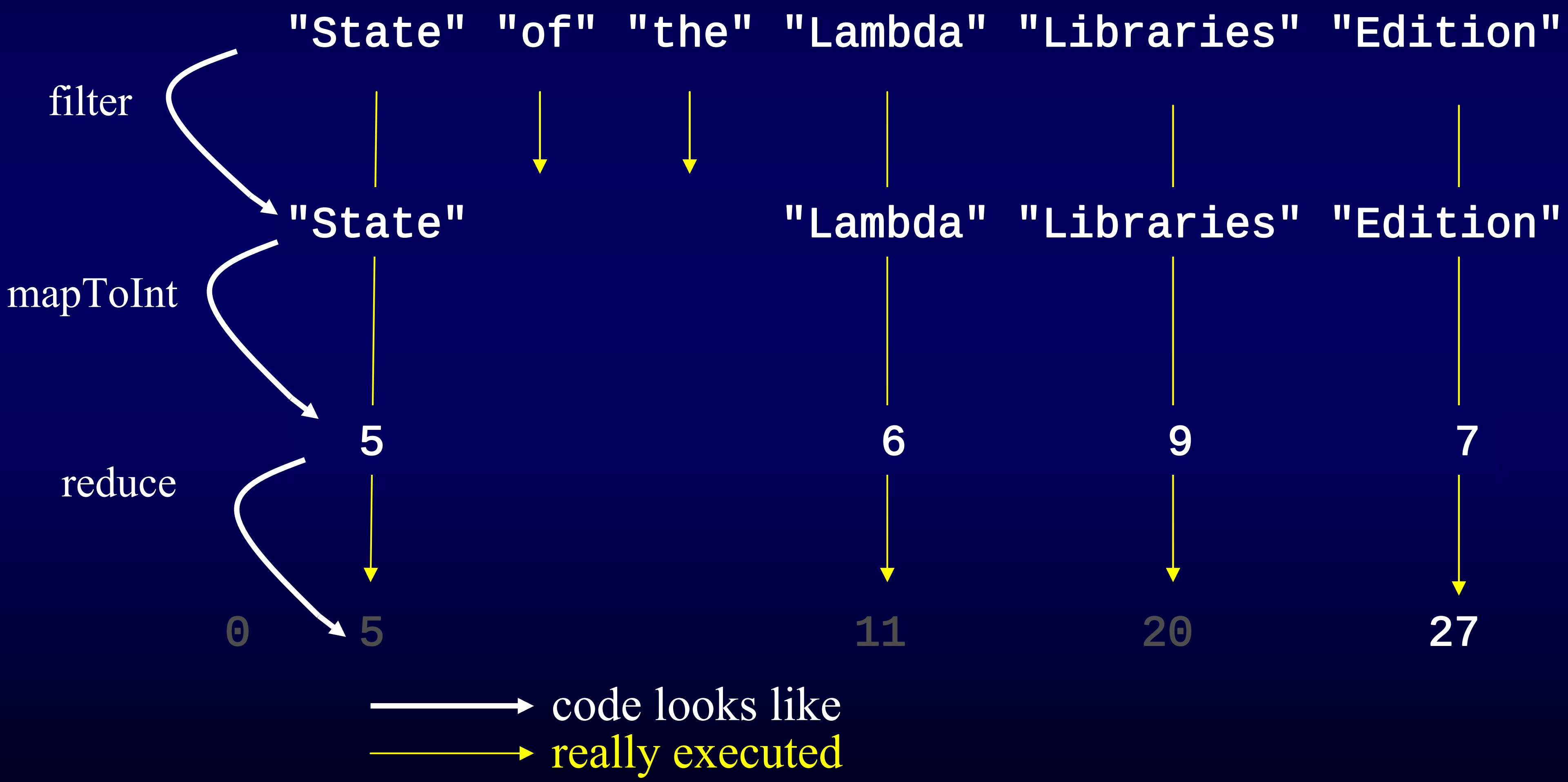
intermediate / terminal - example

```
String[] txt = {"State","of","the","Lambda","Libraries","Edition"};

IntStream is = Arrays.stream(txt).filter(s -> s.length() > 3)
                                 .mapToInt(s -> s.length());

int sum = is.reduce(0, (l1, l2) -> l1 + l2);
```

- `filter()` and `mapToInt()` return streams
  => intermediate
- `reduce()` returns `int`
  => terminal
- intermediate stream not evaluated
  - until a terminal operation is invoked

stream is evaluated when terminal op. is invoked

```
Arrays.stream(txt).filter(s -> s.length() > 3)
                  .mapToInt(s -> s.length())
                  .reduce(0, (l1, l2) -> l1 + l2);
```



reason: performance

- code optimization
- no buffering of intermediate stream results
- easier to handle parallel streams

terminal operations ≈ consuming operations

- terminal operations are consuming operations

```
IntStream is = Arrays.stream(txt).filter(s -> s.length() > 3)
                                 .mapToInt(s -> s.length());

is.forEach(l -> System.out.print(l +", "));
System.out.println();

int sum = is.reduce(0, (l1, l2) -> l1 + l2);
```

```
5, 6, 9, 7,
Exception in thread "main" java.lang.IllegalStateException:
   stream has already been operated upon or closed
```

recommendation: use fluent programming

- best avoid reference variables to stream objects
- instead:
  - construct the stream
  - apply a sequence of intermediate stream operations
  - terminate with an terminal stream operation
  - one statement
  - *fluent programming*
    - build next operation on top of result of previous one

```
int sum = Arrays.stream(txt).filter(s -> s.length() > 3)
                             .mapToInt(s -> s.length())
                             .reduce(0, (l1, l2) -> l1 + l2);
```

agenda

- lambda expression
- method references
- a first glance at streams
- intermediate vs. terminal operations
- **stateful vs. stateless operations**
- flatMap()
- collectors
- parallel streams
- internals of a collector

stateless intermediate operations

- *stateless*ness means ...
  - need only stream element + functionality
  - in order to perform operation
- examples:
  - `filter()`, `map()`, ...
- e.g. `filter()`
  - predicate applied to the element evaluates to
    - `true` – element goes to the next stage
    - `false` – element gets dropped
- easy to handle
  - even in parallel execution

stateful intermediate operations

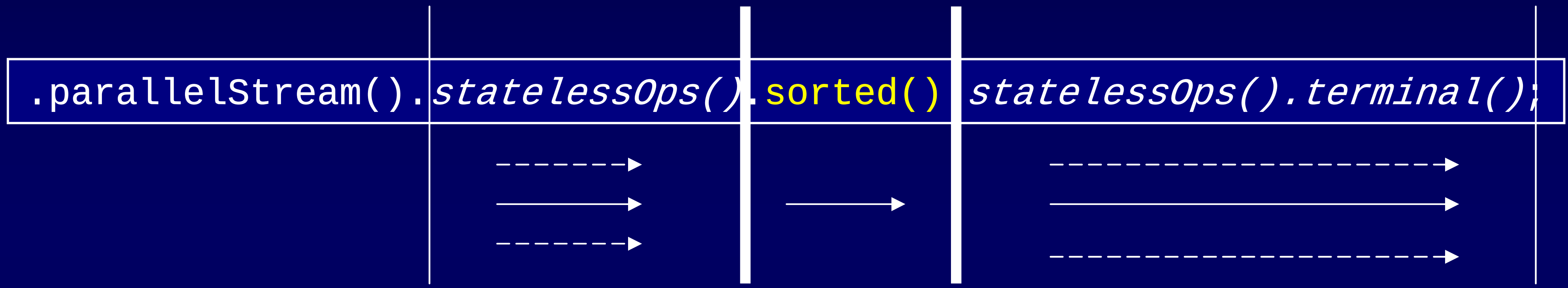
- *statefulness* means ...
  - need stream element + functionality + additional state
  - in order to perform operation
- stateful intermediate

```
Stream<T> limit(long maxSize)
Stream<T> skip(long start)
Stream<T> distinct()
Stream<T> sorted(Comparator<? super T> c)
Stream<T> sorted()
```

- e.g. `distinct()`
  - element goes to the next stage, if it hasn't already appeared before
- not so easy to handle
  - especially in parallel execution

sorted() is stateful

- `sorted()`
  - is the most complex and restrictive stateful operation



- two barriers => stream is sliced
  - stream is buffered at the end of each slice (at the barrier)
  - downstream slice is started after upstream slice has finished
  - i.e. processing is done differently !

overhead of stateful operations

- for sequential streams

  - all operations (except `sorted()`)
    - behave like stateless operations
    - i.e., no barriers, but additional state

  - `sorted()`
    - only operation with extreme slicing
    - two barriers
      - stores all elements in an array (or `ArrayList`) + sorts it
    - uses only one thread in the sorting slice
      - even in parallel case

overhead of stateful operations

- for parallel streams
  - `distinct()`
    - one barrier
      - stores all elements in a `LinkedHashSet`
      - or `ConcurrentHashMap` (if unordered)
  - `limit() / skip()`
    - adjust the spliterator (if stream size is known)
    - no spliterator adjustment after filter (as size is unknown)
      - counting instead => expensive in parallel case

distinct() is stateful

- `distinct()`
  - less restrictive stateful operation

```
.parallelStream().statelessOps().distinct().statelessOps().terminal();
```

- one barrier => only two slices
  - resulting stream is buffered at the end of `distinct()`

wrap-up

- distinction between intermediate & terminal operations
  – deferred execution in a pipeline
  – triggered by terminal operation
- stateful operations introduce barriers
  – expensive in case of parallel execution

agenda

- lambda expression
- method references
- a first glance at streams
- intermediate vs. terminal operations
- stateful vs. stateless operations
- **flatMap()**
- collectors
- parallel streams
- internals of a collector

flatMap()

- a classic operation from functional programming

```
<R> Stream<R>
flatMap(Function<? super T,
                 ? extends Stream<? extends R>> mapper)
```

  - maps an element to a stream of elements

  - result is the concatenation of these streams (i.e. `Stream<R>`)
    - not a stream of streams (i.e. not `Stream<Stream<R>>`)
    - hence the term "*flat*"-map
    - good, because `Stream<R>` can easily be processed

  - corresponding methods that flat-map to a primitive stream

flatMap() - example

- count all non-space characters in a text file

```
try (BufferedReader in
       = new BufferedReader(new FileReader("text.txt"))){
  long cnt = in.lines()                                  // Stream<String>
       .flatMapToInt(String::chars)                      // IntStream
       .filter(Character::isSpaceChar.negate())          // IntStream
       .count();                                         // long
   System.out.println("non-spaces="+cnt);
} catch (IOException | UncheckedIOException e) {
   e.printStackTrace();
}
```

  – create `Stream<String>` via `Reader::lines`
  – map all lines to a single character stream via `String::chars`
  – eliminate all space characters
    - needs opposite of `Character::isSpaceChar`
  – count remaining non-spaces

type inference glitch

- create opposite of `Character::isSpaceChar`
  – via `negate()` in interface `Predicate`
- `Character::isSpaceChar.negate()` does not compile
  – *method invocation* is no inference context
- must insert cast to predicate type
  – *cast* is an inference context

```
...
in.lines()
  .flatMapToInt(String::chars)
  .filter((IntPredicate)Character::isSpaceChar).negate() )
  .count();
...
```

different view

- `flatMap()` is the flexible cousin of `map()`
  - `map()`
    - maps each element to exactly one element
  - `flatMap()`
    - maps each element to none, one, or multiple element(s)

- powerful
  - especially with user-defined mappers

agenda

- lambda expression
- method references
- a first glance at streams
- intermediate vs. terminal operations
- stateful vs. stateless operations
- flatMap()
- **collectors**
- parallel streams
- internals of a collector

collect()

- terminal operation

  `<R> R collect(Collector<? super T, A, R> collector)`

  – collect stream elements into ‘something’
    - looks relatively simple and innocent, but is powerful and complex

  – kind of a ‘kitchen sink’ approach
    - not `collect()` does the work, but the `Collector`

  - `java.util.stream.Collectors`
    - is a class with 30+ factory methods for collectors
    - look here before implementing your own collector

collect to a collection

- factory methods in `Collectors`

```
Collector<T, ?, List<T>> toList()

Collector<T, ?, Set<T>>  toSet()

Collector<T, C>
toCollection(Supplier<C> collectionFactory)
```

  - work with parallel streams, too
    - can handle unsynchronized collections

- example: numbers 0 … 31 into an `ArrayList`

```
List<Integer> ints
= IntStream.range(0,32)
           .boxed()
           .collect(toCollection(ArrayList::new));
```

joining() collectors

- factory methods in `Collectors`

  `Collector<CharSequence, ?, String> joining()`

  – concatenate stream of `CharSequence` to a `String`

  – use a `StringBuilder` internally

    - more efficient than ….`reduce("", String::concat)`

  – further versions with delimiter, prefix + suffix

- example: string representations of numbers 0 … 7

  – concatenated into one string

```
System.out.println(
   IntStream.range(0,8)
            .mapToObj(Integer::toString)
            .collect(Collectors.joining(" ","-> "," <-")));
```

```
-> 0 1 2 3 4 5 6 7 <-
```

collect to a map

- factory methods in `Collectors`

```
Collector<T,?,Map<K,U>>
toMap(Function<? super T, ? extends K> keyMapper,
      Function<? super T, ? extends U> valueMapper)
```

  - further versions with `mergeFunction`
    - to resolve collisions between values associated with same key
  - and with `mapSupplier`
    - e.g. `TreeMap::new`

```
String[] txt = {"State","of","the","Lambda","Libraries","Edition"};
Map<String,Integer> lengths
= Arrays.stream(txt)
        .collect(Collectors.toMap(s->s, String::length));

System.out.println(lengths);
```

```
{the=3, State=5, of=2, Libraries=9, Lambda=6, Edition=7}
```

grouping collectors

- factory methods in `Collectors`

```
Collector<T,?,Map<K,List<T>>>
groupingBy(Function<? super T,? extends K> classifier)
```

  – further versions for concurrent grouping, with map factory, and with downstream collector

- example:

```
String[] txt = { "one", "two", "three", "four", "five", "six" };

Map<Integer,List<String>> lengths
= Arrays.stream(txt)
        .collect(Collectors.groupingBy(String::length));

System.out.println(lengths);
```

```
{3=[one, two, six], 4=[four, five], 5=[three]}
```

partitioning collectors

- factory methods in `Collectors`
  ```
  Collector<T,?,Map<Boolean,List<T>>>
  partitioningBy(Predicate<? super T> predicate)
  ```
  - further versions with map factory and downstream collector

- example:

```
String[] txt = { "one", "two", "three", "four", "five", "six" };

Map<Boolean,List<String>> lengthLT4 = Arrays.stream(txt)
        .collect(Collectors.partitioningBy(s -> s.length() < 4));

System.out.println(lengthLT4);
```

```
{false=[three, four, five], true=[one, two, six]}
```

grouping - example

- count space and non-space characters in one pass through a text file

```
try (BufferedReader in
      = new BufferedReader(new FileReader("text.txt"))) {
  Map<Boolean,List<Integer>> map = inFile
   .lines()                               // Stream<String>
   .flatMapToInt(String::chars)           // IntStream
   .boxed()                               // Stream<Integer>
   .collect(Collectors.partitioningBy     // Map<Boolean,List<Integer>>
                    (Character::isSpaceChar));
   int chars  = map.get(false).size();
   int spaces = map.get(true).size();
} catch (IOException | UncheckedIOException e) {
   e.printStackTrace();
}
```

  - group by `isSpaceChar()`
    - yields `Map<Boolean, List<Integer>>`
    - associates true => list of space characters
false => list of non-space characters

collectors w/ downstream collectors

- factory methods in `Collectors`

```
Collector<T,?,Map<K,D>>
groupingBy(Function<? super T,? extends K> classifier,
           Collector<? super T,A,D> downstream)
```

- examples of downstream collectors

```
Collector<T,?,Optional<T>>
maxBy(Comparator<? super T> comparator)

Collector<T,?,Long> counting()

Collector<T,?,Optional<T>>
reducing(BinaryOperator<T> op)
```

grouping example revisited

- use `counting()` downstream collector

```
...
Map<Boolean,Long> map = inFile
    .lines()                                // Stream<String>
    .flatMapToInt(String::chars)            // IntStream
    .boxed()                                // Stream<Integer>
    .collect(Collectors.partitioningBy      // Map<Boolean,Long>
        (Character::isSpaceChar,Collectors.counting()));

long chars  = map.get(false);
long spaces = map.get(true);
...
```

  - classify by `isSpaceChar()`
    - yields `Map<Boolean, List<Integer>>`
  - then count elements in each list
    - yields `Map<Boolean, Long>`

wrap-up

- `flatMap()` is a flexible version of `map()`
  - flattens "stream of streams" to a plain stream

- collectors place results into a sink
  - sophisticated collectors for classification
  - downstreams collectors for result processing after classification

agenda

- lambda expression
- method references
- a first glance at streams
- intermediate vs. terminal operations
- stateful vs. stateless operations
- flatMap()
- collectors
- **parallel streams**
- internals of a collector

parallel execution - another example

- ... to illustrate implementation details

```
int[] ints = new int[64];
ThreadLocalRandom rand = ThreadLocalRandom.current();
for (int i=0; i<SIZE; i++) ints[i] = rand.nextInt();

Arrays.stream(ints).parallel()
      .reduce((i,j) -> Math.max(i,j))
      .ifPresent(m->System.out.println(“max is: ” + m));
```

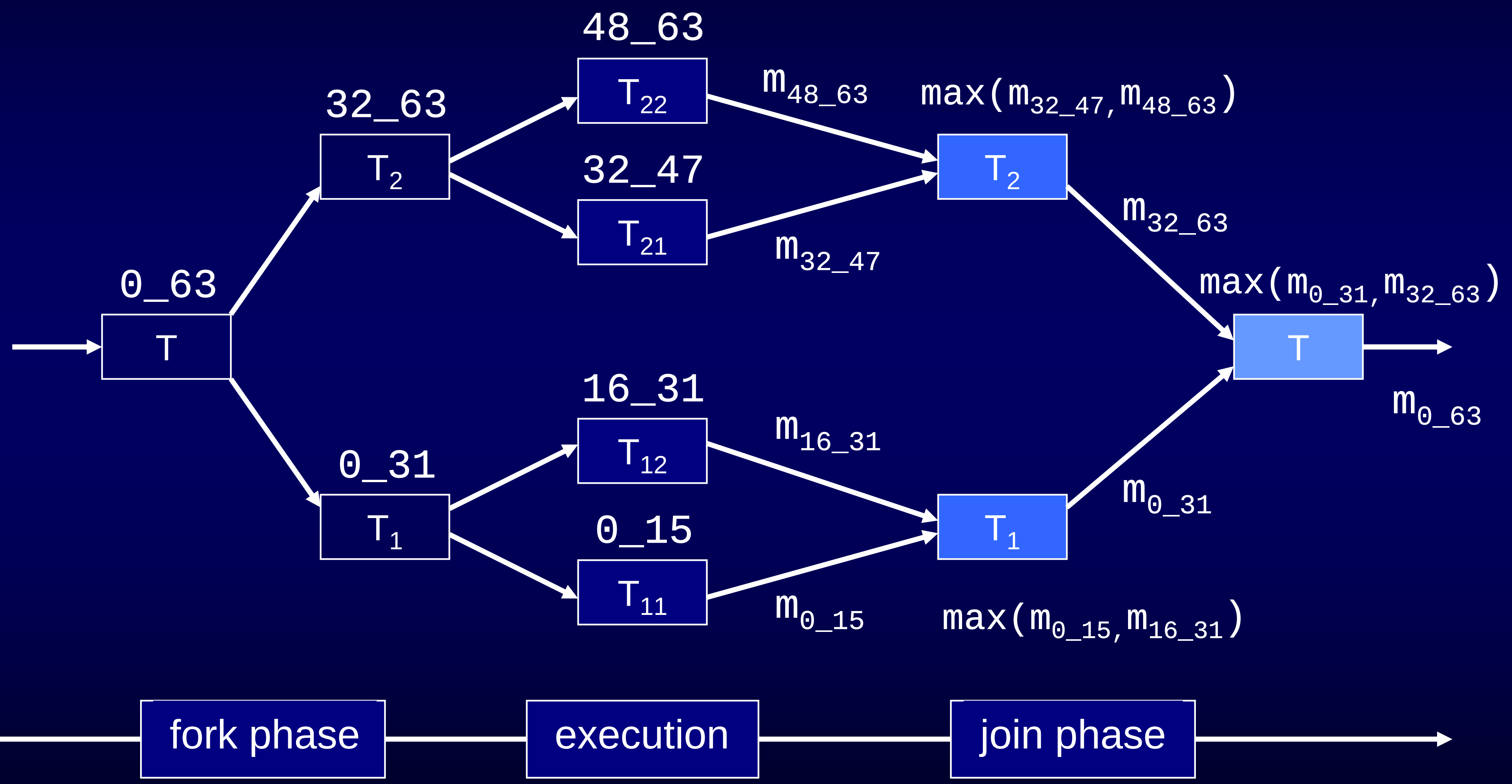
- find (in parallel) the largest element in an `int` array
  - can be implemented shorter via `max()`
  - `reduce()` is more illustrative:
    - points out: one comparison with each element
- `parallelStream()`'s functionality is based on fork-join framework

fork-join tasks

- original task is divided into two sub-tasks
  - by splitting stream source into two parts
    - original task's result is based on sub-tasks' results
    - sub-tasks are divided again => *fork phase*
- at a certain depth, partitioning stops
  - tasks at this level (leaf tasks) are executed
  - *execution phase*
- completed sub-task results are 'combined'
  - to super-task results
  - *join phase*

find largest element with parallel stream

```
reduce((i,j) -> Math.max(i,j));
```



parallel streams + `forEach()`

- what if the terminal operation is `forEach()` ?

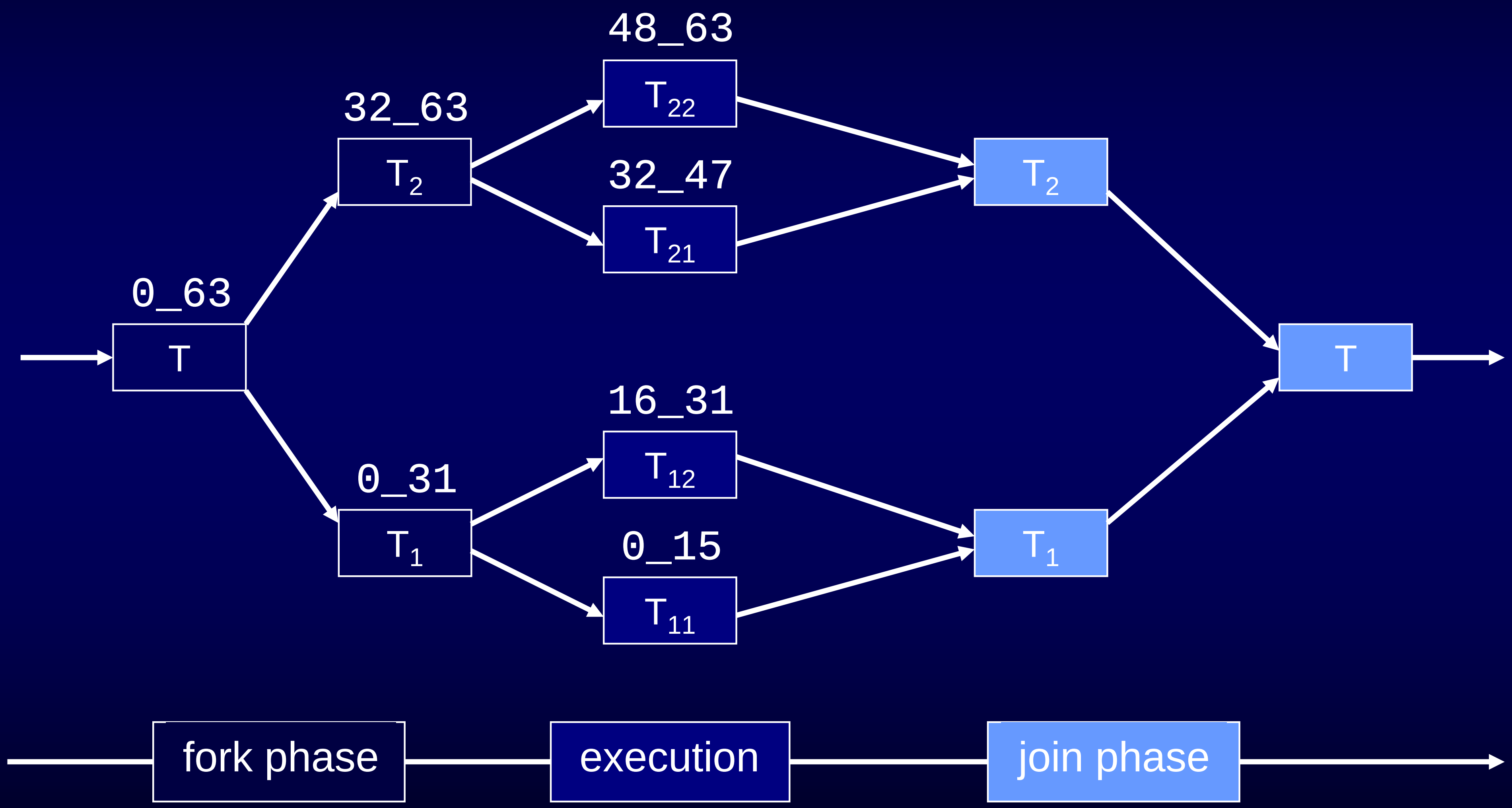
- example:

```
int[] ints = new int[64];
ThreadLocalRandom rand = ThreadLocalRandom.current();
for (int i=0; i<SIZE; i++) ints[i] = rand.nextInt();

Arrays.stream(ints).parallel()
      .forEach(i -> System.out.println("value is: " + i)));
```

=> rudimentary join phase

parallel `forEach()`

```
forEach(i -> System.out.println("value is: " + i)));
```



parallel streams + intermediate operations

what if ...

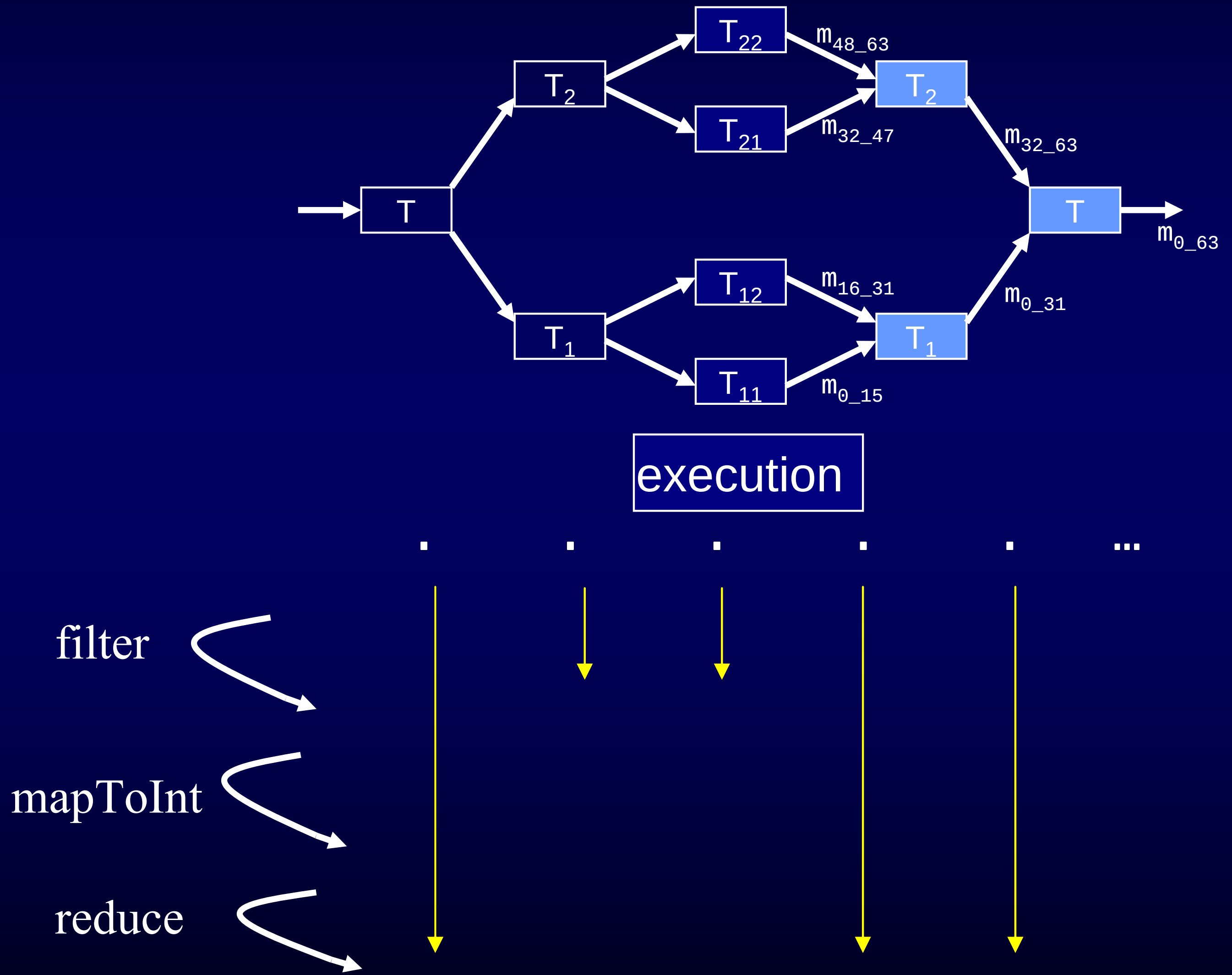
... stream contains upstream (stateless) intermediate operations ?

```
....parallel().filter(...)
              .mapToInt(...)
              .reduce((i,j) -> Math.max(i,j));
```

– when/where are intermediate operations applied to the stream ?

parallel intermediate ops + reduce()

```
filter(...).mapToInt(...).reduce((i,j) -> Math.max(i,j));
```



java.util.Spliterator<T>

- splitter + iterator = spliterator
  - interface with 8 methods

- main functionality:
  - split a stream source into 2 (more or less) equally sized halves
    - the *splitter part*

      `Spliterator<T> trySplit()`
  - sequentially apply execution functionality to each stream element
    - the *iterator part*

      `void forEachRemaining(Consumer<? super T> action)`

spliterator usage

- each type of stream source has its own spliterator type
  - knows how to split and iterate the source
  - e.g. `ArrayList.ArrayListSpliterator`
- parallel streams
  - need splitting (and `Spliterator`)
- sequential streams
  - do <u>not</u> need splitting (and `Spliterator`)
  - but also use it for consistency reasons

reduction with accumulator & combiner

- complex stream operation for reduction to a different type

```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

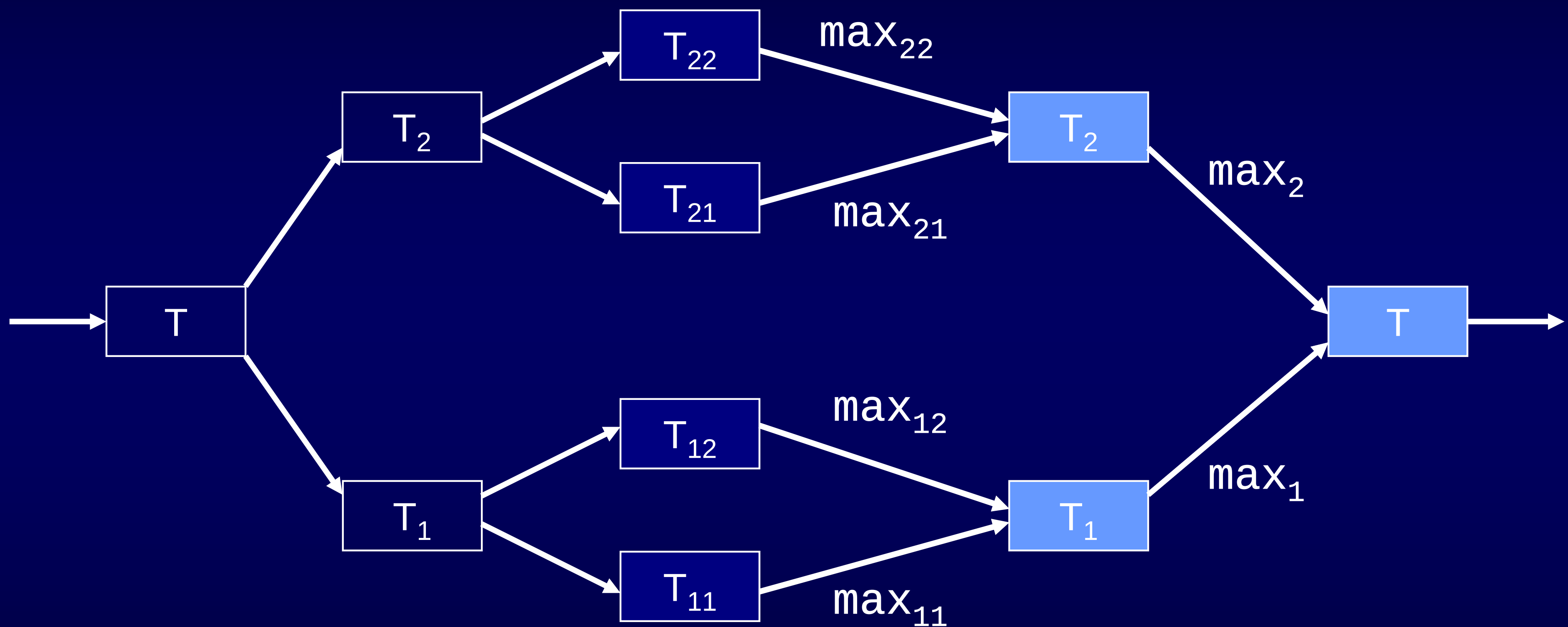
  - `T` is the element type and `U` the result type

- example: reduce `ints` to `double` (their maximum square root)

```
int[] ints = new int[64];
ThreadLocalRandom rand = ThreadLocalRandom.current();
for (int i=0; i<SIZE; i++) ints[i] = rand.nextInt();

double max = Arrays.stream(ints).parallel().boxed()
    .reduce(Double.MIN_VALUE,                           // identity
            (d,i)->Math.max(d,Math.sqrt(Math.abs(i))),  // accumulator
            Math::max                                   // combiner
    );
System.out.format("max is: "+max);
```

role of accumulator & combiner

`... .reduce(` *identity*`,` *accumulator*`,` *combiner* `);`



– accumulator used in "execution" phase (add `int` to `double`)

– combiner used in "join" phase (add two `double`s)

combiner not used for sequential case

```
double max = Arrays.stream(ints).parallel().boxed()
    .reduce(Double.MIN_VALUE,
            (d,i)->Math.max(d,Math.sqrt(Math.abs(i))),   <= execution
            Math::max                                    <= join
    );
```

- combiner must be supplied even for sequential reduce
  - but is not used
  - there is no `reduce()` operation without it
  - using `null` (as 3rd argument) gives a `NullPointerException`
  - idea: same for sequential and parallel operations

the more obvious solution

- use `map()` and `max()` operations

reduction

```
double max = Arrays.stream(ints).parallel().boxed()
    .reduce(Double.MIN_VALUE,
            (d,i)->Math.max(d,Math.sqrt(Math.abs(i))),
            Math::max
    );
```

mapping

```
double max = Arrays.stream(ints).parallel()
                .map(Math::abs)
                .mapToDouble(Math::sqrt)
                .max()
                .getAsDouble();
```

agenda

- lambda expression
- method references
- a first glance at streams
- intermediate vs. terminal operations
- stateful vs. stateless operations
- flatMap()
- collectors
- parallel streams
- **internals of a collector**

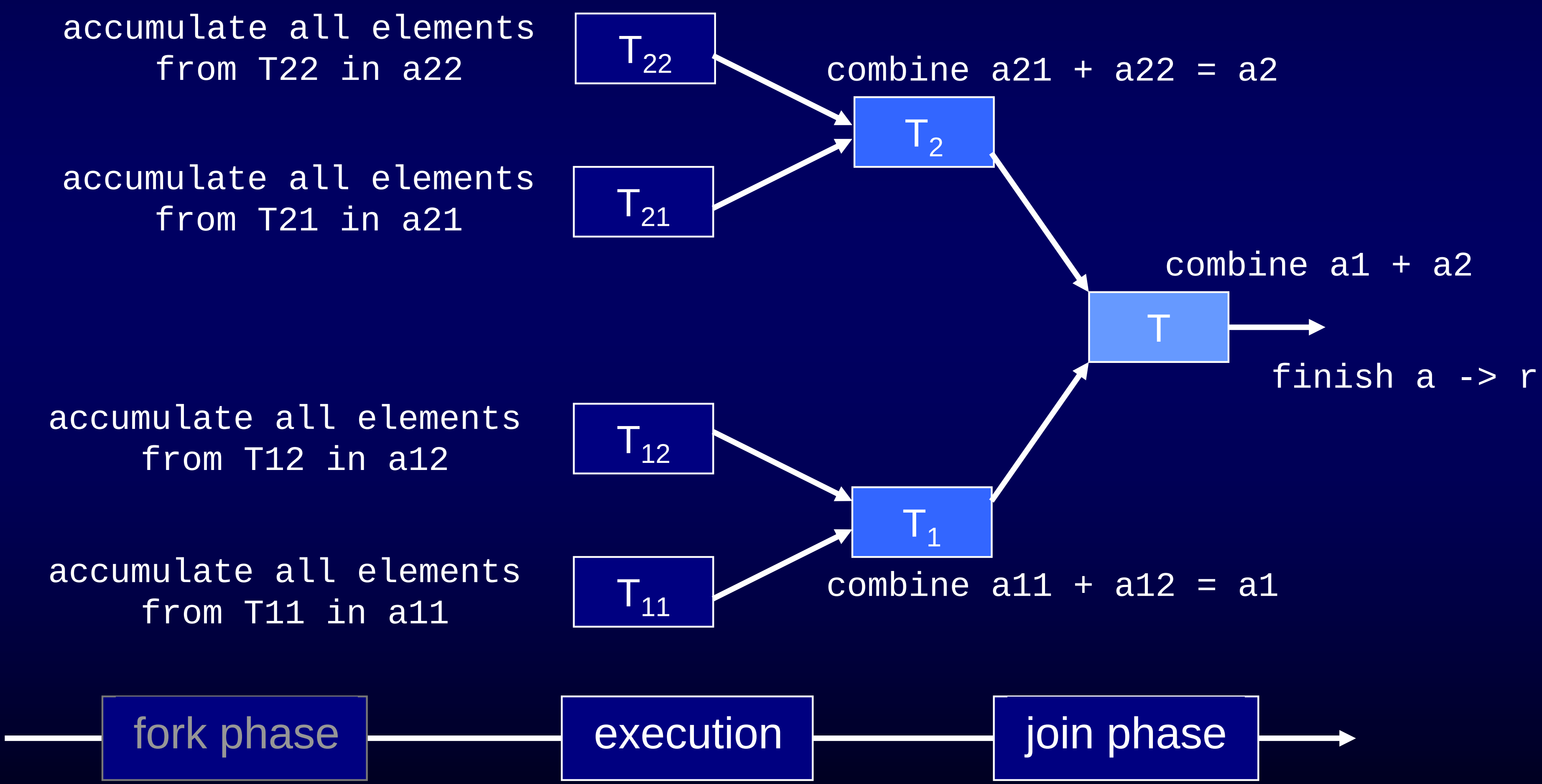
internals of a collector

- a collector is an implementation of the `Collector` interface
  - see examples in a minute

- three different collect algorithms
  - for sequential streams
    - accumulating collect
  - for parallel streams
    - reducing collect
    - concurrently accumulating collect

reducing collect

- fully parallel
  - although `ArrayList` and `HashSet` are not thread-safe



collector for JDK collections

- abstract methods in interface `Collector`
  - and their implementations (for `toList()` and `toSet()`)

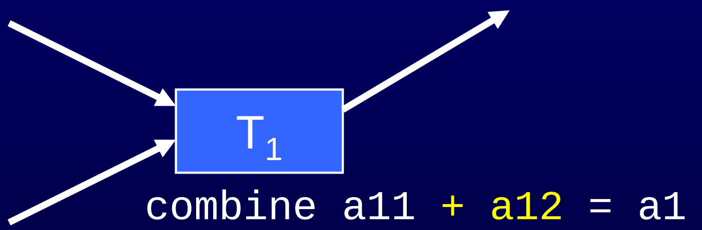
```
Supplier<A> supplier();
          [Collection<T>]::new
BiConsumer<A,T> accumulator();
          Collection<T>::add
BinaryOperator<A> combiner();
          (c1, c2) -> { c1.addAll(c2); return c1; }
Function<A,R> finisher()
          c -> (Collection<T>) c
Set<Characteristics> characteristics();
          IDENTITY_FINISH, not: CONCURRENT, UNORDERED
```

(ArrayList / HashSet)

reducing collect - evaluation

- advantage
  - no locks
  - works for non-thread-safe abstractions
  - (stateless) intermediate operations executed in parallel

- disadvantage
  - algorithmic overhead compared to sequential
  - more `addAll()` invocations (for combine)



accumulating collect

- on sequential streams
  - only the *forEachRemaining* part of the spliterator is used
  - only the *accumulator* part of the collector is used

- on parallel streams
  - the full spliterator functionality is used
  - only the *accumulator* part of the collector is used

- an example of a concurrent accumulation:
  - `toConcurrentMap()` collector

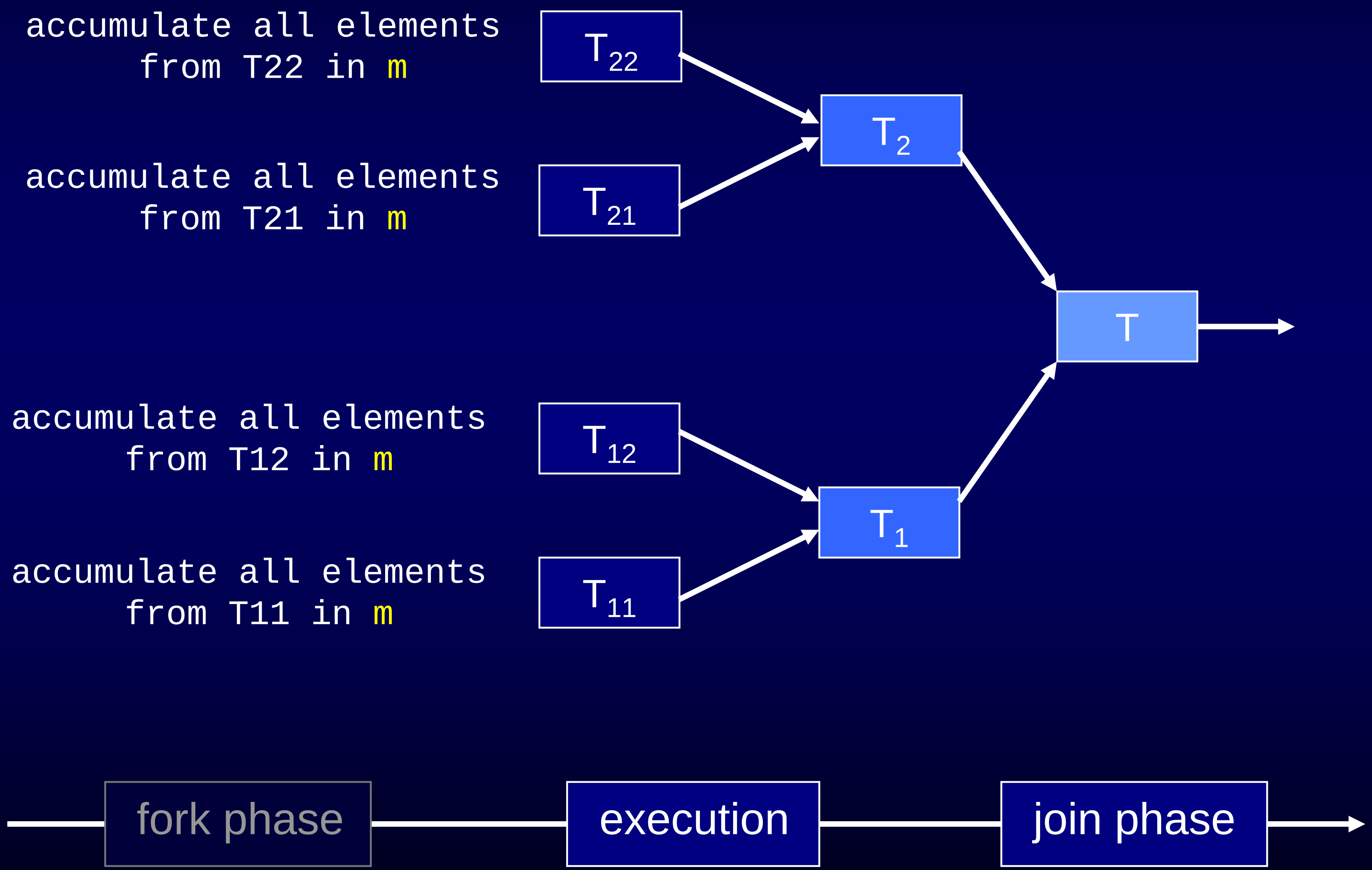
collector for `toConcurrentMap()`

- abstract methods in interface `Collector`
  - and their implementations (for `toConcurrentMap()`)

```
Supplier<A> supplier();
    ConcurrentHashMap::new
BiConsumer<A,T> accumulator();
    (m,e) -> m.merge(keyMapper.apply(e),
                     valueMapper.apply(e), mergeFct)
BinaryOperator<A> combiner();
    (m1,m2) -> { for (Map.Entry<K,V> e : m2.entrySet())
                     m1.merge(e.getKey(),e.getValue(),mergeFct);
                 return m1; }
Function<A,R> finisher()
    m -> (ConcurrentMap<K,V>) m
Set<Characteristics> characteristics();
    CONCURRENT, UNORDERED, IDENTITY_FINISH
```

`mergeFct` throws `IllegalStateException` !!!

concurrently accumulating collect



note on `toConcurrentMap()` collector

- `toConcurrentMap()` uses same implementation as `toMap()`
  - for `accumulator`, `combiner` and `finisher`
  - although `toMap()` is a reducing collector and `toConcurrentMap()` is a concurrently accumulating collector
  - accumulation simply ignores the `combiner`

- characteristics control collect algorithm
  - `CONCURRENT` flag set => accumulation
  - no `CONCURRENT` flag => reduction

wrap-up

- parallel execution adds overhead
  - state in general is an impediment
  - collect uses different algorithms (for thread-safe/-unsafe sinks)
- not predictable which one is faster
  - reducing collect adds overhead for partial results
  - concurrent accumulation relies on synchronization
- rule of thumb
  - don't guess, measure! => run benchmarks
  - large input sequence + cpu-intensive (stateless) intermediate computation => performance gain via parallelism

wrap-up

- lambdas & method references
  - provide concise notation for functionality
- streams provide convenient parallelization
  - of bulk operations on sequences
- complex performance model for parallel execution
  - intermediate operations are pipelined
  - stateful intermediate operations are expensive (due to barriers)
- collectors work even for thread-<u>un</u>safe sinks
  - via reducing collect
  - thread-safe sinks may use concurrently accumulating collect

Angelika Langer & Klaus Kreft

http://www.AngelikaLanger.com
twitter: @AngelikaLanger

related reading:

Lambda & Streams Tutorial/Reference

`AngelikaLanger.com\Lambdas\Lambdas.html`

related seminar:

Programming with Lambdas & Streams in Java 8

`Angelikalanger.com\Courses\LambdasStreams.html`

Q & A