Java 8

Java 8 New Features

- Java 8 Why should you care?
- Passing code
- Lambda expressions
- Processing data with streams
- Collecting data with streams
- Parallel data processing and performance
- Refactoring, testing, and debugging
- Default methods
- Optional: a better alternative to null
- New Date and Time API

Lambda expressions

Chapter 3

As a starter

- The passing code pattern is useful for coping with frequent requirement changes in your code
- a block of code can be passed to another method
- the behaviour of this other method depends on and uses the code passed to it.

Lambda’s in a nutshell

- a lambda expression can be understood as a kind of anonymous function
- that can be passed around
- it doesn’t have a name
- but it has a list of parameters
- a body
- a return type
- and also possibly a list of exceptions that can be thrown

More about Lambda’s

- anonymous because it doesn’t have an explicit name like a method would normally have
- called a function because a lambda isn’t associated with a particular class like a method is
- but it has all the characteristics of a method
- a lambda expression can be passed as argument to a method.
- concise: no need to write a lot of boilerplate like you do for anonymous classes

As an example

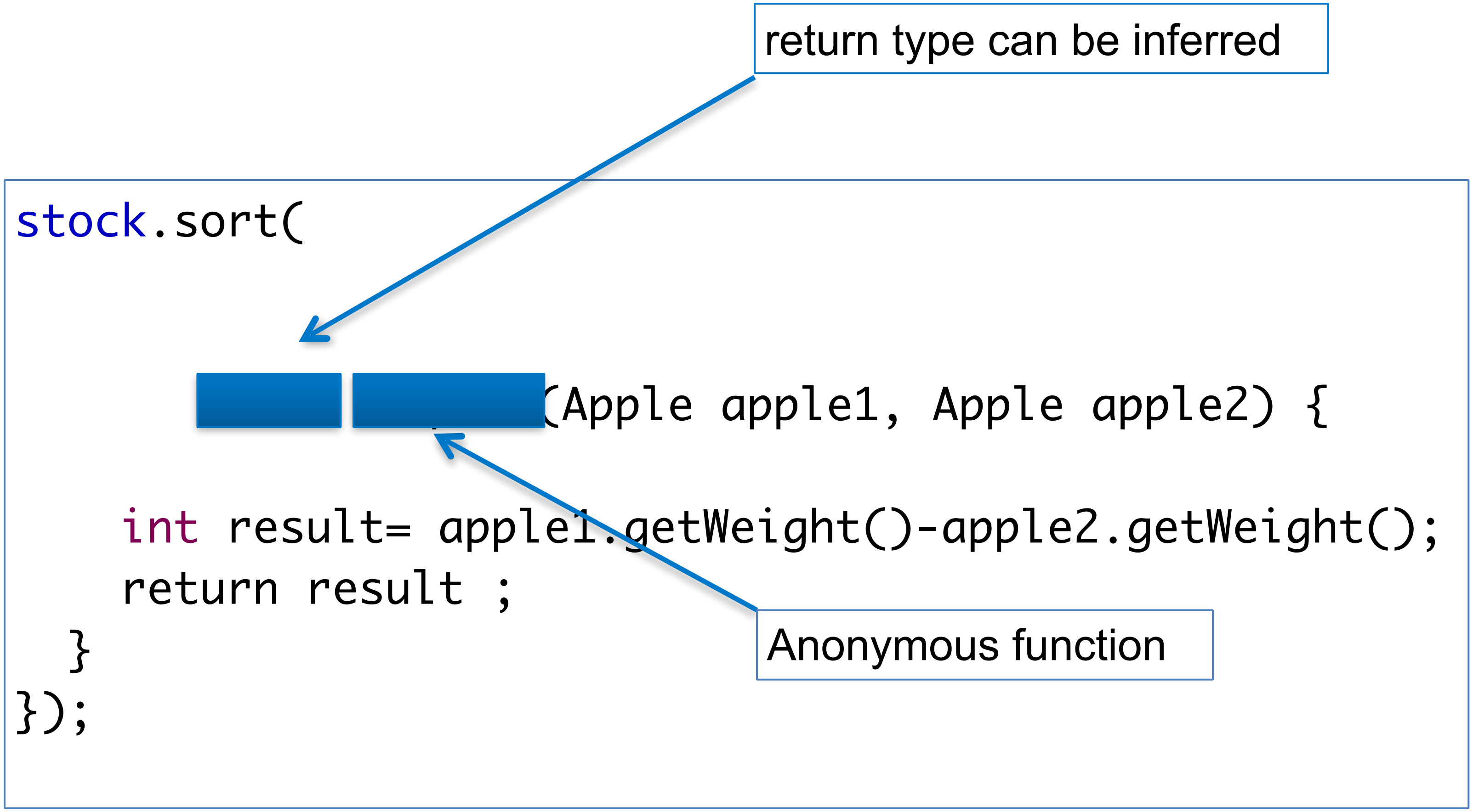
- Without lambda’s

Boilerplate code!

```
stock.sort([redacted]


  [redacted]int compare(Apple apple1, Apple apple2) {

     int result= apple1.getWeight()-apple2.getWeight();
     return result ;
[redacted] }
});
```

As an example

- Without lambda’s



```
stock.sort(


       [redacted] [redacted](Apple apple1, Apple apple2) {

    int result= apple1.getWeight()-apple2.getWeight();
    return result ;
  }
});
```

return type can be inferred

Anonymous function

As an example

- With lambda’s

Introduction → operator to signify the lambda expression

```
stock.sort(

             (Apple apple1, Apple apple2) -> {

    int result= apple1.getWeight()-apple2.getWeight();
    return result ;
  }
);
```

As an example

- With lambda’s

```
stock.sort(


              (Apple apple1, Apple apple2) -> {

    int result= apple1.getWeight()-apple2.getWeight();
    return result ;
  }
);
```

This is an expression; the value of it is implicitly returned when it is the only statement within the body. As an extra bonus the {,} and ; can be omitted

In a concise form

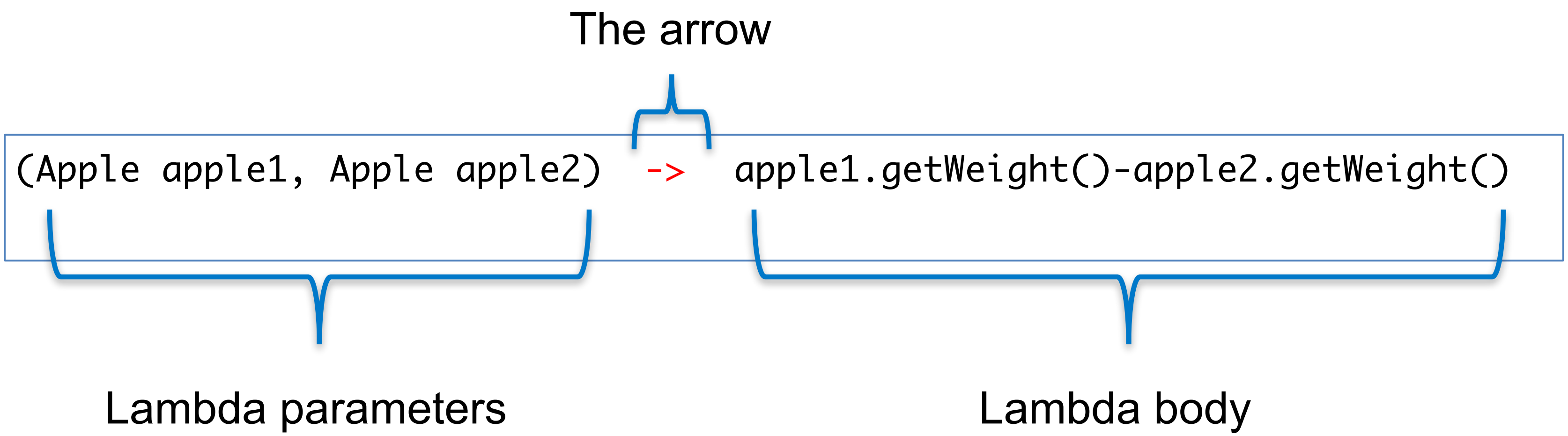
- With lambda’s

```
stock.sort(


            (Apple apple1, Apple apple2) ->


            apple1.getWeight()-apple2.getWeight()
         );
```

- Written on one line

```
stock.sort((Apple apple1, Apple apple2) -> apple1.getWeight()-apple2.getWeight());
```

Composition of Lambda expression



The arrow

```
(Apple apple1, Apple apple2)  ->  apple1.getWeight()-apple2.getWeight()
```

Lambda parameters

Lambda body

Some Lambda expression examples

- (String s) -> s.length()
- (Apple a) -> a.getWeight() > 150
- (int x, int y) -> { System.out.println(x+y); }

  - The first expects a String argument and implicitly the length is returned
  - The second expects an parameter of type Apple and returns a boolean
  - The last expects two int parameters and returns void

Some Lambda expression examples

- () -> 42
- (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())

  - The first lambda expression has no arguments and returns the int 42
  - The last expects two Apple parameters and returns an int

Basic syntax of lambda expression

```
(parameters) -> expression
```

OR

```
(parameters) -> { statements; }
```

Quiz which are valid lambda expressions?

- () -> {}
- () -> "Raoul"
- () -> {return "Mario";}
- (Integer i) -> return "Alan" + i;
- (String s) -> {"Iron Man";}

Where and how to use lambdas?

- Where are lambda expressions allowed?
- Java allows us to use lambdas where a type is expected
- That brings up the question?
  - What is the type of lambda expressions?

A functional interface

- A functional interface is an interface that declares exactly one abstract method
- Later on we will see that from java 8 on, interface can also define so called “default methods” , but that is for later
- i.e. Comparator, Runnable

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

```
public interface Runnable{
public void run();
}
```

@FunctionalInterface

- In the new Java 8 API the @FunctionalInterface is used to indicate that the interface is a functional one
- The compiler gives a warning when an interface is annotated with @FunctionalInterface but, for example, declares more than one abstract method

Quiz,which interface is functional?

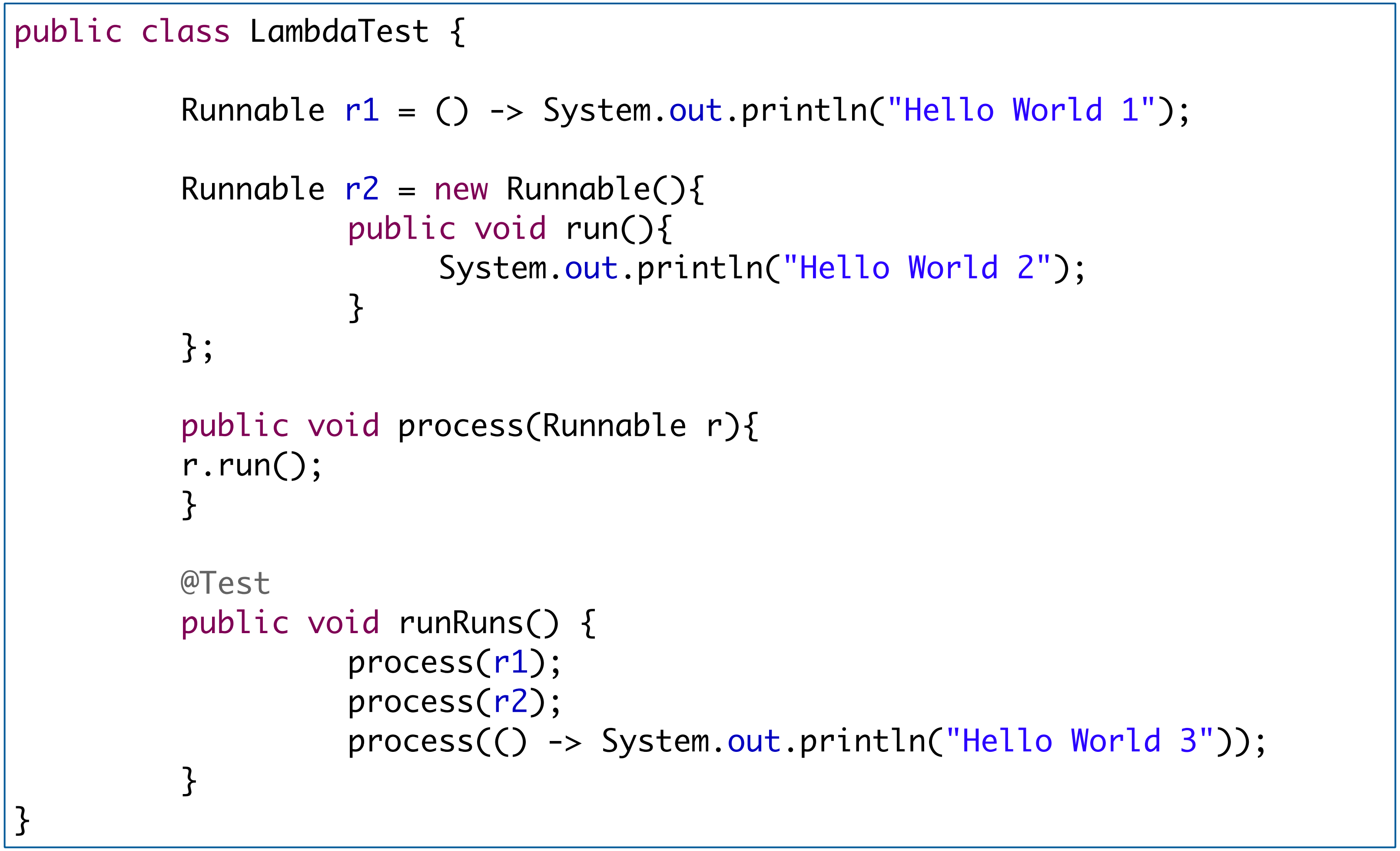
```
public interface Adder{
        public int add(int a, int b);
}

public interface SmartAdder extends Adder{
        public int add(double a, double b);
}

public interface Nothing{
}
```

Why functional interfaces?

- Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline
- the whole expression is treated as an instance of a functional interface.

Example with Runnable

- Runnable is a functionalInterface



```
public class LambdaTest {

         Runnable r1 = () -> System.out.println("Hello World 1");

         Runnable r2 = new Runnable(){
                  public void run(){
                       System.out.println("Hello World 2");
                  }
         };

         public void process(Runnable r){
         r.run();
         }

         @Test
         public void runRuns() {
                  process(r1);
                  process(r2);
                  process(() -> System.out.println("Hello World 3"));
         }
}
```

Function descriptor

- The signature of the abstract method of the functional Interface describes the signature of the lambda expression
- This is called a function descriptor
- i.e. The Runnable interface describes the signature of a function that doesn’t has a parameter and returns void
- Neither name of interface or method matter
- Actually () -> void is enough

What are valid usages of lambdas?

```
public class WhichAreValidExpressions {

        public void execute(Runnable r){
                r.run();
        }

        public Callable<String> fetch() {
                return () -> "Tricky example ;-)";
        }
        @Test
        public void testLambdaExpressions() {
                //1
                execute(() -> {});
                //2
                fetch();
                //3
                ApplePredicate p = (Apple a) -> a.getWeight();
        }
}
```

Putting lambdas in practice

- A recurrent pattern in resource processing is
  - to open a resource
  - do some processing on it
  - and then close the resource
- The setup and cleanup phases are always similar and appear around the important code doing the processing
- This is called the execute around pattern

An example

```
public class ExecuteAroundTest {

 @Test
 public void test() throws IOException {
        processFile();
 }
 public static String processFile() throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
        String oneLineFromFile= br.readLine();
        System.out.println(oneLineFromFile);
        return oneLineFromFile;
  }
 }
}
```

- The current code is limited, only one line can be read
- What If 2 lines or something else is needed?

Make code more general

- Reuse code to do startup and cleanup and tell the processFile method what to do with the file
- We need to parameterize the behaviour of the processFile method
- How to supply this behaviour → lambdas
- Basically a lambda that needs a BufferedReader and returns a String, suffices

To use lambdas we need …

- A functional interface with a signature:
  - (BufferedReader br) -> String

```
@FunctionalInterface
public interface BufferedReaderProcessor {
        public String process(BufferedReader b) throws IOException;
}
```

- use this interface as the argument to your new processFile method

```
public static String processFile(BufferedReaderProcessor p) throwsIOException {
…..}
```

executing a behavior

- Lambda expressions of the form:
- (BufferedReader br) -> String can now be passed
  - Now, execute the behaviour

```
public static String processFile(BufferedReaderProcessor p) throws
                IOException {
try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
return p.process(br);
  }
}
```

Passing lambdas

```java
@Test
public void testLambdaReturnOneLine() throws IOException {
  String output = processFile(
                               (BufferedReader br)-> br.readLine()
                              );
  System.out.println(output);
}



@Test
public void testLambdaReturningTwoLines() throws IOException {
 String output = processFile(
                             (BufferedReader br) -> br.readLine() + br.readLine()
                             );
  System.out.println(output);
}
```

Exercise

- Create a functional interface
  - The signature of the function accepts a BufferedReader and returns a String
- Create a static method that accepts a reference to this interface
  - In the body of this static method a BufferedReader is instantiated and coupled to an ad hoc File which you created and filled with test data
  - The processing of the file is specified in an anonymous function

Exercise continued

- The anonymous function should return a String which is printed to the console
- Make different Implementation for the anonymous function in different Unit tests
- A function that returns as a String:

1. The first line
2. The last line
3. The longest line
4. The longest word
5. Try to implement functionality you think fit

New Functional interfaces in Java8!

- There are already several functional interfaces available in the Java API such as Comparable, Runnable and Callable
- New in Java 8 a whole family of interfaces
  - Inside the java.util.function package
- Among others: Predicate, Consumer, Function

The Predicate interface

```
@FunctionalInterface
public interface Predicate<T>{
        public boolean test(T t);
}
```

- Interface can be used when you need a boolean expression using an object of type T

```
public class UsingPredicateTest {

 public static List<String> filter(List<String> list, Predicate<String> p) {
   List<String> results = new ArrayList<>();
   for(String s: list){
     if(p.test(s)){
        results.add(s);
     }
   }
   return results;
}
 @Test
 public void test() {
   List<String> listOfStrings = Arrays.asList("a", "ab", "", "abc");
   Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
   List<String> nonEmpt = filter(listOfStrings, nonEmptyStringPredicate);
 }
}
```

Consumer, what is it’s function?

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

```
public class ConsumerTest {

        public static void forEach(List<Integer> list, Consumer<Integer> c) {
                for (Integer i : list) {
                        c.accept(i);
                }
        }

        @Test
        public void printIntegers() {
                forEach(Arrays.asList(1, 2, 3, 4, 5),
                                (Integer i) -> System.out.println(i));
        }

}
```

use this interface when you need to access an object of type T and perform some operations on it

Function, what can you do with it?

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

```
public class TestFunctionInterface {

 public static List<Integer> map(List<String> list,Function<String, Integer> f) {
   List<Integer> result = new ArrayList<>();
   for (String s : list) {
       result.add(f.apply(s));
   }
 return result;
 }
 @Test
 public void mapStringsToIntAccordingToTheirLength() {
   List<Integer> lengths
   = map(Arrays.asList("lambdas", "in", "action"), (String s) -> s.length());

  for(int length:lengths) {
      System.out.println(length);
  }
 }
}
```

Predefined Functional interface

New in Java 8	In java.util.function
Consumer<T>	*Represents an operation that accepts a single input argument and returns no result*
Function<T,R>	*Represents a function that accepts one argument of type T and produces a result of R*
Predicate<T>	*Represents a predicate (boolean-valued function) of one argument*
Supplier<R>	*Represents a supplier of results R*
UnaryOperator<T>	*Represents an operation on a single operand that produces a result of the same type T*
BinaryOperator<T>	*Represents an operation upon two operands of the same type T, producing a result of T*
And a lot of specialisations	*BiConsumer<T,U>, BiFunction<T,U,R>, BiPredicate<T,U>, DoubleFunction<R>,*

Quiz:

- Which functional interface would you use?
  1. T -> R
  2. (int,int) -> int
  3. T -> void
  4. () -> T
  5. (T,U) -> R

Answers:

- Which functional interface would you use?
  1. T -> R
     - Function<T,R> typical usage is converting of type T into something of type R
  2. (int,int) -> int
     - int IntBinaryOperator has one method applyAsInt
  3. T -> void
     - Consumer<T> has one method accept()
  4. () -> T
     - Supplier<T> has a single method get()
  5. (T,U) -> R See 1 but then BiFunction<T,U,R>

What is printed by this code?

```
@Test
public void meaningOfThis() {
     Runnable r = new Runnable() {
          public void run() {
              System.out.println(this.getClass().getName());
              System.out.println(OuterClass.this.getClass().getName());
          }
     };
  r.run();
}

//Compare output with:
@Test
public void meaningOfThis() {
     Runnable r =() -> {
              System.out.println(this.getClass().getName());
              System.out.println(OuterClass.this.getClass().getName());
          };
  r.run();
}
```

Type checking, inference and restrictions

It’s like climbing a steep hill

Type of a Lambda

- Lambda expressions enable the generation of an instance of a functional interface
- But a lambda expression doesn’t contain the information itself about which functional interface it’s implementing
- So what is the actual type of a lambda?

Typechecking

- The type of a lambda is deduced from context
  - assignment
  - method invocation
  - cast expression
  - and so on
- The same lambda expression can be associated with different functional interfaces
  - Example: both PriviledgedAction and Callable expect a function that expects nothing and returns a generic type T

An example

```
@Test
public void test() {
   //1.
   Callable<Integer> c = () -> 42;
   //2.
   PrivilegedAction<Integer> p = () -> 42;

}
```

- The type expected inside a context for the lambda expression is called the target type
- In 1. the target type is Callable<Integer>
- In 2.   target type is PrivilegedAction<Integer>

How does type checking occur?

- Take as an example:
- `Callable<Integer> c = () -> 42;`

1. Lookup the one method on this interface
   - This is the public T call() method
2. This interface describes a function (call()) that expects no arguments and returns something of type T
3. The lambda expression is consistent with this description

Special void compatibility rule

- If a lambda has a statement expression as body, it’s compatible with a function descriptor that returns void

```
@Test
public void test() {
        List<String> list = Arrays.asList("1","2","3","4");
        // Predicate has a boolean return
        Predicate<String> p = s -> list.add(s);
        // Consumer has a void return
        Consumer<String> b = s -> list.add(s);
}
```

Type inference

- The compiler can deduce an appropriate signature for the lambda, because the function descriptor is available through the target type
- the compiler has access to the types of the parameters of a lambda expression, and they can be omitted in the lambda syntax
- That is, the compiler infers the types of the parameters of a lambda

Example

- Without inference

```
@Test
public void sortAllApplesOnWeightWithLambda() {

    stock.sort((Apple apple1, Apple apple2) ->
                    apple1.getWeight()-apple2.getWeight());

    forEachT(stock, (Apple a)-> System.out.println(a));
}
```

- With inference

```
@Test
public void sortAllApplesOnWeightWithLambda() {

    stock.sort((apple1, apple2) ->
                    apple1.getWeight()-apple2.getWeight());

    forEachT(stock, (a)-> System.out.println(a));
}
```

- when a lambda has one parameter the parentheses can be omitted

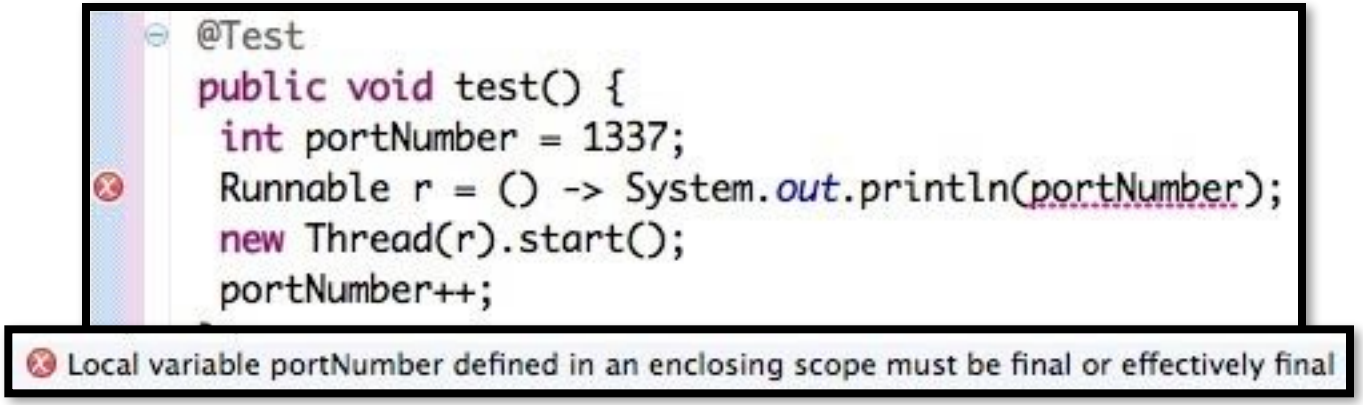
Capturing variables within lambda

- lambda expressions are allowed to use variables defined in an outer scope just like anonymous classes can
- They’re called capturing lambdas
  - Example this lambda captures the variable portNumber:

```
@Test
public void testCapturingLocalVariable() {
        int portNumber = 1337;
        Runnable r = () -> System.out.println(portNumber);
        new Thread(r).start();
}
```

Restrictions on capturing

- Lambdas are allowed to capture without restrictions
  - instance variables
  - static variables
- local variables have to be explicitly
  - declared final or
  - Are effectively final



```
@Test
public void test() {
 int portNumber = 1337;
 Runnable r = () -> System.out.println(portNumber);
 new Thread(r).start();
 portNumber++;
```

Local variable portNumber defined in an enclosing scope must be final or effectively final

Method references

- Method references allow reuse of existing method definitions and pass them like lambdas
- Can increase readability and feel more natural than using lambda expressions

In a nutshell

- Method references are a shorthand to lambdas only calling a specific method
- Often enhance readability
- They are compiled differently

Examples of static methods

- the following lambda expressions forward their arguments to static methods

```
Runnable r= ()-> Thread.dumpStack();
Consumer<String> c=(str) -> System.out.println(str);
```

- Repeating the parameters isn’t necessary
- a method reference is a lambda expression from an existing method implementation
- Use the :: operator

```
Runnable r1= Thread::dumpStack;
Consumer<String> cs1=System.out::println;
```

An example, note ::

```
@Test
 public void test() {

 store.sort((apple1, apple2)-> apple1.getWeight()-apple2.getWeight());

 forEach(store,   apple-> {System.out.println(apple);}   );
 }

 private void forEach(List<Apple> store, Consumer<Apple> consumer) {
  for (Apple apple : store) {
        consumer.accept(apple);
  }
 }
```

```
   store.sort((apple1, apple2)-> apple1.getWeight()-apple2.getWeight());

   forEach(store, System.out::println);
 }
```

A method reference ::

```
   store.sort( (apple1, apple2)-> apple1.getWeight()-apple2.getWeight() );

   forEach(store, System.out::println);
}
```

- With a special method comparing from java.util.Comparator.comparing

```
     store.sort(comparing(Apple::getWeight));

     forEach(store,System.out::println);
}
```

- reverse comparing with:

```
     store.sort(comparing(Apple::getWeight).reversed());
}
```

Examples of instance methods

```
ToIntFunction<Apple>f=a -> a.getWeight();

ToIntFunction<Apple>f=Apple::getWeight;
```

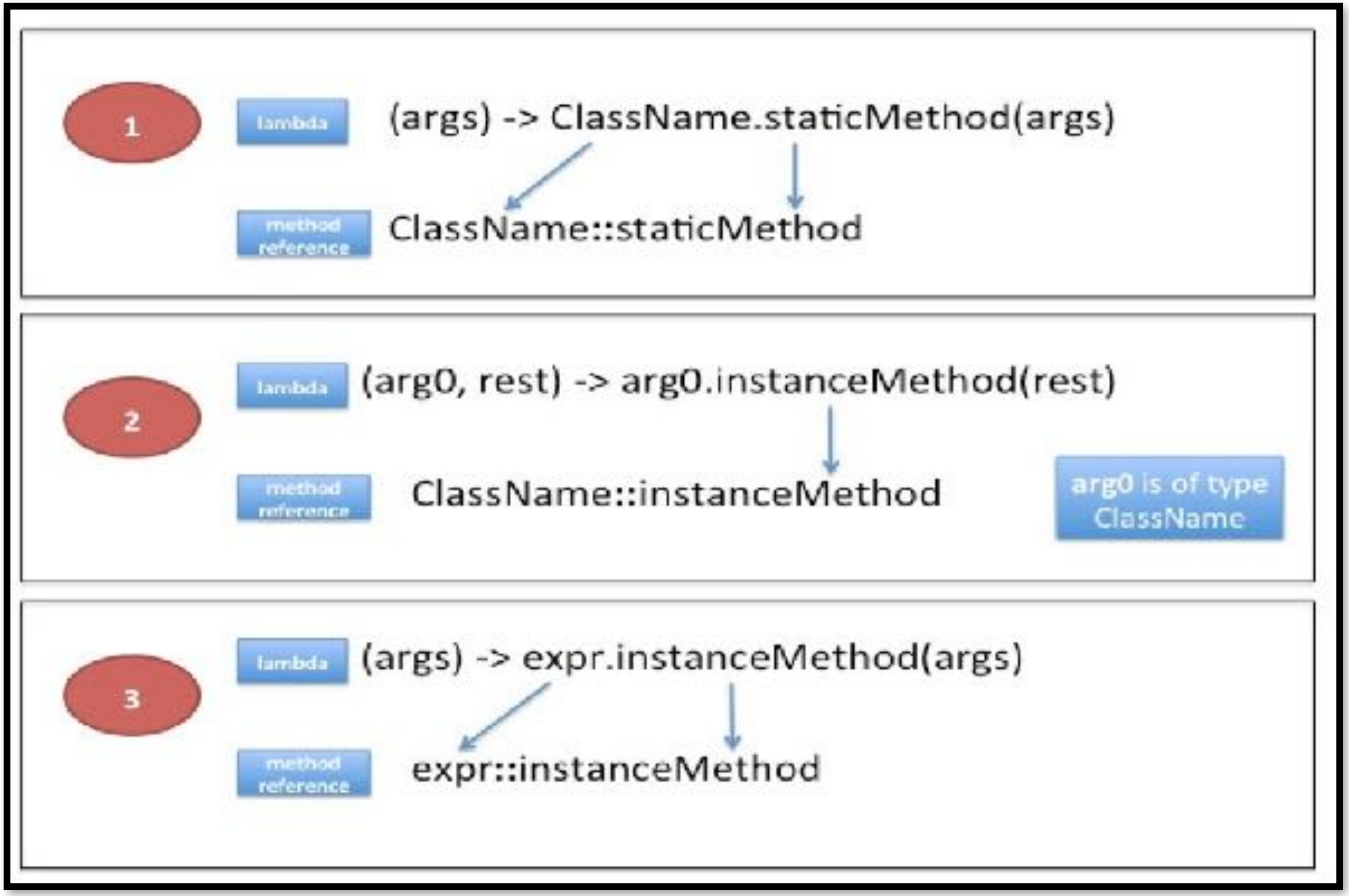
- the following lambda uses the first argument as a receiver object and forward the other arguments to an instance method

```
BiFunction<String, Integer,String> bf=(str, i) -> str.substring(i);
```

```
BiFunction<String, Integer,String> bf=String::substring;
```

- by referring to a method name explicitly, your code can gain better readability

Recipes



1

```
(args) -> ClassName.staticMethod(args)
```

```
ClassName::staticMethod
```

2

```
(arg0, rest) -> arg0.instanceMethod(rest)
```

```
ClassName::instanceMethod
```

3

```
(args) -> expr.instanceMethod(args)
```

```
expr::instanceMethod
```

Exercise

- Assign method references of some methods of the System class to function variables
- As a reference: a table of general FunctionalInterfaces is shown on the next slide
- See the eclipse project for a more in depth description

Predefined Functional interface

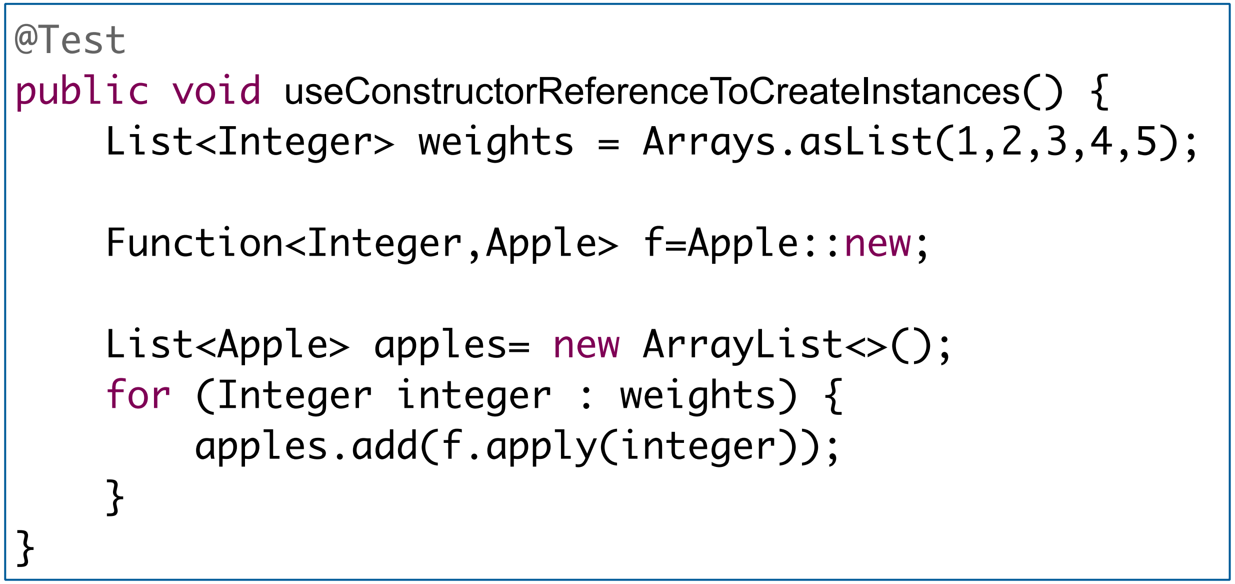
New in Java 8	In java.util.function
Consumer<T>	*Represents an operation that accepts a single input argument and returns no result*
Function<T,R>	*Represents a function that accepts one argument of type T and produces a result of R*
Predicate<T>	*Represents a predicate (boolean-valued function) of one argument*
Supplier<R>	*Represents a supplier of results R*
UnaryOperator<T>	*Represents an operation on a single operand that produces a result of the same type T*
BinaryOperator<T>	*Represents an operation upon two operands of the same type T, producing a result of T*
And a lot of specialisations	*BiConsumer<T,U>, BiFunction<T,U,R>, BiPredicate<T,U>, DoubleFunction<R>,*

Exercise (similar to the former)

- Similar to the last exercise
- Assign method references of some methods of the String class to function variables
- As a reference: a table of general FunctionalInterfaces is shown on the former slide

Constructor references

- Use ClassName::new to create new instance



```java
@Test
public void useConstructorReferenceToCreateInstances() {
    List<Integer> weights = Arrays.asList(1,2,3,4,5);

    Function<Integer,Apple> f=Apple::new;

    List<Apple> apples= new ArrayList<>();
    for (Integer integer : weights) {
        apples.add(f.apply(integer));
    }
}
```

- Define map function to increase readability

Example with map function

```
@Test
public void useMapFunctionToIncreaseReadability() {
        List<Integer> weights = Arrays.asList(1,2,3,4,5);
        Function<Integer,Apple> f=Apple::new;

        List<Apple> apples = map(weights, f);

        for (Apple apple : apples) {
                System.out.println(apple);
        }
}

public <T> List<T> map(List<Integer> list,Function<Integer,T> newClass) {
        List<T> freshInstancesList = new ArrayList<>();
        for (Integer integer : list) {
                freshInstancesList.add(newClass.apply(integer));
        }
        return freshInstancesList;
}
```

FactoryPattern and Constructor ref

- The capability of referring to a constructor without instantiating it enables interesting applications

```
Function<Integer, Apple> fruitFactory=Apple::new;
```

- For example, a Map can be used to associate constructors with a string value

Exercise

- Create an appleFactory method
  - Make a test in which you define an array of String, length m and an array of int, length n
  - Call the appleFactory method with these 2 arrays and an anonymous function which adheres the signature BiFunction<String,Integer,Apple>
  - The appleFactory should return a List<Apple> which in size equals n*m
- See a more in depth description in the eclipse project

Now it really becomes interesting!

Introduction to stream processing

- A stream is a sequence of data items produced one at a time
- A practical existing example is on the unix,linux platforms where many programs
  - operate by reading data form stdin
  - operate on the data
  - then writing data to stdout
- The unix cmdline allows these programs to be linked together with pipes (|)
  - cat file1 file2 | tr “[A-Z]” “[a-z]” | sort | tail -3

Explanation Unix example

- cat file1 file2 | tr “[A-Z]” “[a-z]” | sort | tail -3
  - Prints the 3 words which appear latest in dictionary order after translating to lowercase
- You can say, sort takes a stream of lines as input and produces an other stream of lines as output
- Note in Unix the programs, cat,tr,sort and tail are executed in parallel
  - sort can process the first few lines from tr before cat and tr are finished

About Collections

- The Collections API is the most heavily used
- Collections are extremely fundamental
- They allow to group data and to process it
- But pre java 8 collections have still a lot of disadvantages
  - Much business logic entails SQL-like operations such as “grouping” or “finding” but we lack a sql like declarative style in java
  - Parallel processing of large collections is difficult and error prone

The answer to our problems:

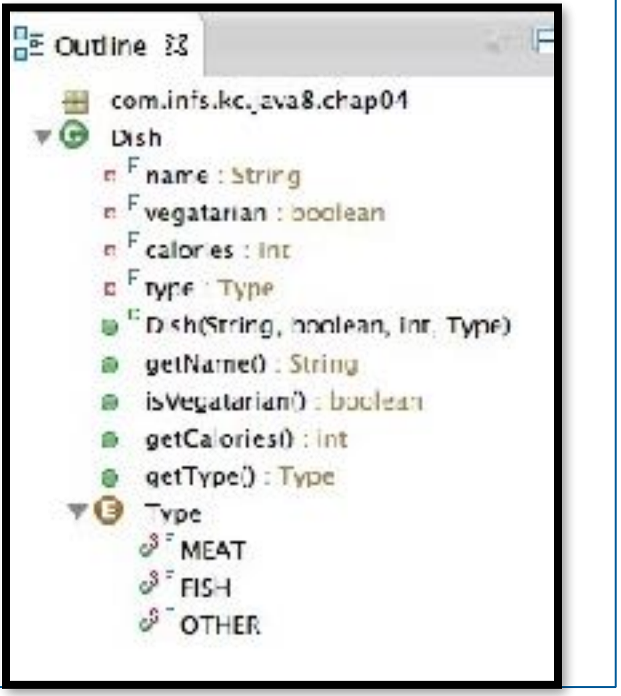
- Streams!
- And although there is a lot to say about collections an easy start point for streams are collections

Setup 1

```
public class StreamTest {

        @Test
        public void aStartPointOfStreams() {

                java.util.stream.Stream<Dish> stream = menu.stream();
        }

        List<Dish> menu = Arrays.asList(
                new Dish("pork", false, 800, Dish.Type.MEAT),
                new Dish("beef", false, 700, Dish.Type.MEAT),
                new Dish("chicken", false, 400, Dish.Type.MEAT),
                new Dish("french fries", true, 530, Dish.Type.OTHER),
                new Dish("rice", true, 350, Dish.Type.OTHER),
                new Dish("season fruit", true, 120, Dish.Type.OTHER),
                new Dish("pizza", true, 550, Dish.Type.OTHER),
                new Dish("prawns", false, 300, Dish.Type.FISH),
                new Dish("salmon", false, 450, Dish.Type.FISH) );

}
```

Setup 2 Note Dish is immutable

```
public class Dish {

private final String name;
private final boolean vegatarian;
private final int calories;
private final Dish.Type type;

public Dish(String name, boolean vegatarian, int calories, Dish.Type type) {
        this.name = name;
        this.vegatarian = vegatarian;
        this.calories = calories;
        this.type = type;
}

public getters …

public enum Type {
        MEAT, FISH, OTHER;
}

}
```



```
Outline
  com.infs.kc.java8.chap04
  Dish
    name : String
    vegatarian : boolean
    calories : int
    type : Type
    Dish(String, boolean, int, Type)
    getName() : String
    isVegatarian() : boolean
    getCalories() : int
    getType() : Type
    Type
      MEAT
      FISH
      OTHER
```

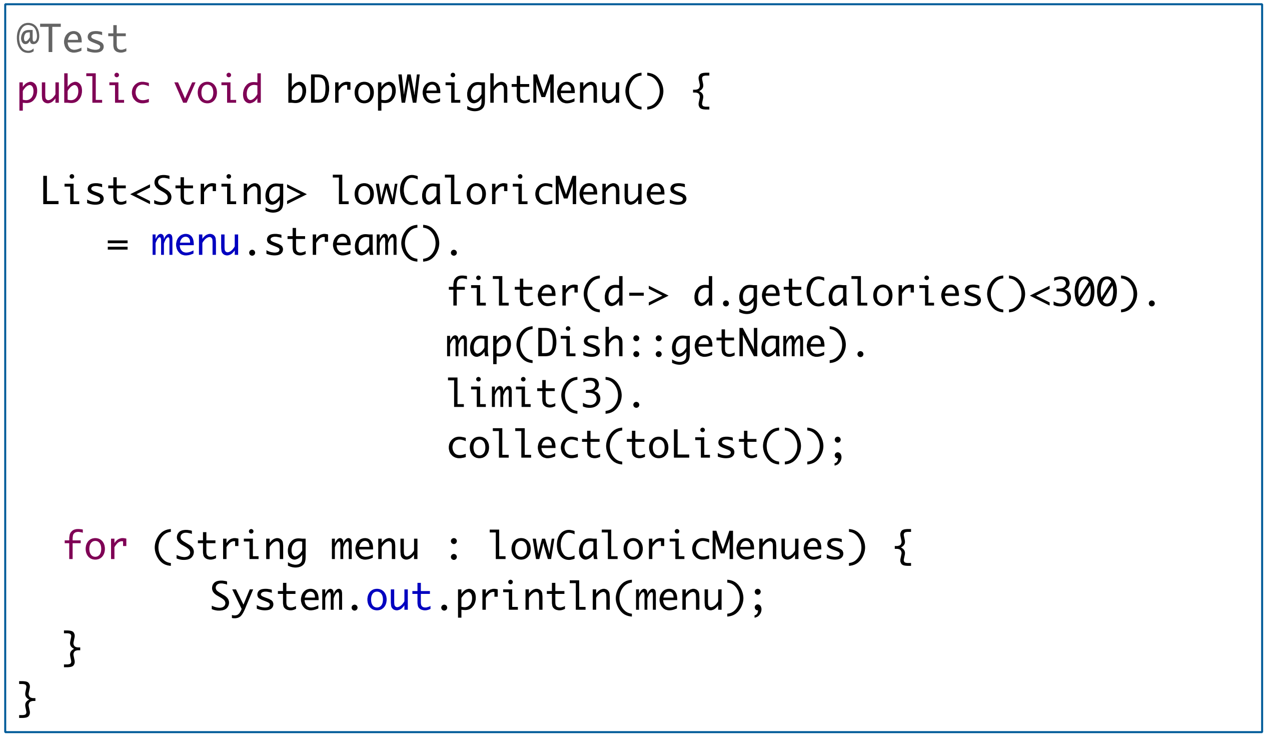
Collection as a starting point

- A short definition is “a sequence of elements from a source that supports aggregate operations
  - Stream provides an interface to a sequenced set of values
    - streams don’t store elements: they are computed “on demand”
- Streams consume from a data-providing source such as Collections, or IO resources
- Aggregate operations: Streams support SQL-like operations

2 important characteristics streams

- Pipelining:
  - Many stream operations return a stream themselves. This allows operations to be chained and form a larger pipeline
  - This enables certain Optimizations like laziness and short-circuiting
  - Compare A pipeline of operations as a “query” on the data source
- Internal iteration
  - no for loop like with collections, it is done behind the scenes

An example



```
@Test
public void bDropWeightMenu() {

 List<String> lowCaloricMenues
    = menu.stream().
                     filter(d-> d.getCalories()<300).
                     map(Dish::getName).
                     limit(3).
                     collect(toList());

  for (String menu : lowCaloricMenues) {
         System.out.println(menu);
  }
}
```

- Filter a stream of dishes, only allow dishes with calories < 300, of the dish “row” select the name (map function), take only the first 3 elements(limit) → sql-like

Note

- filtering, extracting (map) or truncating (limit) functionalities are available through the Streams library
- As a result, the Streams API has more flexibility to decide how to optimize this pipeline

Differences Collection and Stream

- Both Collections and the new notion of Streams provide interfaces to a sequenced set of elements
- A Collection is an in-memory data structure, which holds all the values that the data structure currently has
- By contrast a Stream is a conceptually fixed data structure, but in reality elements are computed on demand

A different view

- a Stream is like a lazily constructed Collection: values are computed when they are solicited by a consumer (demand-driven, or even just-in-time, manufacturing)
- In contrast, a Collection is eagerly constructed (supplier-driven)

TRAVERSABLE ONLY ONCE

- Note that, similarly to Iterators, a Stream can only be traversed once
- After that a Stream is said to be “consumed”

```
@Test
public void printAllMenues() {
        Stream<Dish> dishStream = menu.stream();
        //1
        dishStream.forEach(System.out::println);
        //2
        dishStream.forEach(System.out::println);
}
```

- Second call results in a runtime exception

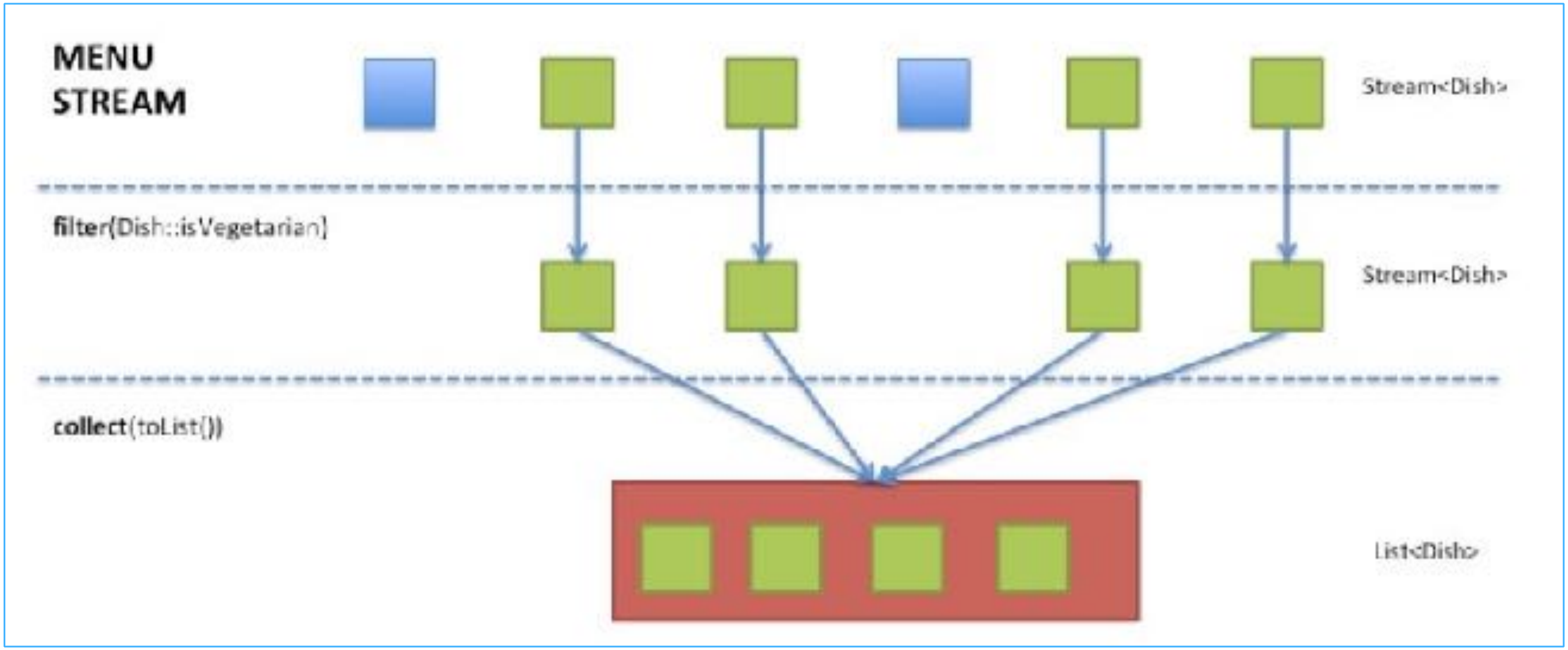
```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

Filtering and slicing

- How to select elements from a stream
  - filtering with a predicate,
  - filtering only unique elements
  - ignoring the first few elements of a stream
  - or truncating a stream to a given size

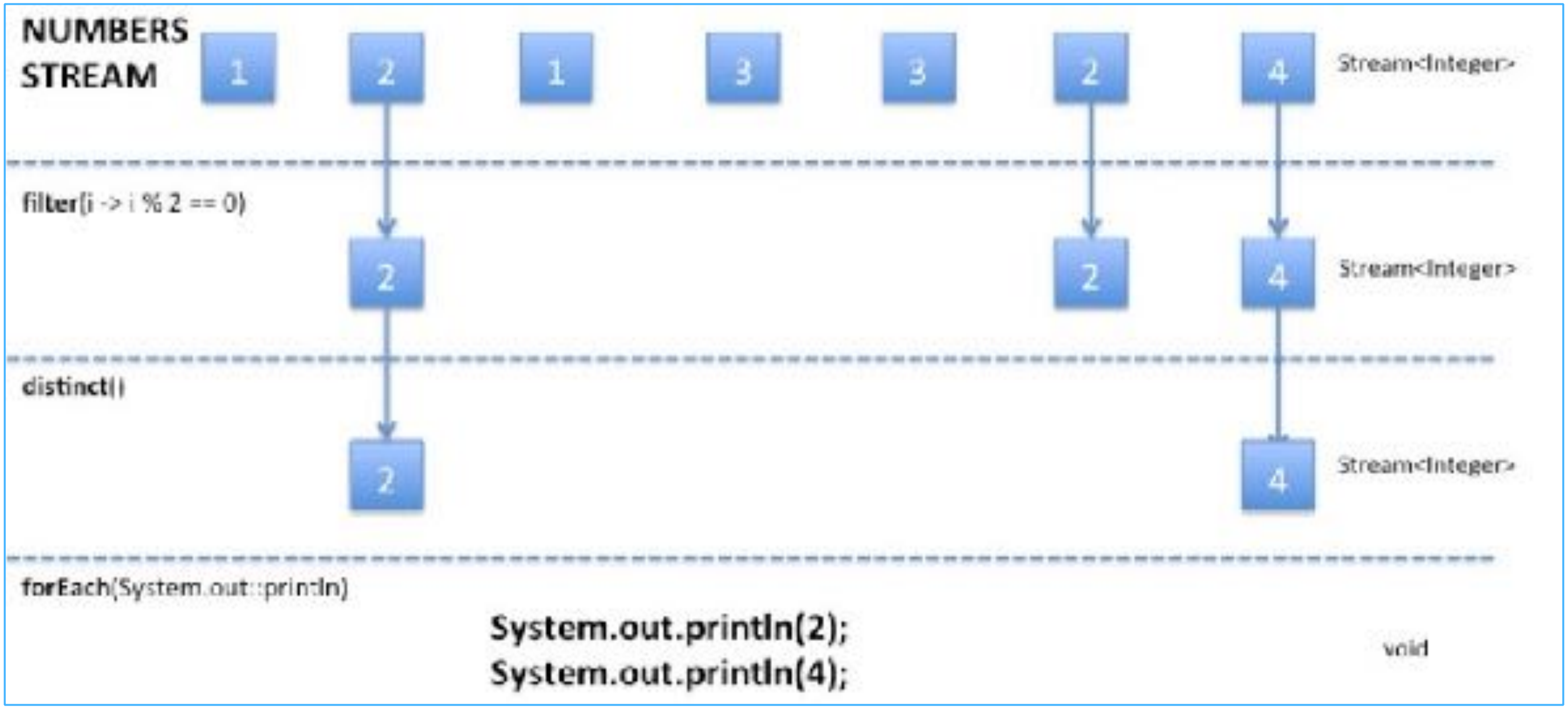
Filtering with a predicate

```
List<Dish> vegetarianMenu =
              menu.stream()
                  .filter(d-> d.isVegatarian())
                  .collect(toList());
```



Filtering unique elements

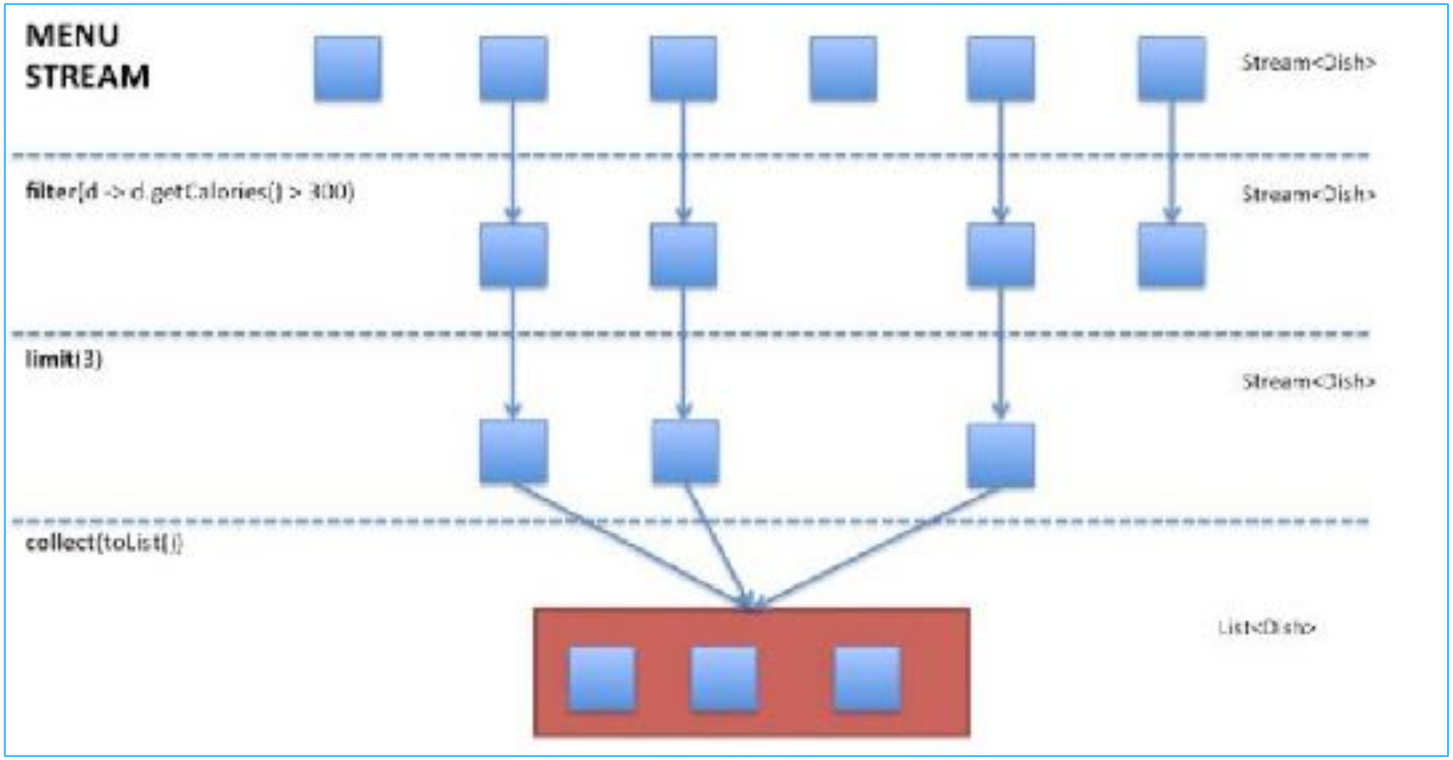
```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
       numbers.stream()
              .filter(i -> i % 2 == 0)
              .distinct()
              .forEach(System.out::println);
```



System.out.println(2);
System.out.println(4);

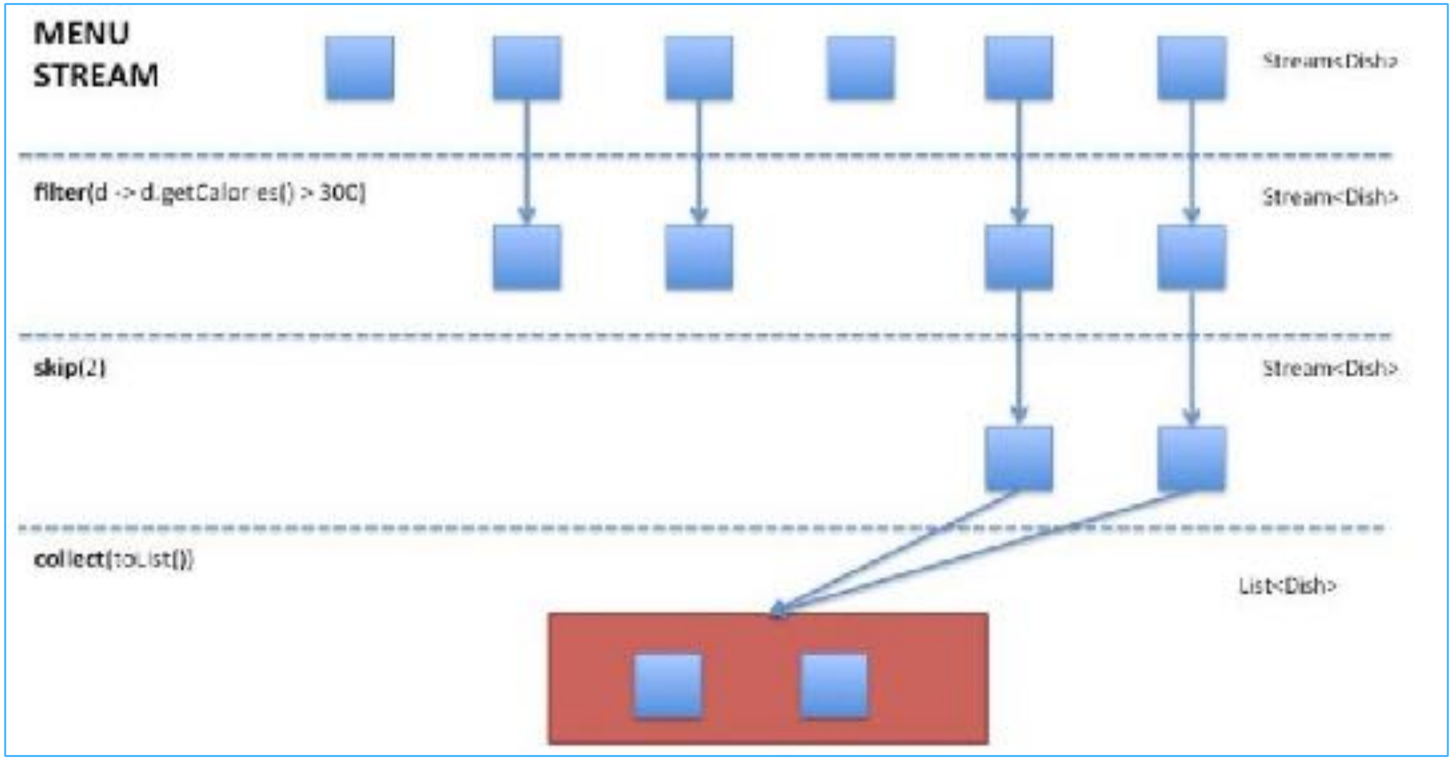
Truncating a stream

```
List<Dish> dishesLimit3 =
           menu.stream()
               .filter(d -> d.getCalories() > 300)
               .limit(3)
               .collect(toList());
```



Skipping elements

```
List<Dish> dishesSkip2 =
            menu.stream()
                .filter(d -> d.getCalories() > 300)
                .skip(2)
                .collect(toList());
```



Mapping

- How to select information from objects?
- The SQL equivalent from: “select” a particular column from a row or rows
- The Streams API provides map and flatMap

- About map
  - Map takes a function as an argument to transform the elements of a stream into another form

Mapping dish to string (dish name)

```
List<String> dishNames = menu.stream()
                             .map(d -> d.getName())
                             .collect(toList());
```

- d -> d.getName() results in a stream of Strings
- map the strings on int’s by taking their length

```
List<Integer> nameLengths = menu.stream()
                                .map(d -> d.getName())
                                .map(s -> s.length())
                                .collect(toList());
```

Flattening Streams

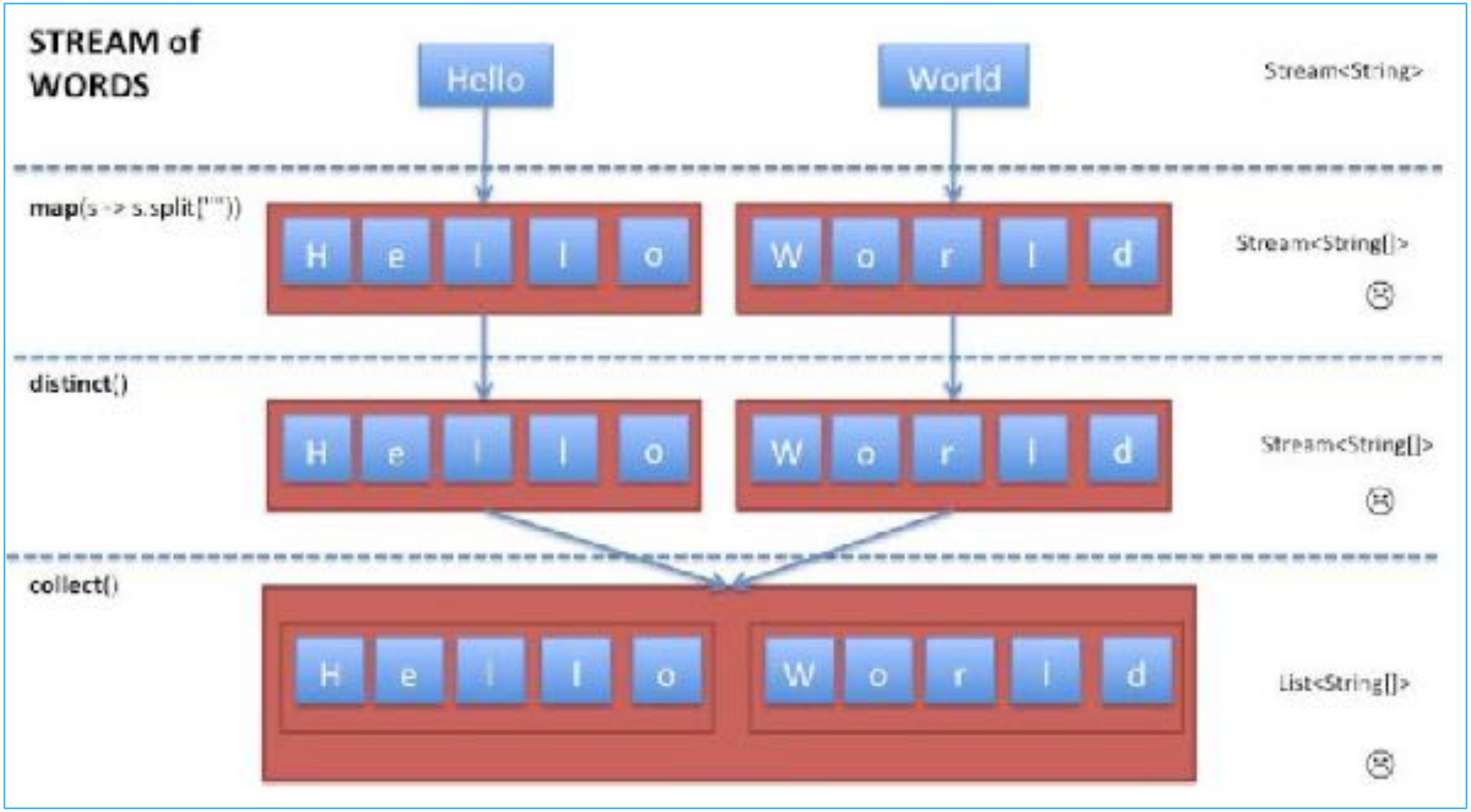
- given the list of words [“Hello”, “World”] return the list [“H”,”e”,”l”,”o”,”W”,”r”,”d”]

```
List<String> lines = Arrays.asList("Hello", "World");
lines.stream()
     .map((String line) -> Arrays.stream(line.split("")))
     .distinct()
     .forEach(System.out::println);
```

- Note: Arrays.stream(array) makes stream functionality available for arrays

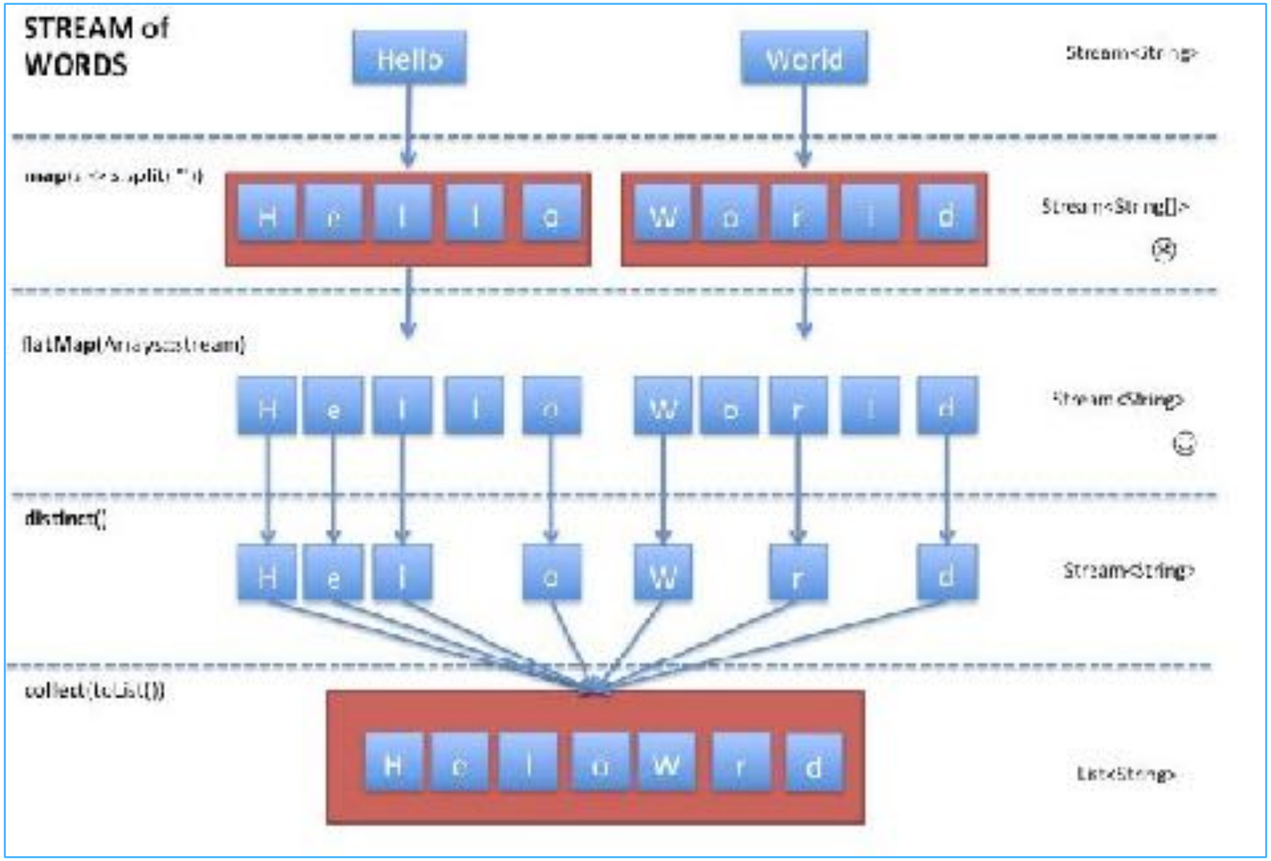
Result, hmm where are the chars?

```
java.util.stream.ReferencePipeline$Head@2f2c9b19
java.util.stream.ReferencePipeline$Head@31befd9f
```



USING FLATMAP

```
lines.stream()
    .flatMap((String line) -> Arrays.stream(line.split("")))
    .distinct()
    .forEach(System.out::println);
```



Exercise

- Given a list of numbers, how would you return a list of the square of each number? e.g. given [1,2,3,4] you should return [1, 4, 9, 16]
- Given two list of numbers, how would you return all pairs of numbers?
  - E.g. given a list (1, 2, 3) and (3, 4) you should return ((1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]. (represent a pair as an array with two elements)
- extend 2) to only return pairs whose sum is divisible by 3?

Special predicates

- Checking a predicate matches at least one element

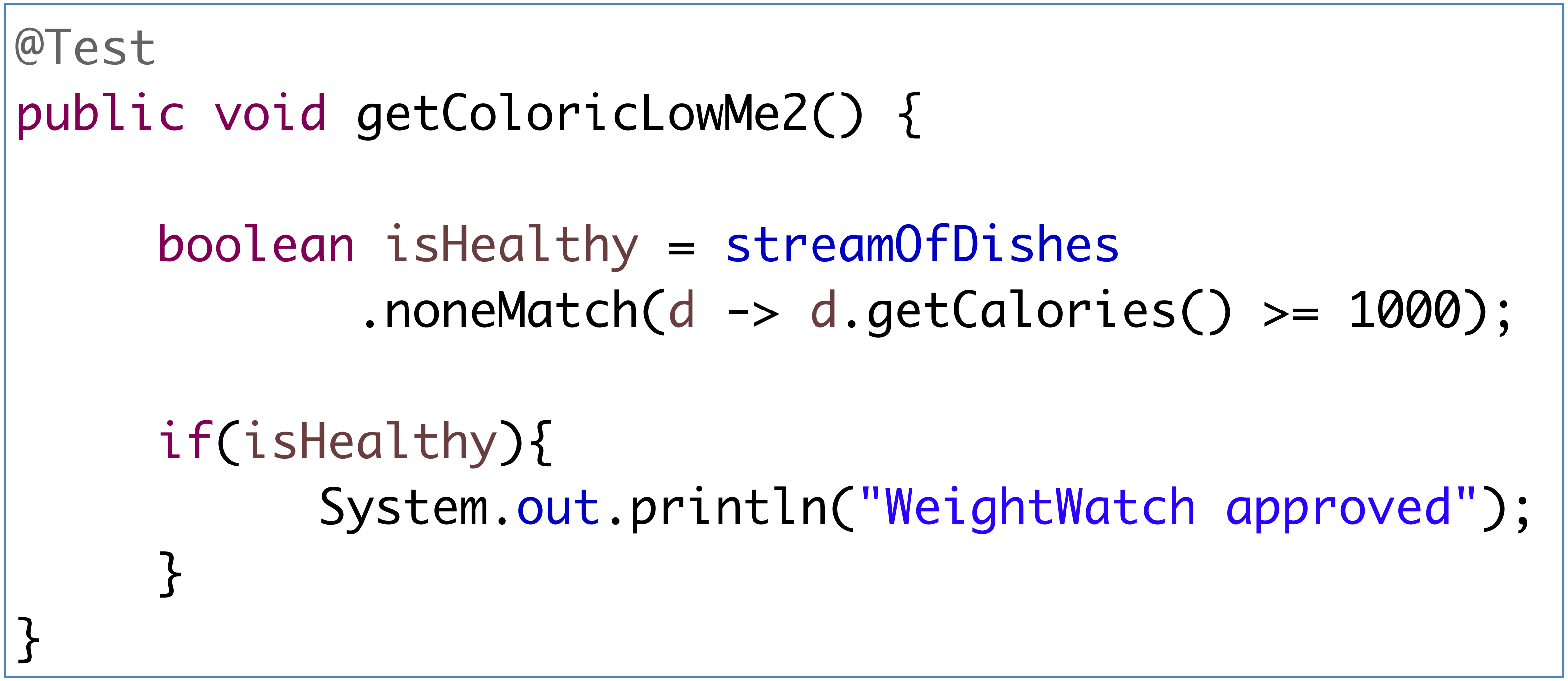
```
public void getAVegatarianFriendlyMeal() {
    if(streamOfDishes.anyMatch(Dish::isVegatarian)){
       System.out.println(”Yes some vegatarian food!");
    }
}
```

- Checking a predicate matches all elements

```
boolean isHealthy = streamOfDishes
                    .allMatch(d -> d.getCalories() < 1000);
```

The dual of anyMatch

- Checking a predicate that matches non of the elements



```
@Test
public void getColoricLowMe2() {

     boolean isHealthy = streamOfDishes
              .noneMatch(d -> d.getCalories() >= 1000);

     if(isHealthy){
           System.out.println("WeightWatch approved");
     }
}
```

Shortcircuiting operations

- These three operations:
  - anyMatch, allMatch and noneMatch
- make use of shortcircuiting,
- a Stream version of familiar Java short-circuiting && and ||" operators
- As soon as an element is found a result can be produced
  - findFirst, findAny and limit are also shortcircuiting operations

Optional

```
@Test
public void findAVegatarianMeal() {

     Optional<Dish> vegaDishesOptional =
        streamOfDishes
        .filter(Dish::isVegatarian)
        .findAny();



}
```

- The stream pipeline is optimized
  - it performs a single pass and finish as soon as a result is found by using short-circuiting
- But, what is an Optional

Optional in a nutshel

- The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value
- It is introduced to avoid bugs related to null-checking! Handy methods are:
  - isPresent
  - T orElse(T default)

The signature of an Optional

- isPresent(): returns true if it contains a value
- ifPresent(Consumer<T> block): executes the given block if a value is present
- T get(): returns the value if it is present otherwise throws a NoSuchElementException
- T orElse(T other): returns the value if present, otherwise returns a default value

Is there an arbitrarily vega-meal?

```
@Test
public void findAVegatarianMeal() {
       streamOfDishes
       .filter(Dish::isVegetarian)
       .findAny()
       .map(Dish::getName)
       .ifPresent(System.out::println);
}
```

- Where does this map(d -> d.getName()) comes from?
- It is a function defined on the Optional!

Finding the first element

```
@Test
public void findTheFirstVegetarianMeal() {
     List<Integer> someNumbers =
                            Arrays.asList(1, 2, 3, 4, 5);
     Optional<Integer> firstSquareDivisibleByThree =
               someNumbers.stream()
               .map(x -> x * x)
               .filter(x-> x % 3 == 0)
               .findFirst();
}
```

- When to use findFirst and findAny?
  - Finding the first element is more constraining in parallel
  - don’t care about which element is return use findAny it is less constraining

Reducing

- Characterizing reduce operations
- Queries that combine all the elements in the stream repeatedly, to produce a single value such as an integer.
- These queries can be classified as a reduction operation
- A stream is reduced to a value
- In functional jargon this is called a fold
  - this operation is seen as “folding” repeatedly a long piece of paper (your stream)

Investigate summing the elements

```
int sum = 0;
for (int x : numbers) {
       sum += x;
}
```

- The list of numbers is reduced into one number by repeatedly using addition
- There are two parameters in this code:

1. An initial value of the sum variable (here) 0
2. An operation to combine all the elements (+)

The reduce operation to the rescue

- Wouldn’t it be great if all the numbers could also be multiply without having to copy and paste this code around
- This is where the reduce operation which abstracts over this pattern of repeated application can help

```
@Test
public void introductionReduce() {

List<Integer> numbers = Arrays.asList(1,2,3,4,5);
int sumTotal = numbers.stream().reduce(0, (sum, x) -> sum + x);
      assertThat(sumTotal, is(15));
}
```

A better look to reduce

```
int sum2 = numbers.stream().reduce(0, (a, b) -> a + b);
```

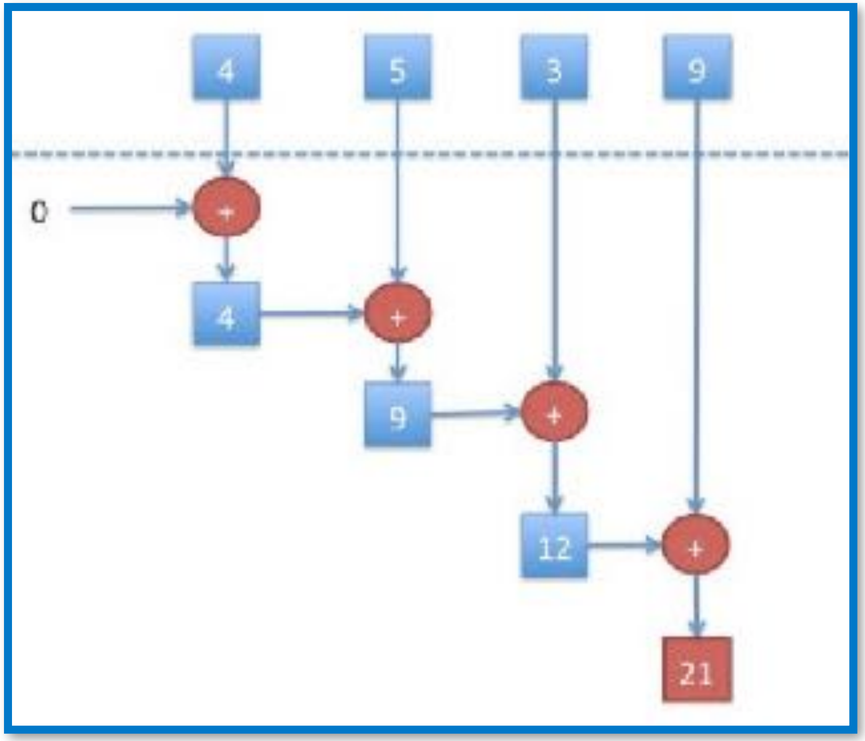
- Reduce takes two arguments:

1. An initial value, here 0
2. A BinaryOperator<T> to combine two elements and produce a new value, here the lambda (a, b) -> a + b is used

- Just as easily all the elements could be multiplied by passing (a, b) -> a * b to reduce

```
int sum2 = numbers.stream().reduce(1, (a, b) -> a * b);
```

reduce(0, (a, b) -> a + b)



- Use method reference to make this code more concise, new in Java 8 Integer.sum

```
numbers.stream().reduce(0, Integer::sum);
```

Maximum and minimum

- Use reduction to compute maxima and minima
- reduce takes two parameters:

1. An initial value
2. A lambda to combine two stream elements, and produces a new value, for maximum:

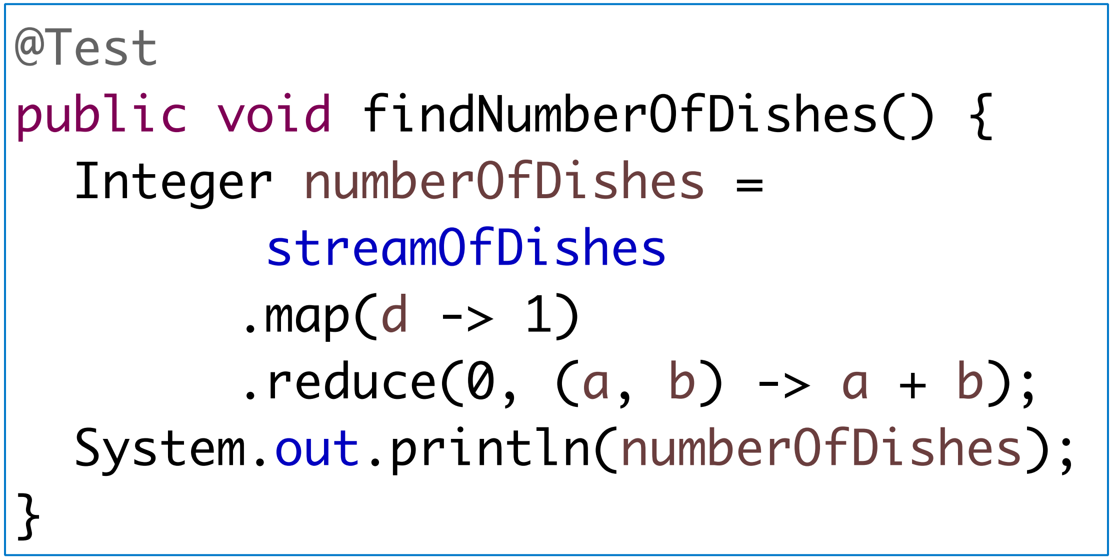
```
        Optional<Integer> max = numbers.stream()
               .reduce((a,b)-> a>=b ?a:b);

                      OR
       Optional<Integer> max = numbers.stream()
               .reduce(Integer::max);
```

Quiz Reducing

- How to count the number of dishes in a stream using the map and reduce methods?

Quiz Solution

- Map each element of a stream into the number “1” and then sum them using reduce



```
@Test
public void findNumberOfDishes() {
  Integer numberOfDishes =
         streamOfDishes
        .map(d -> 1)
        .reduce(0, (a, b) -> a + b);
  System.out.println(numberOfDishes);
}
```

- A chain of map and reduce is commonly known as the “map-reduce” pattern
  – made famous by Google’s use of it for web-searching because it can be easily parallelized

Exercises on memoryscottdb

- Familiarize yourself with the object model of Employees,Departments and SalaryGrades
- Print all the employees and departments
- Find all employees with job is Clerk
- Find all employees with job is Clerk and sort by salary (small to high).
- What are all the unique jobs of the employees?
- Find all employees working on department 10 and sort them by name.

Exercises on memoryscottdb

- Return a string of all employees names sorted alphabetically
- Are there any employees based in New York?
- Print all employees having a salary in scale 2
- How many employees earn a salary in scale 2 or 3
- What's the highest value of all the salaries?
- Find the salary with the smallest value

Exercises on memorypubsdb

- Familiarize yourself with the object model of this domain
- Print all Author objects and all Title objects
- Print all Authors living in CA
- Print all the full names, (auFname + auLname) sort them alphabetically
- Count the number of titles
- Print the values of the advance property
- Does some property advance contain a null?

Exercises on memorypubsdb

- Print all business books
- Count the number of authors that wrote business books
- Print all the authors that wrote business books starting from the Title stream
- Idem as former question but now starting from the Author stream
- Print all the authors that did not write business books

Numeric streams

- Numeric streams: Why do we need them?

```
Integer numberOfDishes =
        streamOfDishes
       .map(d -> 1)
       .reduce(0, (a, b) -> a + b);
```

- The problem with the code above is:
  – Boxing costs (can be large)
- In addition, wouldn't it be nicer if we could call a sum() method directly as follows?

```
Integer numberOfDishes =
        streamOfDishes
       .map(d -> 1)
       .sum()
```

Primitive streams specializations

- The method map generates a Stream<T> which disallows generation of a stream of primitives
- Primitive streams come to the rescue

```
IntStream intStream =
   streamOfDishes.mapToInt(Dish::getCalories);
```

- Other methods are:
  - mapToLong producing a stream of long type
  - mapToDoubleproducing a stream of double type
- PrimitiveStreams offer functions like sum(), min(),max()

To,From object -> primitive stream

```
IntStream intStream =
   streamOfDishes.mapToInt(Dish::getCalories);

Stream<Integer> stream = intStream.boxed();
```

- Default values

```
OptionalInt maxCalories =
                    streamOfDishes
                    .mapToInt(Dish::getCalories)
                    .max();
```

- max operation has no default, so an Optional is returned, choose your own default

```
int max = maxCalories.orElse(1);
```

Numeric ranges

- A common use case with numbers is working with ranges of numeric values
- on IntStream, DoubleStream and LongStream
  - – range (exclusive upperbound)
  - – rangeClosed (inclusive upperbound)

```
IntStream numbers = IntStream.rangeClosed(1, 100);
```

Building streams

- Streams can be created in many ways
- Examples follow for create a stream
  - from a sequence of values
  - from an array
  - from a file
  - from a generative function to create infinite streams

Stream from values

- Using the static method Stream.of

```
Stream<String> stream =
              Stream.of("Java 8 ", "Lambdas ", "Streams");
```

- Using the empty method to get empty stream

```
Stream<String> emptyStream = Stream.empty();
```

- Using the static method Arrays.stream

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

Stream from a File

- Java’s NIO API has been updated to take advantage of the Stream API
- Many static methods in java.nio.file.Files return a stream

```
Stream<String> lines =
Files.lines(Paths.get("data.txt"), Charset.defaultCharset());

long uniqueWords =
        lines.flatMap(line -> Arrays.stream(line.split(" ")))
                .distinct()
                .count();
```

Creating infinite streams

- The Stream API provides two static methods to generate a stream from a function

1. Stream.iterate
2. Stream.generate

- Such Streams create values on demand given a function and can calculate values forever
- It is generally sensible to use limit(n) on such Streams

Examples of “infinite” streams

```
Stream.iterate(0, n -> n + 2)
                .forEach(System.out::println);
```

- About the example
- The iterate method takes an
  - initial value, here 0
  - a lambda (of type UnaryOperator<T>)
- The lambda is successively applied on each new value produced
- This results in an infinite stream, the stream is unbounded

The generate method

- the method generate also produces an infinite stream of values
- It doesn’t apply successively a function on each new produced value however
- It takes a lambda of type Supplier<T> to provide new values

Example generate method

```
Stream.generate(Math::random)
.limit(5)
.forEach(System.out::println);
```

```
0.029727101463219885
0.2628896161656359
0.3841613118657313
0.3668537270009897
0.574106550966172
```

- The supplier that is used is stateless (a referece to the Math.random() methode)
- A stateful supplier can be used but that limits the use of it to sequential streams which is not to be preferred

Collection data with streams

Chapter 5

Grouping the hard pre Java 8 way

```
@Test
public void howToDoSomethingSimpleAsGroupingApplesByColour() {

    Map<String,List<Apple>> listsGroupedByColour= new HashMap<>();
    for (Apple apple : stock) {
        String colour = apple.getColour();
        if(listsGroupedByColour.get(colour)==null){
            ArrayList<Apple> fixedColourList = new ArrayList<>();
            listsGroupedByColour.put(colour,fixedColourList);
        }
        listsGroupedByColour.get(colour).add(apple);
    }

    Set<String> colours = listsGroupedByColour.keySet();

    printMap(listsGroupedByColour, colours);
}
```

The Java 8 way

```
@Test
public void groupApplesByColour() {
    Stream<Apple> appleStream = stock.stream();

    Map<String, List<Apple>> appleGroups =

    appleStream.collect(groupingBy(Apple::getColour));

    Set<String> colours = appleGroups.keySet();

    printMap(appleGroups, colours);

}
```

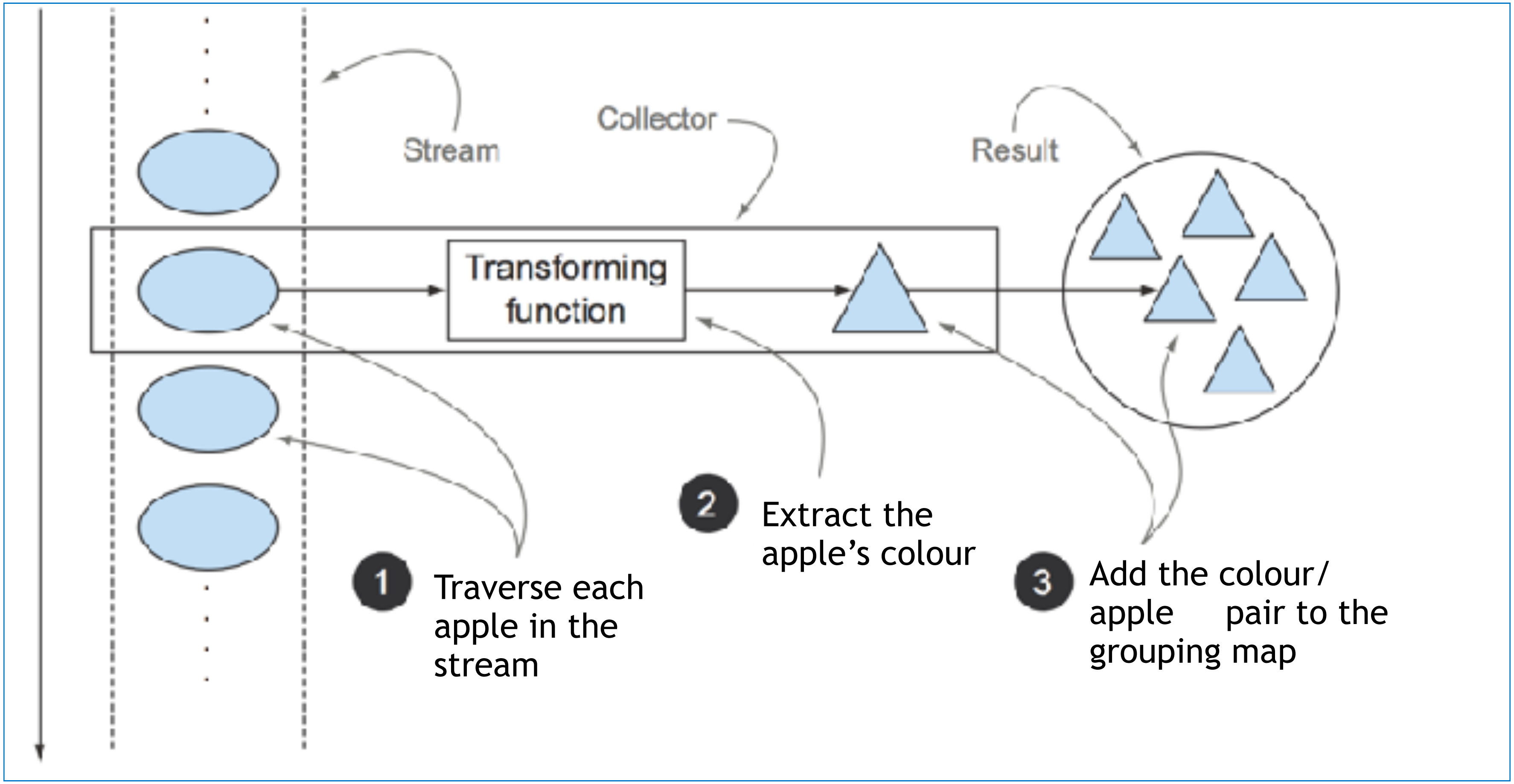
the Java 8 way

- Specify the what
- Not the how
- Better to read
- Better to maintain especially in multi level groupings

Collector in a nutshell

- The collect method expects an implementation of the Collector interface
- Collector is an interface of how to build a summary of elements in the stream
- the toList says: make a list of all the elements in the stream
- likewise groupingBy says make a map whose keys are colour buckets and whose values are Lists of apples of that colour

A picture of collection process



1 Traverse each apple in the stream

2 Extract the apple's colour

3 Add the colour/apple pair to the grouping map

Collectors as advanced reductions

- invoking the collect method on a stream
- triggers a reduction operation
- parameterized by the Collector interface
- Typically the Collector does the following
  - applies a transformation (often identity)
  - and accumulates the result in a data structure

Predefined collectors

- The Collectors class defines collectors

- Fall into 3 different groups:
- Reducing and summarising to one value
- Grouping elements
- Partitioning elements

Reducing and summarising to one value

```
@Test
public void countTheNumberOfApplesInStock() throws Exception {

    Collector<Apple, ?, Long> appleCounter = Collectors.counting();
    Long numberOfApples = stock.stream().collect(appleCounter);

    assertThat(numberOfApples,is(4L));

    //Note: same can be achieved with:

    stock.stream().count();

    //Collectors.counting() will show it's usefulness
}
```

minBy and maxBy

```
@Test
public void findingMinAndMaxValues() throws Exception {

    Comparator<Apple> appleWeightComparator=
            (apple1,apple2)->apple1.getWeight()-apple2.getWeight();

    Optional<Apple> hwApple =
          stock.stream().collect(Collectors.maxBy(appleWeightComparator));

    Optional<Apple> lwApple =
          stock.stream().collect(Collectors.minBy(appleWeightComparator));

    hwApple.ifPresent(System.out::println);

    lwApple.ifPresent(System.out::println);

}
```

Summarization

```
@Test
public void countingTheTotalWeightOfTheStockOfApples() throws Exception {

    Integer totalWeight =
            stock.stream().collect(summingInt((apple)->apple.getWeight()));

    System.out.println(totalWeight);

    IntSummaryStatistics appleStatistics =
            stock.stream().collect(summarizingInt(Apple::getWeight));

    System.out.println(appleStatistics);
}
```

```
Output println: IntSummaryStatistics
                {count=4, sum=550, min=80, average=137.500000, max=220}
```

Generalized summarization

```java
@Test
public void generalSummarization() throws Exception {
    Integer totalWeightSummingInt =
            stock.stream().collect(summingInt((apple)->apple.getWeight()));

    //is special case of:

    Function<Apple,Integer> transformer;
    BinaryOperator<Integer> aggregator;
    Integer startValue;

    startValue=0;
    transformer=Apple::getWeight;
    aggregator=(i,j)-> i+j;
    Integer totalWeightGeneralizedReduction =

            stock.stream().collect(reducing(startValue,transformer,aggregator));

    assertThat(totalWeightSummingInt, is(totalWeightGeneralizedReduction));

}
```

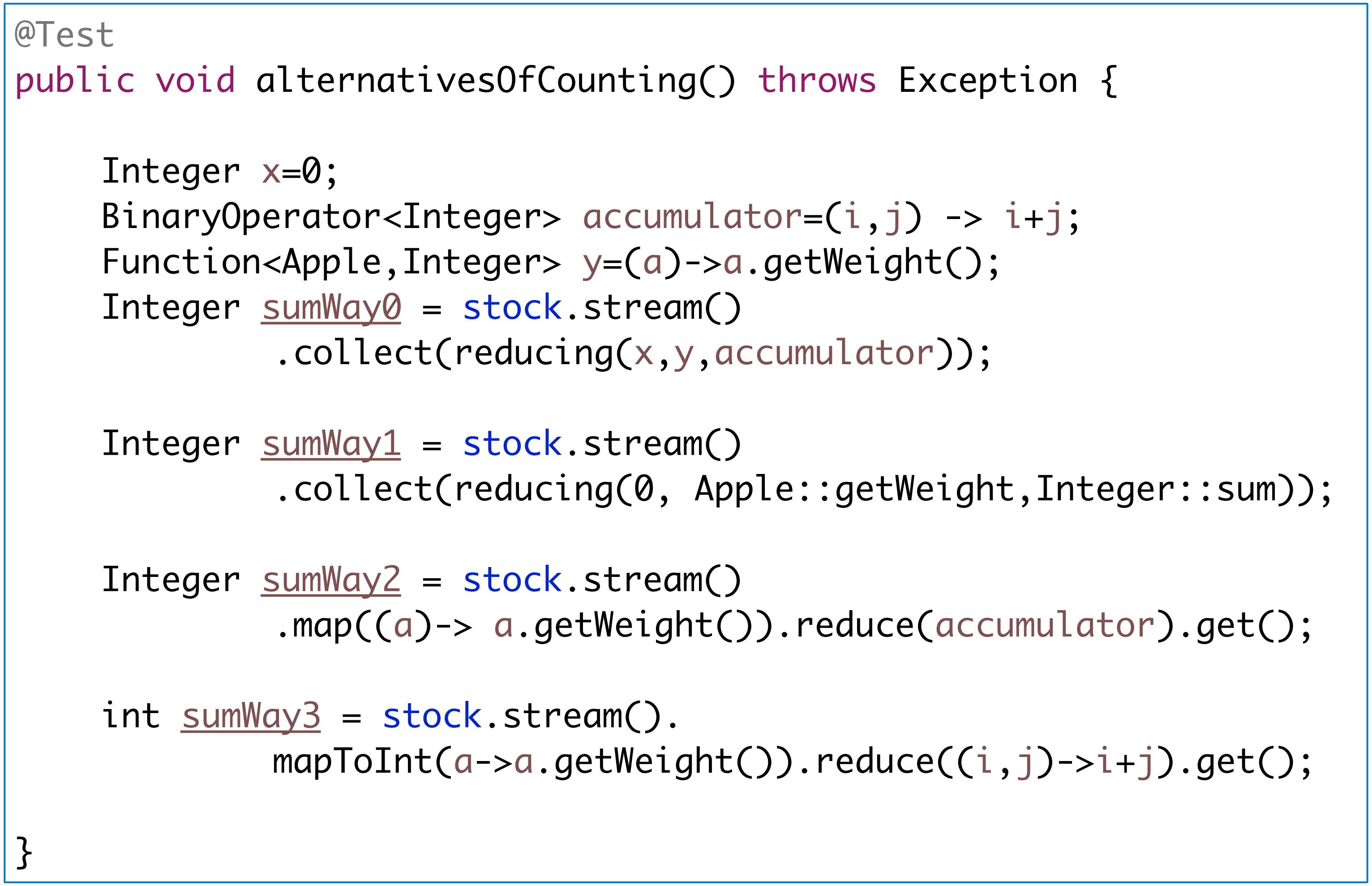
Reducing example

```
@Test
public void reducingWithOnlyAnAggregator()  {

    Function<Apple,Apple> transformer;
    BinaryOperator<Apple> aggregator;
    Apple firstAppleInTheStream;

    firstAppleInTheStream=new Apple("dummy",0);
    aggregator=(apple1,apple2)->
                     apple1.getWeight()>apple2.getWeight()?apple1:apple2;
    transformer=(apple)-> apple;//indentity operation

    Apple heaviestAppleByReduction1 =
             stock.stream().collect(reducing(firstAppleInTheStream,
                                             transformer,aggregator
                                             )
                                    );

    //Nearly equal to reducing(aggregator)
    Optional<Apple> heaviestAppleByReduction2 =
         stock.stream().collect(reducing(aggregator));
}
```

Collect versus reduce

- reduce is meant to combine 2 values and produce a new one
- i.e. reduce is an immutable reduction
- the collect method is meant to mutate a container that is supposed to accumulate the result

multiple ways to perform the same operation



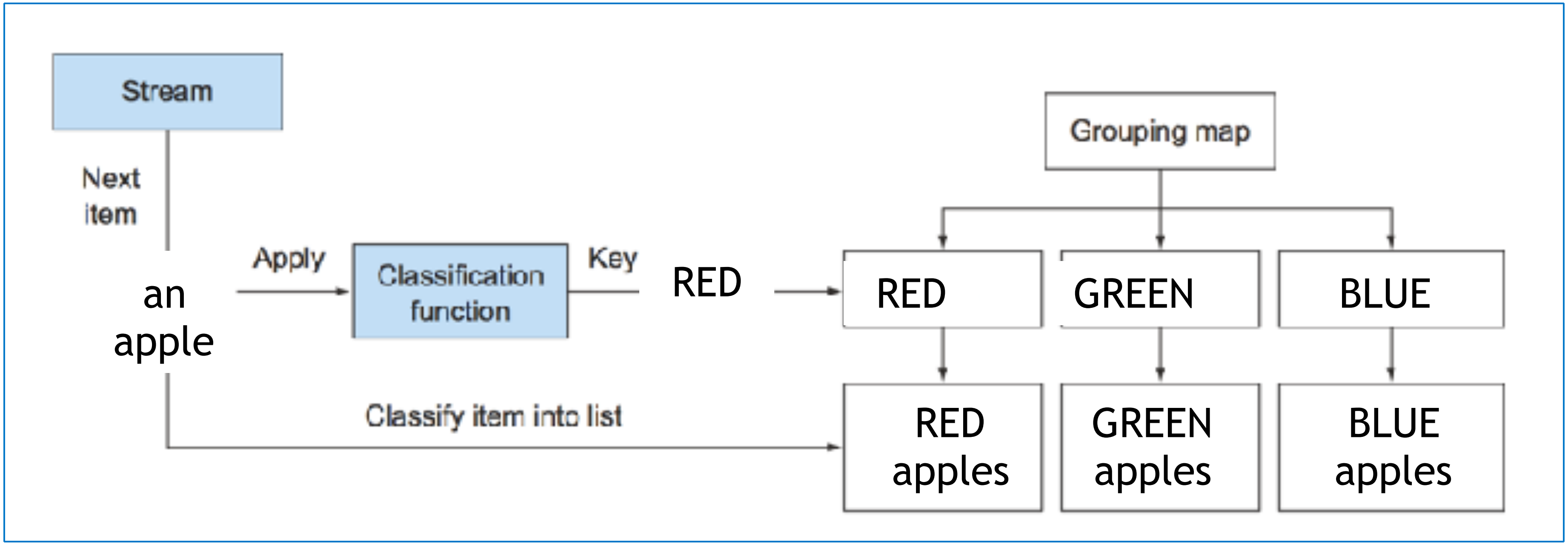
```
@Test
public void alternativesOfCounting() throws Exception {

    Integer x=0;
    BinaryOperator<Integer> accumulator=(i,j) -> i+j;
    Function<Apple,Integer> y=(a)->a.getWeight();
    Integer sumWay0 = stock.stream()
            .collect(reducing(x,y,accumulator));

    Integer sumWay1 = stock.stream()
            .collect(reducing(0, Apple::getWeight,Integer::sum));

    Integer sumWay2 = stock.stream()
            .map((a)-> a.getWeight()).reduce(accumulator).get();

    int sumWay3 = stock.stream().
            mapToInt(a->a.getWeight()).reduce((i,j)->i+j).get();

}
```

what to choose?

- Suggestion
  - explore the largest number of solutions
  - choose most specialised and general enough to solve the problem at hand
  - this goes often hand in hand with readability and performance

- In our example
  - calculate total weight of stock prefer last example
  - it's concise and no (un)boxing overhead

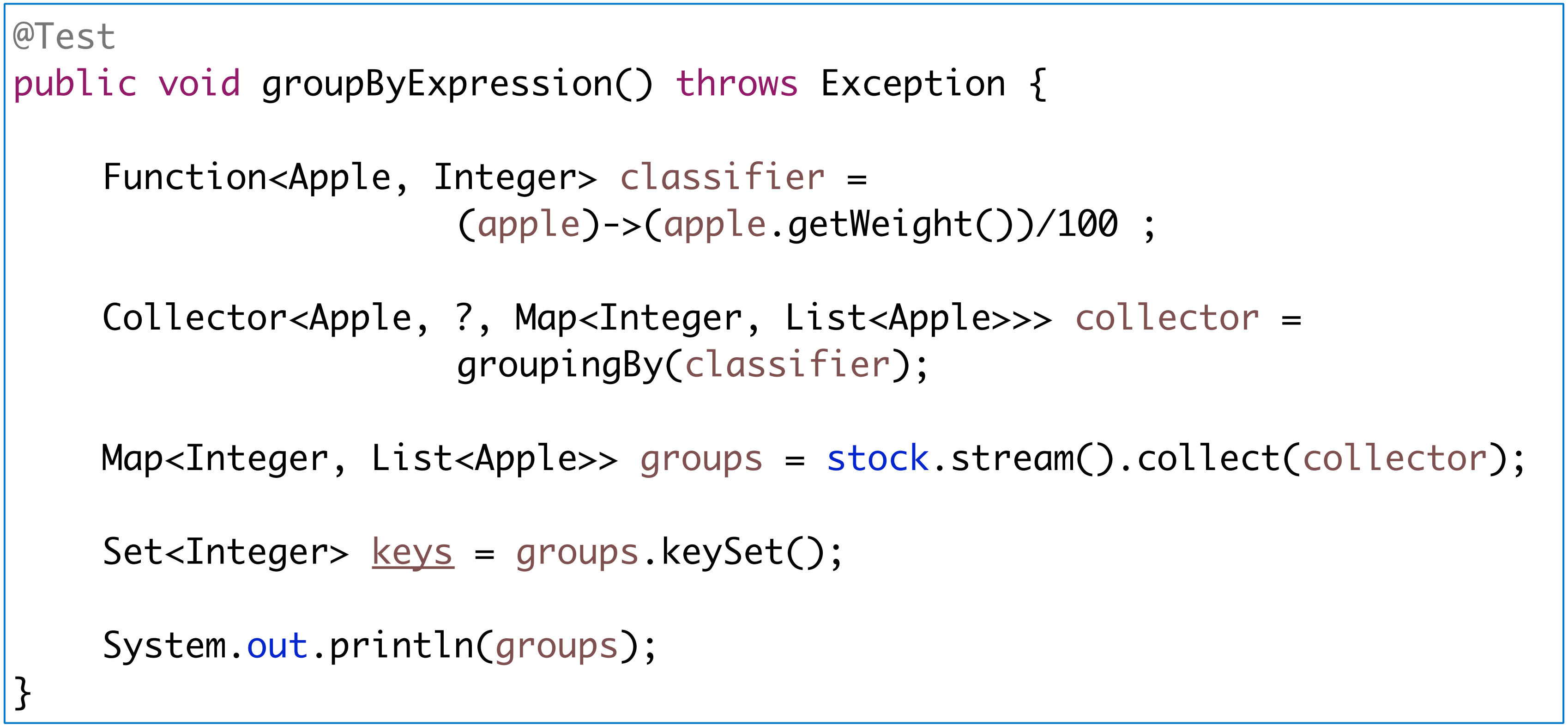
groupingBy


Stream
Next item
an apple
Apply
Classification function
Key
RED
Grouping map
RED
GREEN
BLUE
Classify item into list
RED apples
GREEN apples
BLUE apples

groupingBy an expression

```
@Test
public void itsTimeToGroup() throws Exception {

    Function<Apple, String> classifier = Apple::getColour;

    Collector<Apple, ?, Map<String, List<Apple>>> collector =
                                                 groupingBy(classifier);

    Map<String, List<Apple>> groups = stock.stream().collect(collector);

    Set<String> keys = groups.keySet();

    for (String colour : keys) {
        for (Apple apple : groups.get(colour)) {
            System.out.println(apple);
        }
    }
}
```

groupingBy an expression

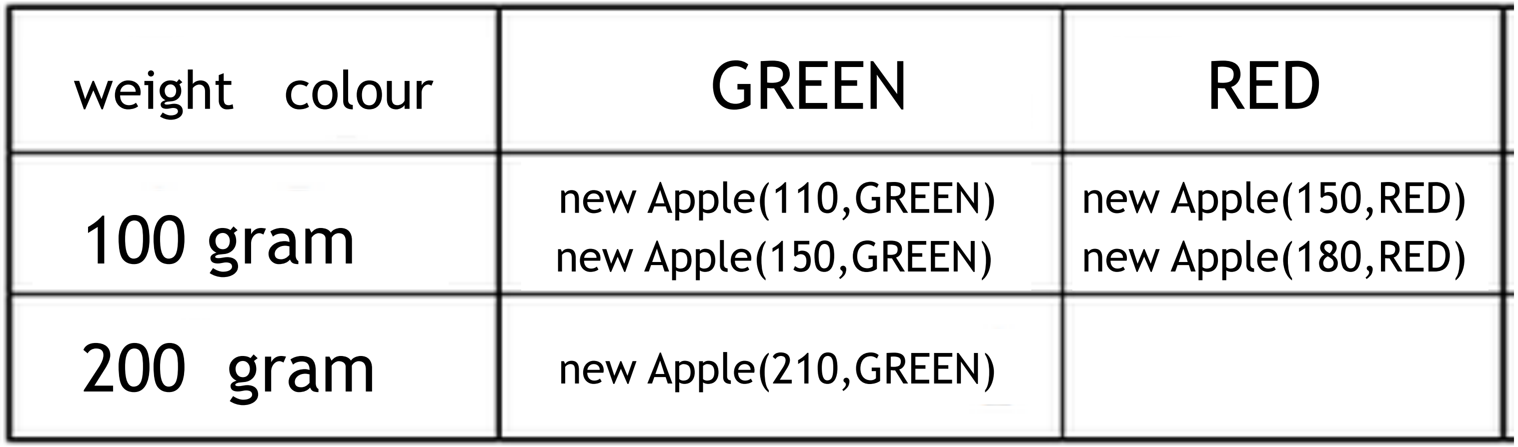


```java
@Test
public void groupByExpression() throws Exception {

    Function<Apple, Integer> classifier =
                    (apple)->(apple.getWeight())/100 ;

    Collector<Apple, ?, Map<Integer, List<Apple>>> collector =
                    groupingBy(classifier);

    Map<Integer, List<Apple>> groups = stock.stream().collect(collector);

    Set<Integer> keys = groups.keySet();

    System.out.println(groups);
}
```

multilevel grouping

```
@Test
public void groupingByWeightAndForEachWeightGroupGroupingByColour(){

    Function<Apple, String> colourClassifier = Apple::getColour;

    Collector<Apple, ?, Map<String, List<Apple>>> colourCollector
                                    = groupingBy(colourClassifier);

    Function<Apple, Integer> weightClassifier
                                    = (apple)->(apple.getWeight())/100 ;

    Collector<Apple, ?, Map<Integer, Map<String, List<Apple>>>> multiCollector
                                    = groupingBy(weightClassifier, colourCollector);
    Map<Integer, Map<String, List<Apple>>> groups
                                    = stock.stream().collect(multiCollector);

    System.out.println(groups);
}
```

Equivalence n-level nested map and n-dimensional classification table



weight colour	GREEN	RED
100 gram	new Apple(110,GREEN) new Apple(150,GREEN)	new Apple(150,RED) new Apple(180,RED)
200 gram	new Apple(210,GREEN)	

Using different type of collector as 2nd element

```
@Test
public void groupingByWeightAndCountingApplesInEachWeightGroup(){

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;
    //Note: reducing(x,y,z);
    // x is start value
    // y is transformer
    // z is aggregator
    Collector<Apple,?,Long> counter= reducing(0L,(apple)-> 1L,(i,j)->i+j);
    // equivalent to:
    counter=Collectors.counting();

    Collector<Apple, ?, Map<Long, Long>> weightCollector
                                    = groupingBy(weightClassifier,counter);
    Map<Long, Long> groups
                                     = stock.stream().collect(weightCollector);

    System.out.println(groups);
}
```

Note: Apple versus Optional<Apple>

```
@Test
public void groupingByWeightAndFindMostHeavyAppleInEachGroup(){

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;
    //Note: reducing(x,y,z);
    // x is start value,y is transformer,z is aggregator
    Collector<Apple,?,Apple> heavyAppleSearcher=
             reducing(stock.get(0),(a)-> a,
                              (a1,a2)->a1.getWeight()>a2.getWeight()?a1:a2);
    // nearly equivalent to:
    Comparator<Apple> comparator=(a1,a2)-> a1.getWeight()-a2.getWeight();
    Collector<Apple, ?, Optional<Apple>> heavyAppleSearcher1 =
                                                 Collectors.maxBy(comparator);

    Collector<Apple, ?, Map<Long, Optional<Apple>>> weightCollector
                             = groupingBy(weightClassifier,heavyAppleSearcher1);
    Map<Long, Optional<Apple>> groups
                              = stock.stream().collect(weightCollector);
    System.out.println(groups);
}
```

Optional<Apple> incidentally used

- Optional<Apple>
  - not expressing absence of value
  - incidentally there because of the signature of the reduce operation maxBy
  - the groupingBy collector lazily adds a key to the map when it finds one
  - the absence of a key is not noticed by design

- To get rid of the Optional
  - change the type returned
  - use collectiongAndThen

collectingAndThen to change the type returned

```
@Test
public void findMostHeavyAppleInEachWeightGroupAndThenChangeTheTypeReturned(){

    Comparator<Apple> comparator=(a1,a2)-> a1.getWeight()-a2.getWeight();
    Collector<Apple, ?, Optional<Apple>> heavyAppleSearcher1 = maxBy(comparator);

    Collector<Apple, ?, Optional<Apple>> downstream=heavyAppleSearcher1;
    Function<Optional<Apple>,Apple> finisher=(optApple)->optApple.get();

    Collector<Apple, ?, Apple> collectorAndTypeChanger =
                                       collectingAndThen(downstream, finisher);

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;

    Collector<Apple, ?, Map<Long, Apple>> weightCollector
                        = groupingBy(weightClassifier,collectorAndTypeChanger);

    Map<Long, Apple> groups = stock.stream().collect(weightCollector);

    System.out.println(groups);
}
```

About the second argument to the groupingBy factory method

- The collector passed as a second argument
  - performs a further reduction operation on all the elements in the stream classified into the same group

mapping: transform elements before collecting

```java
@Test public void mappingElementsInAGroup(){

    Function<Apple, Long> weightClassifier = (apple)->(apple.getWeight())/100L ;

    Collector<Apple,?,Set<CaloricLevel>> mapper=
            mapping((Apple apple)-> {
                                        if(apple.getWeight()<120) {
                                            return CaloricLevel.DIET;
                                        }else if(apple.getWeight()<180) {
                                            return CaloricLevel.NORMAL;
                                        }else {
                                            return CaloricLevel.HEAVY;
                                        }
                                    },toSet());
    Collector<Apple, ?, Map<Long, Set<CaloricLevel>>> weightCollector
                        = groupingBy(weightClassifier,mapper);
    Map<Long, Set<CaloricLevel>> groups
                         = stock.stream().collect(weightCollector);
    System.out.println(groups);
}
```

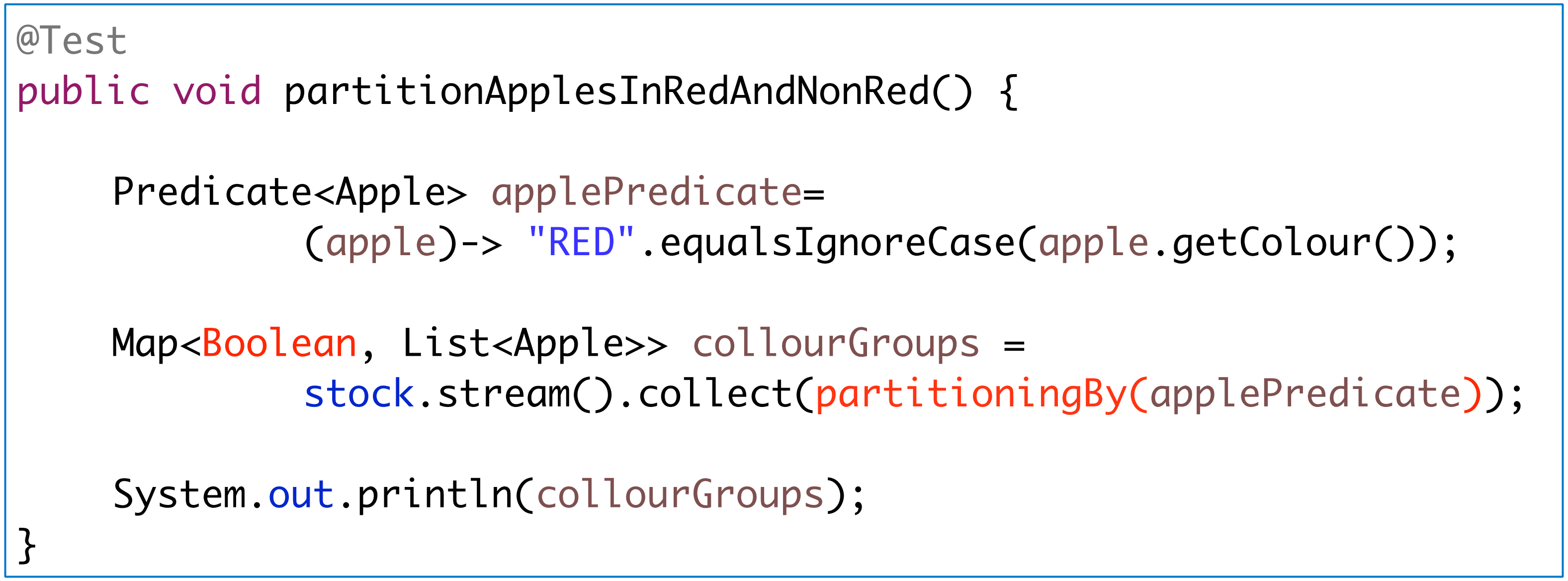
Notes about former slide

- Use the toSet method to remove duplicates
- When you want to specify the type of Set returned use the following syntax

```
Supplier<HashSet<CaloricLevel>> collectionFactory=HashSet::new;

Collector<Apple,?,HashSet<CaloricLevel>> mapper=
         mapping((Apple apple)-> {
                                       if(apple.getWeight()<120) {
                                           return CaloricLevel.DIET;
                                       }else if(apple.getWeight()<180) {
                                           return CaloricLevel.NORMAL;
                                       }else {
                                           return CaloricLevel.HEAVY;
                                       }

                                   },
                                   toCollection(collectionFactory));
```

Partitioning: a special case of groupingBy

- having a predicate as a partitioning function that serves as a classifier
- result in 2 groups
  - group for which predicate returns true and
  - a group for which predicate return false

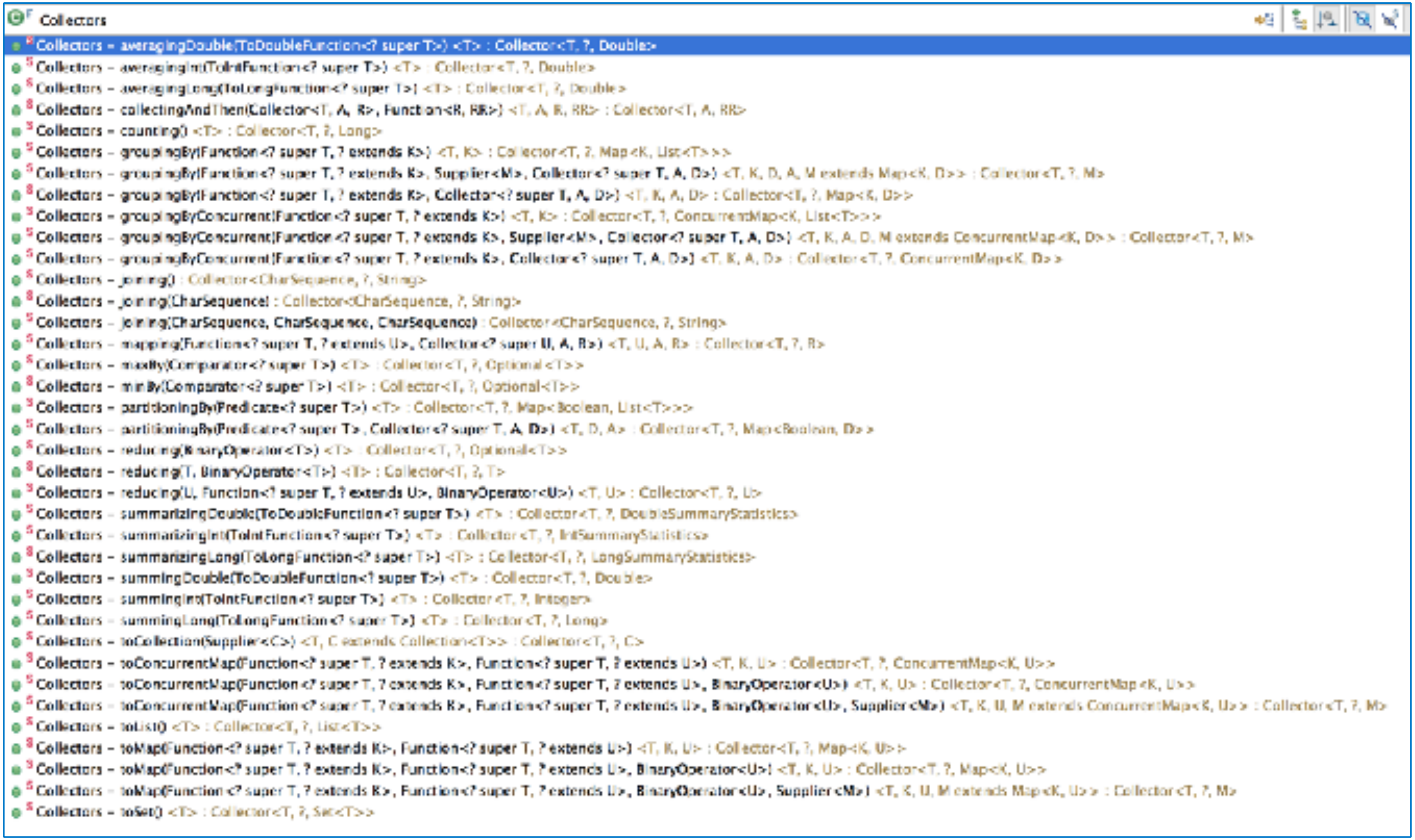


```
@Test
public void partitionApplesInRedAndNonRed() {

    Predicate<Apple> applePredicate=
            (apple)-> "RED".equalsIgnoreCase(apple.getColour());

    Map<Boolean, List<Apple>> collourGroups =
            stock.stream().collect(partitioningBy(applePredicate));

    System.out.println(collourGroups);
}
```

Divide numbers in prime and non prime numbers

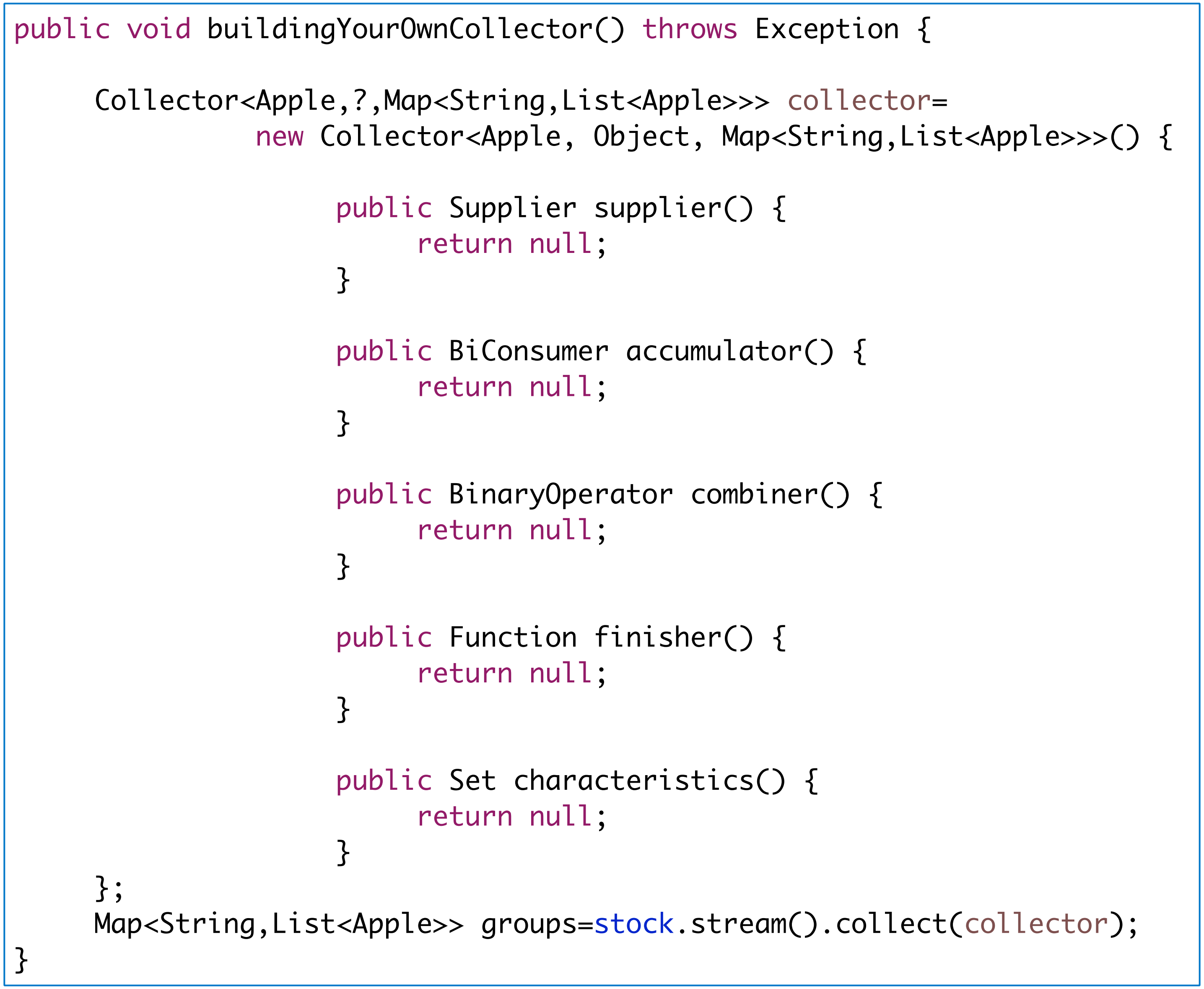
```java
public boolean isPrime(int n) {
    int upperLimit=(int)Math.sqrt(n);
    //Note: start at 2, everything is divisible by 1
    IntStream rangeClosed = IntStream.rangeClosed(2,upperLimit);
    IntPredicate intPredicate =(i)-> (n %i)==0;
    return (rangeClosed.noneMatch(intPredicate ));
}


@Test
public void partitionARangeOfValuesINPrimeAndNonPrime() throws Exception {
    final int n=1000;
    IntStream range = IntStream.range(1, n);

    Predicate<Integer> numberPredicate=i ->isPrime(i);
    Map<Boolean, List<Integer>> primeAndNonPrimeGroups =
            range.boxed().collect(partitioningBy(numberPredicate));
    System.out.println(primeAndNonPrimeGroups);
}
```

See static members of Collectors class for collectors



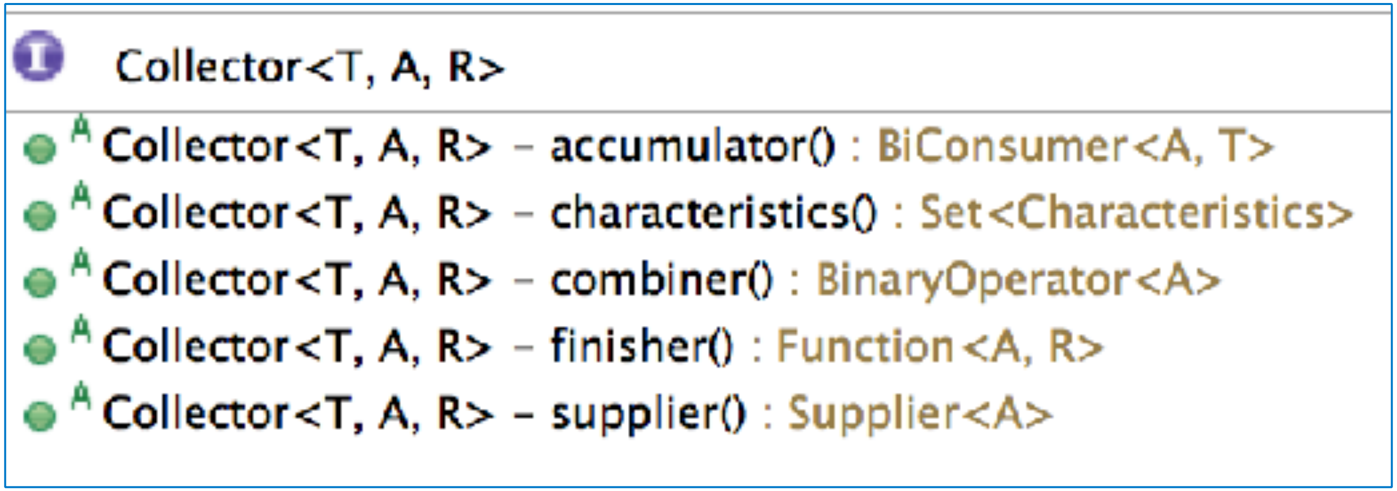
```
Collectors
Collectors - averagingDouble(ToDoubleFunction<? super T>) <T> : Collector<T, ?, Double>
Collectors - averagingInt(ToIntFunction<? super T>) <T> : Collector<T, ?, Double>
Collectors - averagingLong(ToLongFunction<? super T>) <T> : Collector<T, ?, Double>
Collectors - collectingAndThen(Collector<T, A, R>, Function<R, RR>) <T, A, R, RR> : Collector<T, A, RR>
Collectors - counting() <T> : Collector<T, ?, Long>
Collectors - groupingBy(Function<? super T, ? extends K>) <T, K> : Collector<T, ?, Map<K, List<T>>>
Collectors - groupingBy(Function<? super T, ? extends K>, Supplier<M>, Collector<? super T, A, D>) <T, K, D, A, M extends Map<K, D>> : Collector<T, ?, M>
Collectors - groupingBy(Function<? super T, ? extends K>, Collector<? super T, A, D>) <T, K, A, D> : Collector<T, ?, Map<K, D>>
Collectors - groupingByConcurrent(Function<? super T, ? extends K>) <T, K> : Collector<T, ?, ConcurrentMap<K, List<T>>>
Collectors - groupingByConcurrent(Function<? super T, ? extends K>, Supplier<M>, Collector<? super T, A, D>) <T, K, A, D, M extends ConcurrentMap<K, D>> : Collector<T, ?, M>
Collectors - groupingByConcurrent(Function<? super T, ? extends K>, Collector<? super T, A, D>) <T, K, A, D> : Collector<T, ?, ConcurrentMap<K, D>>
Collectors - joining() : Collector<CharSequence, ?, String>
Collectors - joining(CharSequence) : Collector<CharSequence, ?, String>
Collectors - joining(CharSequence, CharSequence, CharSequence) : Collector<CharSequence, ?, String>
Collectors - mapping(Function<? super T, ? extends U>, Collector<? super U, A, R>) <T, U, A, R> : Collector<T, ?, R>
Collectors - maxBy(Comparator<? super T>) <T> : Collector<T, ?, Optional<T>>
Collectors - minBy(Comparator<? super T>) <T> : Collector<T, ?, Optional<T>>
Collectors - partitioningBy(Predicate<? super T>) <T> : Collector<T, ?, Map<Boolean, List<T>>>
Collectors - partitioningBy(Predicate<? super T>, Collector<? super T, A, D>) <T, D, A> : Collector<T, ?, Map<Boolean, D>>
Collectors - reducing(BinaryOperator<T>) <T> : Collector<T, ?, Optional<T>>
Collectors - reducing(T, BinaryOperator<T>) <T> : Collector<T, ?, T>
Collectors - reducing(U, Function<? super T, ? extends U>, BinaryOperator<U>) <T, U> : Collector<T, ?, U>
Collectors - summarizingDouble(ToDoubleFunction<? super T>) <T> : Collector<T, ?, DoubleSummaryStatistics>
Collectors - summarizingInt(ToIntFunction<? super T>) <T> : Collector<T, ?, IntSummaryStatistics>
Collectors - summarizingLong(ToLongFunction<? super T>) <T> : Collector<T, ?, LongSummaryStatistics>
Collectors - summingDouble(ToDoubleFunction<? super T>) <T> : Collector<T, ?, Double>
Collectors - summingInt(ToIntFunction<? super T>) <T> : Collector<T, ?, Integer>
Collectors - summingLong(ToLongFunction<? super T>) <T> : Collector<T, ?, Long>
Collectors - toCollection(Supplier<C>) <T, C extends Collection<T>> : Collector<T, ?, C>
Collectors - toConcurrentMap(Function<? super T, ? extends K>, Function<? super T, ? extends U>) <T, K, U> : Collector<T, ?, ConcurrentMap<K, U>>
Collectors - toConcurrentMap(Function<? super T, ? extends K>, Function<? super T, ? extends U>, BinaryOperator<U>) <T, K, U> : Collector<T, ?, ConcurrentMap<K, U>>
Collectors - toConcurrentMap(Function<? super T, ? extends K>, Function<? super T, ? extends U>, BinaryOperator<U>, Supplier<M>) <T, K, U, M extends ConcurrentMap<K, U>> : Collector<T, ?, M>
Collectors - toList() <T> : Collector<T, ?, List<T>>
Collectors - toMap(Function<? super T, ? extends K>, Function<? super T, ? extends U>) <T, K, U> : Collector<T, ?, Map<K, U>>
Collectors - toMap(Function<? super T, ? extends K>, Function<? super T, ? extends U>, BinaryOperator<U>) <T, K, U> : Collector<T, ?, Map<K, U>>
Collectors - toMap(Function<? super T, ? extends K>, Function<? super T, ? extends U>, BinaryOperator<U>, Supplier<M>) <T, K, U, M extends Map<K, U>> : Collector<T, ?, M>
Collectors - toSet() <T> : Collector<T, ?, Set<T>>
```

Creating your own collectors



```
public void buildingYourOwnCollector() throws Exception {

     Collector<Apple,?,Map<String,List<Apple>>> collector=
               new Collector<Apple, Object, Map<String,List<Apple>>>() {

                    public Supplier supplier() {
                         return null;
                    }

                    public BiConsumer accumulator() {
                         return null;
                    }

                    public BinaryOperator combiner() {
                         return null;
                    }

                    public Function finisher() {
                         return null;
                    }

                    public Set characteristics() {
                         return null;
                    }
     };
     Map<String,List<Apple>> groups=stock.stream().collect(collector);
}
```

The Collector interface



Collector<T, A, R>

- Collector<T, A, R> – accumulator() : BiConsumer<A, T>
- Collector<T, A, R> – characteristics() : Set<Characteristics>
- Collector<T, A, R> – combiner() : BinaryOperator<A>
- Collector<T, A, R> – finisher() : Function<A, R>
- Collector<T, A, R> – supplier() : Supplier<A>

- T is the type of the items in the stream to be collected
- A is the type of the accumulator, the object on which the partial result will be accumulated during the collection process
- R is the type of the object (typically, but not always, the collection) resulting from the collect operation

MAKING A NEW RESULT CONTAINER: THE SUPPLIER METHOD

```
@Test
public void buildingYourOwnCollector() {

    Collector<Apple,Map<String,List<Apple>>,Map<String,List<Apple>>> collector=

      new Collector<Apple, Map<String,List<Apple>>,Map<String,List<Apple>>>() {

            //should create and return a new mutable result container
            //supplier delivers starting value for the accumulator
            public Supplier<Map<String,List<Apple>>> supplier() {
                return () -> new HashMap<String,List<Apple>>();
            }
            ……. .
    };
    Map<String,List<Apple>> groups=stock.stream().collect(colllector);
}
```

ACCUMULATING ITEMS TO THE CONTAINER: THE ACCUMULATOR METHOD

```
public void buildingYourOwnCollector() throws Exception {

    Collector<Apple,Map<String,List<Apple>>,Map<String,List<Apple>>> collector=
        new Collector<Apple, Map<String,List<Apple>>, Map<String,List<Apple>>>() {

                ….. 
                //should  fold a value into a mutable result container
                public BiConsumer<Map<String,List<Apple>>,Apple> accumulator() {
                    return (map, a)->{
                        if(map.get(a.getColour())==null) {
                            map.put(a.getColour(), new ArrayList<Apple>());
                        }
                        map.get(a.getColour()).add(a);};
                }

    …..

    Map<String,List<Apple>> groups=stock.stream().collect(colllector);
}
```

combiner, finisher and characteristics

```
Collector<Apple,Map<String,List<Apple>>,Map<String,List<Apple>>> collector=
    new Collector<Apple, Map<String,List<Apple>>, Map<String,List<Apple>>>() {

    ….
    // should accept two partial results and merges them
    public BinaryOperator<Map<String,List<Apple>>> combiner() {
        return (map1,map2)->{
            Map<String,List<Apple>> sumMap=new HashMap<String,List<Apple>>();
                    sumMap.putAll(map1);
                    sumMap.putAll(map2);
                    return sumMap;
                };
    }

    // should transform the intermediate result to the final result
    public Function<Map<String,List<Apple>>,Map<String,List<Apple>>> finisher() {
            return Function.identity();
    }
    // describes properties of the collector
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
    }
};
```

combiner, finisher

- the combiner
  - allows a parallel reduction of the stream
  - uses the fork/join framework
  - stream is recursively split in substreams until a stop condition is met
  - combiner combines the partial results
- the finisher
  - has to return a function that’s invoked at the end of the accumulation process
  - transforms the accumulator object into the final result
  - often the accumulator coincides with the expected result so the finisher equals the Funtion.identity()

characteristics method

- the characteristics method
  - returns an immutable set of Characteristics, defining the behavior of the collector
  - providing hints
  - can the stream be reduced in parallel

- Characteristics is an enumeration
  - UNORDERED— result of the reduction independent of order of traversal or accumulation
  - CONCURRENT— accumulator can be called concurrently
  - IDENTITY_FINISH—says the finisher method is the identity, accumulator is the final result

Parallel data processing and performance

Chapter 6

Agenda

- Processing data in parallel with parallel Streams
- Performance analysis of parallel Streams
- The Fork/Join framework
- Splitting a Stream of data using a Spliterator

Processing data in parallel

- With the new Stream interface collections of data can be manipulate in a declarative way
- the shift from external to internal iteration enables the native Java library to gain control over how to iterate the elements of a Stream
- This opens the possibility to execute a pipeline of operations on these collections that automatically makes use of the multiple cores on the computer

Parallel processing before Java 7

1. split the data structure containing your data into subparts
2. assign each of these subparts to a different thread
3. synchronise them opportunely to avoid unwanted race conditions
4. wait for the completion of all threads
5. finally reaggregate the partial results
6. quite cumbersome!

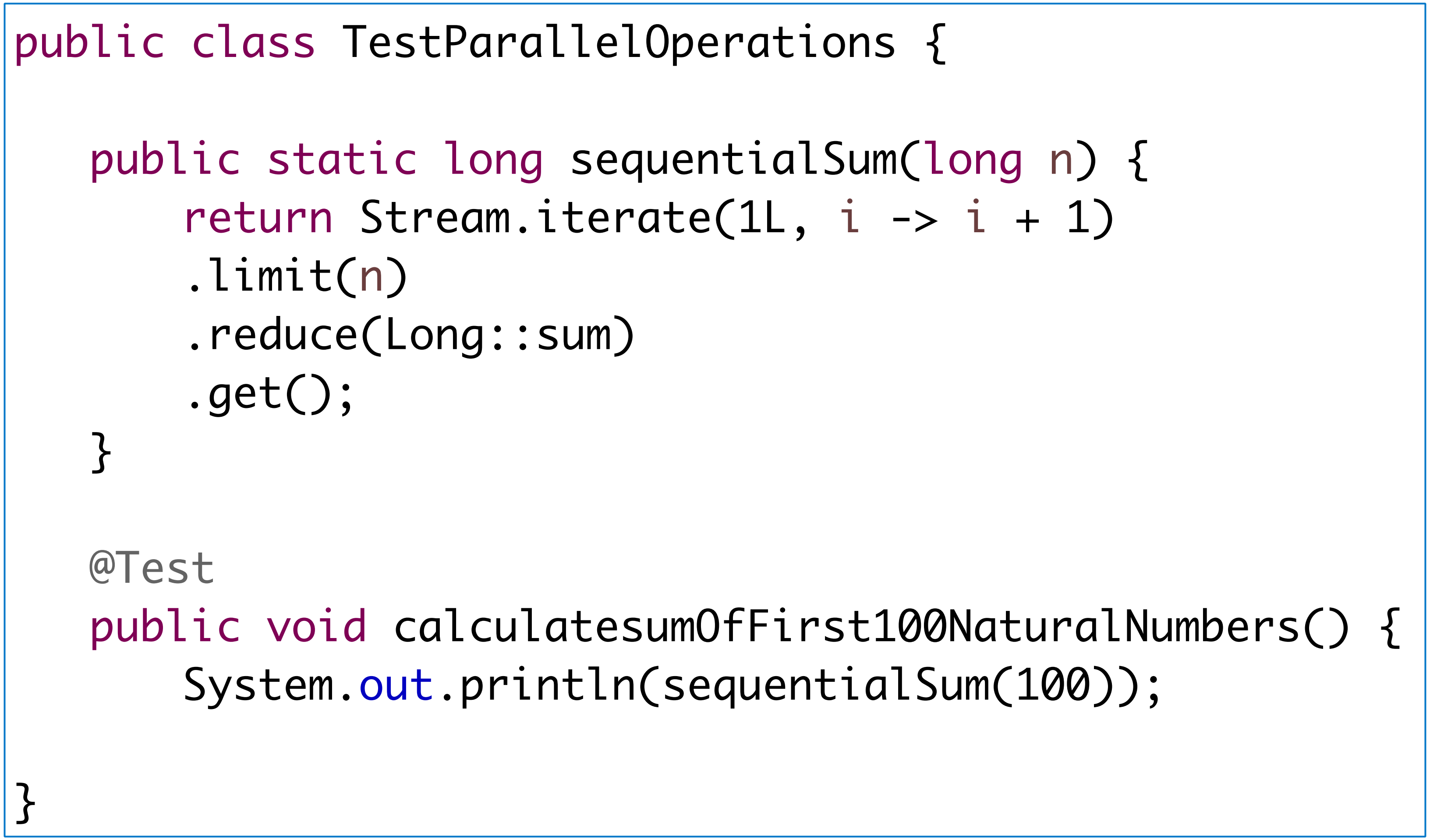
Fork/Join framework

- Java 7 introduced the Fork/Join framework
  - to perform these operations more
  - consistently and
  - in a less error prone way
- Still difficult to use
- Later on we will look at this possibility

parallel Streams

- important to know how parallel Streams work internally
- if this aspect is ignored, unexpected results could appear
- the way a parallel Stream gets divided into chunks before processing the different chunks in parallel can be the origin of odd results
- take control of this splitting process by implementing and using custom Spliterator

An example: Calculate sum



```java
public class TestParallelOperations {

   public static long sequentialSum(long n) {
       return Stream.iterate(1L, i -> i + 1)
       .limit(n)
       .reduce(Long::sum)
       .get();
   }

   @Test
   public void calculatesumOfFirst100NaturalNumbers() {
       System.out.println(sequentialSum(100));

}
```

Compare code with traditional way

```
public static long sequentialSum(long n) {
      return Stream.iterate(1L, i -> i + 1)
      .limit(n)
      .reduce(Long::sum)
      .get();
  }
```

```
public static long iterativeSum(long n) {
       long result = 0;
       for (long i = 1; i <= n; i++) {
              result += i;
       }
       return result;
       }
}
```

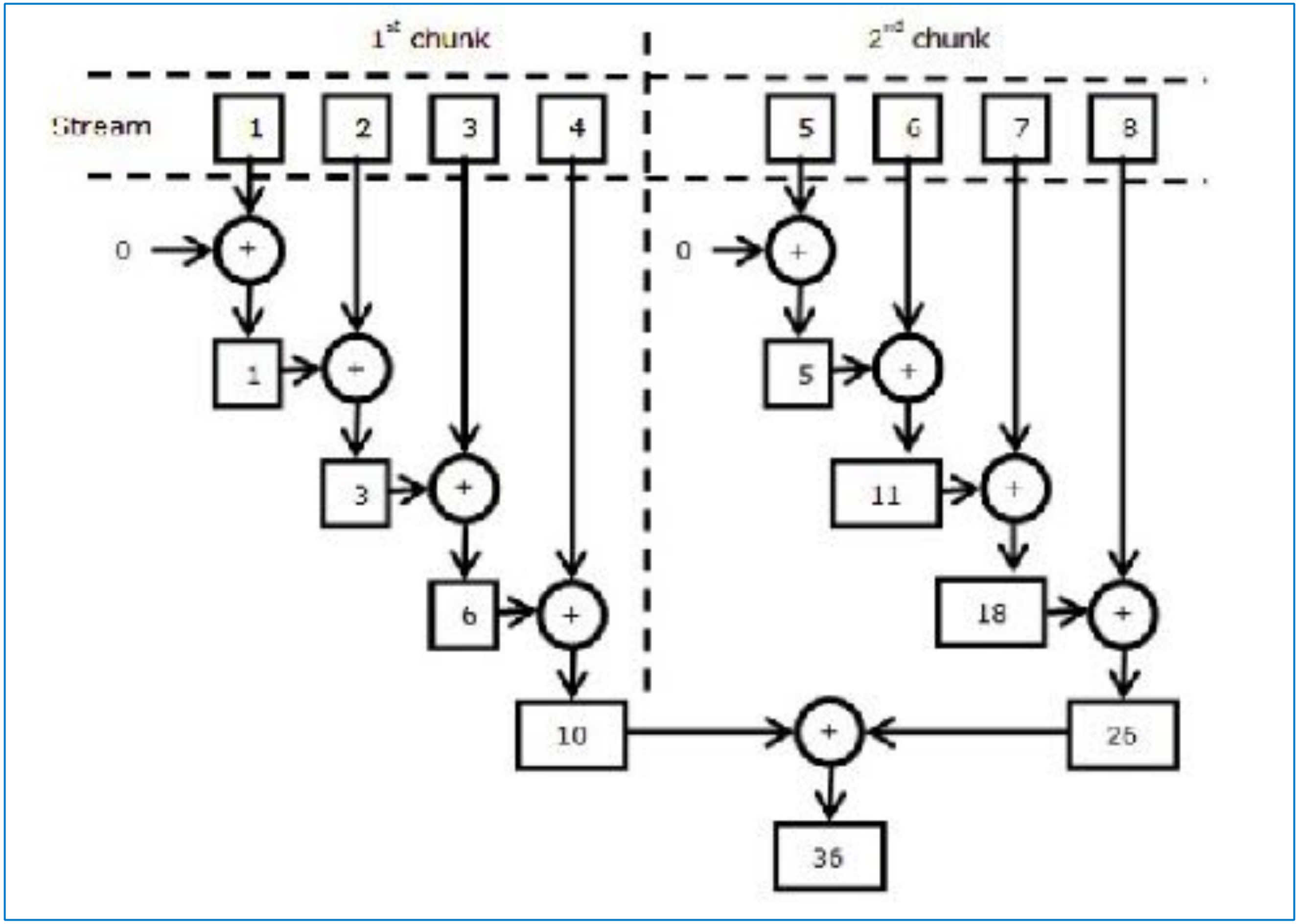
What must be done

- This operation seems to be a good candidate to leverage parallelization especially for large values of **n**
- However where to start?
- Is synchronising on the result variable a good idea?
- How many threads to use?
- Who does the generation of numbers?
- Who adds them up?
- Stop worrying adopt parallel streams!

Turn stream into a parallel one

```
public static long parallelSum(long n) {

      return Stream.iterate(1L, i -> i + 1)
                   .limit(n)
                   .parallel()
                   .reduce(Long::sum)
                   .get();
}
```

Parallel stream uses chunks



- The stream is internally divided into multiple chunks

What happens when calling parallel?

- Note: calling parallel() on a sequential Stream doesn’t imply any concrete transformation on the Stream itself
- Internally, a boolean flag is set to signal that all the operations that follow the parallel() invocation must be run in parallel
- Switching from and to parallel/sequential processing can be achieved by calling parallel() and sequential()

Parallel, sequential just say so

```
public static long m (long n) {
return Stream.iterate(……)
 .limit(n)
 .parallel()
 .filter(….)
 .sequential()
 .map(…)
 .parallel()
 .reduce(….);
}
```

- The filter() and reduce() operations will be performed in parallel,
- The map() operation sequentially

Put the parallel option to the test

- Compare:

```
public static long parallelSum(long n) {
      return Stream.iterate(1L, i -> i + 1)
                   .limit(n)
                   .parallel()
                   .reduce(Long::sum)
                   .get();
}
```

- and:

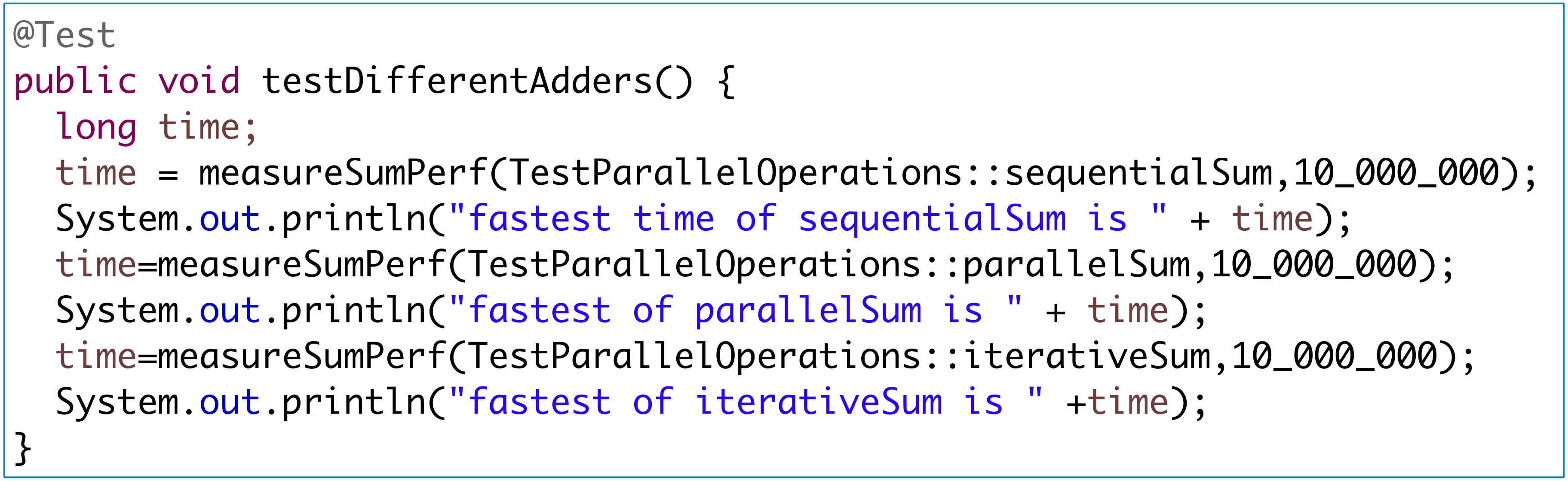
```
 public static long sequentialSum(long n) {
      return Stream.iterate(1L, i -> i + 1)
      .limit(n)
      .reduce(Long::sum)
      .get();
   }
```

The testharness

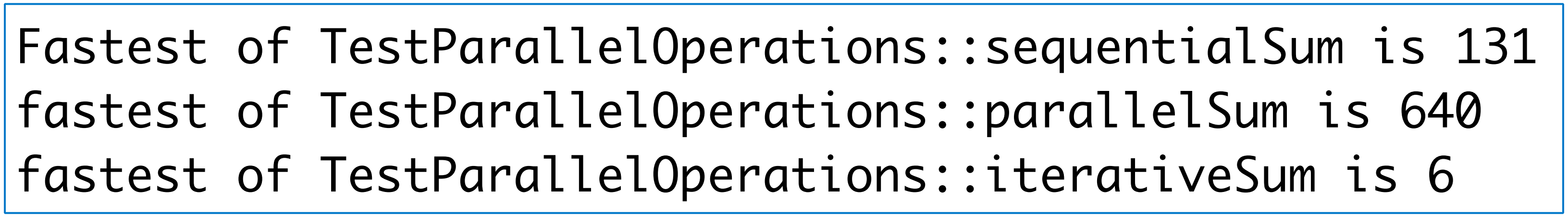
```java
public long measureSumPerf(Function<Long, Long> adder,
                                                    long n) {
   long fastest = Long.MAX_VALUE;
   for (int i = 0; i < 10; i++) {
      long start = System.nanoTime();
      long sum = adder.apply(n);
      long duration = (System.nanoTime() - start) / 1_000_000;
      System.out.println("Result: " + sum);
      if (duration < fastest)
         fastest = duration;
    }
  return fastest;
}
```

- Each method parallelSum and sequentialSum will be called 10-times, the fastest execution time is printed

The raw data



```
@Test
public void testDifferentAdders() {
  long time;
  time = measureSumPerf(TestParallelOperations::sequentialSum,10_000_000);
  System.out.println("fastest time of sequentialSum is " + time);
  time=measureSumPerf(TestParallelOperations::parallelSum,10_000_000);
  System.out.println("fastest of parallelSum is " + time);
  time=measureSumPerf(TestParallelOperations::iterativeSum,10_000_000);
  System.out.println("fastest of iterativeSum is " +time);
}
```



```
Fastest of TestParallelOperations::sequentialSum is 131
fastest of TestParallelOperations::parallelSum is 640
fastest of TestParallelOperations::iterativeSum is 6
```

- Very disappointing! The parallelSum is by far the slowest! Why?
- Note: iterative, the old fashioned java way is fast because summing and generation of numbers is combined

Why this unexpected result?

- There are two issues

1. iterate() generates boxed objects, which have to be unboxed to numbers before they can be added
2. iterate() is difficult to divide into independent chunks to execute in parallel

- a mental model that some stream operations are more “parallelizable” than others is handy

Behind the scenes

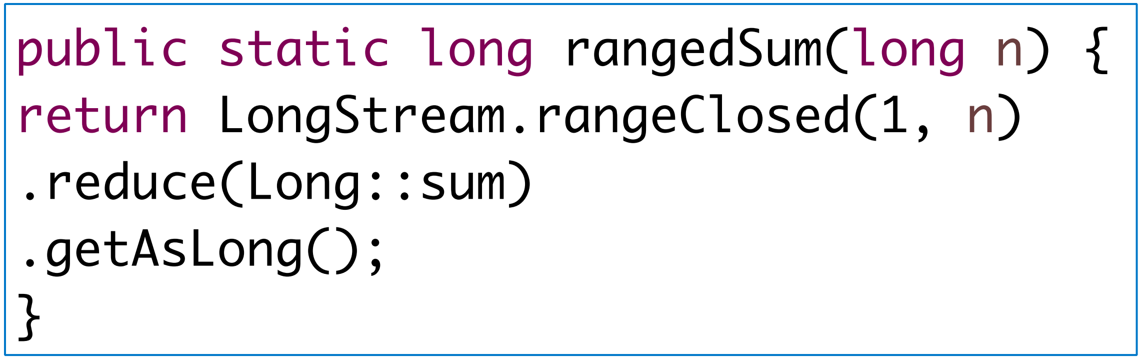
- The whole list of numbers is not available at the beginning of the reduction process
- The Stream can’t partition itself efficiently in chunks to be processed in Parallel
- By flagging the Stream as parallel only overhead of allocating each sum operation on a different thread is added to the sequential processing
- parallel programming can be tricky!

Using more specialized methods

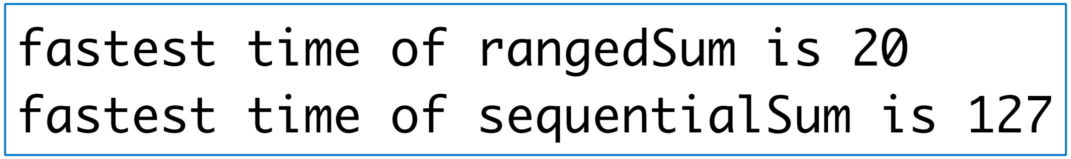
- how can the parallel Stream be used to leverage the multicore processors in an effective way
- Use LongStream.rangeClosed()
  - 2 advantages compared to iterate()

1. It works on primitive long numbers directly so there’s no boxing and unboxing overhead
2. It produces ranges of numbers, which can be easily splitted into independent chunks.

What is the (un)box overhead?



```
public static long rangedSum(long n) {
return LongStream.rangeClosed(1, n)
.reduce(Long::sum)
.getAsLong();
}
```



```
fastest time of rangedSum is 20
fastest time of sequentialSum is 127
```

- Note ranged sum is not yet parallelized
- Apparently there is a huge overhead in boxing and unboxing

Can parallelize help further?

```java
public static long parallelRangedSum(long n) {
return LongStream.rangeClosed(1, n)
       .parallel()
       .reduce(Long::sum)
       .getAsLong();
}
```

```
fastest of parallelRangedSum is 3
Finally an improvement over the old “pre” java 8 iteration
fastest of iterativeSum is 6
```

```java
public static long iterativeSum(long n) {
  long result = 0;
  for (long i = 1; i <= n; i++) {
       result += i;
  }
  return result;
}
```

Prerequisite for parallel succes

1. using the right data structure
2. and then making it work in parallel
3. Important:

The parallelization process isn’t free

- Requires to recursively partition the Stream
- Assign the reduction operation of each sub stream to a different thread
- Reaggregate the results of these operations in a single value
- Moving data between multiple cores can be expensive
  - so it’s important that work done in parallel on another core takes longer than the time taken to transfer the data from one core to another

Using parallel Streams correctly

- Plausible alternative

```
public class Accumulator {
    public long total = 0;
    public void add(long value) { total += value; }
}

public static long sideEffectSum(long n) {
   Accumulator accumulator = new Accumulator();
   LongStream.rangeClosed(1, n)
             .parallel()
             .forEach(accumulator::add);
   return accumulator.total;
}
```

```
fastest time of parallelSum is 2
```

Times aren’t everything

- Note: the swaggering result!

```
Result: 10308601425092
Result: 14207428618068
Result: 2099609531250
Result: 4938067001829
Result: 6676516192027
Result: 4270900971005
Result: 8000289514804
Result: 6459467974634
Result: 7897230220057
Result: 5516525834428

Instead of:
Result: 50000005000000
```

- Shared state isn’t save with muliple threads!

Recommendations

- Some qualitative advice that could be useful when considering to use a parallel Stream
- If in doubt, measure
- Watch out for boxing
  - Java 8 includes primitive Streams (IntStream, DoubleStream etc.) for this reason
    - they often outweigh the benefits of parallel Streams

Recommendations

- Some operations naturally perform worse on a parallel Stream
  - i.e. limit and findFirst that rely on the order of the elements
  - i.e. findAny will perform better than findFirst because it is not constrained to operate in the encounter order
- You can always turn an ordered Stream into an unordered one

Recommendations

- For small amount of data, choosing a parallel Stream is almost never a winning decision
- Take into account how well the data structure underlying the Stream decomposes.
  - i.e. ArrayList can be split much more efficiently than a LinkedList, because the first can be evenly divided without traversing it

Recommendations

- The characteristics of a Stream, and how the intermediate operations can modify them
  - a SIZED Stream can be divided into equal parts, and be processed in parallel more effectively
  - a filter operation can throw away an unpredictable number of elements, making the size of the Stream itself unknown
- Consider whether a terminal operation has a cheap or expensive merge step
  - If expensive then re-aggregation of the partial results can outweigh the parallelization

The ForkJoin framework

- Designed to:
  - recursively split a parallelizable task
  - then combine the results of each subtask
- Implements the ExecutorService interface
  - This implementation distributes those subtasks to worker threads
  - The threads are part of a thread pool, called the ForkJoinPool

Working with RecursiveTask

- The ForkJoinPool expects tasks that
  - subclasses RecursiveTask<R> R is the resulttype of the parallelized task
  - or implementations of RecursiveAction if the task returns no result

```
class ParallelTask extends RecursiveTask<T>{

   @Override
   protected T compute(){
      return null;
   }
}
NOTE:
public abstract class RecursiveTask<V> extends ForkJoinTask<V>
```

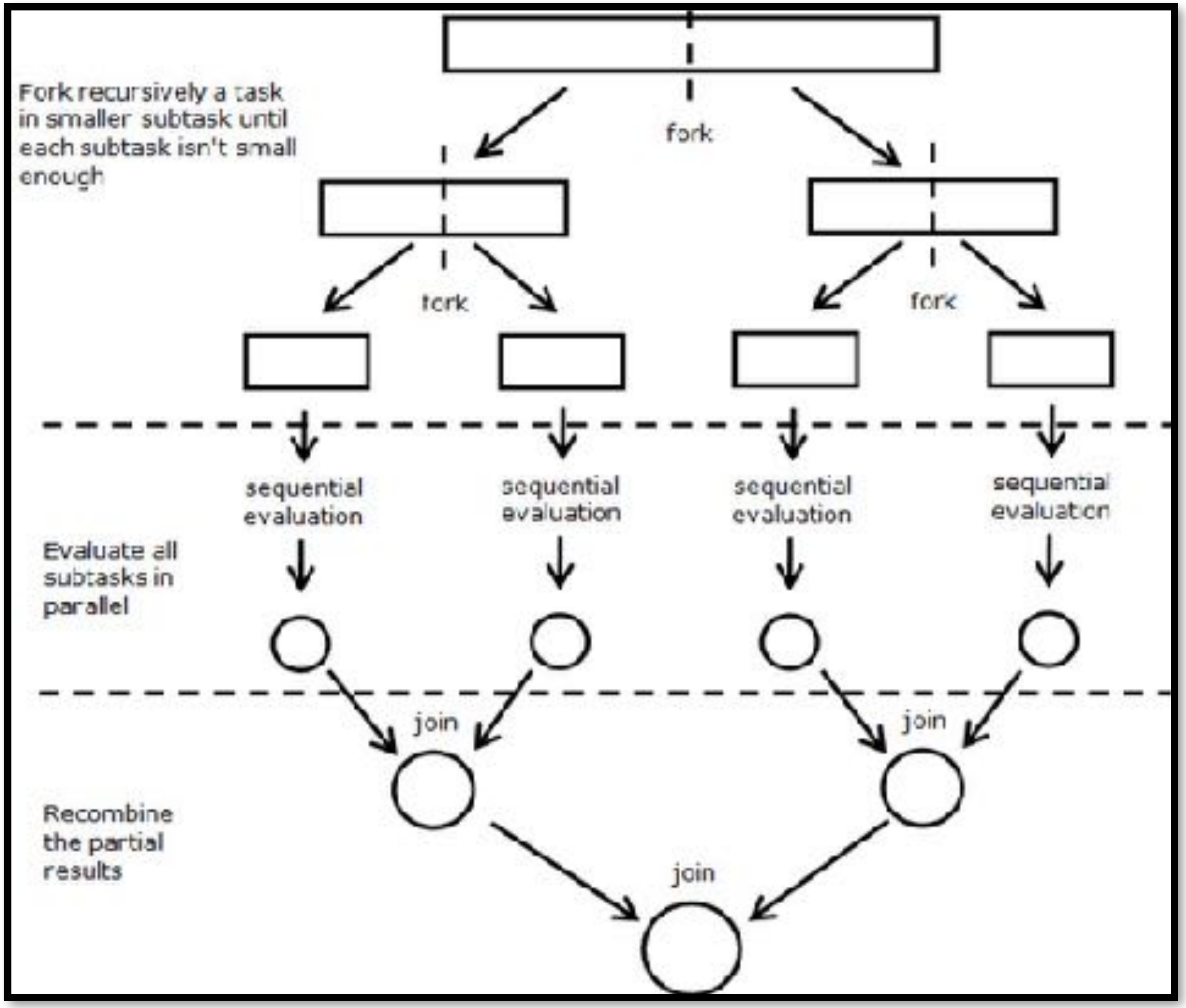
protected abstract T compute()

- Should contain logic for:
  - splitting the task at hand into subtasks
  - and the algorithm to produce the result of a single subtask when it’s no longer convenient to further divide it

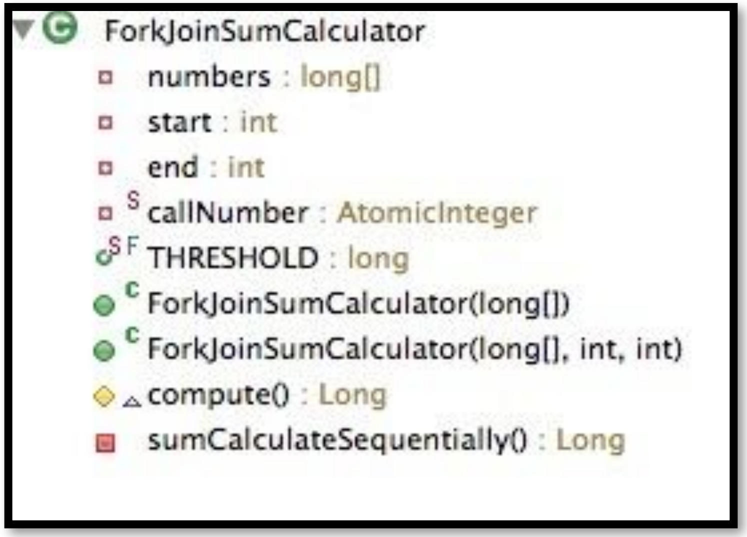
General pattern

```
if (task is small enough or no longer divisible) {
        compute task sequentially
} else {
        split task in two subtasks
        call this method recursively possibly further splitting each subtask
        wait for the completion of all subtasks
        combine the results of each subtask
}
```

In a picture


Fork recursively a task in smaller subtask until each subtask isn't small enough
fork
fork
fork
sequential evaluation
sequential evaluation
sequential evaluation
sequential evaluation
Evaluate all subtasks in parallel
join
join
Recombine the partial results
join

An example: Summing long array



```
ForkJoinSumCalculator
    numbers : long[]
    start : int
    end : int
    callNumber : AtomicInteger
    THRESHOLD : long
    ForkJoinSumCalculator(long[])
    ForkJoinSumCalculator(long[], int, int)
    compute() : Long
    sumCalculateSequentially() : Long
```

```
@Test
public void test() {
  long[] numbers = LongStream.rangeClosed(0L, 1_000_000L)
                             .toArray();
  ForkJoinSumCalculator fjSum =
           new ForkJoinSumCalculator(numbers);
  Long sum = new ForkJoinPool().invoke(fjSum);
  System.out.println(sum);
}
```

Structure of ForkJoinSumCalculator

```
public class ForkJoinSumCalculator extends RecursiveTask<Long>{
    private long [] numbers;
    private int start;
    private int end;
    private static  AtomicInteger callNumber = new AtomicInteger(0);

   public static final long THRESHOLD=10_000 ;

   public ForkJoinSumCalculator(long[] numbers) {
        super();
        this.numbers = numbers;
        start=0;
        end=numbers.length;
   }
   public ForkJoinSumCalculator(long[] numbers, int start, int eind) {
        System.out.printf("callNumber  %4d start =  %d  einde = %d\n",
                          callNumber.getAndIncrement(),start,eind);
    this.numbers=numbers;
    this.start=start;
    this.end=eind;
}
```

….continued on next page

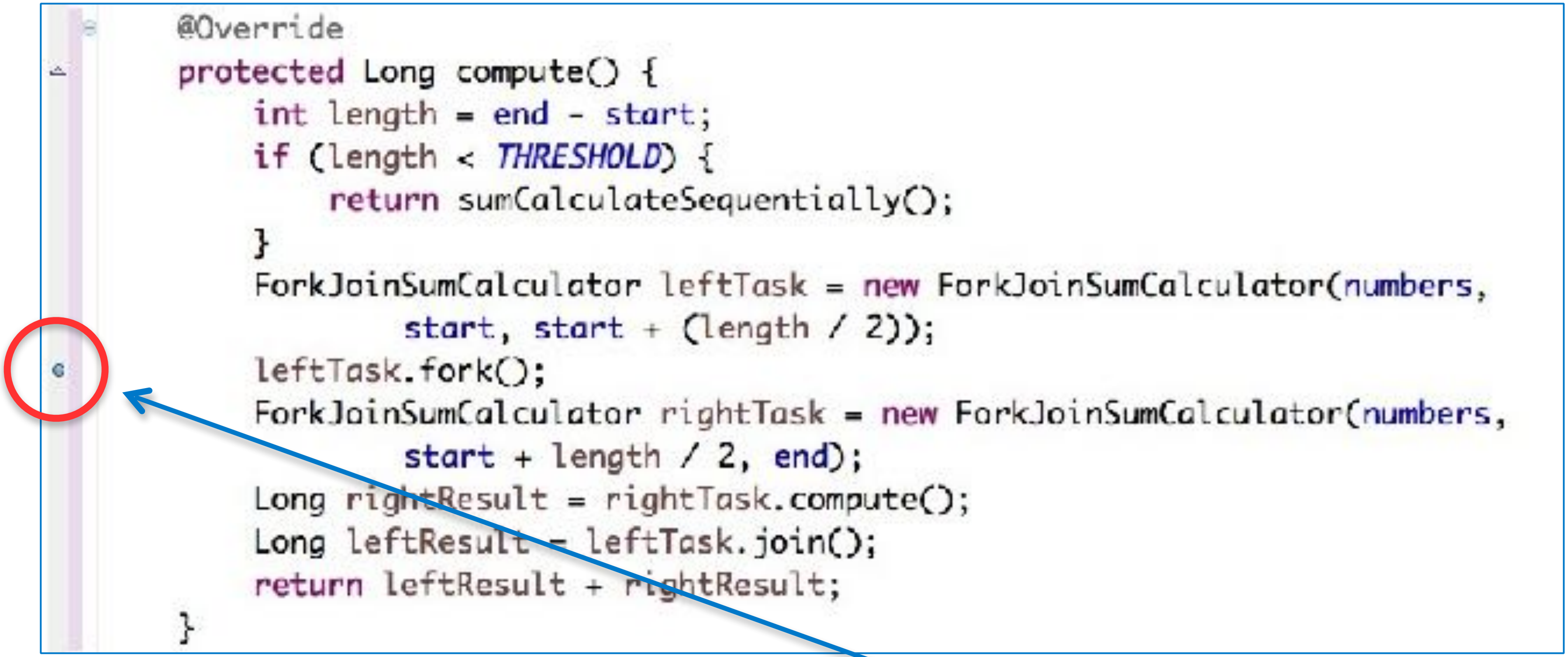
ForkJoinSumCalculator continued

```java
@Override
protected Long compute() {
    int length=end-start;
    if(length<THRESHOLD) {
        return sumCalculateSequentially();
    }
    ForkJoinSumCalculator leftTask =
        new ForkJoinSumCalculator(numbers,start,start+(length/2));
    leftTask.fork();
    ForkJoinSumCalculator rightTask =
        new ForkJoinSumCalculator(numbers, start + length/2, end);
    Long rightResult = rightTask.compute();
    Long leftResult = leftTask.join();
    return leftResult + rightResult;
}
private Long sumCalculateSequentially() {
    long sum = 0;
    for (int i = start; i < end; i++) {
    sum += numbers[i];
    }
    return sum;
}
```

Three interesting calls

```
protected Long compute() {
    …
    ForkJoinSumCalculator leftTask =
    new ForkJoinSumCalculator(arguments..);
    //1 Why a fork?
    leftTask.fork();
    ForkJoinSumCalculator rightTask =
    new ForkJoinSumCalculator(arguments);
    //2 Why a compute?
    Long rightResult = rightTask.compute();
    //3 Why a join?
    Long leftResult = leftTask.join();
    return leftResult + rightResult;
}
```

To get an idea, debug our program

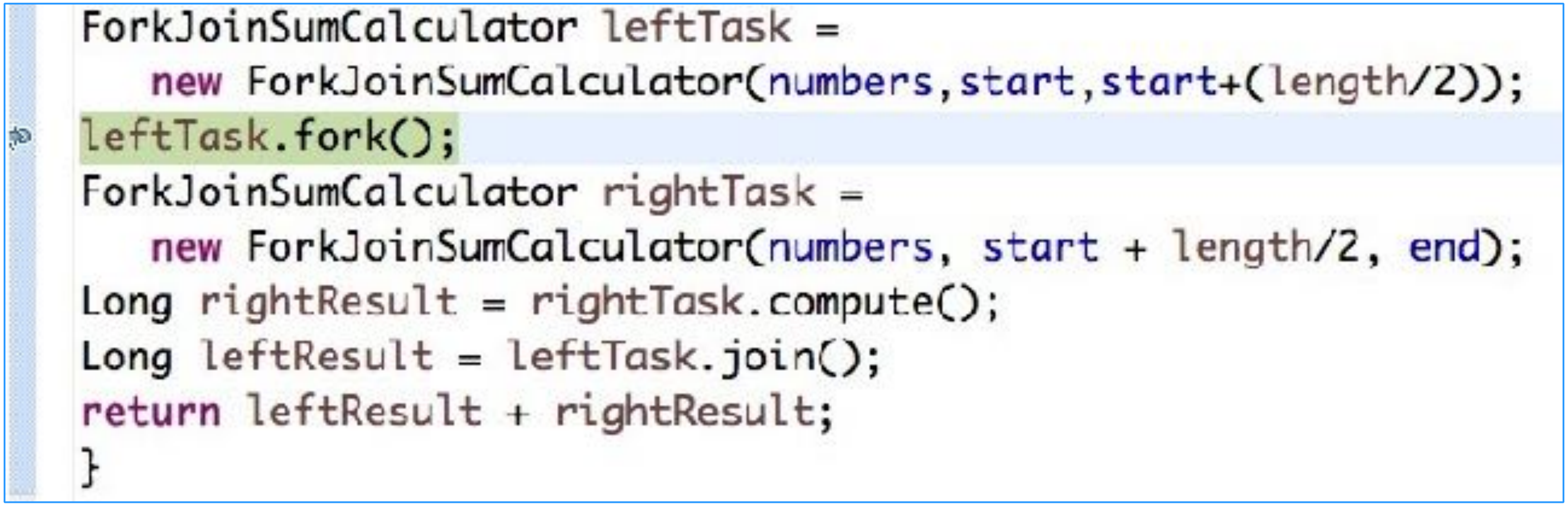


```
@Override
protected Long compute() {
    int length = end - start;
    if (length < THRESHOLD) {
        return sumCalculateSequentially();
    }
    ForkJoinSumCalculator leftTask = new ForkJoinSumCalculator(numbers,
            start, start + (length / 2));
    leftTask.fork();
    ForkJoinSumCalculator rightTask = new ForkJoinSumCalculator(numbers,
            start + length / 2, end);
    Long rightResult = rightTask.compute();
    Long leftResult = leftTask.join();
    return leftResult + rightResult;
}
```

- On the fork call is an active breakpoint

The first “run” of program

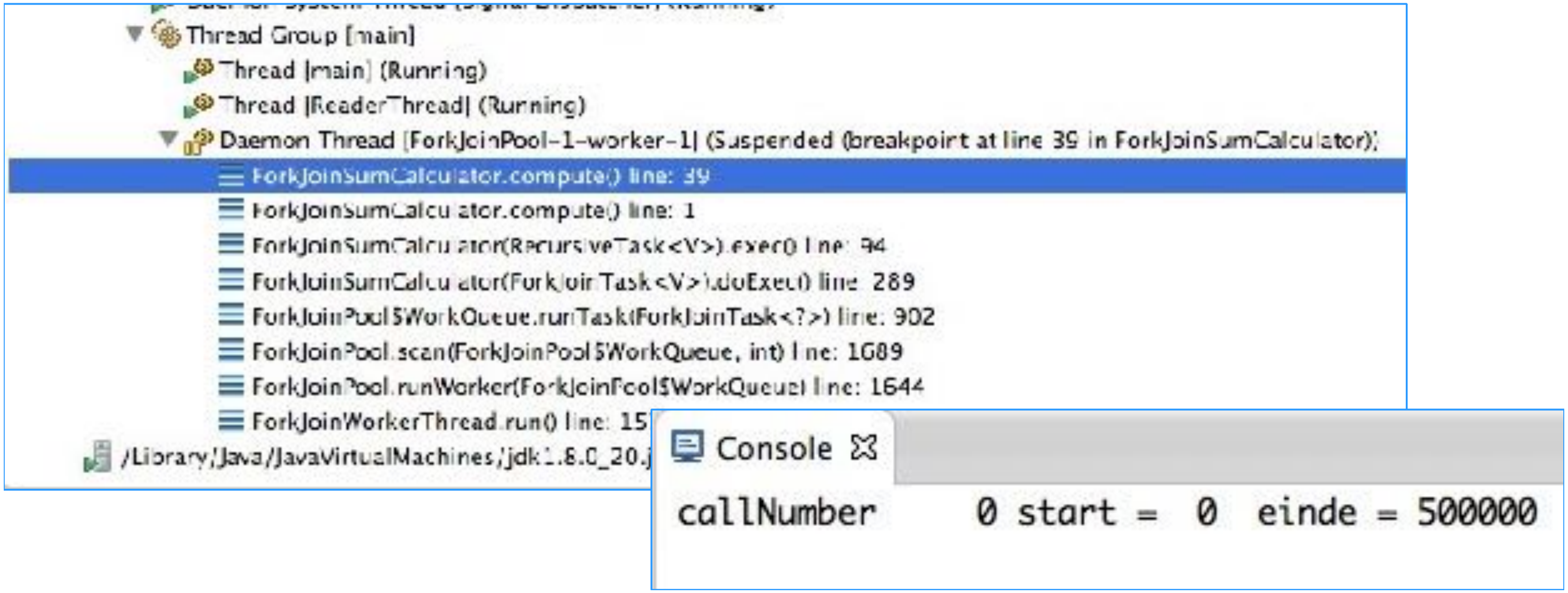
This figure shows the relevant part of the compute method



```
ForkJoinSumCalculator leftTask =
   new ForkJoinSumCalculator(numbers,start,start+(length/2));
leftTask.fork();
ForkJoinSumCalculator rightTask =
   new ForkJoinSumCalculator(numbers, start + length/2, end);
Long rightResult = rightTask.compute();
Long leftResult = leftTask.join();
return leftResult + rightResult;
}
```

- The next slide shows the relevant windows of the eclipse debug perspective
- Note the length of array is 1_000_000L

Result first "run" of program

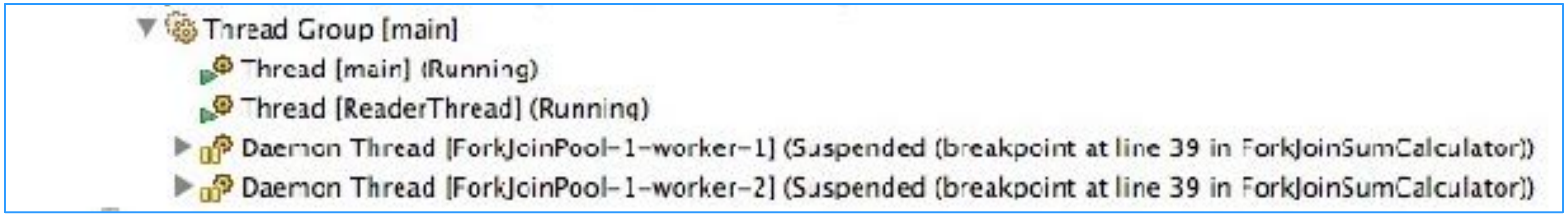


```
Daemon System Thread [Signal Dispatcher] (Running)
▼ Thread Group [main]
    Thread [main] (Running)
    Thread [ReaderThread] (Running)
  ▼ Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
      ForkJoinSumCalculator.compute() line: 39
      ForkJoinSumCalculator.compute() line: 1
      ForkJoinSumCalculator(RecursiveTask<V>).exec() line: 94
      ForkJoinSumCalculator(ForkJoinTask<V>).doExec() line: 289
      ForkJoinPool$WorkQueue.runTask(ForkJoinTask<?>) line: 902
      ForkJoinPool.scan(ForkJoinPool$WorkQueue, int) line: 1689
      ForkJoinPool.runWorker(ForkJoinPool$WorkQueue) line: 1644
      ForkJoinWorkerThread.run() line: 15
 /Library/Java/JavaVirtualMachines/jdk1.8.0_20.j
```

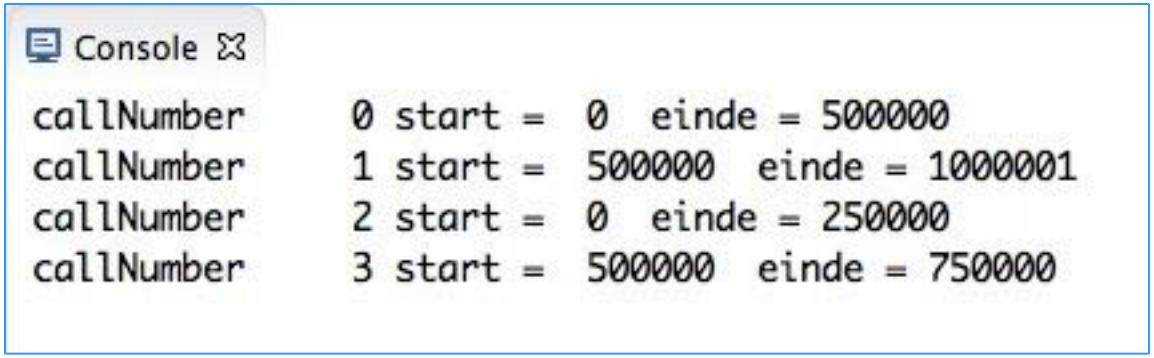
```
callNumber      0 start =  0  einde = 500000
```

- The first task is added to the forkjoinpool as a result of: new ForkJoinPool().invoke(fjsum);
- The compute method will be called
- The first thing done is calling the constructor

Resuming the first worker thread



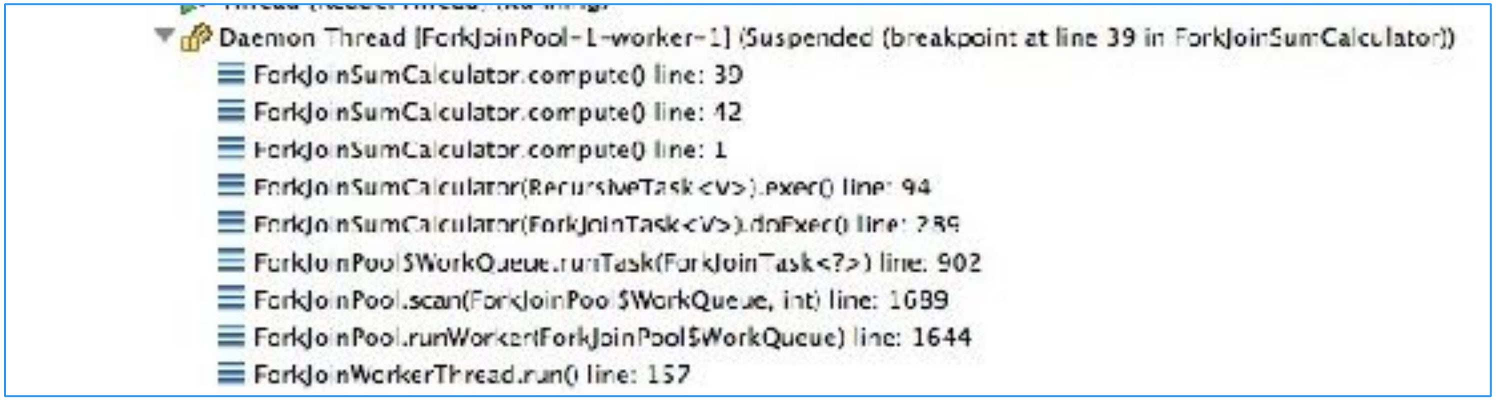
```
Thread Group [main]
    Thread [main] (Running)
    Thread [ReaderThread] (Running)
  Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  Daemon Thread [ForkJoinPool-1-worker-2] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
```



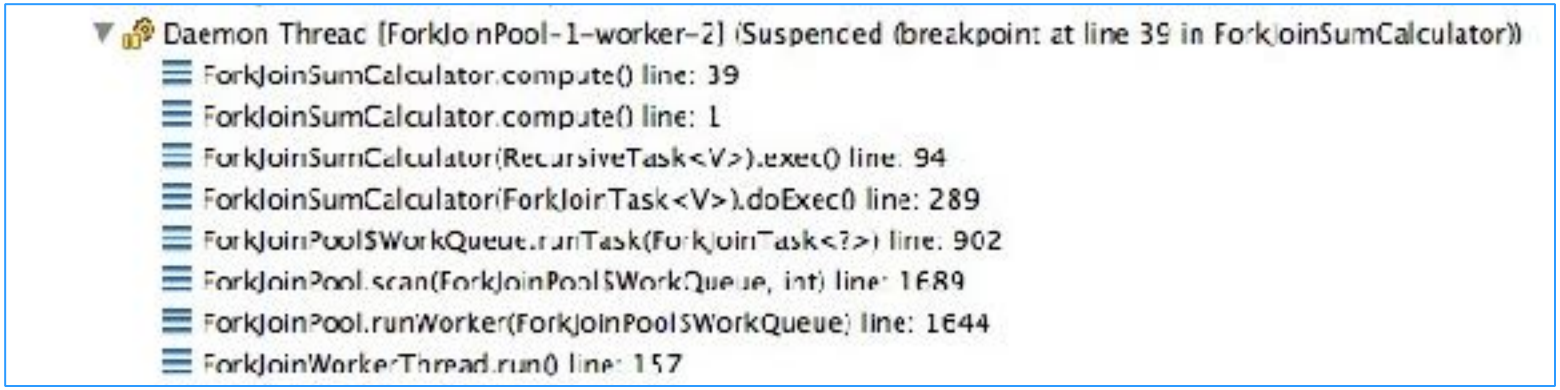
```
callNumber     0 start =  0  einde = 500000
callNumber     1 start =  500000  einde = 1000001
callNumber     2 start =  0  einde = 250000
callNumber     3 start =  500000  einde = 750000
```

- The first daemon thread is resumed
  - It will define a new task (rightTask) and call compute, a synchronous call, on rightTask
  - compute will call fork and a second thread becomes active

Threads 1,2 suspended on fork call



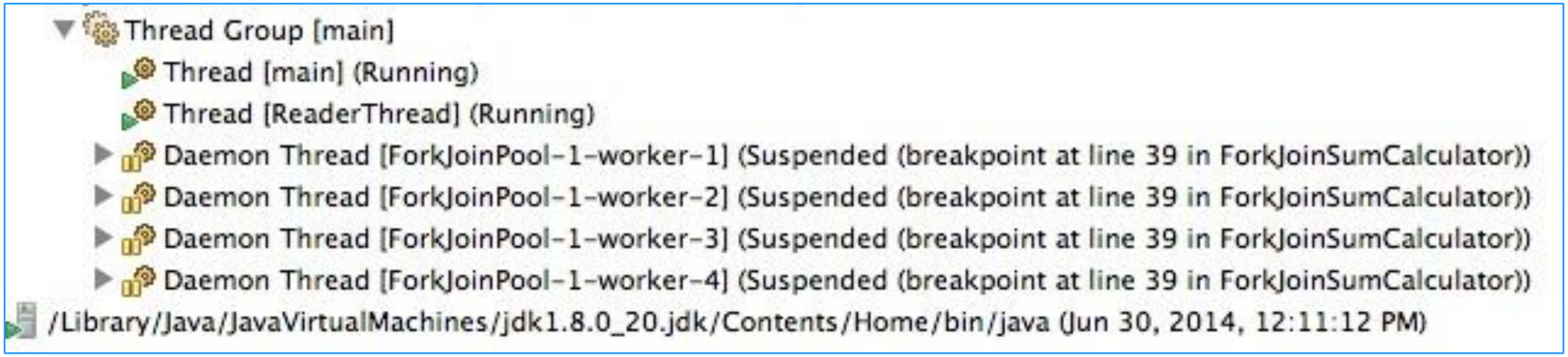
```
Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
    ForkJoinSumCalculator.compute() line: 39
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 1
    ForkJoinSumCalculator(RecursiveTask<V>).exec() line: 94
    ForkJoinSumCalculator(ForkJoinTask<V>).doExec() line: 289
    ForkJoinPool$WorkQueue.runTask(ForkJoinTask<?>) line: 902
    ForkJoinPool.scan(ForkJoinPool$WorkQueue, int) line: 1689
    ForkJoinPool.runWorker(ForkJoinPool$WorkQueue) line: 1644
    ForkJoinWorkerThread.run() line: 157
```



```
Daemon Thread [ForkJoinPool-1-worker-2] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
    ForkJoinSumCalculator.compute() line: 39
    ForkJoinSumCalculator.compute() line: 1
    ForkJoinSumCalculator(RecursiveTask<V>).exec() line: 94
    ForkJoinSumCalculator(ForkJoinTask<V>).doExec() line: 289
    ForkJoinPool$WorkQueue.runTask(ForkJoinTask<?>) line: 902
    ForkJoinPool.scan(ForkJoinPool$WorkQueue, int) line: 1689
    ForkJoinPool.runWorker(ForkJoinPool$WorkQueue) line: 1644
    ForkJoinWorkerThread.run() line: 157
```

- Note that the first thread has already done a recursive call on compute while for the second, new thread, it is the first call

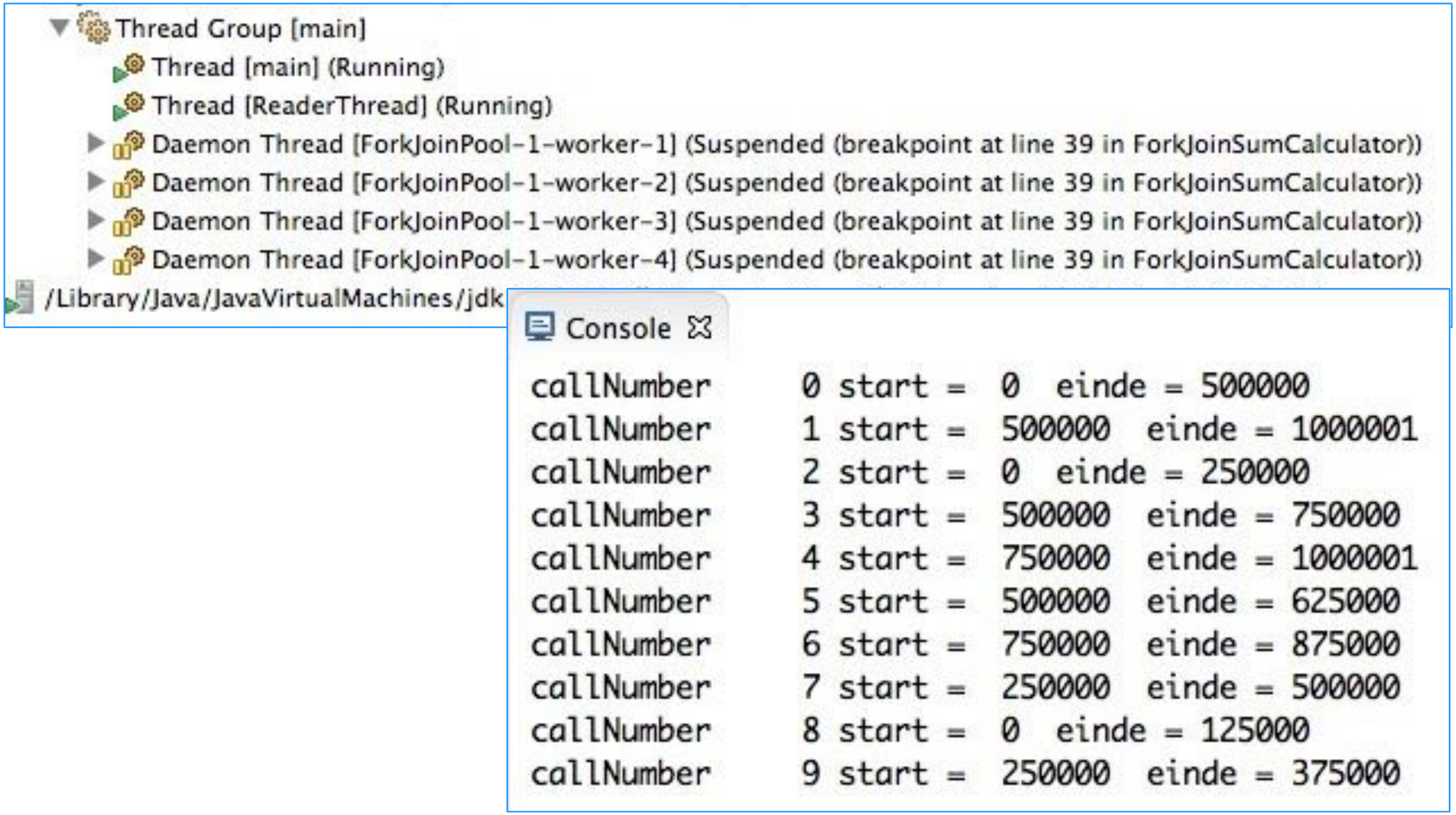
Playing for the scheduler



```
▼ Thread Group [main]
      Thread [main] (Running)
      Thread [ReaderThread] (Running)
   ▶ Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
   ▶ Daemon Thread [ForkJoinPool-1-worker-2] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
   ▶ Daemon Thread [ForkJoinPool-1-worker-3] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
   ▶ Daemon Thread [ForkJoinPool-1-worker-4] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
/Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home/bin/java (Jun 30, 2014, 12:11:12 PM)
```

- In reality threads are scheduled by a scheduler
- In this case the processor has 8 cores and the forkjoin threadpool has 8 workerthreads
- The output of the debugger is generated by sequentially calling resume on thread 1 and 2

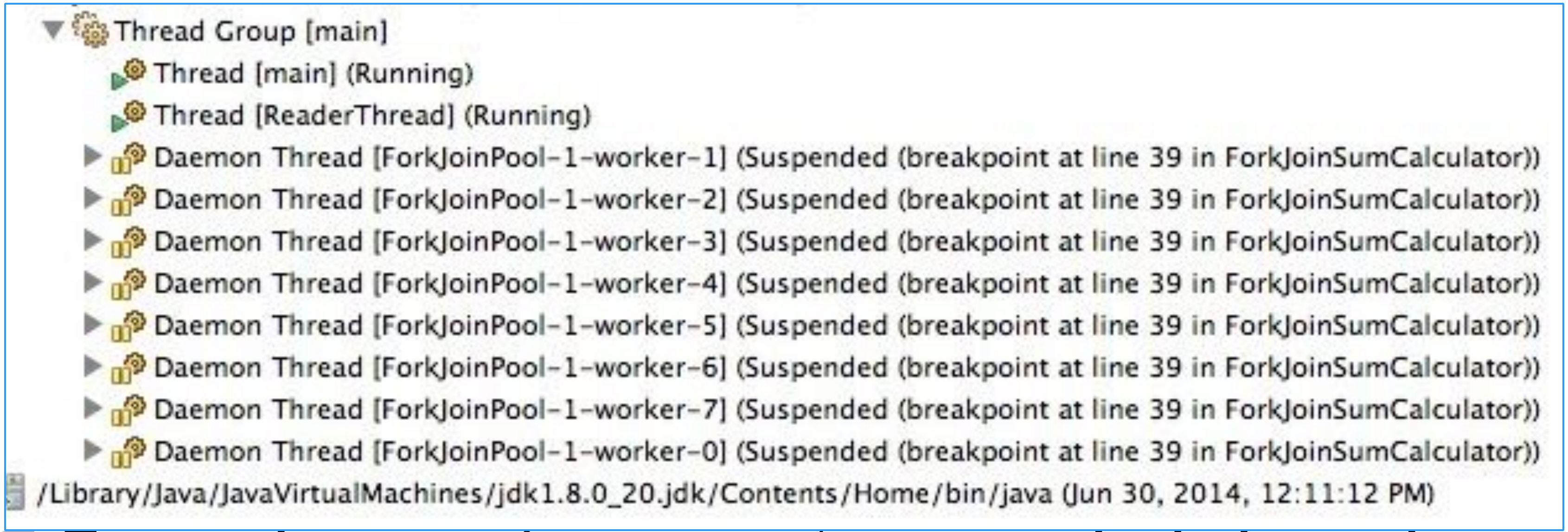
Output after 4 threads are started



```
▼ Thread Group [main]
      Thread [main] (Running)
      Thread [ReaderThread] (Running)
  ▶ Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-2] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-3] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-4] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
/Library/Java/JavaVirtualMachines/jdk
```

```
callNumber      0 start =  0  einde = 500000
callNumber      1 start =  500000  einde = 1000001
callNumber      2 start =  0  einde = 250000
callNumber      3 start =  500000  einde = 750000
callNumber      4 start =  750000  einde = 1000001
callNumber      5 start =  500000  einde = 625000
callNumber      6 start =  750000  einde = 875000
callNumber      7 start =  250000  einde = 500000
callNumber      8 start =  0  einde = 125000
callNumber      9 start =  250000  einde = 375000
```

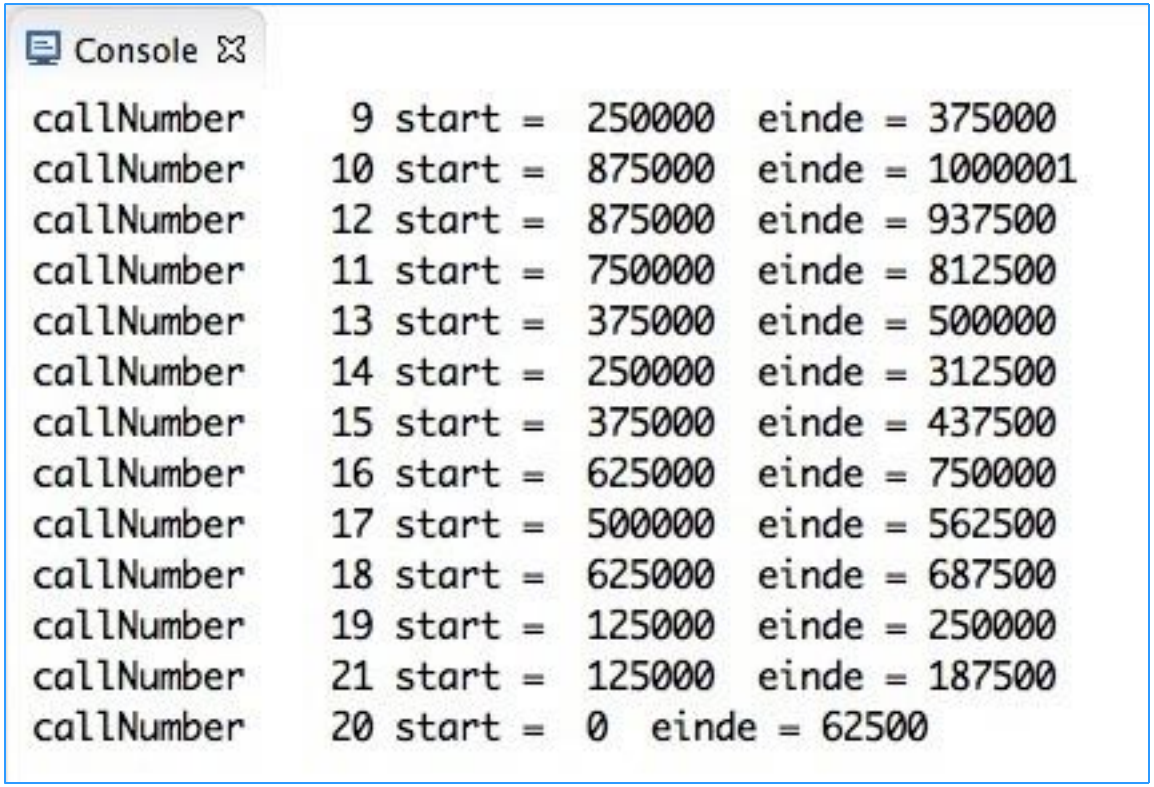
Putting all our cores to work



```
▼ Thread Group [main]
    Thread [main] (Running)
    Thread [ReaderThread] (Running)
  ▶ Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-2] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-3] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-4] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-5] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-6] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-7] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-0] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
/Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home/bin/java (Jun 30, 2014, 12:11:12 PM)
```

- From the previous run 4 suspended threads are available
- Mimicking a multi core environment all threads are (sequentially) resumed
- This results in 8 threads

Output after 8 threads are started

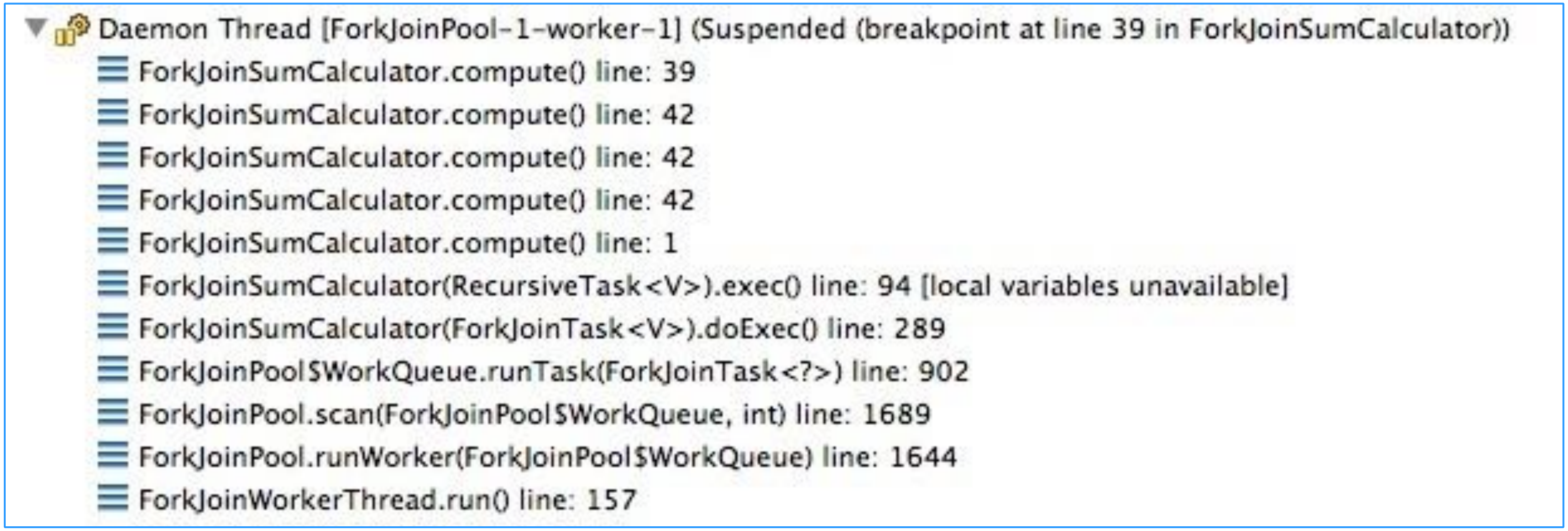


Console

```
callNumber     9 start =  250000  einde = 375000
callNumber    10 start =  875000  einde = 1000001
callNumber    12 start =  875000  einde = 937500
callNumber    11 start =  750000  einde = 812500
callNumber    13 start =  375000  einde = 500000
callNumber    14 start =  250000  einde = 312500
callNumber    15 start =  375000  einde = 437500
callNumber    16 start =  625000  einde = 750000
callNumber    17 start =  500000  einde = 562500
callNumber    18 start =  625000  einde = 687500
callNumber    19 start =  125000  einde = 250000
callNumber    21 start =  125000  einde = 187500
callNumber    20 start =  0  einde = 62500
```

- Note the threshold to start recursive task is 10_000L so new task will be created

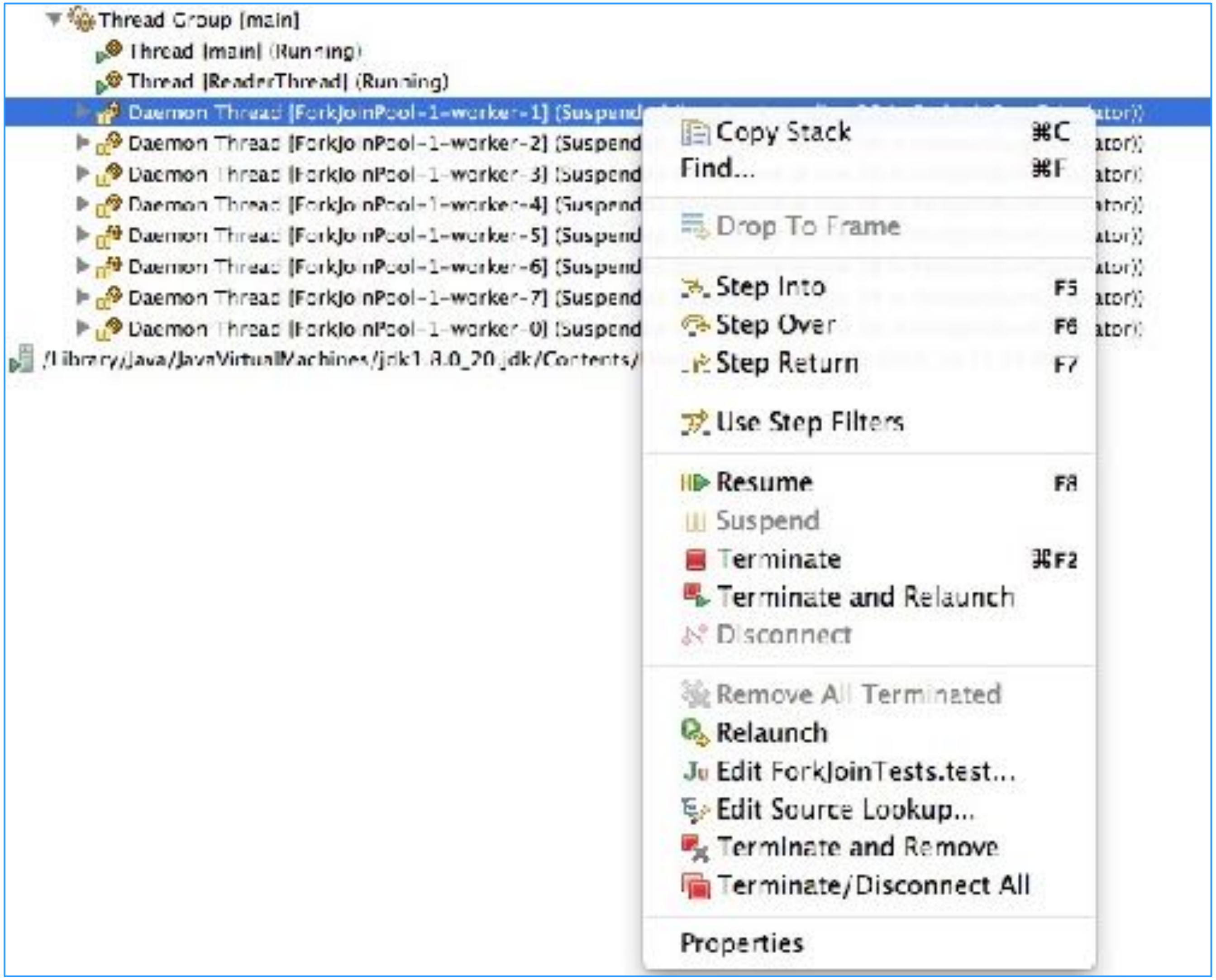
Resuming our threads



```
Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
    ForkJoinSumCalculator.compute() line: 39
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 1
    ForkJoinSumCalculator(RecursiveTask<V>).exec() line: 94 [local variables unavailable]
    ForkJoinSumCalculator(ForkJoinTask<V>).doExec() line: 289
    ForkJoinPool$WorkQueue.runTask(ForkJoinTask<?>) line: 902
    ForkJoinPool.scan(ForkJoinPool$WorkQueue, int) line: 1689
    ForkJoinPool.runWorker(ForkJoinPool$WorkQueue) line: 1644
    ForkJoinWorkerThread.run() line: 157
```

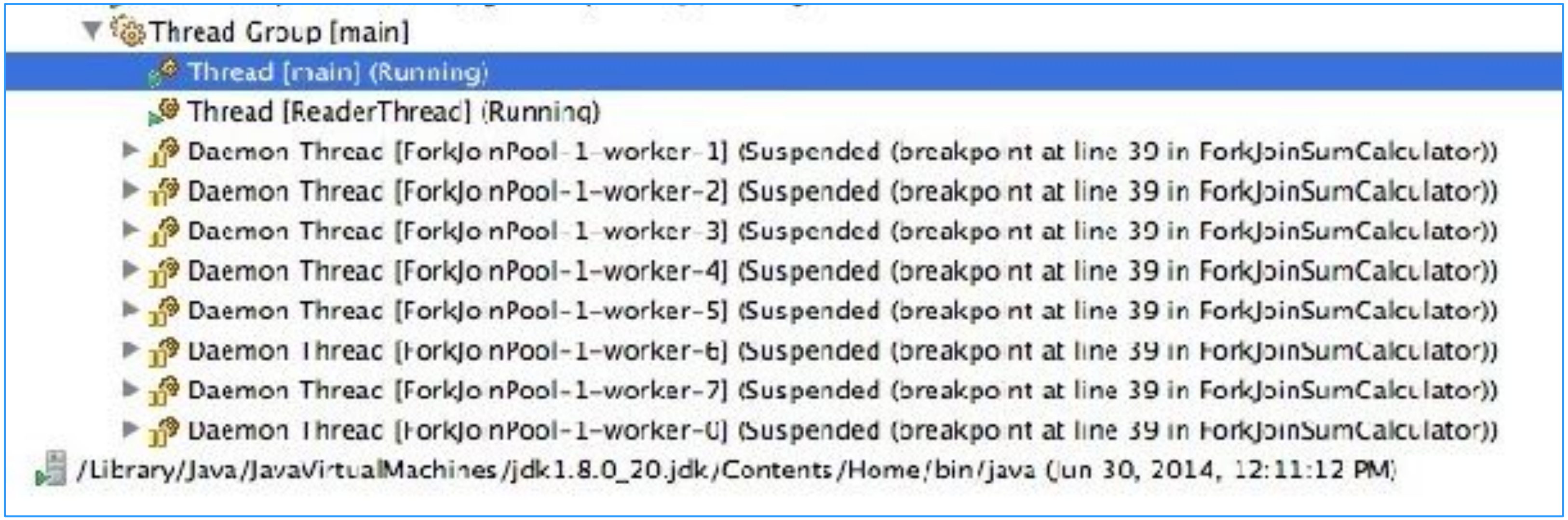
- After 8 threads no new threads are started
- New recursive task are queued to the queue of the pool
- Because the threshold is not yet reached new recursive calls to compute are made

Calling resume on each thread



Thread Group [main]
Thread [main] (Running)
Thread [ReaderThread] (Running)
Daemon Thread [ForkJoinPool-1-worker-1] (Suspend
Daemon Thread [ForkJoinPool-1-worker-2] (Suspend
Daemon Thread [ForkJoinPool-1-worker-3] (Suspend
Daemon Thread [ForkJoinPool-1-worker-4] (Suspend
Daemon Thread [ForkJoinPool-1-worker-5] (Suspend
Daemon Thread [ForkJoinPool-1-worker-6] (Suspend
Daemon Thread [ForkJoinPool-1-worker-7] (Suspend
Daemon Thread [ForkJoinPool-1-worker-0] (Suspend
/Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/
Copy Stack ⌘C
Find... ⌘F
Drop To Frame
Step Into F5
Step Over F6
Step Return F7
Use Step Filters
Resume F8
Suspend
Terminate ⌘F2
Terminate and Relaunch
Disconnect
Remove All Terminated
Relaunch
Edit ForkJoinTests.test...
Edit Source Lookup...
Terminate and Remove
Terminate/Disconnect All
Properties

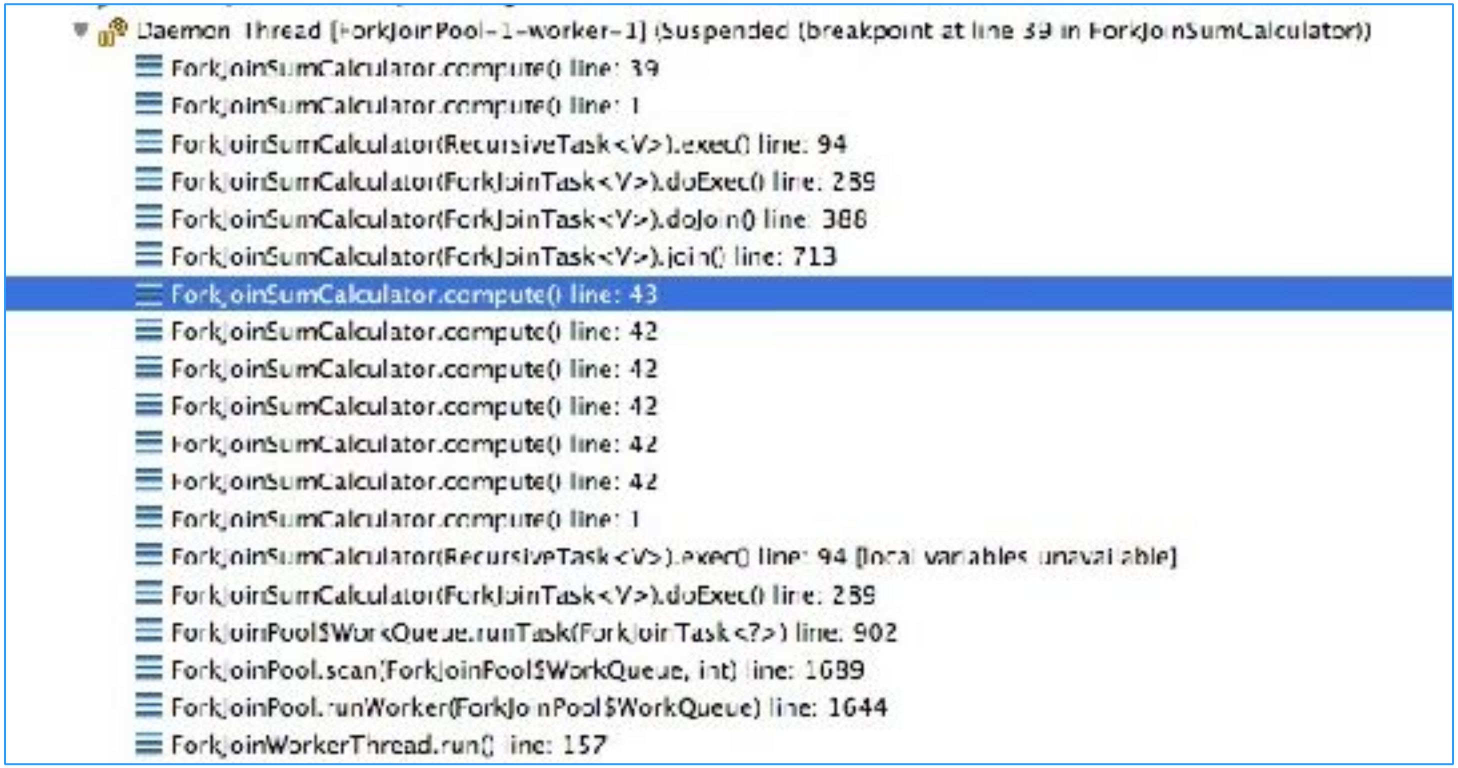
Resuming threads



```
▼ Thread Group [main]
    Thread [main] (Running)
    Thread [ReaderThread] (Running)
  ▶ Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-2] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-3] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-4] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-5] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-6] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-7] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
  ▶ Daemon Thread [ForkJoinPool-1-worker-0] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
/Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home/bin/java (Jun 30, 2014, 12:11:12 PM)
```

- Resuming threads did not result in starting new Daemon Threads after the pool had 8 threads

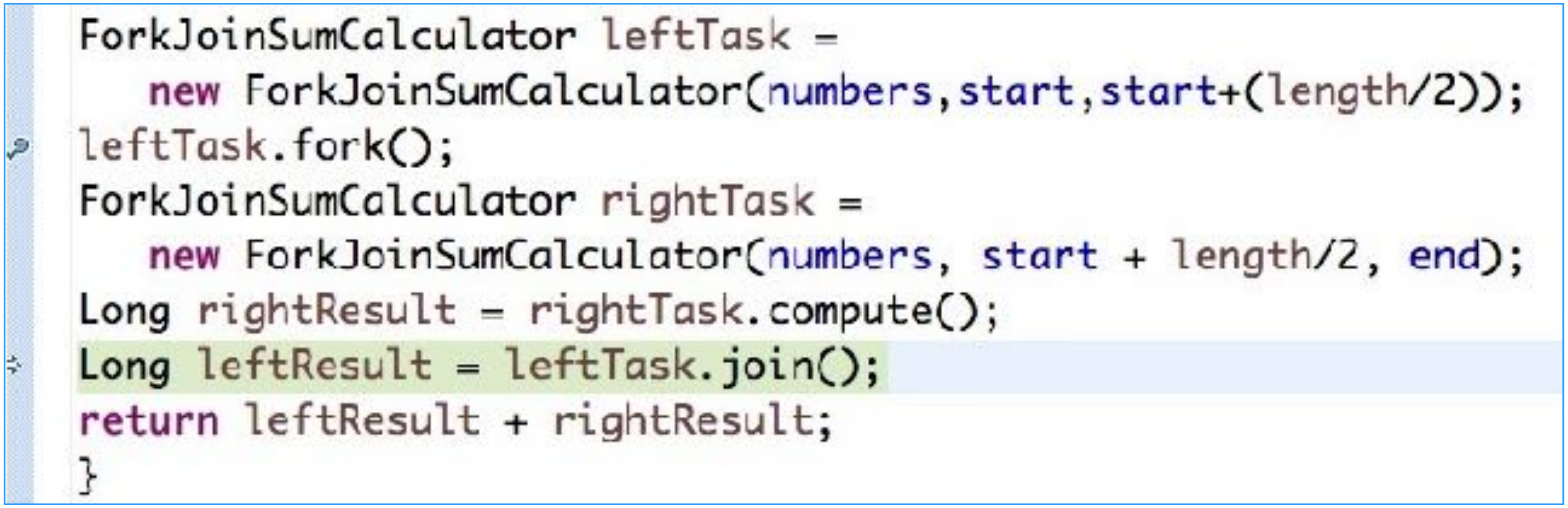
Hitting the bottom



```
Daemon Thread [ForkJoinPool-1-worker-1] (Suspended (breakpoint at line 39 in ForkJoinSumCalculator))
    ForkJoinSumCalculator.compute() line: 39
    ForkJoinSumCalculator.compute() line: 1
    ForkJoinSumCalculator(RecursiveTask<V>).exec() line: 94
    ForkJoinSumCalculator(ForkJoinTask<V>).doExec() line: 289
    ForkJoinSumCalculator(ForkJoinTask<V>).doJoin() line: 388
    ForkJoinSumCalculator(ForkJoinTask<V>).join() line: 713
    ForkJoinSumCalculator.compute() line: 43
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 42
    ForkJoinSumCalculator.compute() line: 1
    ForkJoinSumCalculator(RecursiveTask<V>).exec() line: 94 [local variables unavailable]
    ForkJoinSumCalculator(ForkJoinTask<V>).doExec() line: 289
    ForkJoinPool$WorkQueue.runTask(ForkJoinTask<?>) line: 902
    ForkJoinPool.scan(ForkJoinPool$WorkQueue, int) line: 1689
    ForkJoinPool.runWorker(ForkJoinPool$WorkQueue) line: 1644
    ForkJoinWorkerThread.run() line: 157
```

- Finally a compute method reaches the threshold starting a sequential sum
- The compute returns and a join is called

The active call on the stack



```
ForkJoinSumCalculator leftTask =
    new ForkJoinSumCalculator(numbers,start,start+(length/2));
leftTask.fork();
ForkJoinSumCalculator rightTask =
    new ForkJoinSumCalculator(numbers, start + length/2, end);
Long rightResult = rightTask.compute();
Long leftResult = leftTask.join();
return leftResult + rightResult;
}
```

Reminder of the code

```
protected Long compute() {
  int length=end-start;
  if(length<THRESHOLD) {
     return sumCalculateSequentially();
  }
  ForkJoinSumCalculator leftTask =
       new ForkJoinSumCalculator(numbers,start,start+(length/2));
  leftTask.fork();
  ForkJoinSumCalculator rightTask =
       new ForkJoinSumCalculator(numbers, start + length/2, end);
  Long rightResult = rightTask.compute();
  Long leftResult = leftTask.join();
  return leftResult + rightResult;
}
private Long sumCalculateSequentially() {
  long sum = 0;
  for (int i = start; i < end; i++) {
         sum += numbers[i];
  }
  return sum;
}
```

Done some experiments

- Played with different scenario’s

1. leftTask.fork, rightTask.compute,leftTask.join
2. leftTask.fork, rightTask.fork, leftTask.join, rightTask.join
3. No big difference in results were observed
4. Note however that we have a 100 task evenly divided over 8 cores
5. Also using a tenfold bigger THRESHOLD did not result in a noticeable difference
6. To be continued

The Spliterator

- new interface added to Java 8
- stands for “splittable” iterator
- are designed to traverse the elements of a source in parallel
- Java 8 provides a default Spliterator implementation for the Collection framework
- Normally no need to write your own
- Understanding spliterators is important to understand parallel streams

The Spliterator interface

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

- tryAdvance consumes the elements one by one, returning true if there are other elements to be traversed; compare next() of Iterator
- trySplit() partitions off some of its elements to a second Spliterator allowing the two to be processed in parallel

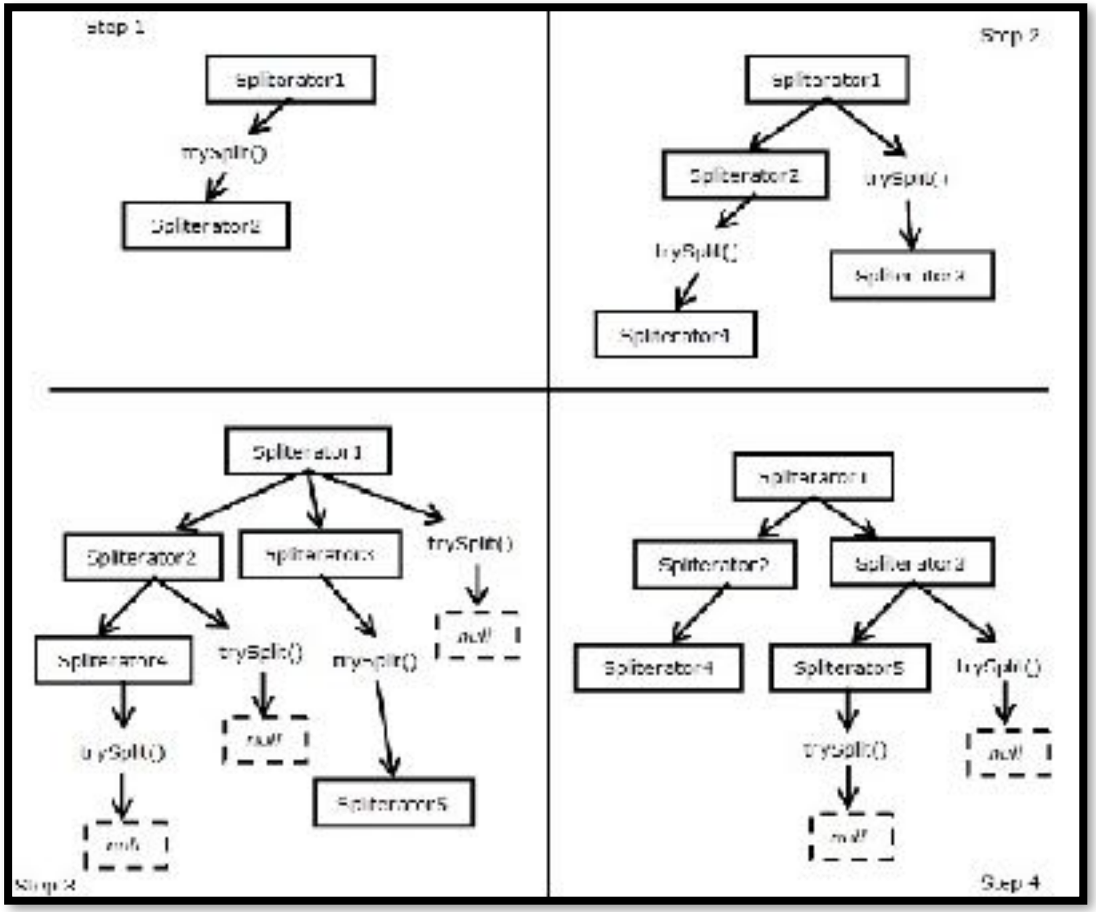
The Spliterator interface continued

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

- estimateSize() provides an estimation of the number of the elements remaining to be traversed; even an inaccurate value can be useful to split the structure
- characteristics() returns an int encoding of the set of characteristics of the Spliterator

The splitting process

- The split algorithm recursively splits a Stream into multiple parts



The Spliterator characteristics

ORDERED	Elements have a defined order (e.g. a List), so the Spliterator enforces this order when traversing and partitioning them
DISTINCT	For each pair of traversed elements x and y, x.equals(y) returns false
SORTED	The traversed elements follow a predefined sort order
SIZED	This Spliterator has been created from a source with a known size (for example, a Set), so the value returned by estimatedSize() is precise
NONNULL	It’s guaranteed that the traversed elements won’t be null
IMMUTABLE	The source of this Spliterator can’t be modified. This implies that no elements can be added, removed, or modified during their traversal
CONCURRENT	The source of this Spliterator may be safely concurrently modified by other threads without any synchronization
SUBSIZED	Both this Spliterator and all further Spliterators resulting from its split are SIZED

Implementing an Spliterator

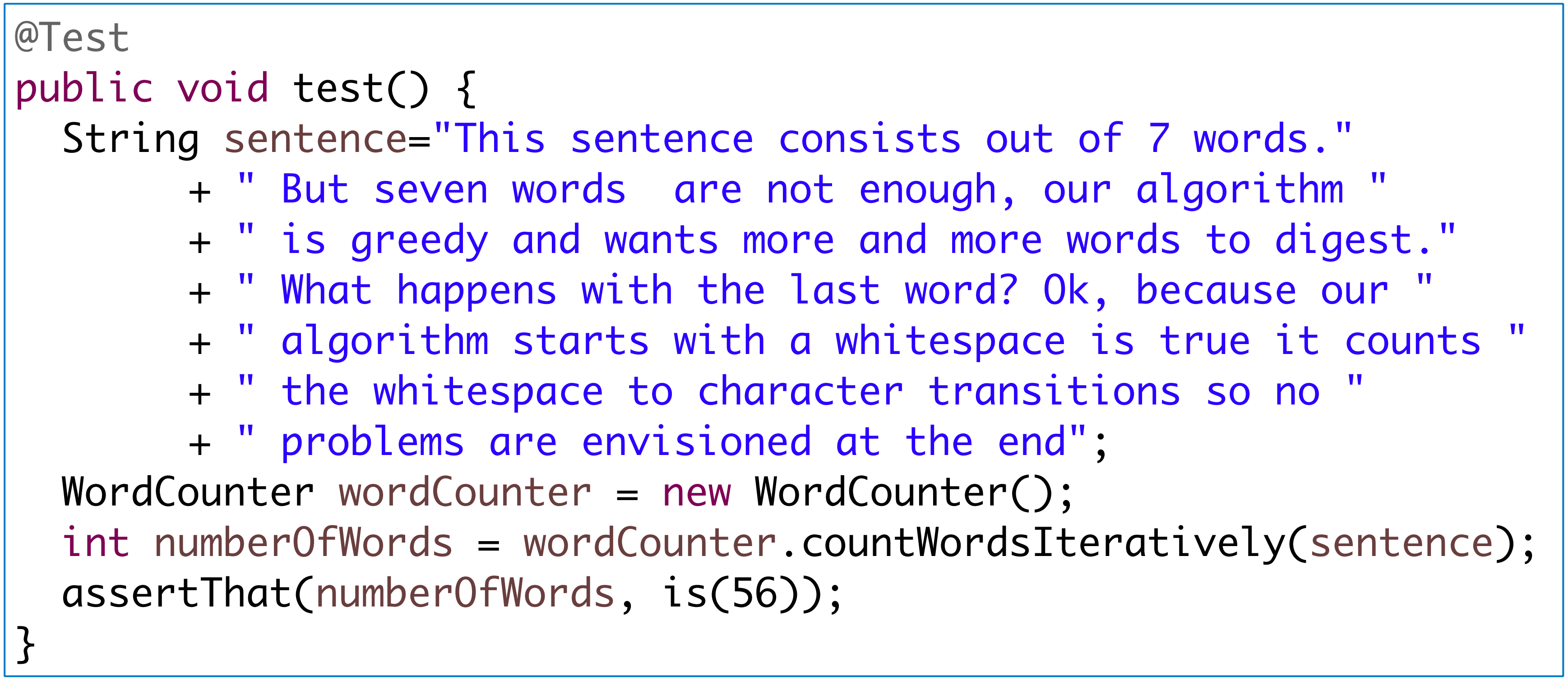
- Counting the number of words in a String
  – An iterative version

```java
public int countWordsIteratively(String s) {
    int counter = 0;
    boolean lastSpace = true;
    for (char c : s.toCharArray()) {
        if (Character.isWhitespace(c)) {
                lastSpace = true;
        } else {
            if (lastSpace) counter++;
                lastSpace = false;
        }
    }
    return counter;
}
```

Idea behind countWordsIteratively

- Check character for character
- When the character is a whitespace character then remember that
  - Whitespace is interpreted as a word separator
- When the character following a whitespace character is not a whitespace character mark this as the start of a new word
  - Increment the word counter
- Start the algorithm as if a whitespace character was detected

It works but...



```
@Test
public void test() {
  String sentence="This sentence consists out of 7 words."
        + " But seven words  are not enough, our algorithm "
        + " is greedy and wants more and more words to digest."
        + " What happens with the last word? Ok, because our "
        + " algorithm starts with a whitespace is true it counts "
        + " the whitespace to character transitions so no "
        + " problems are envisioned at the end";
  WordCounter wordCounter = new WordCounter();
  int numberOfWords = wordCounter.countWordsIteratively(sentence);
  assertThat(numberOfWords, is(56));
}
```

- Rewrite problem in a more functional style
- In such a way that we can use parallel options of a stream without explicitly using threads

A functional approach

```
//No primitive charStream exists
Stream<Character> stream =
    //produces a range of numbers
    //exactly matching with the char positions
    IntStream.range(0, sentence.length())
     //each number is matched with char on that position
             .mapToObj(sentence::charAt);
```

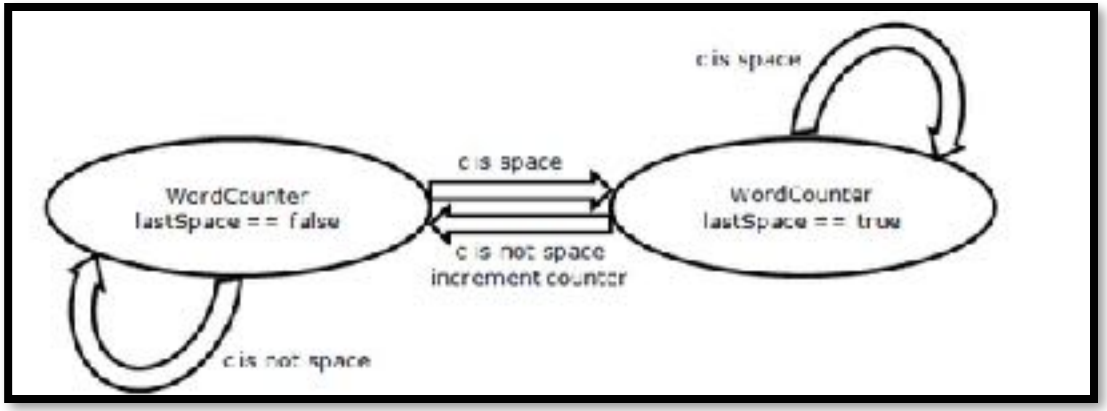
- Count words by reducing the Stream
- The reduction needs 2 variables to keep state:
  - an int counting the number of words found so far
  - a boolean to remember if the last encountered character was a space or not

Immutable helper class WordCounter

```
public class WordCounter {
    private final boolean lastSpace;
    private final int counter;

    public WordCounter(boolean lastSpace, int counter) {
        this.lastSpace = lastSpace;
        this.counter = counter;
    }
    public WordCounter accumulate(Character c) {
        if (Character.isWhitespace(c)) {
           return lastSpace ?this :new WordCounter(true, counter);
        } else {
           return lastSpace ?new WordCounter(false, counter+1):this;
        }
    }
    public WordCounter combine(WordCounter wordCounter) {
        return new WordCounter(wordCounter.lastSpace,
                                     counter + wordCounter.counter);
    }
    public int getCounter() {
        return counter;
    }
}
```

Note for every new word or character whitespace transition we get a new WordCounter instance

State diagram



```
public WordCounter accumulate(Character c) {
  if (Character.isWhitespace(c)) {
     return lastSpace ?this :new WordCounter(true, counter);
  } else {
   return lastSpace ?new WordCounter(false, counter+1):this;
  }
}
```

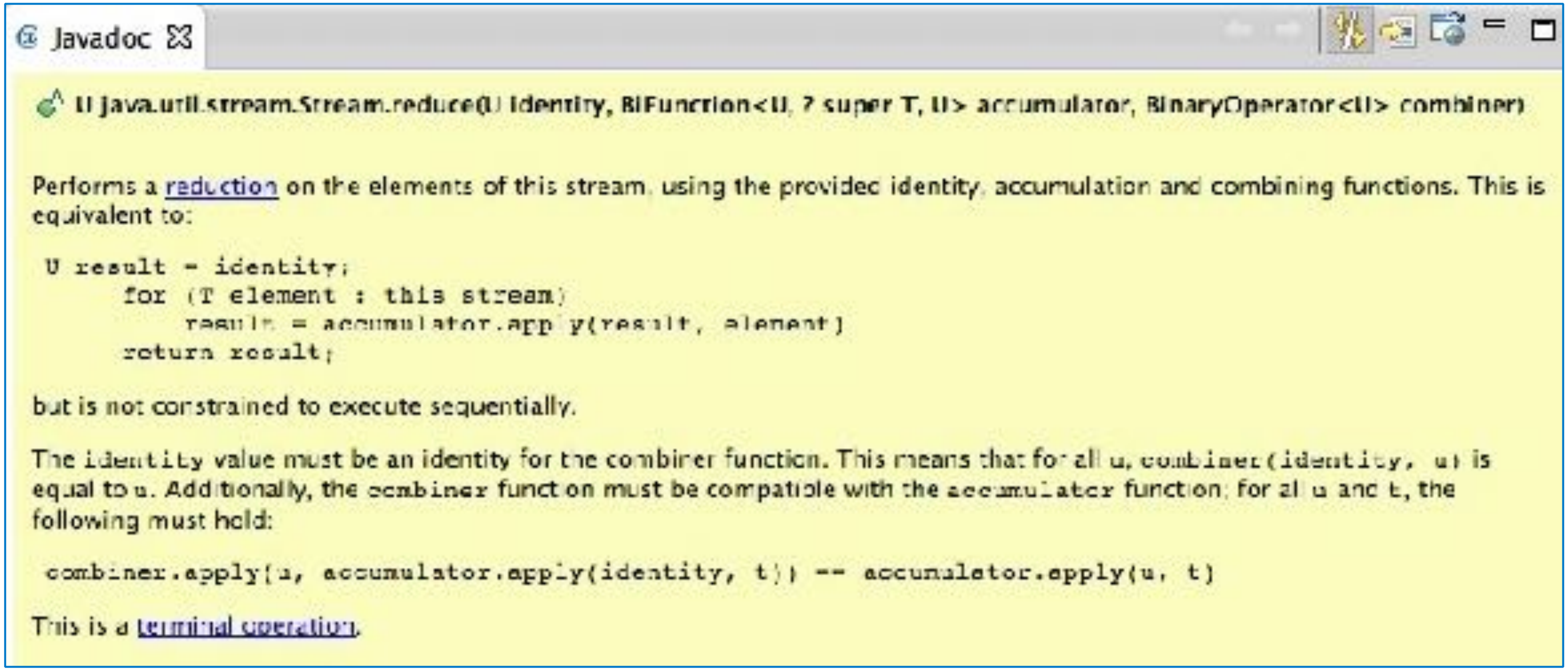
- the method accumulate() is called whenever a new Character of the Stream is traversed

Combining to subparts

```
public WordCounter combine(WordCounter wordCounter) {

       return new WordCounter(wordCounter.lastSpace,
                       counter + wordCounter.counter);
}
```

- combine(), is invoked to aggregate the partial results of two WordCounters operating on two different subparts of the Stream

From the API



U java.util.stream.Stream.reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. This is equivalent to:

```
 U result = identity;
     for (T element : this stream)
         result = accumulator.apply(result, element)
     return result;
```

but is not constrained to execute sequentially.

The `identity` value must be an identity for the combiner function. This means that for all u, `combiner(identity, u)` is equal to u. Additionally, the `combiner` function must be compatible with the `accumulator` function; for all u and t, the following must hold:

```
 combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)
```

This is a terminal operation.

- Our WordCounter (WC) has an
  - accumulator bifunction expecting WC and a Char
  - a combiner expecting 2 WC’s and returning a WC
  - the start value is new WC(true,0)

Putting to the test, it works but

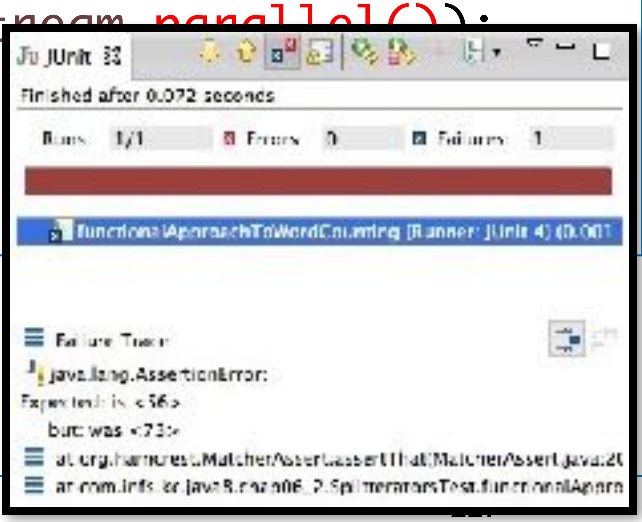
```java
@Test
public void functionalApproachToWordCounting() {
  String sentence = "This sentence consists out … 56 words.”;
  Stream<Character> characterStream =
  IntStream.range(0, sentence.length())
           .mapToObj(sentence::charAt);

  int wordCount = countWords(characterStream);
  assertThat(wordCount, is(56));
}
private int countWords(Stream<Character> stream) {
    WordCounter wordCounter =
       stream.reduce(new WordCounter(true,0),
                    WordCounter::accumulate,
                    WordCounter::combine);
 return wordCounter.getCounter();
}
```

A questionmark

- The purpose of the functional style solution is to use it in a parallel stream

```
@Test
public void functionalApproachToWordCounting() {
  String sentence = "This sentence consists out … 56 words.”;
  Stream<Character> characterStream =
  IntStream.range(0, sentence.length())
           .mapToObj(sentence::charAt);

  int wordCount = countWords(characterStream.parallel());
  assertThat(wordCount, is(56));
}
```

- What is going wrong?
- It is counting words twice

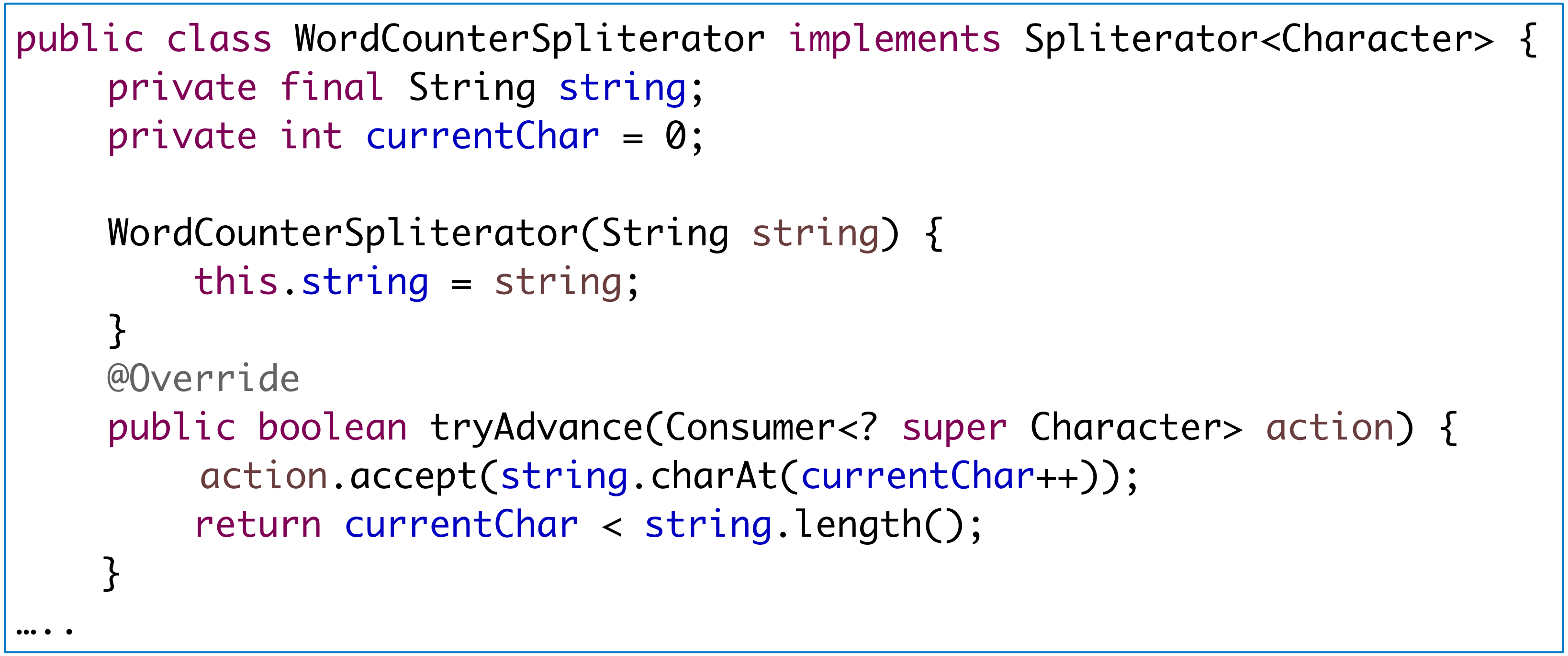


```
Finished after 0.072 seconds
Runs: 1/1   Errors: 0   Failures: 1
functionalApproachToWordCounting [Runner: JUnit 4] (0.001
Failure Trace
java.lang.AssertionError:
Expected: is <56>
     but: was <73>
at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20
at com.infosupport.java8.chap06_2.SpliteratorsTest.functionalAppro
```

What is wrong with the solution?

- Sometimes a word is split in two parts by the spliterator which means it is counted twice
- A spliterator is needed that only splits on whitespace
- In general: going from a sequential to a parallel stream can go astray when the result depends on where the stream is split

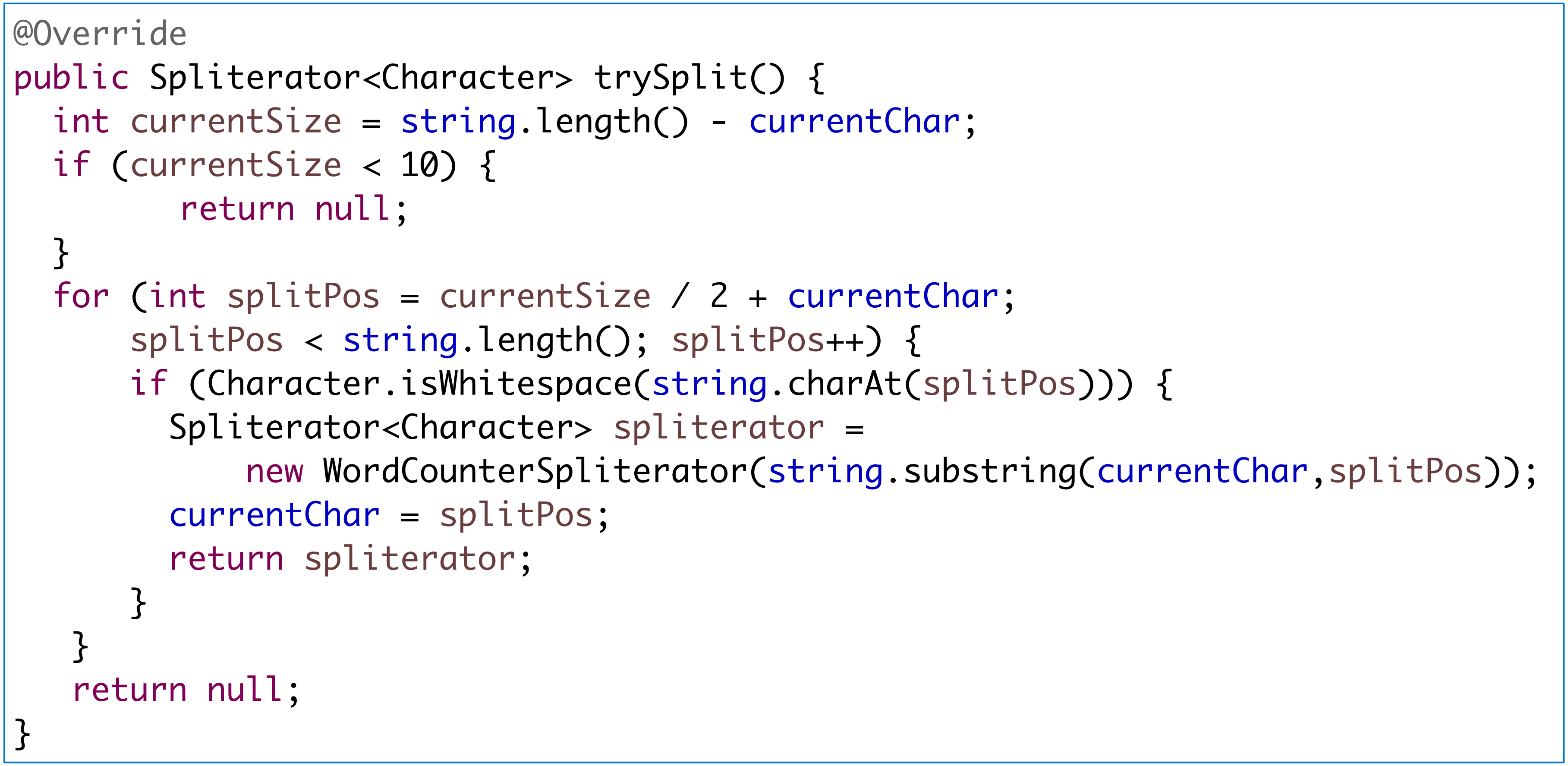
Custom Spliterator (Partial)



```
public class WordCounterSpliterator implements Spliterator<Character> {
    private final String string;
    private int currentChar = 0;

    WordCounterSpliterator(String string) {
        this.string = string;
    }
    @Override
    public boolean tryAdvance(Consumer<? super Character> action) {
        action.accept(string.charAt(currentChar++));
        return currentChar < string.length();
    }
…..
```

- tryAdvance() method feeds the Consumer with the Character
- the consumed Character is forwarded to a function, here the accumulator of reduce

The trySplit() method

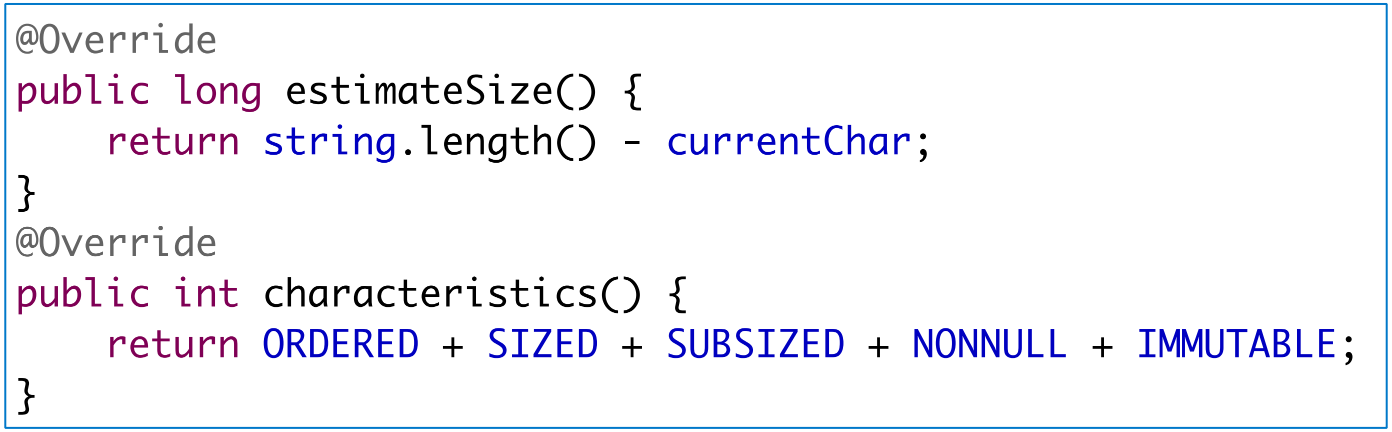


```java
@Override
public Spliterator<Character> trySplit() {
  int currentSize = string.length() - currentChar;
  if (currentSize < 10) {
         return null;
  }
  for (int splitPos = currentSize / 2 + currentChar;
      splitPos < string.length(); splitPos++) {
      if (Character.isWhitespace(string.charAt(splitPos))) {
        Spliterator<Character> spliterator =
            new WordCounterSpliterator(string.substring(currentChar,splitPos));
        currentChar = splitPos;
        return spliterator;
      }
   }
   return null;
}
```

The trySplit() method

- the trySplit() defines the logic used to split the data structure
- First: set a limit under which no further splits are needed
- If number of remaining Characters to be traversed is under this limit
  - Return null to signal that no further split is needed
- If number is above this limit find a whitespace character to split the stream and create a new Spliterator

The 2 remaining methods



```
@Override
public long estimateSize() {
    return string.length() - currentChar;
}
@Override
public int characteristics() {
    return ORDERED + SIZED + SUBSIZED + NONNULL + IMMUTABLE;
}
```

- The estimatedSize() of elements still to be traversed is the total length of the String parsed by this Spliterator minus the position currently iterated
- the characteristic() method signals to the framework that this Spliterator has distinct properties

Using the spliterator

```
@Test
public void functionalApproachToWordCountingGoingWrongWithParalllel() {
    String sentence = "This sentence consists out of … 56 words.”;

    Spliterator<Character> spliterator = new WordCounterSpliterator(sentence);
    Stream<Character> characterStream = StreamSupport.stream(spliterator, true);
    int wordCount = countWords(characterStream);

    assertThat(wordCount, is(56));
}
```

- Using StreamSupport.stream(spliterator, true)
  - – true, the second argument of the stream method signifies a parallel stream

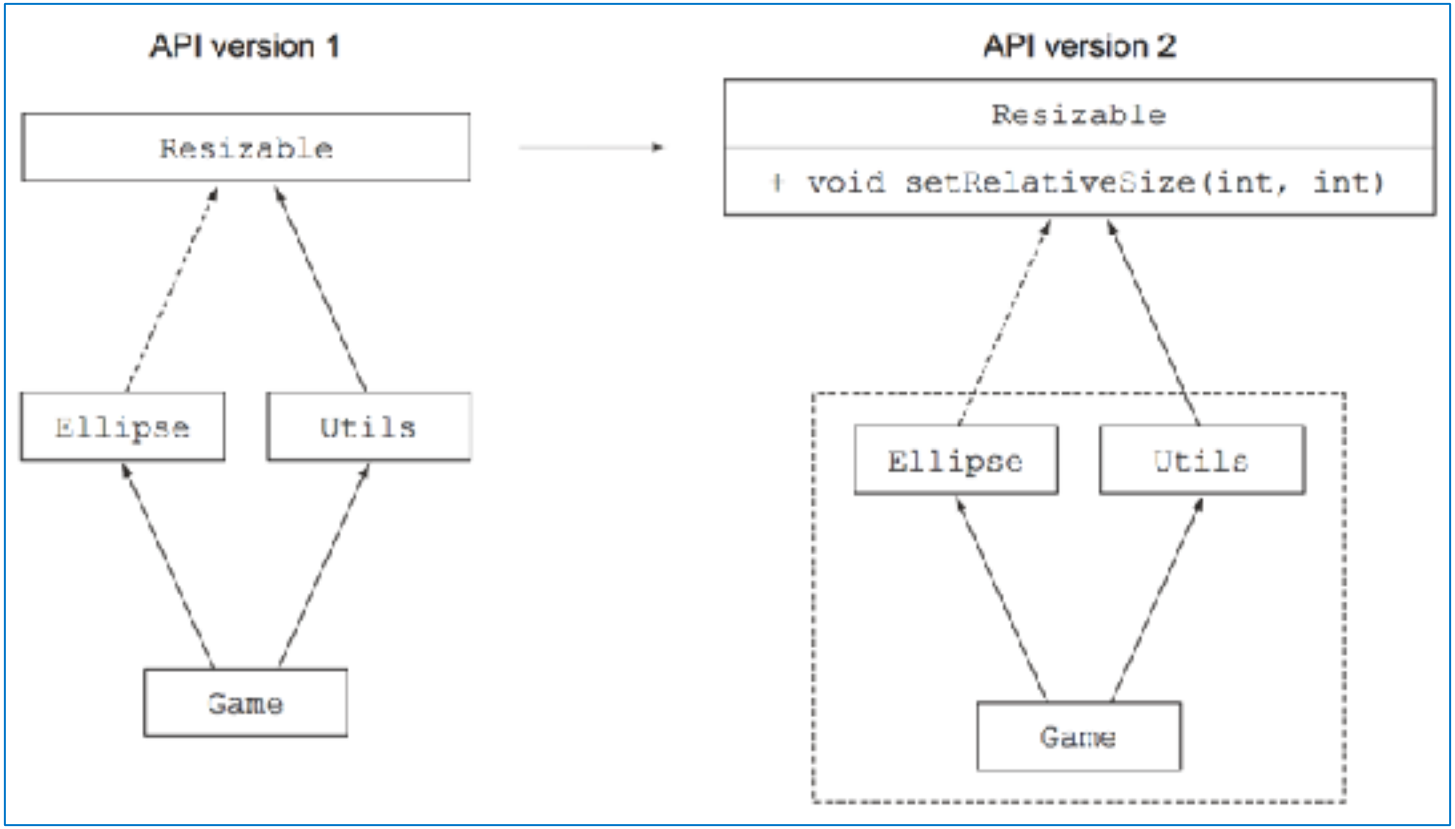
Default methods

Interfaces revisited in Java 8

```
public interface Action {

    static void doSomeStuff() {
        System.out.println("This can't be true");
    };

    default void doAction() {
        System.out.println("What is this default keyword?");
    }

}
```

- In Java 8 the above interface compiles nicely
- static methods are starting from Java 8 allowed on interfaces
- A default method is introduced to be backwards compatible

evolution of an API

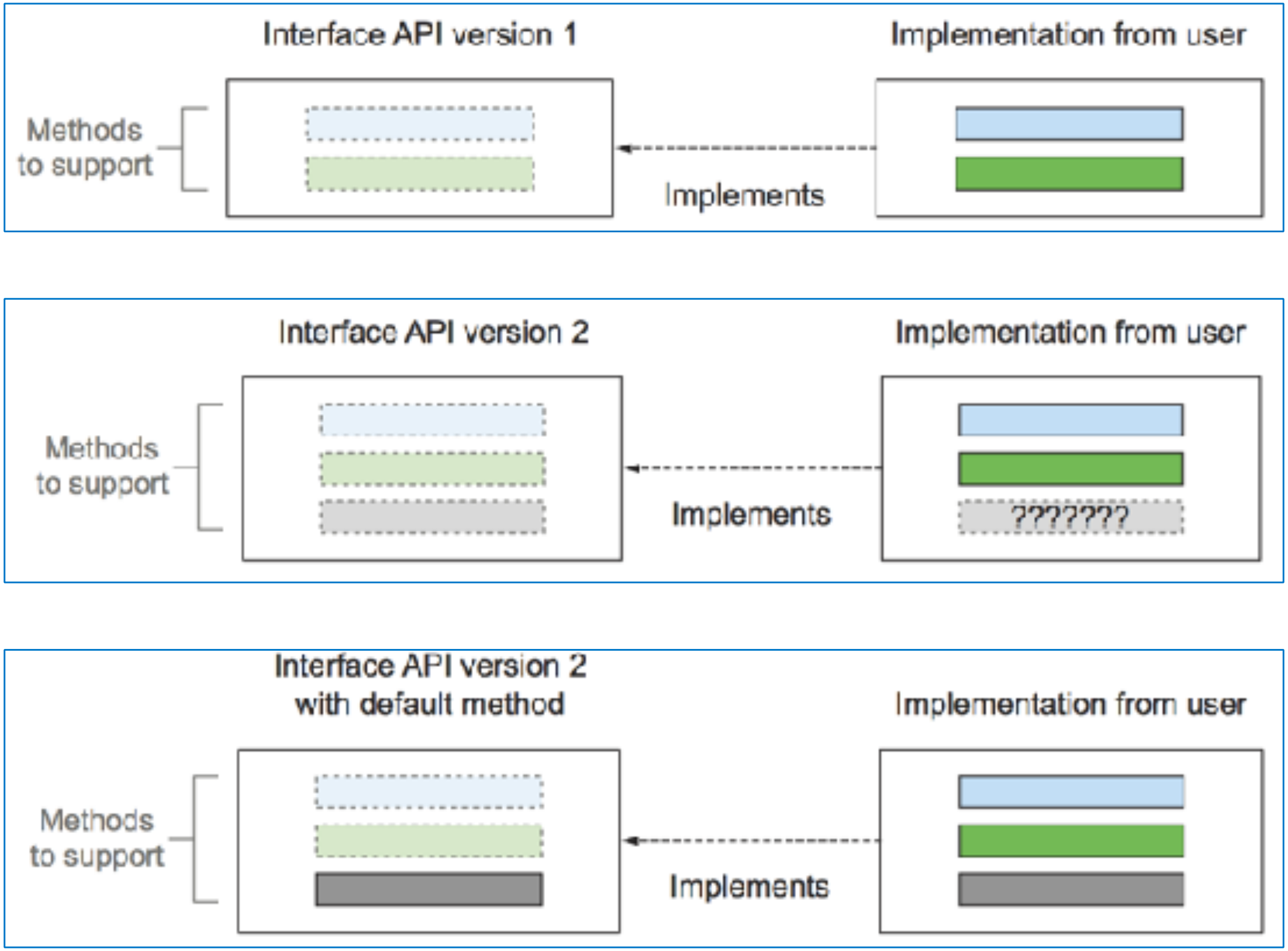


Ellipse and Utils depend on v1, recompiling against v2 results in exception

about default methods

- default methods allow you to provide a default implementation for methods in an interface
- existing classes implementing this interface will inherit the default implementations if they don’t provide one themselves (and they probably don’t)

in pictures


Interface API version 1
Implementation from user
Methods to support
Implements
Interface API version 2
Implementation from user
Methods to support
Implements
???????
Interface API version 2 with default method
Implementation from user
Methods to support
Implements

Different types of compatibilities

- binary
  - existing binaries running without errors continue to link (which involves verification, preparation, and resolution) without error after introducing a change
  - i.e. adding a method to an interface; because if it’s not called, existing methods of the interface can still run
- source
  - an existing program will still compile after introducing a change
- behavioural
  - running a program after a change with the same inputs results in the same behaviour

2 Usage patterns for default methods

- optional methods and
- multiple inheritance of behaviour

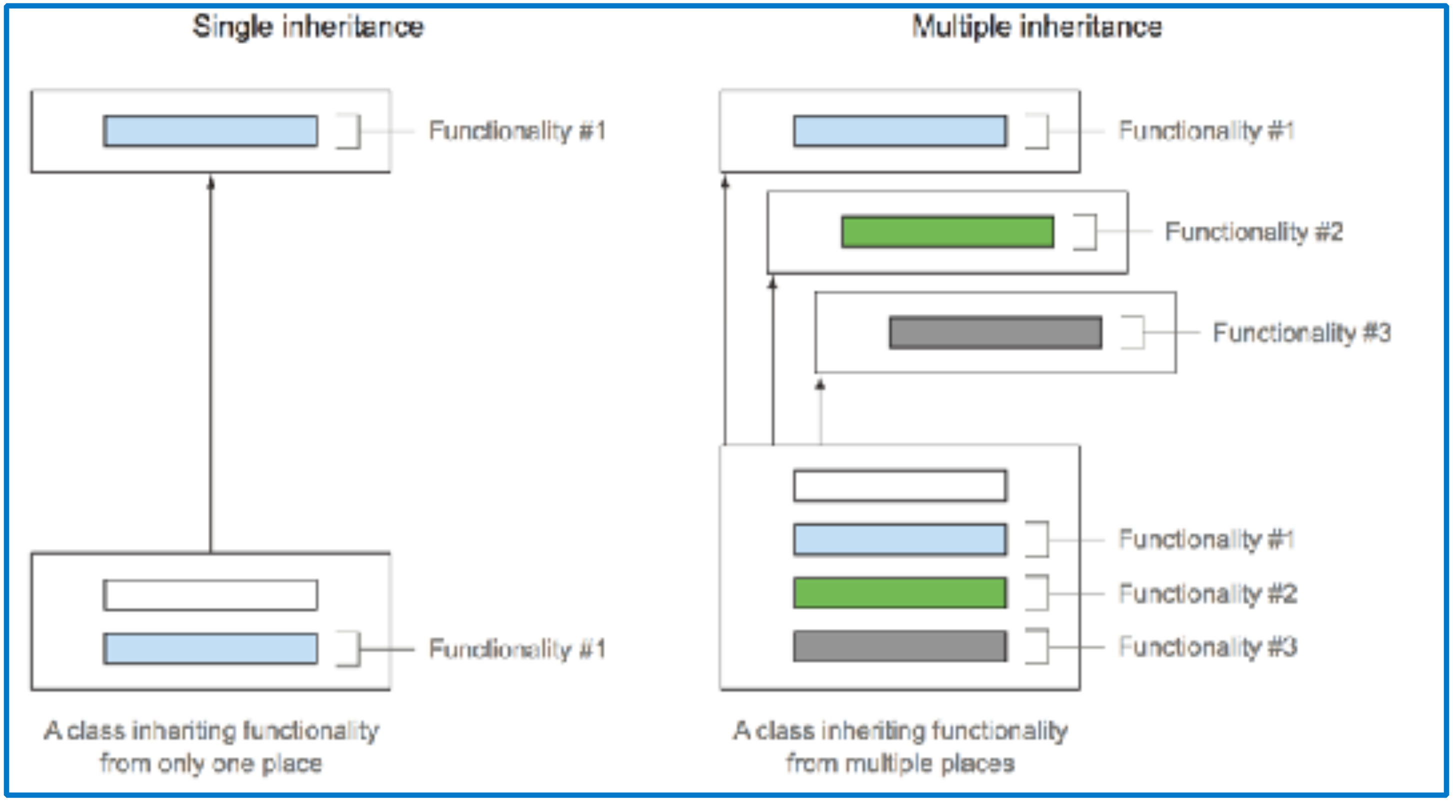
- About optional methods
  - use case: a class implements an interface but some methods are left empty
  - Java 8 solution: provide a default implementation for such methods
  - concrete classes don’t need to explicitly provide an empty implementation

2 Usage patterns for default methods

- optional methods and
- multiple inheritance of behaviour

- About multiple inheritance of behaviour
  - This is the ability of a class to reuse code from multiple places
  - classes can inherit from only one other class, but they can implement multiple interfaces
  - this opens up interesting possibilities

multiple inheritance of behaviour



MINIMAL INTERFACES WITH ORTHOGONAL FUNCTIONALITIES

- Define simple Interfaces around a distinct task
  - call them X,Y and Z
  - define abstract methods (get/setters) in X,Y and Z to parameterize the task
  - define a default method in the interface that executes the task
  - classes can “magically” acquire behaviour by implementing such an interface.

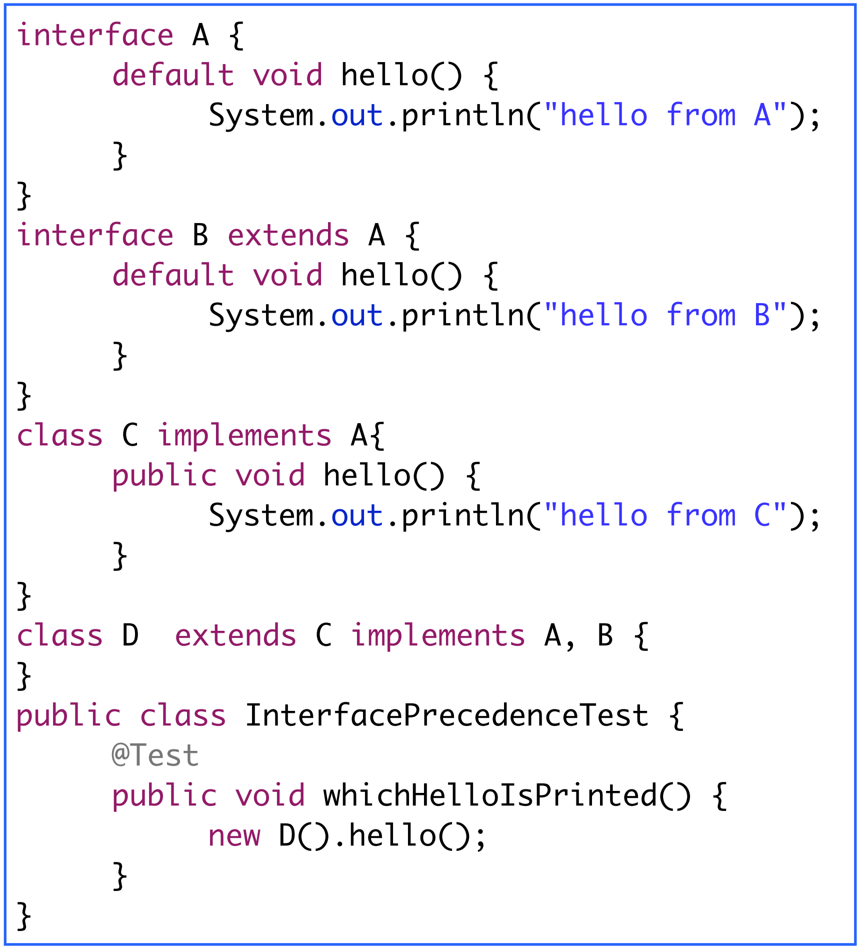
MINIMAL INTERFACES WITH ORTHOGONAL FUNCTIONALITIES

- optional methods and
- multiple inheritance of behaviour

- about multiple inheritance of behaviour
  - this is the ability of a class to reuse code from multiple places
  - classes can inherit from only one other class, but they can implement multiple interfaces
  - this opens up interesting possibilities

Diamond problem in Java 8 ?

- Consider the following setup of interfaces A & B and classes C and D



```
interface A {
	default void hello() {
		System.out.println("hello from A");
	}
}
interface B extends A {
	default void hello() {
		System.out.println("hello from B");
	}
}
class C implements A{
	public void hello() {
		System.out.println("hello from C");
	}
}
class D  extends C implements A, B {
}
public class InterfacePrecedenceTest {
	@Test
	public void whichHelloIsPrinted() {
		new D().hello();
	}
}
```

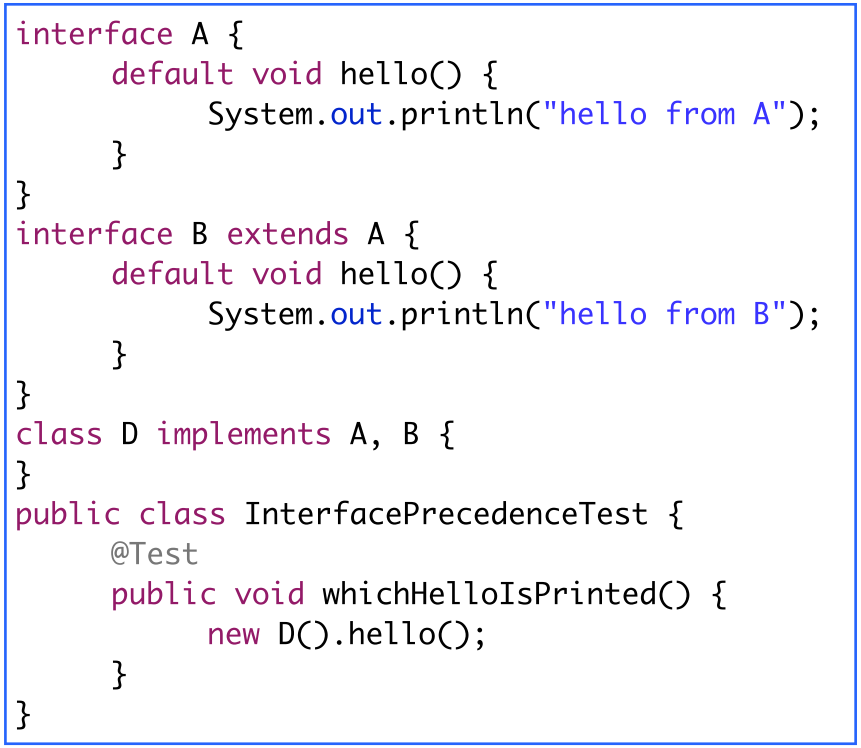
- Which implementation is chosen?
- hello from C

Diamond problem in Java 8 ?

- Three resolution rules to address this problem
  1. Classes always win: a method in the class or superclass takes priority over any default method declaration
  2. Next, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected
  3. Finally, if the choice is still ambiguous, the class has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly

Diamond problem in Java 8 ?

- Consider the following setup of interfaces A & B and class D



```
interface A {
     default void hello() {
          System.out.println("hello from A");
     }
}
interface B extends A {
     default void hello() {
          System.out.println("hello from B");
     }
}
class D implements A, B {
}
public class InterfacePrecedenceTest {
     @Test
     public void whichHelloIsPrinted() {
          new D().hello();
     }
}
```

- Which implementation is chosen?
- Note class C is removed
- hello from B

Diamond problem in Java 8 ?

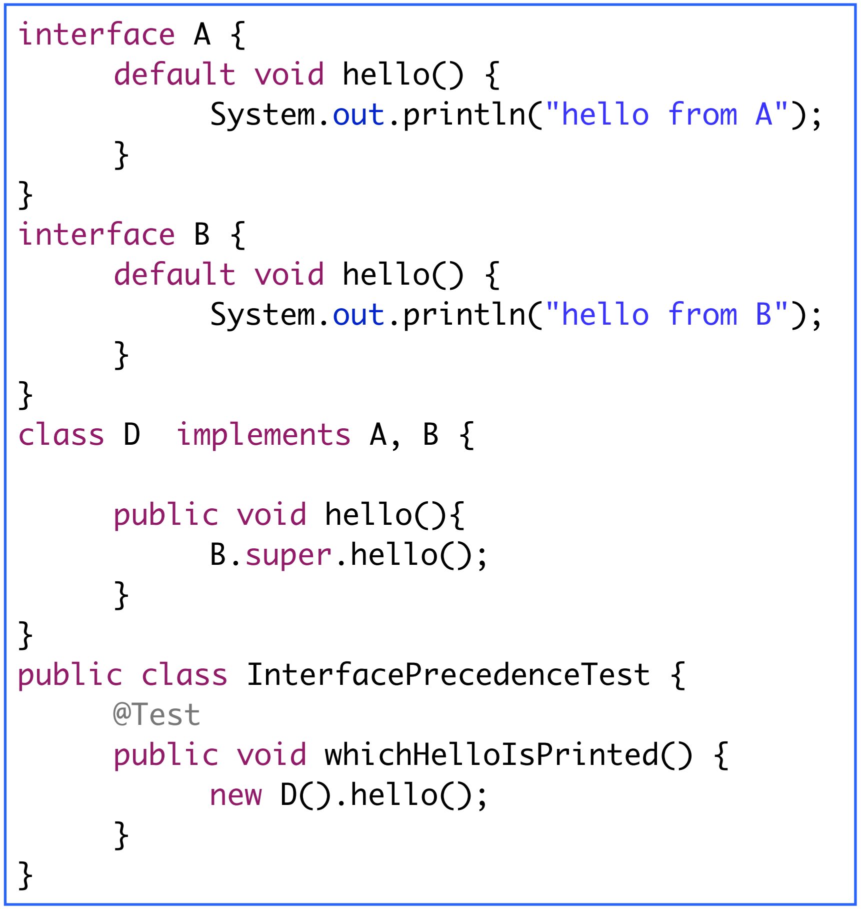
- Consider the following setup of interfaces A & B and class D

```
interface A {
	default void hello() {
		System.out.println("hello from A");
	}
}
interface B {
	default void hello() {
		System.out.println("hello from B");
	}
}
class D implements A, B {
}
public class InterfacePrecedenceTest {
	@Test
	public void whichHelloIsPrinted() {
		new D().hello();
	}
}
```

- Which implementation is chosen?
- Note B does not extend A anymore
- Compilation error

Diamond problem in Java 8 ?

- Consider the following setup of interfaces A & B and class D



```
interface A {
	default void hello() {
		System.out.println("hello from A");
	}
}
interface B {
	default void hello() {
		System.out.println("hello from B");
	}
}
class D  implements A, B {

	public void hello(){
		B.super.hello();
	}
}
public class InterfacePrecedenceTest {
	@Test
	public void whichHelloIsPrinted() {
		new D().hello();
	}
}
```

- Which implementation is chosen?
- Note B does not extend A anymore
- We have to explicitly say which interface to use

Optional<T>

A better alternative to null

Agenda

- What’s wrong with null references and why you should avoid them
- From null to Optional: rewriting your domain model in a null-safe way
- Putting Optionals to work: removing null checks from your code
- Different ways to read the value possibly contained in an Optional
- Rethinking programming given potentially missing values

How to model the absence of a value?

The datamodel

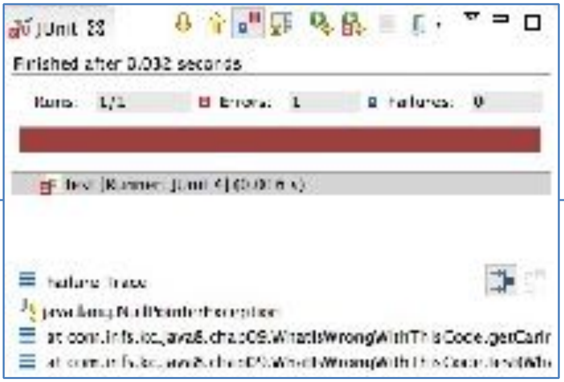
```
public class Person {
  private Car car;

  public Car getCar() {
          return car;
  }

  public void setCar(Car car) {
          this.car = car;
  }
}
```

```
public class Car {
  private Insurance insurance;

  public Insurance getInsurance() {
          return insurance;
  }

  public void setInsurance(Insurance insurance) {
          this.insurance = insurance;
  }
}
```

```
public class Insurance {
  private String name;

  public String getName() {
          return name;
  }

   public void setName(String name) {
          this.name = name;
  }
}
```

Brain breaker

```java
public class WhatIsWrongWithThisCode {

  @Test
  public void test() {
    Person person = new Person();
    String carInsuranceName = getCarInsuranceName(person);
    System.out.println(carInsuranceName);
  }

  public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
  }
}
```



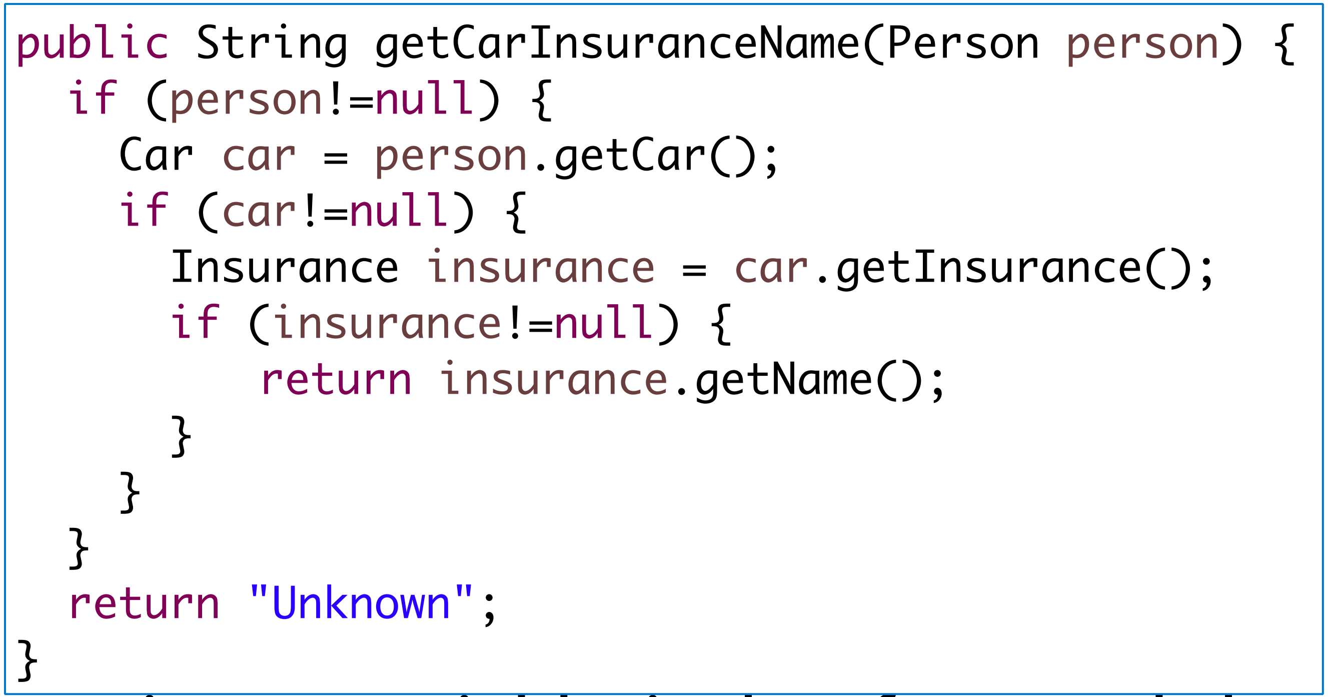
```
Finished after 0.032 seconds
Runs: 1/1    Errors: 1    Failures: 0
test [Runner: JUnit 4] (0.0[illegible] s)
Failure Trace
java.lang.NullPointerException
at com.infs.oc_java8.chap09.WhatIsWrongWithThisCode.getCarIr
at com.infs.oc_java8.chap09.WhatIsWrongWithThisCode.test(Wha
```

Code looks reasonable

- But persons don’t necessary own a car
- What to return when no car is present?
- Unfortunately the null reference is often returned to indicate the absence of a value

- What to do about it?

Standard reflex: null checking



```java
public String getCarInsuranceName(Person person) {
  if (person!=null) {
    Car car = person.getCar();
    if (car!=null) {
      Insurance insurance = car.getInsurance();
      if (insurance!=null) {
          return insurance.getName();
      }
    }
  }
  return "Unknown";
}
```

- Every time a variable is dereferenced that could be null perform a check
- Not null check on name (getName) is omitted because in this domain name can’t be null
- This not null property is not derivable

Other line of attack

- Former approach results in nested if statements which is poorly readable
- Trying to get rid of nested if results in:

```
public String getCarInsuranceName(Person person) {
  if (person==null) return "Unknown";
  Car car = person.getCar();
  if (car==null) return "Unknown";
  Insurance insurance = car.getInsurance();
  if (insurance==null) return "Unknown";
  return insurance.getName();
}
```

- This approach results in multiple exit points
- Easy to forget to check a null property

Problems with null

- It’s a source of NullPointerException, by far the most common Exception in Java
- null worsens readability by code with often deeply nested null checks
- null doesn’t have any semantic meaning
- null represents the wrong way to model the absence of a value in a statically typed language.
- Java always hides pointers from developers except in one case: the null pointer

Even more problems with null

- null creates a hole in the type system
  - Null carries no type or other information, meaning it can be assigned to any reference type
  - This is a problem because, when it’s propagated to another part of the system,you have no idea what that null was initially supposed to be

- ok, null’s are bad but what is the alternative?
- is there an alternative?

Look at Groovey

- Groovey introduced a safe navigation operator, represented by “?.”
- The java code becomes in Groovey:
  - def carInsuranceName =person?.car?.insurance?.name
- The Groovy ?. operator allows navigation without throwing a NullPointerException by propagating the null reference returning a null in the when any value in the chain is a null

A safe navigation operator in Java ?

- Proposed for Java 7 but discarded
- Often a developer’s reflex to a NullPointerException is introducing an if to check that a value is not null
  - Can be ok but you must first ask if it is allowed that this value is null in this particular situation?
  - Is it allowed that an algorithm or data model presents a null in this specific situation
- If this question is overlooked a bug may not be fixed but will be hidden
- A “?.” operator doesn’t address this pitfall

Inspiration from Haskell & Scala

- Haskell and Scala, functional languages , take a different approach
- Haskell includes a Maybe type, which essentially encapsulates an optional value
- A value of type Maybe contains either a value of a given type or nothing
- There is no concept of a null reference.
- Scala has a similar construct called Option[T] to encapsulate the presence or absence of a value of type T

Java Optional<T>

- With these types you have to explicitly check whether a value is present or not using operations available on the these types
- This enforces the idea of "null checking”
- No longer is it possible to "forget to do it" because it is enforced by the type system!
- Along the same lines Java adopted a new type Optional<T>

Introducing the Optional class

- Java 8 introduces a new class called java.util.Optional<T>
  - When a value is present, the Optional class just wraps it
- When a value is not present the absence of a value is modelled with Optional.empty()
- Optional.empty() is a static factory method which returns a singleton instance of the Optional class

Internals of Optional I.

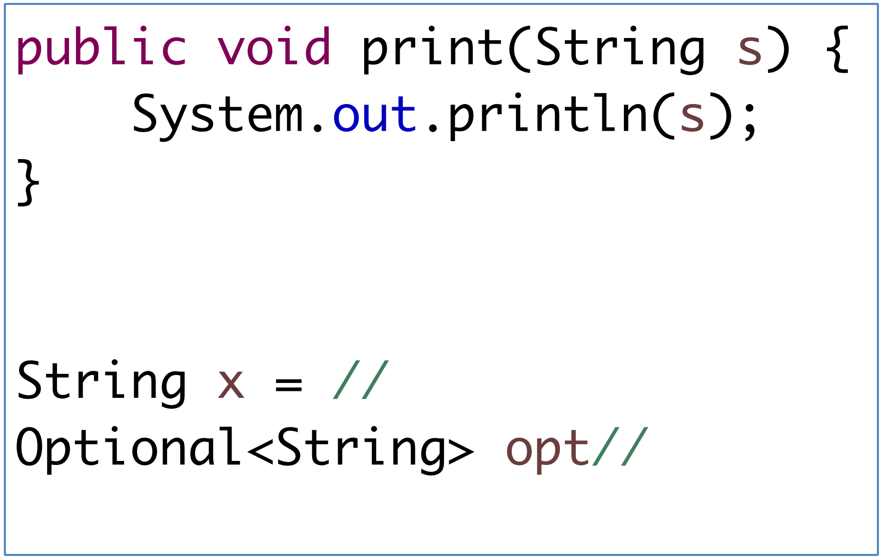
```
public final class Optional<T> {
    /**
     * Common instance for empty()
     */
    private static final Optional<?> EMPTY = new Optional<>();

    /**
     * If non-null, the value;
     * if null, indicates no value is present
     */
    private final T value;

    private Optional() {
        this.value = null;
    }
}
```

Internals of Optional II.

```
public T get() {
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
}
public boolean isPresent() {
    return value != null;
}
public T orElse(T other) {
    return value != null ? value : other;
}
public Optional<T> filter(Predicate<? super T> predicate) {
    Objects.requireNonNull(predicate);
    if (!isPresent())
        return this;
    else
        return predicate.test(value) ? this : empty();
}
```

Examples with Option type

- In the following examples given variables and types are used:



```
public void print(String s) {
    System.out.println(s);
}


String x = //
Optional<String> opt//
```

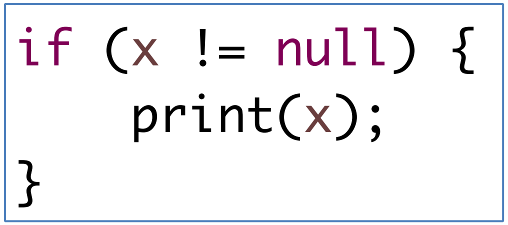
- x may be null but opt is never null , but may or may not contain some value

How to create Optional instances?

```
String notNull="if null an exception is throwed!";
opt = Optional.of(notNull);

String mayBeNull="either return empty or present (set) ";
opt = Optional.ofNullable(mayBeNull);

String remark=" always returns empty , "  +
              "corresponding to null (Singleton) ";
opt = Optional.empty();
```

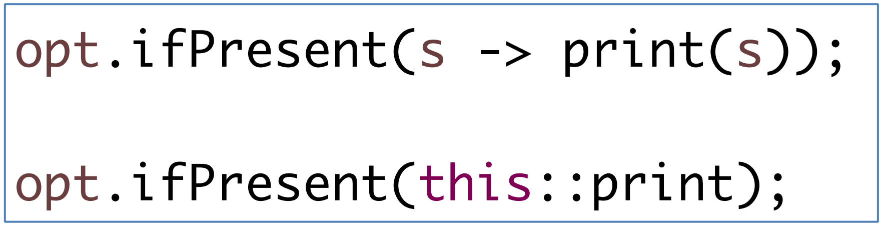
Do something when optional is set

- if statements to safeguard us for NullPointerExceptions can be omitted



```
if (x != null) {
    print(x);
}
```

- Use the methods on the optional instead



```
opt.ifPresent(s -> print(s));

opt.ifPresent(this::print);
```

reject, filter certain optional values

- Sometimes not only the reference must be set but also a certain condition must be met
- Pre Java 8 approach:

```
if (x != null && x.contains("criteria")) {
    print(x);
}
```

- Approach using Optional:

```
Opt.filter(s -> s.contains("ab"))
   .ifPresent(this::print);
```

transform value if present

- Often a transformation on a value must be applied, but only if it’s not null

```
if (x != null) {
    String t = x.trim();
    if (t.length() > 1) {
        print(t);
    }
}
```

- Achieve this using the map function

```
opt.
    map(String::trim).
    filter(t -> t.length() > 1).
    ifPresent(this::print);
}
```

Some trickery look at opt.map( )

```
opt.
    map(String::trim).
    filter(t -> t.length() > 1).
    ifPresent(this::print);
}
```

- If opt contains no value nothing happens and empty() is returned
- The transformation is type-safe

```
Optional<String> opt =Optional.of("1") ;

Optional<Integer> len = opt.map(String::length);
```

- if opt was empty, map() does nothing except changing generic type.

Turn empty Optional in default value

- Pre Java 8 solution

```
int length = (x != null)? x.length() : -1;
```

- Java 8 solution

```
int length = opt.map(String::length).orElse(-1);
```

- if computing default value is slow, expensive or has side-effects use the Supplier<T> version

```
length = opt.
             map(String::length).
             orElseGet(() -> slowOrExpensiveDefault());
```

What if method returns Optional

```java
public Optional<String> tryFindSimilar(String s){..}
```

- Flatten Optional<Optional<T> -> Optional<T> with flatmap

```java
Optional<Optional<String>> bad =
                              opt.map(this::tryFindSimilar);

System.out.println(bad.get().get());

Optional<String> nicer =
                           opt.flatMap(this::tryFindSimilar);

System.out.println(nicer.get());
```

Lazily throw exceptions

- Throw an exception if value is not available

```
public char firstChar(String s) {
    if (s != null && !s.isEmpty())
        return s.charAt(0);
    else
        throw new IllegalArgumentException();
}
```

- Don’t create an instance of exception in advance because it can be expensive

```
opt.
filter(s -> !s.isEmpty()).
map(s -> s.charAt(0)).
orElseThrow(IllegalArgumentException::new);
```