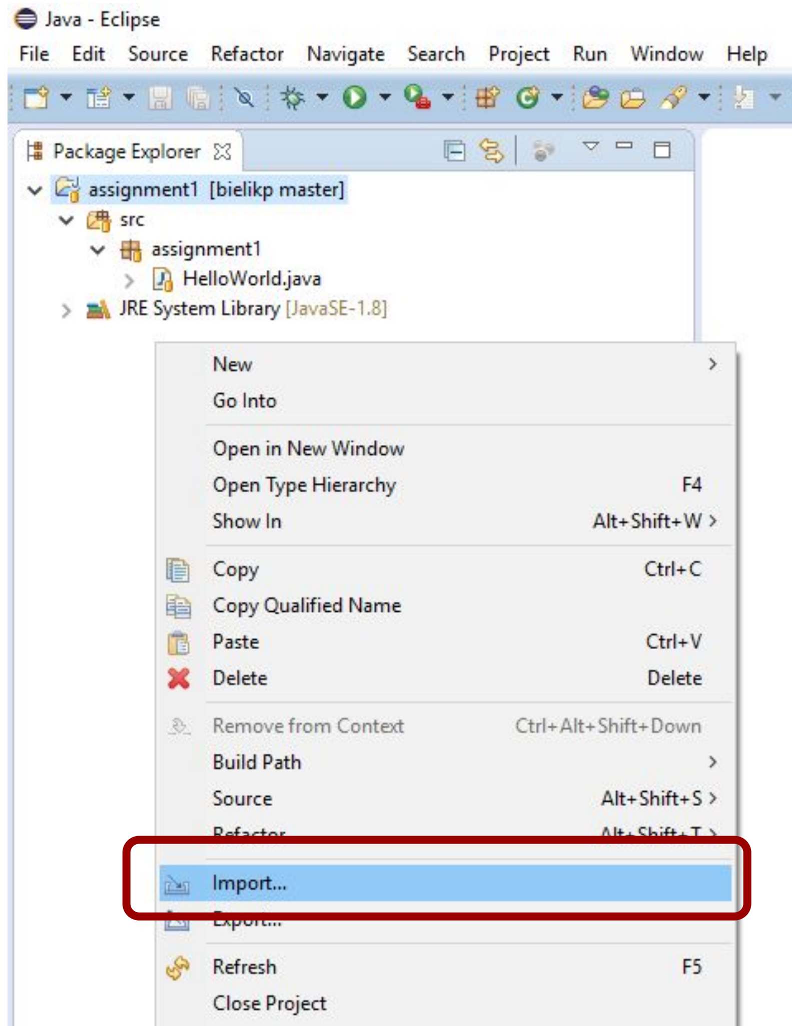
Parallel Programming Exercise Session 2

Spring 2020

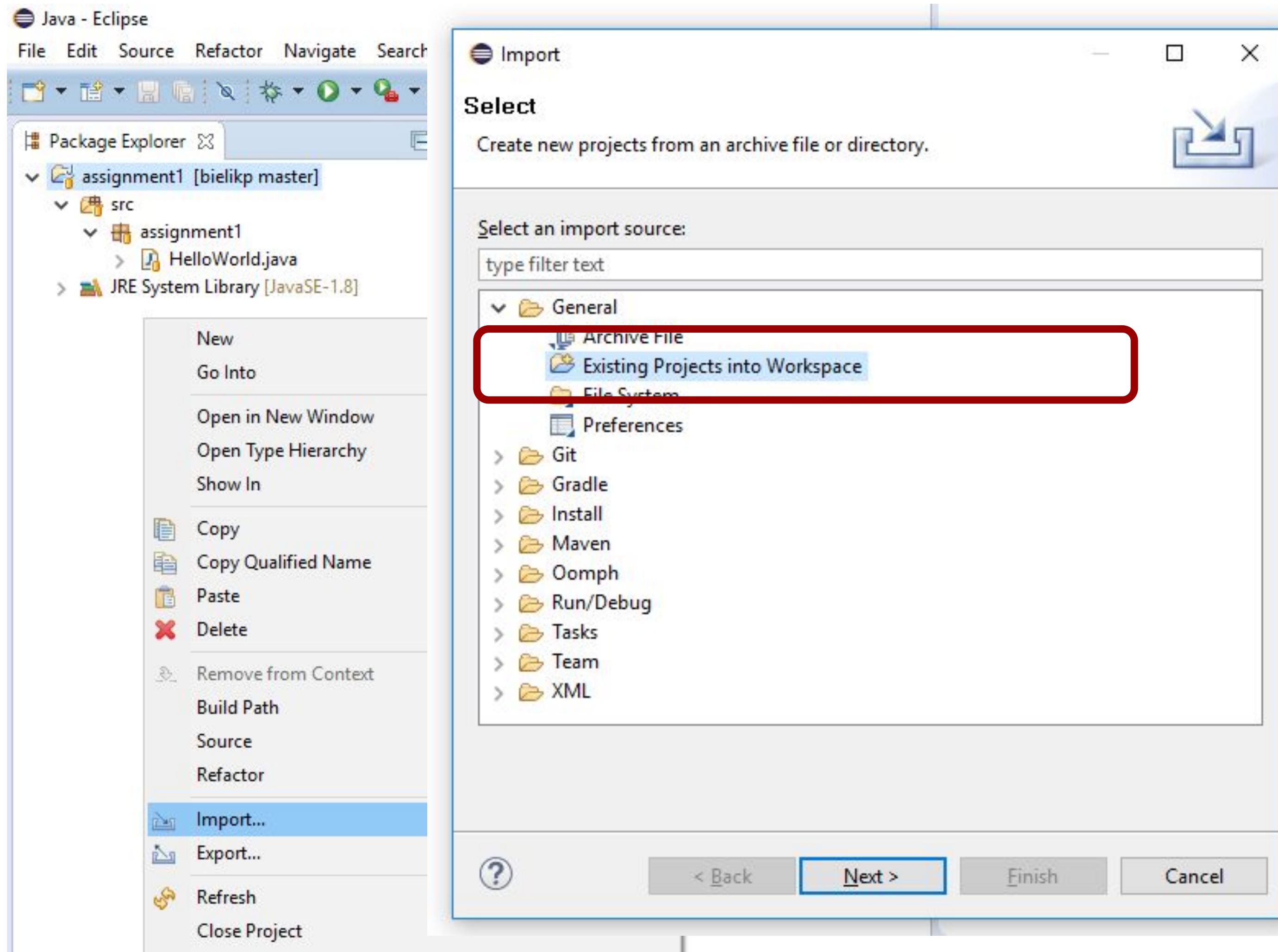
Preparations

1. Import assignment2.zip in Eclipse
2. Run the projects unit-tests in Eclipse
3. Understand output of unit-tests
    - Did the test fail or succeed?
    - Why did the test fail?
4. Start coding and keep checking if tests pass

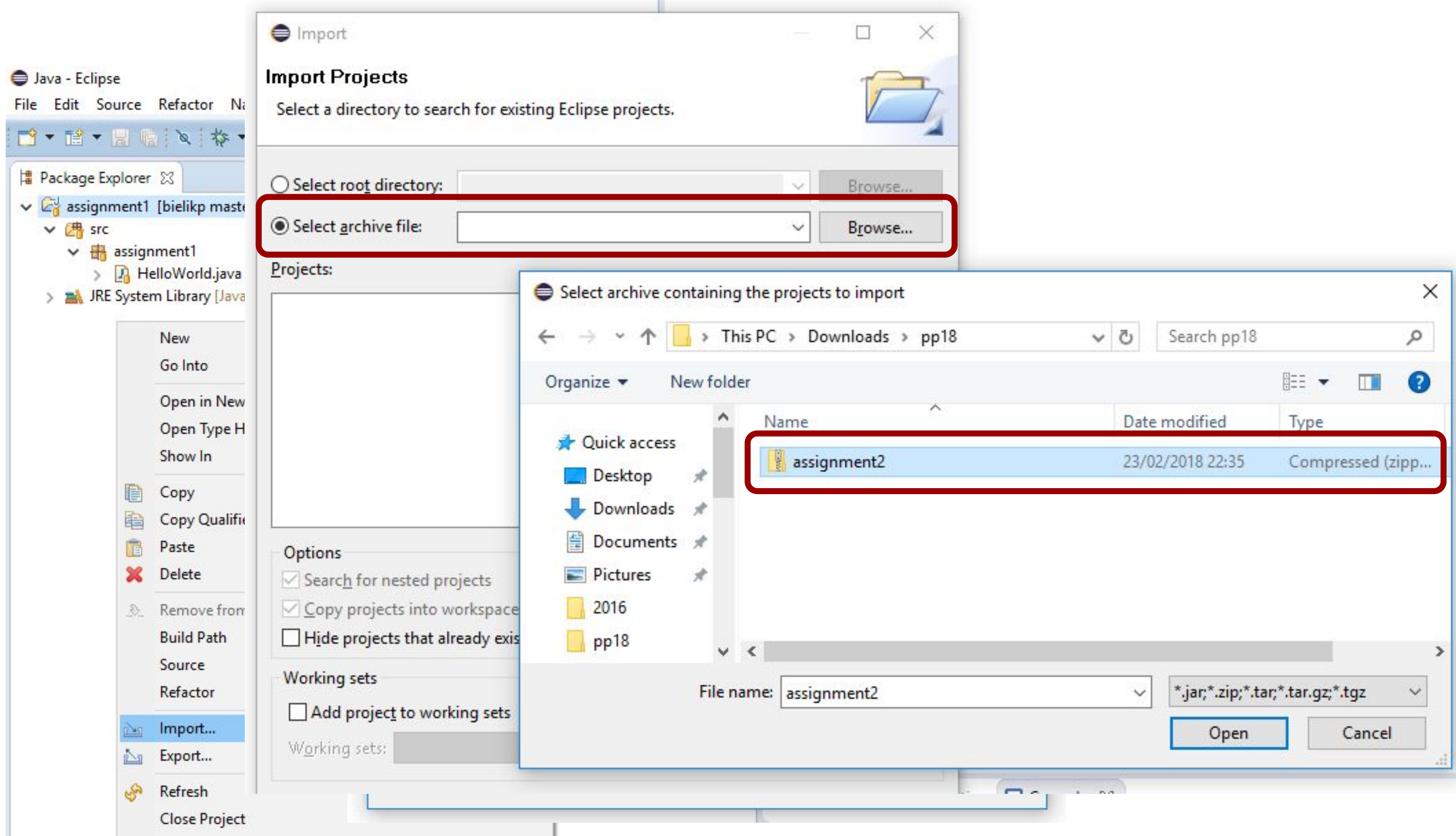
Eclipse: import project



Eclipse: import project

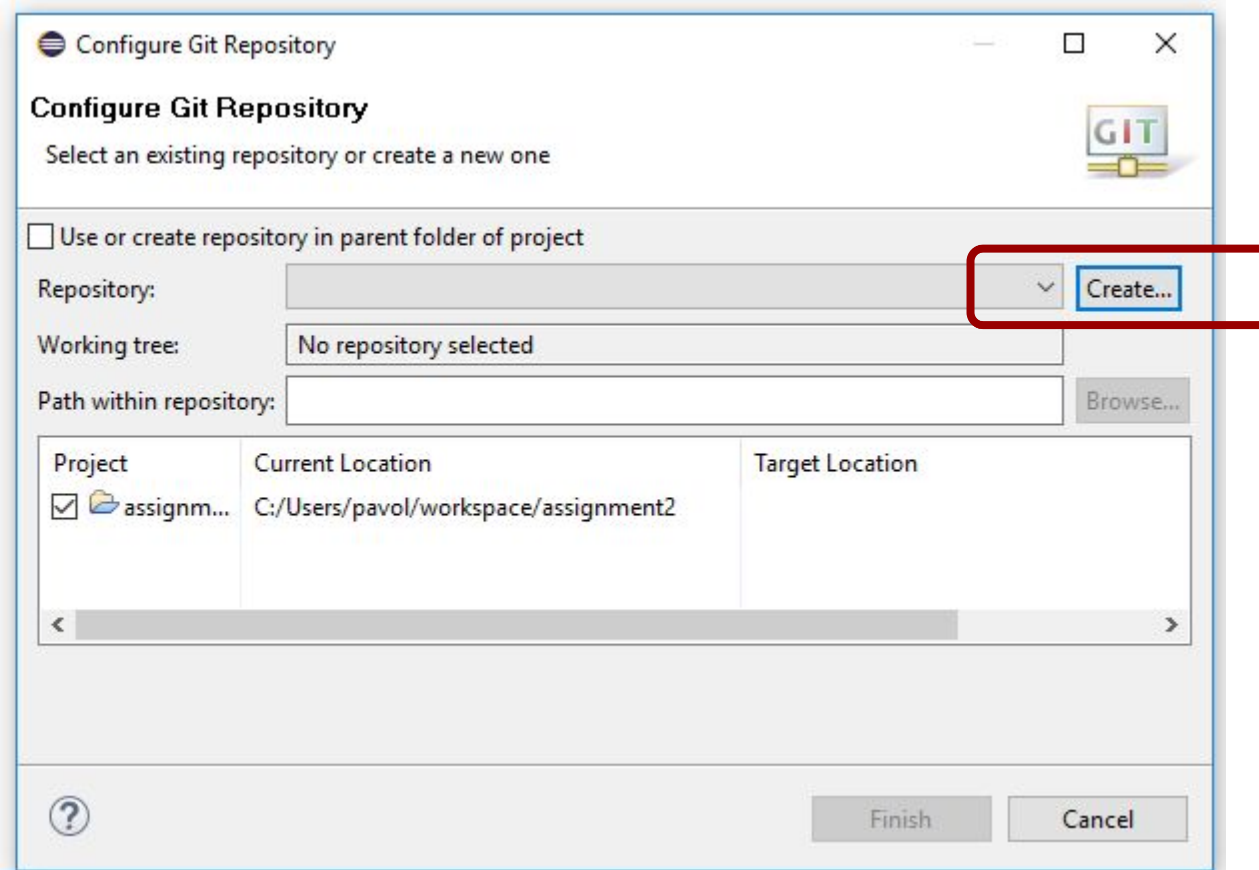


Eclipse: import project

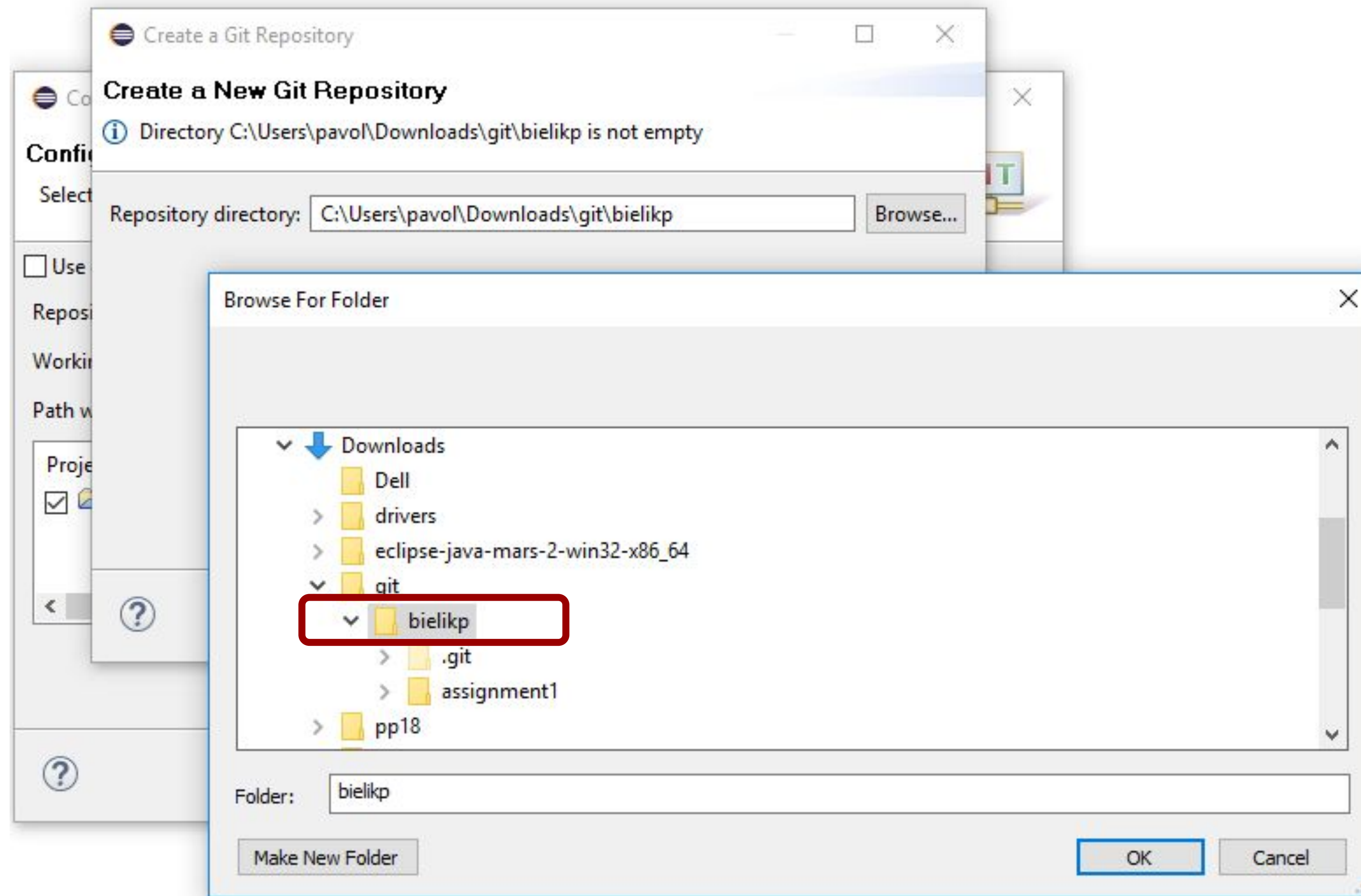


Eclipse: add to git

Team -> Share Project ...

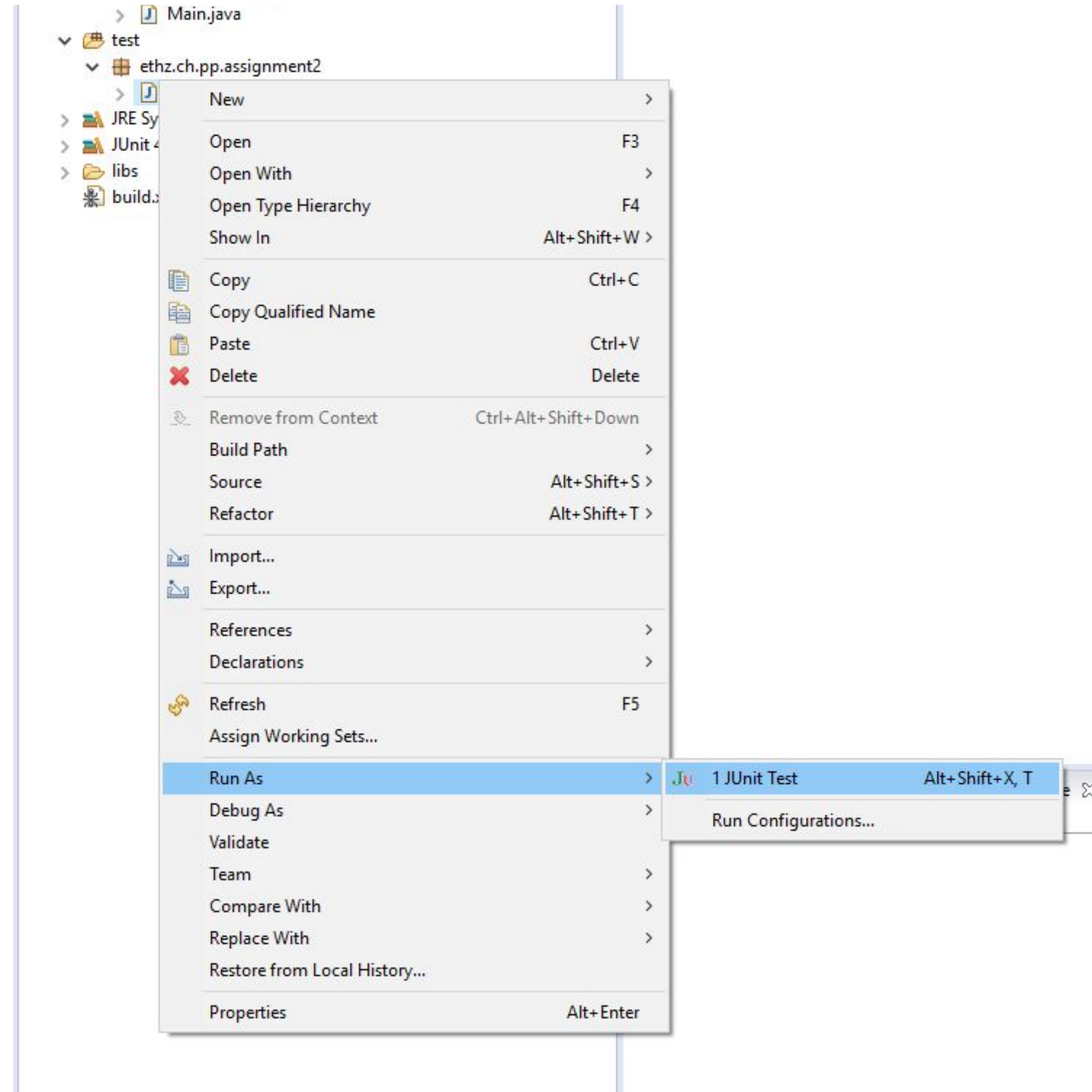


Eclipse: add to git

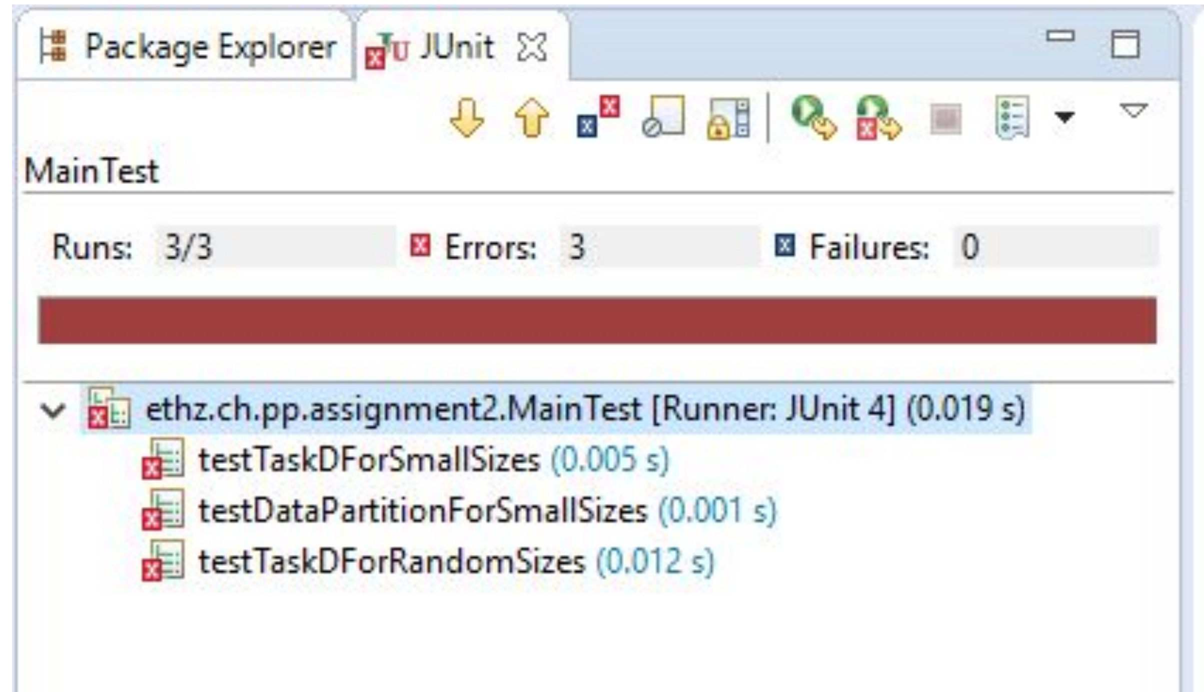


Important: Select same directory as for assignment 1

Eclipse: running JUnit tests (1)



Eclipse: running JUnit tests (2)

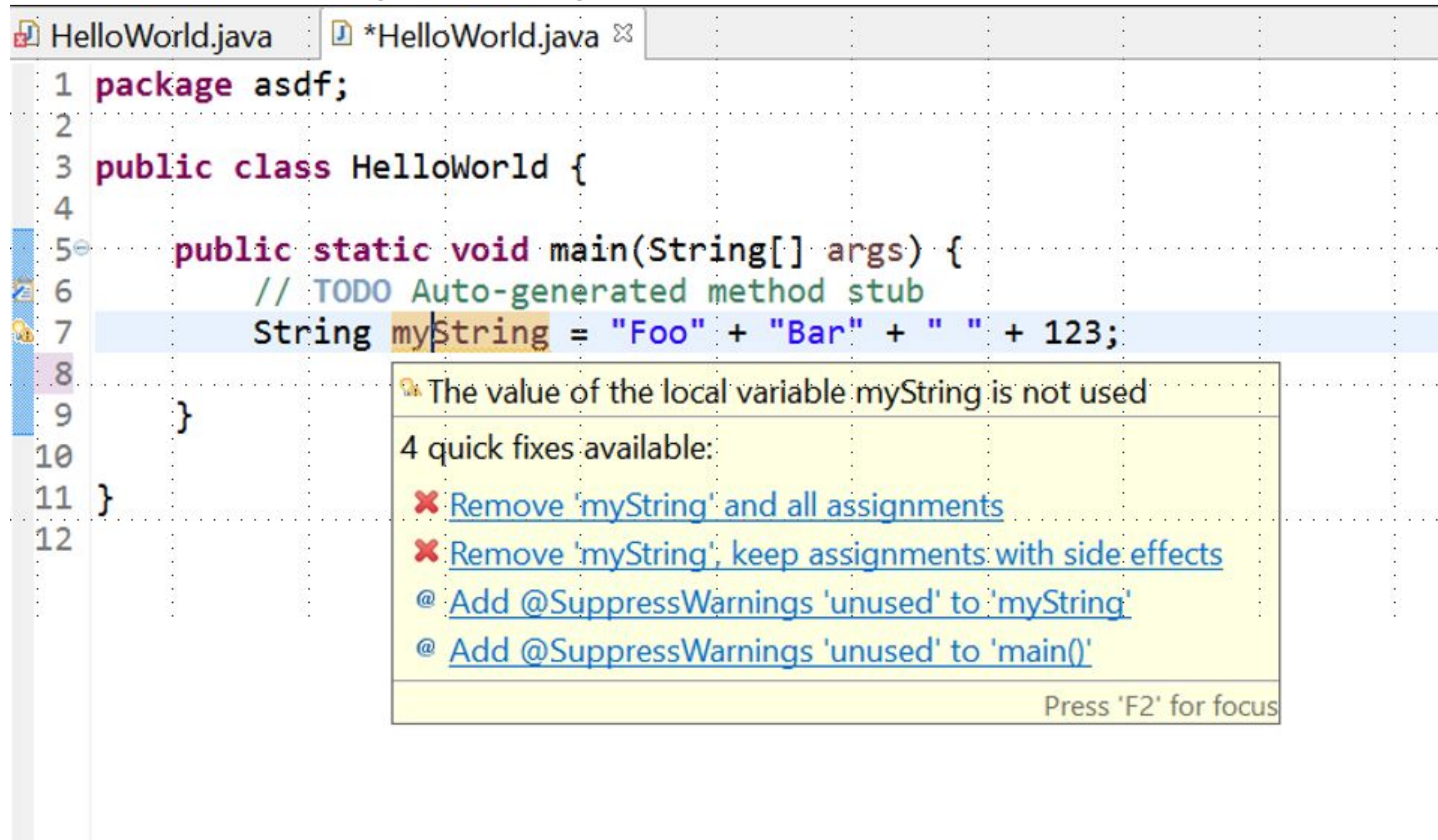


Code Style

- Try to make your code as readable as possible
  (Use Eclipse formatter <CTRL>+<SHIFT>+F)
- Include high-level comments that explain why you are doing something (much better than a line-by-line commentary of your code)

Code Style / Errors

Keep attention what Eclipse reports:



Java Doc (http://docs.oracle.com/javase/7/docs/api/)

java.text
java.text.spi
java.util
java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.jar
java.util.logging
java.util.prefs
java.util.regex
java.util.spi
java.util.zip
javax.accessibility
javax.activation
javax.activity

java.util

Interfaces

Collection
Comparator
Deque
Enumeration
EventListener
Formattable
Iterator
List
ListIterator
Map
Map.Entry
NavigableMap
NavigableSet
Observer
Queue
RandomAccess
Set
SortedMap
SortedSet

Classes

AbstractCollection
AbstractList
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractQueue
AbstractSequentialList
AbstractSet
ArrayDeque

Overview Package Class Use Tree Deprecated Index Help — *Java™ Platform Standard Ed. 7*

Prev Class **Next Class** Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

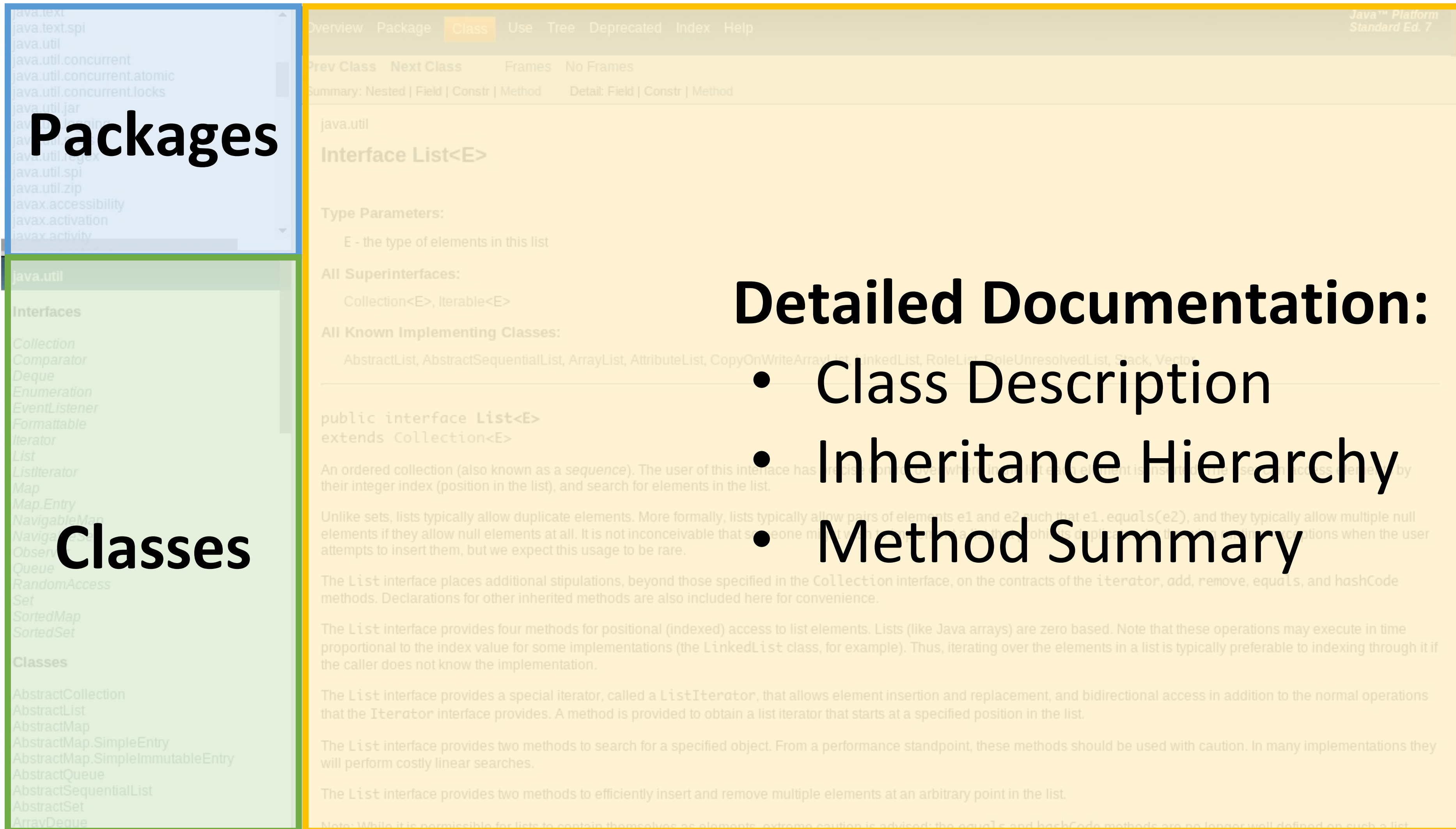
The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the `Iterator` interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The `List` interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The `List` interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a list.

Java Doc (http://docs.oracle.com/javase/7/docs/api/)



java.text
java.text.spi
java.util
java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.jar
java.util.logging
java.util.prefs
java.util.regex
java.util.spi
java.util.zip
javax.accessibility
javax.activation
javax.activity

java.util

Interfaces

Collection
Comparator
Deque
Enumeration
EventListener
Formattable
Iterator
List
ListIterator
Map
Map.Entry
NavigableMap
NavigableSet
Observer
Queue
RandomAccess
Set
SortedMap
SortedSet

Classes

AbstractCollection
AbstractList
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractQueue
AbstractSequentialList
AbstractSet
ArrayDeque

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

add

```
void add(int index,
       E element)
```

Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Parameters:

`index` - index at which the specified element is to be inserted

`element` - element to be inserted

Throws:

`UnsupportedOperationException` - if the *add* operation is not supported by this list

`ClassCastException` - if the class of the specified element prevents it from being added to this list

`NullPointerException` - if the specified element is null and this list does not permit null elements

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this list

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index > size()`)

Method Signature

Semantic description what the method does

Parameter description

Possible occurring errors

remove

```
E remove(int index)
```

Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.

Parameters:

`index` - the index of the element to be removed

Returns:

the element previously at the specified position

Throws:

`UnsupportedOperationException` - if the `remove` operation is not supported by this list

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

Task A

To start with, print to the console "Hello Thread!" from a new thread. How do you check that the statement was indeed printed from a thread that is different to the main thread of your application? Furthermore, ensure that you program (i.e., the execution of main thread) finishes only after the thread execution finishes.

Task A: How to create and start a new thread?

option 1: Extend class Thread

```
class ConcurrWriter extends Thread { …
     public void run() { … }
}
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start();   // calls ConcurrWriter.run()
```

option 2: Implement Runnable

```
public class ConcurrReader implements Runnable {
     …
     public void run() { …
          … code here executes concurrently with caller … }
}

ConcurrReader readerThread = new ConcurrReader();
Thread t = new Thread(readerThread);
t.start();   // calls ConcurrReader.run() automatically
```

Task B

Run the method computePrimeFactors in a single thread other than the main thread. Measure the execution time of sequential execution (on the main thread) and execution using a single thread. Is there any noticeable difference?

Task C

Design and run an experiment that would measure the overhead of creating and executing a thread.

Task C

option 1: Measures real time elapsed including time when the thread is not running.

```
long time = System.nanoTime();
//compute something
time =  System.nanoTime() - time;
```

option 2: Measures thread cpu time excluding time when the thread is not running.

```
ThreadMXBean tmxb = ManagementFactory.getThreadMXBean();
long time = tmxb.getCurrentThreadCpuTime();
//compute something
time = tmxb.getCurrentThreadCpuTime()-time;
```

Task D

Before you parallelize the loop in Task E, design how the work should be split between the threads by implementing method PartitionData. Each thread should process roughly equal amount of elements. Briefly describe you solution and discuss alternative ways to split the work?

Task D: Split the work between the threads

```
PartitionData(int length, int numPartitions) { ... }
```

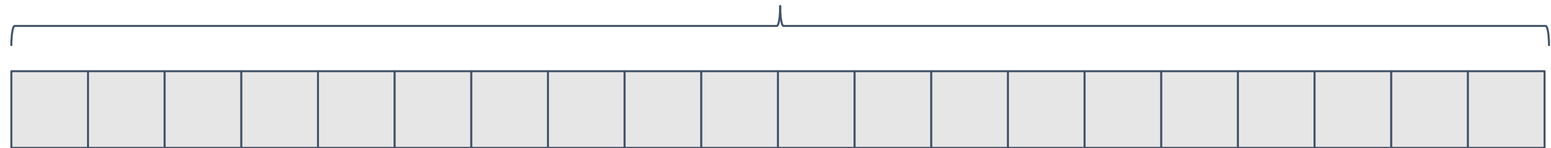
a) PartitionData(20,1) ?

b) PartitionData(20,2) ?

c) PartitionData(20,3) ?

Task D: Split the work between the threads

PartitionData(int length, int numPartitions) { … }

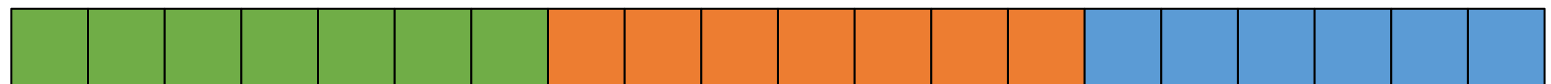


Input

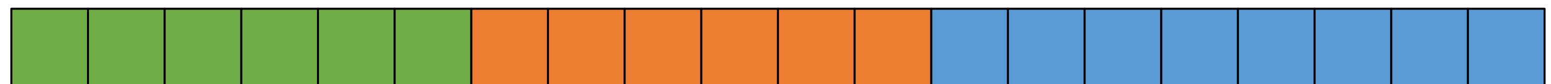
a) PartitionData(20,1)

b) PartitionData(20,2)

c) PartitionData(20,3)



d) PartitionData(20,3)



both c) and d) are correct solutions for this exercise

Task D

- What about (length>0 and numPartitions>0) and length<numPartitions?
    - ??
    - ??
- And (length<=0 or numPartitions<=0)?
    - ??
    - ??

```
PartitionData(int length, int numPartitions) {  …  }
```

Task D

- What about (length>0 and numPartitions>0) and length<numPartitions?
    - Throw an exception?
    - Return m = min(m,n) splits?
- And (length<=0 or numPartitions<=0)?
    - Throw an exception?
    - Create a default return value (e.g. new ArraySplit[0])?
- In any case, write your assumptions in JavaDoc

```
PartitionData(int length, int numPartitions) {  …  }
```

Task E

Parallelize the loop execution in computePrimeFactors using a configurable amount of threads.

Task F

Think of how would a plot that shows the execution speed-up of your implementation, for n = 1, 2, 4, 8, 16, 32, 64, 128 threads and the input array size of 100, 1000, 10000, 100000 look like

Task G

Measure the execution time of your parallel implementation for n = 1, 2, 4, 8, 16, 32, 64, 128 threads and the input array size of input.length = 100, 1000, 10000, 100000. Discuss the differences in the two plots from task F and G.