JUnit Testing + Generics

Lecture 5 – Various left-overs!

Dr Jonas Lundberg

Jonas.Lundberg@lnu.se

Slides and Java examples are available in Moodle

5 februari 2020

Agenda

- Software Testing in General.
- JUnit Testing
- Generics

Reading Instructions
Testing: Only these slides plus web instructions from junit.org
Generics: Sections 19.1-19.8 (Skip 19.9)

Jacv Examples (avilable in Moodle)
A complete JUnit test case for an **IntList** implementation
A generic version of an **IntList** implementation

Software Testing

Testing: the process of executing a program with the intent of finding errors.

Myers, 1979

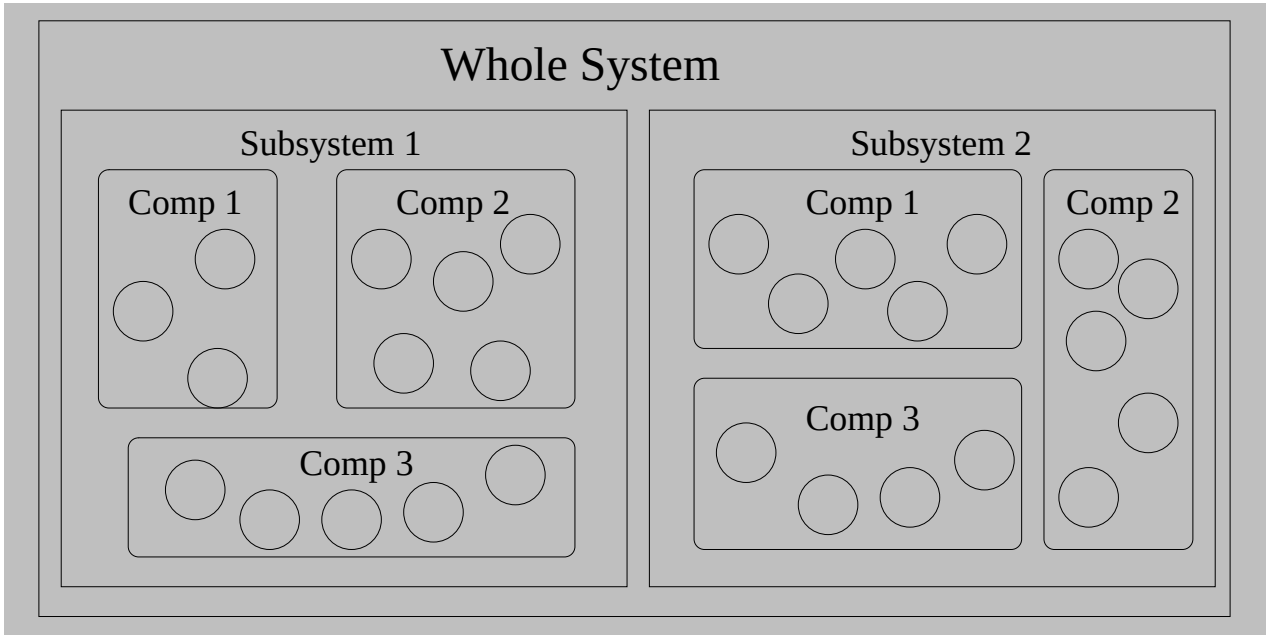
Testing in General

- Software Development
- Motivation
- Different types of testing
- Limitations

Unit Testing

- Unit testing
- JUnit
  - Tool Presentation
  - Test Cases
  - JUnit Design

A Hierarchical View of an OO System



- A **system** consists of many **subsystems** ...
- ... which consists of many **components** ...
- ... which consists of many **classes**.
- A class is the smallest unit in an object-oriented system.

Different Types of Testing

- **Component** or **Unit**
  Test of isolated components in order to find errors.
- **Integration**
  Test the interaction of components and subsystems in order to find errors.
- **Regression**
  Test that takes place after a code change/update.
- **Performance**
  Test to verify acceptable time and memory costs.
- **Release** (α-testing)
  Test the whole system in order to verify that all requirments are satisfied.
- **Acceptance** (β-testing)
  Actual client/customer test the system in their own enviroment using realistic data.

Independent, Black- and White-box Testing

- **Independent Testing**: Developers have a tendency to write tests that "fits" their implementation. Not becuase they are lazy or evil, but because they are trapped in a way of thinking.
  - Advantage: Better chans to find logical errors.
  - Disadvantage: More costly in terms of hours (or money).

- **Black-Box Testing**: The test designer has no access/knowledge of implementations details. An interface test checking if a system/component fulfills its requirement spec.
  - Advantage: Independent testing possible
  - Disadvantage: Impossible to make sure that every part of the code is tested.

- **White-Box Testing:** The test designer has access to the source code and can make sure that every part of the code is tested.
  - Advantage: Every part of the code can be tested.
  - Disadvantage: Practically impossible for larger system.

Test Limitations

Testing can only show the presence of errors, not their absence.
Dijkstra, 1972

In general

- Infinite set of possible input data
- Infinite set of possible user scenarios

$\Rightarrow$ We can not test all possible executions
$\Rightarrow$ We can not prove absence of errors.

Verification: Theoretical proofs that a program (or method) always behaves correctly. Extremely time-consuming $\Rightarrow$ rarely used in practice.

Why Testing Then?

Testing is a process intended to build confidence in the software.
Sommerville, 2004

Thus, tests do not verify the absence of errors,
they only make them less likely.

Reasons for testing

1. Detect Defects:
   - System crashes
   - Incorrect computations
   - Data corruption

2. Verify (indicate) that system meets its requirements

When Should Testing Take Place?

Possible times for testing

- After system development but before release
  - The old fashion approach
  - Proven to be very costly for larger systems
  - Still popular among students
- As an integrated part of the development process
  - The recommended approach today
  - Errors are discovered as early as possible

In general, we want to find problems/errors as early as possible.

Components and Unit Testing

- Components are rather small and self-consisting parts of a system
- Typical components:
  - Data structures (e.g. lists, trees, graphs, sets)
  - Algorithms (e.g. Merge sort, Strongly Connected Components)
  - Well-defined entities in general
- In this context: Every part of the program that can be tested in isolation is a component.

Unit Testing

- Goal: Convince ourselves that a component is working properly in isolation (and detect errors indicating that it doesn't).
- When: During (preferred) or directly after implementation
- Who: The component developers
- Type: White-box

Suitable fixtures will be discussed later

Example: A Sorting Routine

- Assume a method `sort(int[] array)` that sorts integers
- Exhaustive testing ⇒ test all possible cases ⇒ impossible!
- My suggestion:
  - `array = new int[0];` (empty list)
  - `array = {7};` (singleton list)
  - `array = {1,34,56,2,-8,9,61,-55};` (even size)
  - `array = {-10,34,8,-6,56,12,78};` (odd size)
  - `array = {3,34,-8,76,-13, ... };` (very large array)
- Reasons:
  - Most problems show up at extreme values
  - Many algorithms (e.g. merge and quick sort) works a bit different for odd and even sized lists.
  - Test reasonable speed and memory requirement using a very large array.

Unit Testing: Choosing Test Data Sets

- General:
  - Test all extreme cases
  - Test a few (not many) standard cases
  - Test scalability by testing a *large* case
    (Only makes sense for certain methods. Pointless for get/set methods.)
- White-box testing:
  - Make sure that every method is tested (Minimum!)
  - Make sure that every statement is tested (Realistic Goal)
- Experience: constructing a suitable test data sets is often more time consuming than writing the actual tests.
- Notice: To figure out what input data that is required to execute a certain statement is in general non-trivial.
- The developers are most suited to handle this type of tests.

Unit Testing: Suggested Approach

The literature suggests

- implement method m_1, test m_1
- implement method m_2, test m_2
- ...

The Extreme Programming (XP) Approach

- write test for m, implement m, test m
- ...

The XP believers claim that:

- Writing tests first is a good preparation.
  (It forces you to identify problematic cases)
- The implementation is more goal directed.
  (Your task is completed when you pass the test.)
- You will save time once you get used to it.

Non-believers say that it doesn't scale to larger projects.

JUnit: Introduction

- JUnit is a tool designed to simplify the testing of Java components
- It is for free. Information and download at

  `http://junit.org/`

- Available in other languages (C++, Visual Basic, Python)
- Supported by Eclipse and IntelliJ
- Uses annotations and reflection $\Rightarrow$ difficult to understand what is going on
- My view of JUnit:
  - A very useful tool (together with Eclipse/IntelliJ)
  - Easy to start using
  - Difficult to understand in detail how it works
- We will use the latest version (**JUnit 5**) that makes use of new Java features like Lambda Expressions
- **JUnit in Eclipse** (Similar in IntelliJ I hope)
  - Eclipse provides plenty of support for JUnit test cases.
  - Execution: Right-click and chose "Run as JUnit Test"

First Example: MyMath.mult(int a, int b)

Method to be tested:

```
// A static method computing a*b using recursion.
// Requires 2nd argument b to be non-negative.
public static int mult(int a,int b) {
   if (b < 0)
      throw new IllegalArgumentException("2nd parameter must be non-negative
   else if (b == 0)
      return 0;
   else
      return a + mult(a,b-1);
}
```

Test Idea

- Test extreme cases: `0*5, 5*0, 1*5, 5*1`
- Test standard cases: `7*5, 12*16`
- Test with large values: `27638*7492`
- Test exception: `10*-1`

First JUnit Test

```
// Support for JUnit annotations (e.g. @Test)
import org.junit.jupiter.api.Test;
 // Make static Assert methods available
import static org.junit.jupiter.api.Assertions.*;

public class MyMathTest {
   @Test public void testMultiply() {
      // Test extreme cases
      assertEquals(0, MyMath.mult(10, 0));  // check 10*0 = 0
      assertEquals(10, MyMath.mult(10, 1));  // check 10*1 = 10
      assertEquals(0, MyMath.mult(0, 10));  // check 0*10 = 0

      // Test a few standard cases, provide
      assertEquals(50, MyMath.mult(10, 5), "10 x 5 = 50");
      assertEquals(-35, MyMath.mult(-7, 5), "-7 x 5 = -35");

      // Test with large numbers, check 23246*7958 = 184991668
      assertEquals(184991668, MyMath.mult(23246, 7958));

   ...
   }
```

Details in First Example

- Required imports:

```
// Support for JUnit annotations (e.g. @Test)
import org.junit.jupiter.api.Test;
 // Make static Assert methods available
import static org.junit.jupiter.api.Assertions.*;
```

- Each method annotated with `@Test` is a *Test Case*
  $\Rightarrow$ A *run* in JUnit that either fails or passes
- Methods starting with `assert...` are the actuals tests

```
assertEquals(0, MyMath.mult(0, 10));
assertEquals(50, MyMath.mult(10, 5), "10 x 5 must be 50");
```

  In the second case we provide a suitable error message
- There are many available assert methods
- There are many available annotations
- Test 1-2 methods in each Test method

Exception Testing

MyMath.mult() should raise an exception when the 2nd argument is negative.

```
public class MyMathTest {
   @Test public void testMultiply() {
      // Test extreme cases
      assertEquals(0, MyMath.mult(10, 0));  // check 10*0 = 0
     ...   other tests

      // Check for IllegalArgumentException
      assertThrows(IllegalArgumentException.class,
                   () -> MyMath.mult(10, -1));   // Trigger exception
```

JUnit 5 use lambda expressions (a Runnable) to trigger exceptions.

assertThrows takes a Runnable and an (expected) exception type as input.

Second Example: MyMath.sort(int[] in)

```
// A static method sorting an integer array using Selection Sort.
public static void sort(int[] in) {
    int sz = in.length;
    for (int i=0; i<sz-1; i++) {
        int first = i;           // position to update
        int min = first;         // initialize min
        for (int j=first+1; j<sz; j++) {  // remaining elements
            if ( in[j] < in[min] )
                min = j;
        }
        /* Swap first and min */
        int tmp = in[first];
        in[first] = in[min];
        in[min] = tmp;
    }
}
```

Test Idea: empty array, singleton array, a few standard cases, one large array

Second JUnit Test Method

```
@Test public void testSorting() {
    // Dropping small tests (empty, singleton, few elements, ... )

    int[] arr5 = random(100,100);   // Random array of size 100
    MyMath.sort(arr5);
    for (int i=0; i<arr5.length-1;i++)
        assertTrue( arr5[i] <= arr5[i+1] );        // Check if sorted

    int[] arr6 = random(100000,1000000);  // About 15s to sort 100000 elements
    MyMath.sort(arr6);
    for (int i=0; i<arr6.length-1;i++)
        assertTrue( arr6[i] <= arr6[i+1] );        // Check if sorted
}
private int[] random(int size,int max) {   // Private help method
    Random rand = new Random();
    int[] arr = new int[size];
    for (int i=0;i<size;i++) {           // "size" elements
        arr[i] =1+rand.nextInt(max);     // in range [1,max]
    }
    return arr;
}
```

The `Assert` Class

```
// JUnit annotations (e.g. @Test, @AfterEach, @BeforeEach)
import org.junit.jupiter.api.Test;
 // Make static Assert methods available
import static org.junit.jupiter.api.Assertions.*;
```

The class `Assert` contains static methods like

- `assertEquals` (uses `equals()` on objects)
- `assertSame`, `assertNotSame` (uses `==`, `!=` on objects)
- `assertTrue`, `assertFalse`
- `assertNull`, `assertNotNull`

The static import has the same effect as

```
import org.junit.jupiter.api.Assertions;  // Make Assertions
                                          // class available
@Test public void testMultiplying() {
     ...
    Assertions. assertEquals(35, MyMath.mult(5,7));
```

A Larger Example: `IntList`

We have a class `IntListImpl` implementing the following interface:

```
public interface IntList {
   /** Add integer n to list */
   public void add(int n);

   /** Remove integer at position index */
   public void remove(int index) throws IndexOutOfBoundsException;

   /** Get integer at position index */
   public int get(int index) throws IndexOutOfBoundsException;

   /** Number of integers currently stored */
   public int size();

   /** Find position of integer n, otherwise return -1 */
   public int indexOf(int n);

   /** Sort list in ascending order */
   public void sort();
}
```

Example: `IntListTest` - Overview

```
public class IntListTest {
   private static int count = 0;

   /* Executed before every test method. */
   @BeforeEach public void setUp() {
      System.out.println("Run test method: "+(++count));
   }
   /* Executed after every test method. */
   @AfterEach public void tearDown() {
      System.out.println(" --- done with test "+count);
   }

   /* My list of test methods */
   @Test public void testInitSize() { ... }
   @Test public void testAddGet() { ...}
   @Test public void testIndexOf() { ... }
   @Test public void testRemove() { ... }
   @Test public void testSort() { ...}
}
```

JUnit Execution

What Happens?

1. JUnit identifies method tagged with `@BeforeAll` and executes it.
   (No such method in the `IntListTest` example.)
2. JUnit identifies methods tagged with `@BeforeEach` and `@AfterEach` ⇒ `setUp()` and `tearDown()` in the `IntListTest` example
3. JUnit identifies all methods annotated with `@Test`
   ⇒ methods named like `testXXXX()` in the `IntListTest` example
4. For each found method `testXXXX()` annotated with `@Test` it executes
   - 4.1 `@BeforeEach setUp()`
   - 4.2 `@Test testXXXX()`
   - 4.3 `@AfterEach tearDown()`
5. JUnit identifies method tagged with `@AfterAll` and executes it.
   (No such method in the `IntListTest` example.)

Methods annotated with `@Before...` and `@After...` are used to prepare (and restore) data used in the various tests.

Example: IntListTest.testAddGet()

```
@Test
public void testAddGet() {
    /* Test add(int n) used in build() */
    IntList list1 = build(5); // list1 = [0,1,2,3,4]
    assertEquals(5,list1.size());

    IntList list2 = build(10);  // [0,1,2,3,4,5,6,7,8,9]
    assertEquals(10,list2.size());

    /* Test get(int n) */
    for (int i=0;i<list2.size();i++)
        assertEquals(i,list2.get(i));

    /* Test if get(int index) throws exception */
    assertThrows(IndexOutOfBoundsException.class, () -> list2.get(-8));
    assertThrows(IndexOutOfBoundsException.class, () -> list2.get(123));
}
```

Example: Generating Test Data

Constructing help methods to generate test data saves time in the long run.

```
private IntList build(int size) {
    IntList list = new IntListImpl();
    for (int i=0;i<size;i++)
        list.add(i);            // [0,1,2,3,4, ... ]
    return list;
}

private IntList random(int size,int max) {
    Random rand = new Random();

    IntList list = new IntListImpl();
    for (int i=0;i<size;i++) {       // Add ‘‘size’’ random numbers to list
        int n = 1 + rand.nextInt(max)    // int in range [1,max]
        list.add(n);
    }
    return list;
}
```

JUnit Suggestions (Isolated Components)

- Implement a method, write a test
- Each test method tests 1-2 methods
- No exhaustive tests:
  - Test a few standard cases
  - Focus on extreme cases
  - Add more tests on demands (when bugs appear)
- Create a few help methods to generate fixtures.
  (e.g. like `build()` and `random()` in our example.)

JUnit - Getting Started

Suggested Approach

- JUnit 5 is installed in newer version of Eclipse (Oxygen or later)
- Create a new JUnit 5 test case:
  `File --> New --> JUnit Test Case`
  Then select `New JUnit Jupiter test`
- Download my JUnit Examples and test them
- Study and understand how they work
- Start working on the JUnit exercises

JUnit 5 Reading

- Plenty of information at `junit.org/junit5`
- User Guide at `junit.org/junit5/docs/current/user-guide/`
- API Documentation at `http://junit.org/junit5/docs/current/api/`
- Google on *JUnit 5 + Tutorial*

Make sure to search for JUnit 5 (which is quite different from JUnit 4).

10 minute break?

zzzzzzzzzzzzzzzzzzzzzzzzzz

Generic Classes – Introduction

- **Generics:** Classes with type parameters
  - Instead of implementing `IntList`, a list class for integers only ...
  - ... we implement `List<T>` and instantiate it as `List<Integer>` (or as `List<String>`, `List<Double>`, `List<Student>`, `...`)
  - $\Rightarrow$ One class definition can generate many different types of objects.
- A generic class is instantiated by providing a concrete type as a parameter to the constructor call.
- It might be more than one type parameter.
- The Java Library has a number of generic data structures
  - `ArrayList<T>`, `LinkedList<T>`, `Stack<T>`, `...`
  - `TreeSet<T>`, `HashSet<T>`, `HashMap<K,V>` (topic of Lecture 8)

Using Generics `GenericMain.java`

```
List<Integer> list = new ArrayList<Integer>();  // A generic integer list
for (int i=1; i<=5; i++)
    list.add(i);                                // add 1,2,3,4,5. Converted to Integer

for (int i=0; i<list.size(); i++) {
    int n = list.get(i);                        // Integer --> int
    System.out.print(" "+n);
}
System.out.println("\n");                       // line break

List raw = new ArrayList();                     // A "raw" integer list
for (int i=1; i<=5; i++)
    raw.add(i);                                 // add 1,2,3,4,5. Converted to Integer

for (int i=0; i<raw.size(); i++) {
    int n = (Integer) raw.get(i);               // Object --> Integer --> int
    System.out.print(" "+n);
}
```

Why Generics?

Why generics, why not use `Object` as a parameter?

- Use `Object` ⇒ repeated down casts ⇒ type checking at run-time
- Generic classes ⇒ static type checking ⇒ errors found at compile time

Using generic classes is simple ⇒ **Use them!**

Implementing Generics

- Implementing generic classes is a bit more tricky

```
public class A<T> {
   public void a(T t) { "do something with t"}
   public T b(T t) { "return something of type T"}
}
```

- Remember: It must work on every possible type of object.

A Simple Example - GenValue.java

```
public class GenValue<T> {
    private T value;

    public GenValue(T val) { value = val; }

    public void setValue(T val) { value = val; }
    public T getValue() { return value; }

    @Override
    public String toString() { return "GV("+value.toString()+")"; }

    ... missing methods
}
```

Usage:

```
GenValue<Integer> i1 = new GenValue<Integer>(7);
i1.setValue(23);
System.out.println("Integers: "+i1.toString()+"\t"+i1.getValue());

GenValue<String> s1 = new GenValue<String>("Hello");
System.out.println("\nStrings: "+s1.toString()+"\t"+s1.getValue());
```

Example - GenValue.java (cont.)

```
public class GenValue<T> {
    private T value;

    public GenValue(T val) { value = val; }

    @Override
    public boolean equals(Object o) {  // Why not T as parameter type?
        if (o instanceof GenValue) {  // Compare with raw version
            GenValue other = (GenValue) o;
            return other.value.equals(value);
        }
        return false;
    }
}
```

Usage:

```
GenValue<String> s1 = new GenValue<String>("Hello");
GenValue<String> s2 = new GenValue<String>("World!");

String msg = s1.equals(s2) ? "equal" : "not equal";
System.out.println("They are "+msg);
```

Implementing Generic Classes

- Generic class ⇒ Class with type parameters (one or more)

```
public class GenValue<T> {
...
}
```

  E and T are often used but any upper case letter will do.

- Using generic classes: Exact type is decided when we create a new object

```
GenValue<Integer> i1 = new GenValue<Integer>(7);

GenValue<String> s1 = new GenValue<String>("Hello");
```

- Must work for all parameter types T ⇒ expect only class Object properties
  ⇒ Methods like `equals()`, `hashCode()`, `toString()`, `clone()`
- The possible types can be limited by *constraining* the type parameter
  ⇒ An increased set of methods to play around with. (Next Example)

Constraining Type Parameters

Generic class declarations often look like this

```
public class MyClass <E extends Comparable<E>>  {
     ....
}
```

- `<E extends Comparable<E> >` $\Rightarrow$ Any class E implementing Comparable
- We are decreasing the number of possible parameter types
- We are increasing the number of available methods to use

A Generic List (GenericList.java)

Non-generic interface

```
public interface IntList extends Iterable<Integer> {
   public void add(int n);           // Add integer n to list.
   public boolean contains(int n);   // true if n is in list, otherwise false.
   public int get(int index);        // Get integer at position index.
   public int size();                // Number of integers currently stored.
   public String toString();         // String of type "[ 7 56 -45 68 ... ]"
   public void addAll(Iterable<Integer> col); // Append iterable collection
                                              // of elements to the list.
}
```

Generic interface

```
public interface GenericList<T> extends Iterable<T> {
   public void add(T t);           // Add element t to list.
   public boolean contains(T t);   // Returns true if element t is in list, othe
   public T get(int index);        // Get element at position index.
   public int size();              // Number of elements currently stored.
   public String toString();       // String displaying  list content on one lin
   public void addAll(Iterable<T> col); // Append iterable collection
                                        // of elements to the list.
}
```

My Generic List (GenericArrayList.java)

```
public class GenericArrayList<T> implements GenericList<T> {
   private int size = 0;
   private T[] values;

   public GenericArrayList() {values = (T[]) new Object[8];}

   public void add(T t) {
      values[size++] = t;
      if (size == values.length) { // increase size
         resize();
   }

   private void  resize() {   // Private help method
      T[] tmp = (T[]) new Object[2*values.length];
      System.arraycopy(values,0,tmp,0,values.length);
      values = tmp;
   }
```

Method `addAll(Iterable<T> col)`

- `addAll(Iterable<T> col)` ⇒ Add elements using any collection class implementing the `Iterable` interface
- For Example: `ArrayList`, `HashSet`
- Implementation is straight forward

```
public void addAll(Iterable<T> col) {
     for (T t : col)
          add(t);      // Our own add method (with resizing)
}
```

Usage:

```
GenericList<String> strList = new GenericArrayList<String>();
ArrayList<String> sL = new ArrayList<String>();     // Implements Iterable
... add strings to ArrayList

strList.addAll(sL);         // Add ArrayList content to strList
System.out.println(strList);
```

It is sometimes very easy ...

Converting a non-generic class to a generic version is sometimes very easy.

```
// Non-generic
public boolean contains(int n) {
   for (int i=0;i<size;i++) {
      int v = values[i];
      if (v == n)
         return true;
   }
   return false;
}
```

```
// Generic
public boolean contains(T t) {
   for (int i=0;i<size;i++) {
      T v = values[i];
       if (v .equals(t))
            return true;
   }
   return false;
}
```

Suggestion

- When new to generics, start by implementing a non-generic version.

Non-Generic Iterators

```
public class ArrayIntList implements Iterable<Integer> {
    private int size = 0;
    private int [] values;

    ... many methods are dropped

    public Iterator<Integer> iterator() { // Required by Iterable interface
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Integer> {
        private int count = 0;

        public Integer next() { return values[count++]; }
        public boolean hasNext() { return count<size; }
        public void remove() {
            throw new RuntimeException("remove() is not implemented");
        }
    }
```

The Iterator (GenericList.java)

```
public class GenericArrayList<T> implements GenericList<T> {
   private int size = 0;
   private T[] values;

   public Iterator<T> iterator() {  // Implements Iterable in GenericList<T>
      return new ListIterator<T>(values);  // Create iterator of type <T>
   }                                       // using ListIterator<X> below

   private class ListIterator<X> implements Iterator<X> {
      private int count = 0;         // Notice: We are not using same type T
      private X[] elements;          // as the enclosing class GenericArrayList

      public ListIterator(X[] xElements) {elements = xElements;}

      public X next() {return elements[count++];}
      public boolean hasNext() {return count<size;}

      public void remove() {  // Marked as optional in Iterator<X>
         throw new RuntimeException("remove() is not implemented");
      }
   }
```

Generics: Readings

- Section 19.1 - 19.8 in book by Horstmann
- Download my example `GenericArrayList<T>` from Moodle
- Study and understand how generics work
- Start working on the generic exercises