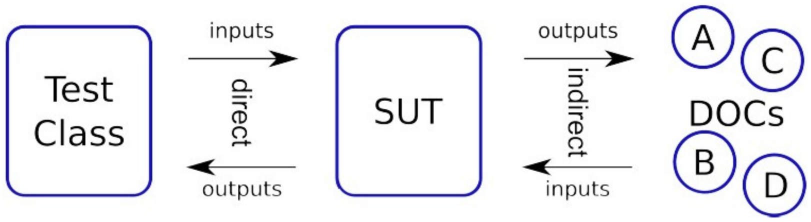
UECS2354 Software Testing
Lab 05: Introduction to Doubles

In this lab, we look at the use of interfaces to introduce polymorphic behaviour into methods in classes. Then we examine how we can use interfaces to refactor our application code using interfaces so that we can introduce test doubles such as stubs and mocks to implement our tests.

Interactions in unit tests

The class containing the application methods that we are testing is called the **system under test (SUT)** (also known as **class under test, or CUT**). So far we have looked at, the SUT usually works in isolation from under classes in the application; that is, the methods in the SUT do not call methods from other classes. This also corresponds with the goal of unit testing, which is to test methods in isolation from other methods.
However, in most practical applications, methods from one class will usually call methods from another class in order to use the returned results to accomplish their functionalities. The additional classes and methods that a SUT interacts with are sometimes called the **Depended Upon Components (DOCs)**. This is illustrated in the diagram below.



The interactions between the SUT and the test class (i.e. the class containing the JUnit test methods) are direct. This means the test class can control the inputs sent to the SUT, and can process the outputs returned from the methods invoked in the SUT.
However, the **test class is not able to control the inputs that the SUT receives from its DOCS**, nor can the test class monitor how often, or what, outputs are sent from the SUT to its DOCs. These inputs and outputs are therefore indirect, from the viewpoint of the tests being conducted.

The indirect nature of the inputs and outputs between the SUT and DOCs creates **two main issues** when it comes to testing:

- Often times, the desired functionality of the SUT will involve sending specific values in these outputs to the DOCs. As the **test class cannot capture** or monitor the outputs sent from the SUT to its DOCs**, it cannot verify** whether these outputs are correct or not.

- The inputs received by the SUT from the DOCs may influence its behaviour. Without knowledge of the values of these inputs, the **test class cannot accurately predict what kind of result** the SUT should produce and therefore cannot verify whether the SUT is functioning correctly or not.

UECS2354 Software Testing

Lab 05: Introduction to Doubles

Doubles, Mocks, Stubs and Test Spies

To deal with these issues that result from the indirect interaction between the SUTs and the DOCs, we introduce the concept of test doubles. A **test double is basically a software component** (usually a class) that is intended to **replace a class from the DOC** so that we can exert more control over the interactions between the DOC and the SUT **for the purposes of testing**.
What is normally done is to substitute the test double for the DOC when testing is being performed, and then to allow the SUT to interact with the DOC in the usual way when the application is running in production mode. The intention is that we can set up the test double in such a way that we can control its behaviour as well as monitor the interactions between it and the SUT.

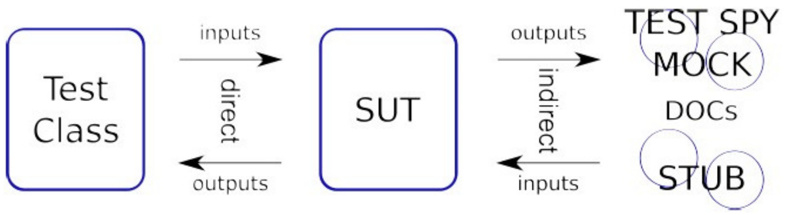
This will resolve the previous 2 issues by allowing us to:

- verify whether the SUT sends the correct output values to the test double
- control the input values sent from the test double to the SUT, and hence predict accurately the output that the SUT should produce

Test doubles can be classified into several types:

Type	Also known as	Description
Dummy object	dummy	Needs to exist, no real collaboration needed
Test stub	stub	Used for passing some values to the SUT (indirect inputs)
Test spy	spy	Used to verify if the SUT calls specific methods of the collaborator (indirect outputs)
Mock object	mock	

All these test doubles will interact with the SUT as a replacement for the DOCs when testing is being performed, illustrated as follows:



Dummies and **stubs** are used to prepare the environment for testing.

- A **dummy** is employed to be passed as a value (e.g. as a parameter of a direct method call)
- A **stub** passes some data to the SUT, substituting for one of its DOCs.

The purpose of test **spies** and **mocks** is to **verify the correctness of the communication** between the SUT and DOCs. They have a slight difference in how they are used in test code. No test double is used for direct outputs of the SUT, as this can be directly observed by the test method in test class.

Theoretically, all test doubles have well defined roles and responsibilities. However, in real-life programming, we will often encounter and probably create test code in which a test double plays more than one role. Typically, this happens with test spies and mocks, which are also frequently used in order to provide indirect inputs to the SUT, thus also playing the role of stubs.

Running the lab

STUB

1. The RandomGeneratorClass class contains a single method getRandomInteger(), which uses its parameter numLimit as a parameter to the nextInt() method of the Random object. This will cause a random number to be generated between 0 and numLimit-1 (Java, like other programming languages, produces pseudo random numbers instead of truly random numbers).

   The addTwoNumbers() method passes the value 4 to the getRandomInteger() method, which results in a random integer between 0 and 3 (inclusive) being returned. It then adds this result with its num1 parameter and returns that as its final result.

   Run this program several times, and note that the values produced for each run is different due to the random nature of num2 in addTwoNumbers().

```java
package my.edu.utar;

import java.util.Random;

class RandomGeneratorClass {

   Random randomGenerator = new Random();

   public int getRandomInteger(int numLimit) {

      int randomInt = randomGenerator.nextInt(numLimit);
      return randomInt;
   }
}

public class RandomAddNumbers {

   RandomGeneratorClass rgc = new RandomGeneratorClass();

   public int addTwoNumbers(int num1) {

      int num2 = rgc.getRandomInteger(4);
      System.out.println("Adding " + num1 + " and " + num2);
      return num1 + num2;
   }

   public static void main(String args[]) {

      RandomAddNumbers ran = new RandomAddNumbers();
      System.out.println(ran.addTwoNumbers(7));
      System.out.println(ran.addTwoNumbers(8));
      System.out.println(ran.addTwoNumbers(9));
   }
}
```

2. If addTwoNumbers() method is to be the SUT, then it involves interaction with a DOC, which will be the getRandomInteger() method from RandomGeneratorClass.

3. Repeat running `RandomAddNumbersTest` as JUnit test about 10 times. Notice that while the test fails in most runs, it also occasionally succeeds.
   The value 3 is passed to `addTwoNumbers()` within the test method, and the expected result that is tested is 5. As this value 3 is added to the random number generated in `addTwoNumbers()`, the actual result returned can be anywhere between 3 and 6. When the actual result returned is 5, the test passes; it will however fail the rest of the time.

   Test behaviour should be predictable. Having **tests that randomly pass and fail is confusing and unhelpful**, because failures in a test should ideally help to **pinpoint a bug** or fault in the application code being tested, and not result from correct but random behaviour of the code itself.

```java
package my.edu.utar;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import my.edu.utar.RandomAddNumbers;

public class RandomAddNumbersTest {

   @Test
   public void testAddTwoNumbers() {

      RandomAddNumbers ran = new RandomAddNumbers();
      int result = ran.addTwoNumbers(3);
      assertEquals(5, result);
   }
}
```

4. Study `addTwoNumbers()` carefully and we can see there are two main parts:

   - first, is the obtaining of a random number, and
   - second is the utilisation of this random number in an addition operation.

   The **main functionality of `addTwoNumbers()` is actually in the second part**, and this is therefore what we need to test. For the test to work, **it does not really matter whether `num2` is random** or not.

   What we really require is a way of making `num2` a predetermined, fixed value while we are running a test on `addTwoNumbers()`, and then allowing it to become random again when we are going to use the class in real life production.

5. The ideal solution would be to somehow allow the `getRandomInteger`() call in `RandomAddNumbers` to display different behaviour depending on whether testing was being conducted (in which it would return a predetermined, fixed integer), or whether normal operation was required (in which it would return a random integer).

   In other words, we require polymorphic behaviour from `getRandomInteger()`. This can be accomplished through the use of either interfaces or inheritance. We will use the interfaces option as the classes involved in this example are not logically related through inheritance.

```java
package my.edu.utar;
import java.util.Random;

interface RandomNumberFunctionality {
   public int getRandomInteger(int numLimit);
}

class NewRandomGeneratorClass implements RandomNumberFunctionality {
    Random randomGenerator = new Random();

   public int getRandomInteger(int numLimit) {
      int randomInt = randomGenerator.nextInt(numLimit);
      return randomInt;
   }
}

class DummyRandomNumber implements RandomNumberFunctionality {
   int valToReturn;

   public DummyRandomNumber(int valToReturn) {
      this.valToReturn = valToReturn;
   }

   public int getRandomInteger(int numLimit) {
      return valToReturn;
   }
}

public class NewRandomAddNumbers {
   RandomNumberFunctionality rnf;

   public NewRandomAddNumbers(RandomNumberFunctionality rnf) {
      this.rnf = rnf;
   }

   public int addTwoNumbers(int num1) {
      int num2 = rnf.getRandomInteger(4);
      System.out.println("Adding " + num1 + " and " + num2);
      return num1 + num2;
   }

   public static void main(String args[]) {
      RandomNumberFunctionality original = new NewRandomGeneratorClass();
      RandomNumberFunctionality dummy = new DummyRandomNumber(5);

      System.out.println ("Working with original random functionality");
      NewRandomAddNumbers nr1 = new NewRandomAddNumbers(original);
      System.out.println(nr1.addTwoNumbers(7));
      System.out.println(nr1.addTwoNumbers(8));
      System.out.println(nr1.addTwoNumbers(9));

      System.out.println ("Working with dummy functionality that returns a
   fixed number");
      NewRandomAddNumbers nr2 = new NewRandomAddNumbers(dummy);
      System.out.println(nr2.addTwoNumbers(7));
      System.out.println(nr2.addTwoNumbers(8));
      System.out.println(nr2.addTwoNumbers(9));
   }
}
```

6. `NewRandomAddNumbers.java` is a refactoring of `RandomAddNumbers()` to allow the introduction of an interface to support polymorphic behaviour.

   Refactoring essentially means rewriting code or changing its design while still maintaining identical functionality.
   The **key change** here is the introduction of the **`RandomNumberFunctionality` interface** which includes the definition of the single method `getRandomInteger()` that we require polymorphic behaviour from.

   `NewRandomGeneratorClass` then implements this interface and provides the desired random functionality as was the case in `RandomAddNumbers`. There is also a new class called `DummyRandomNumber` whose implementation of `getRandomInteger()` simply returns its instance variable `valToReturn`, whose value is set by the constructor.

7. The `NewRandomAddNumbers` class now includes an instance variable that is of the interface reference type `RandomNumberFunctionality`, and this interface reference is instantiated in the constructor. The `addTwoNumbers()` method is nearly identical to that in the original `RandomAddNumbers` class, with the **key difference being that the call to `getRandomInteger()` is invoked on an interface reference variable**, rather than on a concrete object as is the case in the original `RandomAddNumbers` class.

   In the `main()` method, we declare two objects from the classes that implement the `RandomNumberFunctionality` interface and assign them to interface reference variables. These are subsequently passed to the constructor of `nr1` and `nr2` respectively.

   The call to `addTwoNumbers()` in **`nr1`** will use the implementation of `getRandomInteger()` in `NewRandomGeneratorClass`, which **returns a random integer**. On the other hand, the call to `addTwoNumbers()` in **`nr2`** will use the implementation of `getRandomInteger()` in `DummyRandomNumber`, which returns **a fixed value of 5**.

   The passing of the object of the appropriate class to be used to the constructor of the class is sometimes known as the **dependency injection pattern**. The DOC is injected into the SUT (or class under test) through its constructor.

8. We have thus achieved our objective of polymorphic behaviour in `getRandomInteger()`. When we want `addTwoNumbers()` to behave in a random manner, we simply instantiate `NewRandomAddNumbers` with a `NewRandomGeneratorClass` object; and when we want `addTwoNumbers()` to behave in a predictable manner, we instantiate `NewRandomAddNumbers` with a `DummyRandomNumber` object.

   In fact, we can continue to create more classes that implement `RandomNumberFunctionality` with their own custom functionality for `getRandomInteger()`, and pass objects of these classes to the constructor of `NewRandomAddNumbers`. Regardless of the particular object being used, the **implementation for `addTwoNumbers()` DOES NOT need to be changed**. This is the key advantage of polymorphic behaviour.

   Run this class several times and observe the outputs to verify that the calls to `nr1` produce different and random results each time, while the calls to `nr2` produce the same result each time.

9. **DummyRandomNumber** class is a simple example of a **test double**. In RandomAddNumbers, the SUT is the addTwoNumbers() method and the DOC is getRandomInteger() from RandomGeneratorClass. We can use DummyRandomNumber to set up test for addTwoNumbers() by obtaining a fixed value from its getRandomInteger implementation, which then allows us to predict correctly the result that should be returned from the SUT. In this context, the **DummyRandomNumber** class functions as a **stub**; passing the fixed value to the SUT in place of its original DOC.

10. The first test method, testAddTwoNumbersOriginal(), in the class NewRandomAddNumbersTest.java is identical in functionality to the test method from RandomAddNumbersTest as the NewRandomGeneratorClass is passed as an object to the constructor of NewRandomNumbers.

    The second test method, testAddTwoNumbersDummy(), sets up the stub DummyRandomNumber so that calling its getRandomInteger() method will always return the value 5. Therefore the result of calling addTwoNumbers() will always be the same, and this test will always pass (or always fail); depending on what is the expected result that it is compared to.

    Run this class as a JUnit test several times to verify that the first test method demonstrates random pass/fail behaviour, while the second test method succeeds all the time.

```java
package my.edu.utar;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class NewRandomAddNumbersTest {

   @Test
   public void testAddTwoNumbersOriginal() {
      //Setting up the test
      RandomNumberFunctionality original = new NewRandomGeneratorClass();
      NewRandomAddNumbers nr1 = new NewRandomAddNumbers(original);

      // Executing the test
      int result = nr1.addTwoNumbers(3);
      assertEquals(5, result);
   }

   @Test
   public void testAddTwoNumbersDummy() {
      // Setting up the test
      RandomNumberFunctionality dummy = new DummyRandomNumber(5);
      NewRandomAddNumbers nr1 = new NewRandomAddNumbers(dummy);

      // Executing the test
      int result = nr1.addTwoNumbers(3);
      assertEquals(8, result);
   }
}
```

11. The `findLargestNumberInRandomArray()` accepts two parameters: `arrLength` and `numLimit`. It then creates an integer array of size `arrLength`, and populates this array with random integers between 0 and `numLimit – 1`. Then it iterates through this array to locate the largest number in the array and returns this number as the final result.
    The random functionality is obtained from the `getRandomInteger()` method in the `RandomGeneratorClass` that we had used earlier. Just as in the case for `addTwoNumbers()` in `RandomAddNumbers.java`, calling the method `findLargestNumberInRandomArray()` repeatedly returns a different result each time. Run `RandomFindLargest` several times to verify that this is so.

```
package my.edu.utar;

public class RandomFindLargest {

   RandomGeneratorClass rgc = new RandomGeneratorClass();

   public int findLargestNumberInRandomArray(int arrLength, int numLimit)
   {
      // first create an integer array of the desired size
      // then populate array with random integers
      int[] numArray = new int[arrLength];
      System.out.println();
      for (int i = 0; i < numArray.length; i++) {
         numArray[i] = rgc.getRandomInteger(numLimit);
         System.out.println ("numArray["+ i +"] = " + numArray[i]);
      }

      // then go through this array to find the largest number
      int bigNum = numArray[0];
      for (int i = 0; i < numArray.length; i++)
         if (bigNum < numArray[i])
               bigNum = numArray[i];
      return bigNum;
   }

   public static void main(String args[]) {

      RandomFindLargest rfl = new RandomFindLargest();

      //create an array of length 8, and fill it with random
      // integers between 0 and 20
      int result = rfl.findLargestNumberInRandomArray(8, 20);
      System.out.println("The largest number is : " + result);
   }
}
```

12. Repeat running `RandomFindLargestTest` as a JUnit test about 10 times and observe the results.

Exercise:
Refactor `RandomFindLargest` so that it includes the use of a stub class in place of `RandomGeneratorClass` (the DOC). Keep in mind that this stub class must return a sequence of predetermined numbers corresponding to the multiple calls to `getRandomInteger()` in `findLargestNumberInRandomArray`. Create a new test method to test the refactored version.

```java
package my.edu.utar;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class RandomFindLargestTest {

   @Test
   public void testFindLargestNumberInRandomArray() {

      RandomFindLargest rfl = new RandomFindLargest();

      //create an array of length 5, and fill it with random
      // integers between 0 and 8
      int result = rfl.findLargestNumberInRandomArray(5, 8);
      assertEquals(7, result);
   }
}
```

MOCK

13. In the previous two examples, we looked at the use of stubs to return values to the SUT. Test doubles can also be used as mocks to verify correct communication in the other direction, i.e. that a SUT is passing the correct output values to the DOC.

    Refer to the class `WorkingWithStrings` below, at the end of the method `checkStringLength()`, a **call is made the `doOtherStuff()`** method in `SomeOtherWork` class. In this case, the SUT is `checkStringLength()` and the DOC is `doOtherStuff()`. The method `checkStringLength()` does not return a result (it is void), and it also does not change the value of any instance variable in `WorkingWithStrings`. This means that **the only way of verifying** whether the `checkStringLength()` method works correctly or not is to **keep track of how many times `doOtherStuff()` is called** by `checkStringLength()`. Modifying the code in `doOtherStuff()` is definitely a bad idea. Thus, we need to use **mock**.

```java
package my.edu.utar;

class SomeOtherWork {

   public void doOtherStuff(String message) {
      // lots of other complex code
   }
}

public class WorkingWithStrings {

   SomeOtherWork sow = new SomeOtherWork();
   // if the length of any of the string elements in the strArray
   // is larger than strLimit, that string element is passed to the method
   // doOtherStuff from the class SomeOtherWork
   public void checkStringLength(String[] strArray, int strLimit) {
      for (int i = 0; i < strArray.length; i++)
         if (strArray[i].length() > strLimit)
            sow.doOtherStuff(strArray[i]);
   }
}
```

14. The **mock** simply **records the number of times it is called by the SUT**, and the values passed in each call. The mock can then be used by the test method to verify the correctness of the SUT.

    Study the code below. The approach is identical to how we created a stub in `NewRandomAddNumbers`. The `doOtherStuff()` is moved to a method in the interface `StuffFunctionality`; this is implemented by `NewSomeOtherWork` and `DummyWork` (the mock class).

    `DummyWork` has an `ArrayList` of type `String`; when its `doOtherStuff()` is called, the string parameter is added to this ArrayList. The `getStrList()` returns this `ArrayList` in the form of an array of `Strings`. The `getStrList()` method will return the array of Strings to verify that the method `checkStringLength()` worked correctly.

```
package my.edu.utar;

import java.util.ArrayList;

interface StuffFunctionality {
   public void doOtherStuff(String message);
}

class NewSomeOtherWork implements StuffFunctionality {

   public void doOtherStuff(String message) {
      // lots of other complex code for actual functionality
   }
}

class DummyWork implements StuffFunctionality {

   ArrayList<String> strList = new ArrayList<String>();

   public void doOtherStuff(String message) {
      strList.add(message);
   }

   // convert ArrayList of strings to an array of Strings and return it
   public String[] getStrList() {
      String[] strArrayToReturn = new String[strList.size()];
      strArrayToReturn = strList.toArray(strArrayToReturn);
      return strArrayToReturn;
   }
}

public class NewWorkingWithStrings {

   StuffFunctionality sf;

   public NewWorkingWithStrings() {
      sf = new NewSomeOtherWork();
   }

   public NewWorkingWithStrings(StuffFunctionality sf) {
      this.sf = sf;
   }
   // if the length of any of the string elements in the strArray
   // is larger than strLimit, that string element is passed to the method
   // doOtherStuff from the class SomeOtherWork

   public void checkStringLength(String[] strArray, int strLimit) {
      for (int i = 0; i < strArray.length; i++)
         if (strArray[i].length() > strLimit)
            sf.doOtherStuff(strArray[i]);
   }

   public static void main(String[] args) {

      String [] strArray = {"cat", "houses", "dog", "elephant", "rat"};

      // With this approach we have no way of knowing
      // whether the method worked

      NewWorkingWithStrings nww1 = new NewWorkingWithStrings();
      nww1.checkStringLength(strArray, 4);


      // With this approach, we can get the array of strings
      // back from DummyWork to see whether it contains all the strings
      // whose length are more than 4

      DummyWork dw = new DummyWork();
      NewWorkingWithStrings nww2 = new NewWorkingWithStrings(dw);
      //return all the strings with length more than 4
      nww2.checkStringLength(strArray, 4);

      String [] results = dw.getStrList();
      System.out.println("Results are : ");
      for (int i = 0; i < results.length; i++)
         System.out.println(results[i]);
   }
}
```

15. The `testCheckStringLength()` test method in `NewWorkingWithStringsTest`, uses a `DummyWork` object to check the result of a single call to `checkStringLength()`.
    The second test method, `testCheckStringLengthV2()`, is a parameterised test involving several arrays using the equivalence partition approach.

```java
package my.edu.utar;

import junitparams.JUnitParamsRunner;
import junitparams.Parameters;

import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;

@RunWith(JUnitParamsRunner.class)
public class NewWorkingWithStringsTest {

   @Test
   public void testCheckStringLength() {

      String [] strArray = {"cat", "houses", "dog", "elephant", "rat"};

      DummyWork dw = new DummyWork();
      NewWorkingWithStrings nww2 = new NewWorkingWithStrings(dw);
      nww2.checkStringLength(strArray, 4);

      String [] results = dw.getStrList();
      String [] expectedResults = {"houses", "elephant"};
      assertArrayEquals(expectedResults, results);
   }

   private Object[] getParamsForTestCheckStringLengthV2() {

      return new Object[] {
         new Object[] {new String[]{"cat", "houses", "dog", "elephant",
   "rat"}, 4, new String[]{"houses", "elephant"}},
         new Object[] {new String[]{"11", "Peter", "22"}, 3, new String[]
   {"Peter"}},
         new Object[] {new String[]{"11", "22"}, 10, new String[] {}}
      };
   }

   @Test
   @Parameters(method = "getParamsForTestCheckStringLengthV2")
   public void testCheckStringLengthV2(String[] strArray, int strLimit,
   String[] expectedResults) {

      DummyWork dw = new DummyWork();
      NewWorkingWithStrings nww2 = new NewWorkingWithStrings(dw);

      nww2.checkStringLength(strArray, strLimit);
      String [] results = dw.getStrList();
      assertArrayEquals(expectedResults, results);
   }
}
```

16. We have now looked at refactoring code to create stubs and mocks for testing purposes. We had to use stubs for the case of `NewRandomAddNumbers` and `RandomFindLargest` since random behaviour was involved and the use of stubs was necessary to provide fixed numbers so that the results of tests were predictable. We had to use a mock for the case of `WorkingWithStrings` because there was no other way to check whether the method worked or not.

17. In practice, a stub can be used every time a SUT obtains values from calling a method in the DOC, even if the value is not random. Also, a mock can be used every time a SUT passes a value to a DOC, even if we can test the SUT properly without using mocks (for example, if the SUT returns a result which we can use in an `AssertXXX` method). The basic motivation for this is that **using mocks and stubs help to isolate the SUT during the testing process**. This ensures that if a failure occurs in the test, we can be assured that the source of the defect or bug is in the SUT. Without the use of mocks or stubs, the test failure of a SUT could be also due to a bug in the DOCs. This will increase the amount of code that needs to be checked in the process of tracking down the source of the bug. As debugging is a tedious and time consuming activity, limiting the amount of code that needs to be checked is very useful.

UECS2354 Software Testing

Lab 05: Introduction to Doubles

Exercise:

There are two DOCs (`FileReaderClass` and `FileWriterClass`) that the SUT (`sampleMethod`) interacts with.

Refactor `SampleClass` and the two DOCs to introduce the use of a stub and mock.

Write a test method for `sampleMethod` which makes use of both the stub and mock.

```
package my.edu.utar;

class FileReaderClass {

   public int getNumberFromFile() {
      // code to open a file and read a number from it and return this number

      int numberToReturn = 10;
      return numberToReturn;
   }
}

class FileWriterClass {

   public void writeDataToFile(int dataToWrite) {
      // code to open a file and write the parameter dataToWrite
   }
}

public class SampleClass {

   FileReaderClass frc = new FileReaderClass();
   FileWriterClass fwc = new FileWriterClass();

   public void sampleMethod() {

      // Create an integer array and fill it with numbers read from file

      int[] numsFromFile = new int[5];
      for (int i = 0; i < 5; i++)
         numsFromFile[i] = frc.getNumberFromFile();


      // Do some processing on numsFromFile

      // Write the new values in numsFromFile to the file

      for (int i = 0; i < 5; i++)
         fwc.writeDataToFile(numsFromFile[i]);
   }
}
```