UECS2354 Software Testing
Lab 04: Structuring Tests

Part A
Overview

We will look at setting up a specific directory structure for application and test code in an Eclipse project. The intention is to maintain a clean separation between these two different code types but at the same time ensure that application classes are accessible to test code. Next, we look at testing based on examining the values of instance variables in the class being tested; rather than the results returned from a method. Then we see how we can test methods that are designed specifically to throw exceptions.

We look at the concept of time out testing, and how we can specify tests to skip if we wish to. Next we examine, how we can write or refactor our test classes so that duplicated code in all the test methods can be extracted out and placed in a common location. This helps in simplifying the process of writing complex tests.

We look at the alternative of using text files to read input parameter values that we use for our tests. Finally, we conclude with a discussion of the test suite as a mechanism for grouping together a number of related test classes.

Reasons for maintaining a clean separation between application code and test code.

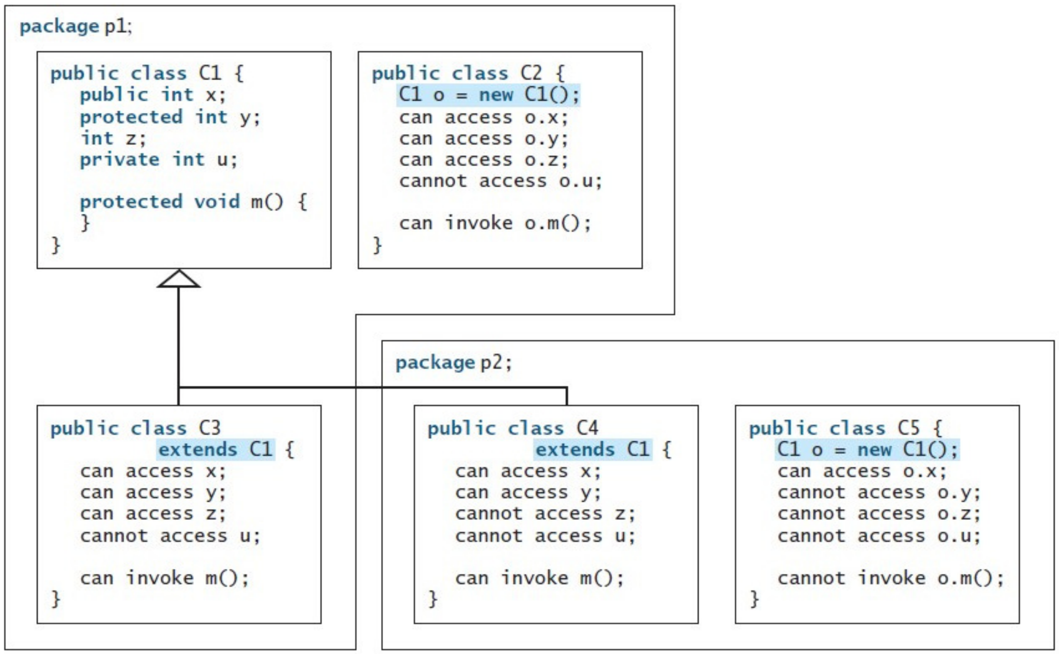
a. The number of test classes (containing the unit tests) may be as many as, or even more, than the application classes. Therefore, if both the test and application code are kept in the same directories, it will quickly become cluttered and cause confusion in the development process.

b. When we are ready to distribute the source code for our application (which is the case for open source projects), we do not usually include the tests created as part of the test driven development approach. Again, if all the test source code is included in the same directory structure as the application code, we may require a considerable amount of time to filter out and remove them.

However, keeping application and test code in different directories has one major drawback: test code may not be able to access certain classes in the package used to store application code, depending on the access modifier used on those classes. Recall that Java has 4 access (or visibility) modifiers which specify the accessibility of classes or members marked with those access modifiers.

Access Level	**From**			
	the same class	**child classes**	**classes in the same package**	**classes in other packages**
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	No	Yes	No
private	Yes	No	No	No

UECS2354 Software Testing
Lab 04: Structuring Tests



Scenario and Issues

Assume that we have a class `Student` declared without any access modifier in the `my.edu.utar.program` package, for example:

```
package my.edu.utar.program;
class Student {
// instance variables and methods for Student
}
```

When **no access modifier** is specified for a class, the **default** (or package) access modifier **applies**. This means that class `Student` is not accessible from another class in a different package from `Student`. Therefore, when we are writing code for test classes in the `my.edu.utar.test` package, we will not be able to access the class `Student` and instantiate objects from it for testing purposes, for example:

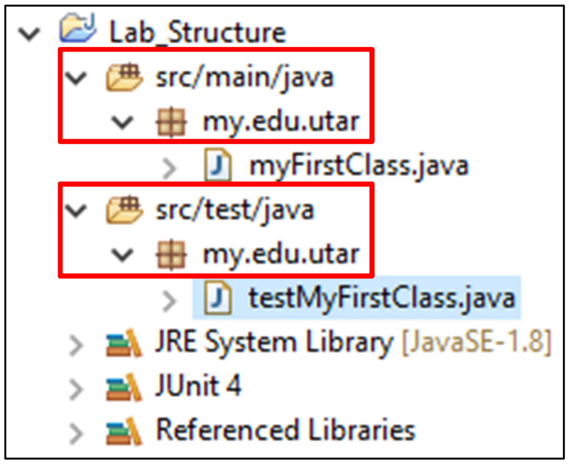
```
package my.edu.utar.test;

import my.edu.utar.program.Student;
// error not possible to access Student from a different package

public class DoSomeTests{
   @Test
   public void doSomeTestsOnStudent(){
      Student s = new Student();
      // error not possible to access Student from a different package
   }
}
```

We would need to change the access modifier for the class `Student` in order to be able to access it and instantiate objects from it for testing purposes. However, the class `Student` may have been intentionally created with the default access modifier for the sole purpose of protecting it from being accessed from other packages. This is in line with the object orientation principle of data hiding. We are therefore obliged to change the access modifier of class `Student` to public when we are compiling the code for testing purposes, and then changing it back to default when we are ready to distribute the code for final release. Such an approach is messy and complicated, and significantly increases the chances of us accidentally introducing errors into our application code.

Solution

The ideal situation therefore is to have the **application code and test code in different directories but within the same package**. This provides the desired separation between test and application code, and at the same time allows test code to access application classes.

The way to accomplish this is to have a different root folder in an Eclipse project to contain the source code for the application and test classes respectively, as shown in the figure below. We will also explicitly specify different directories to store the compiled byte code from both types of classes.



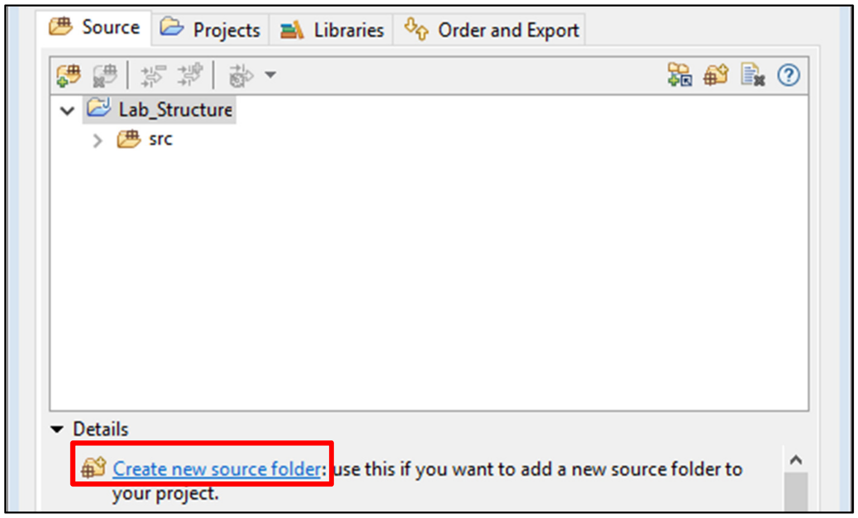
```
Lab_Structure
  src/main/java
    my.edu.utar
      myFirstClass.java
  src/test/java
    my.edu.utar
      testMyFirstClass.java
  JRE System Library [JavaSE-1.8]
  JUnit 4
  Referenced Libraries
```

Running the lab

Setting up a directory structure for application and test code

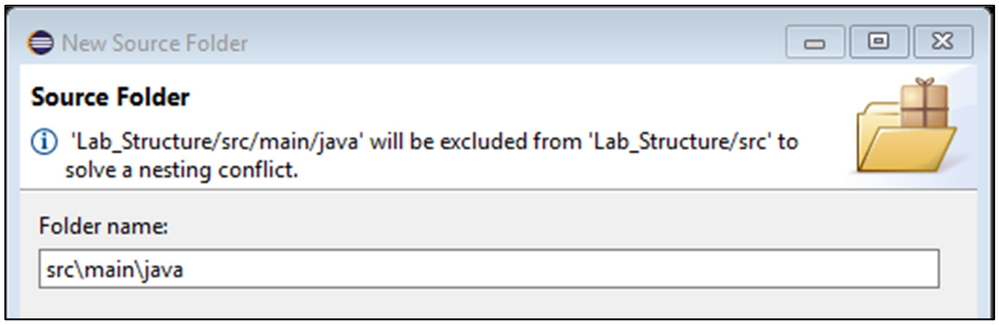
1. Create a new project in Eclipse. In the *Create a Java Project* dialog box, specify the project name and click on *Next*.

2. In the *Java Settings* dialog box with the ***Source*** tab selected, you will see the ***src*** directory specified as the default directory to store all the source code for this project. Click on the ***Create new source folder*** option in the *Details* list immediately below the list box.



Create new source folder: use this if you want to add a new source folder to your project.

3. In the *Source Folder* dialog box, specify the folder name as ***src\main\java***. Click [Finish].

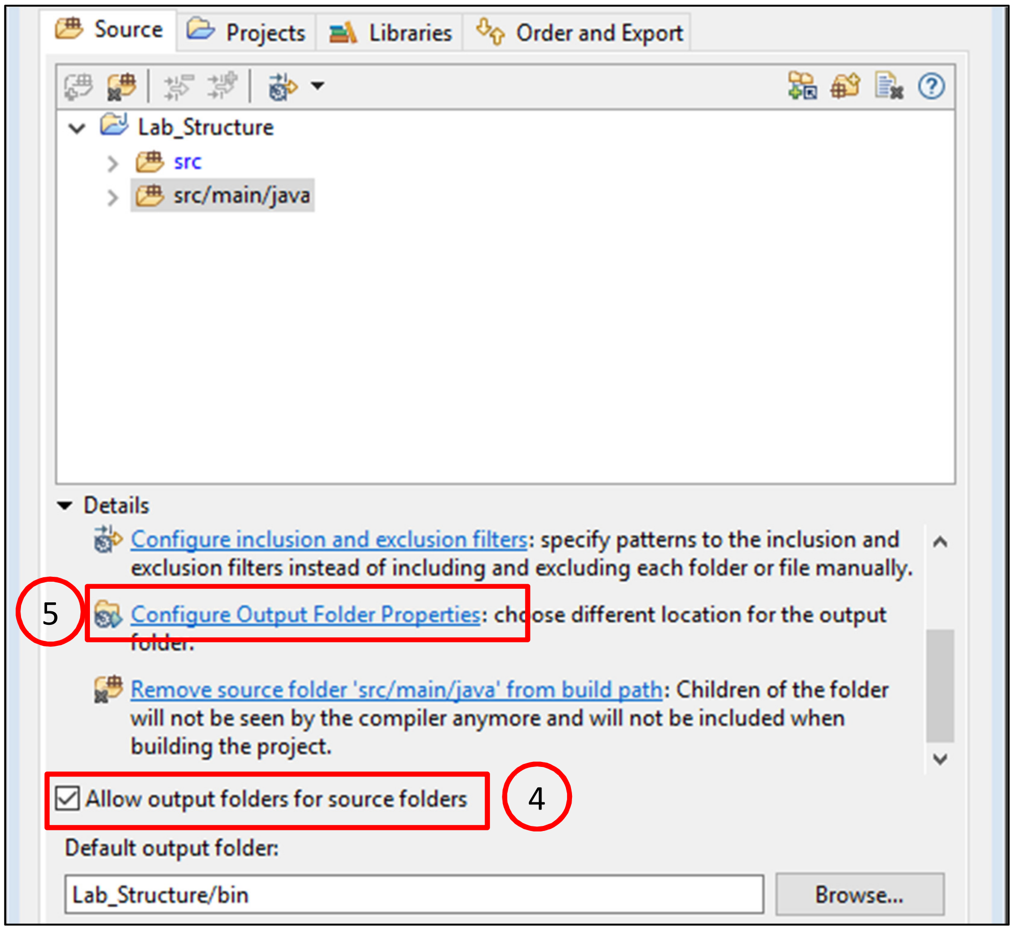


Source Folder

'Lab_Structure/src/main/java' will be excluded from 'Lab_Structure/src' to solve a nesting conflict.

Folder name: src\main\java

4. In the *Java Settings* dialog box, ensure that the ***src\main\java* folder is highlighted**. Then **check on** ☑ the box ***Allow output folders for source folders***. ④



Configure inclusion and exclusion filters: specify patterns to the inclusion and exclusion filters instead of including and excluding each folder or file manually.

Configure Output Folder Properties: choose different location for the output folder.

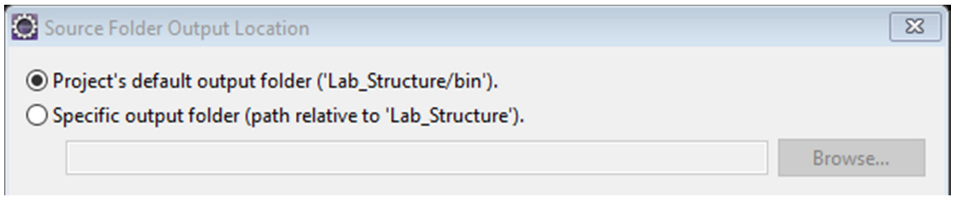
Remove source folder 'src/main/java' from build path: Children of the folder will not be seen by the compiler anymore and will not be included when building the project.

☑ Allow output folders for source folders

Default output folder:

Lab_Structure/bin

5. Click on the ***Configure Output Folder Properties*** ⑤ in the **Details** list immediately below the list box. In the *Source Folder Output Location* dialog box that appears, check the radio button for ***Project's default output folder (Lab_Structure/bin)***, then click [OK]. At this point, we have created the source folder to store application code and also specified the folder to store the compiled byte code from the application classes.



● Project's default output folder ('Lab_Structure/bin').
○ Specific output folder (path relative to 'Lab_Structure').

6. Repeat this process to create a separate source and output folder to store the test classes and its compiled byte code. In the *Java Settings* dialog box with the *Source* tab selected, click on the ***Create new source folder*** option in the *Details* list immediately below the list box.

   In the *Source Folder* dialog box, specify the folder name as ***src\test\java***. Click [Finish].
   In the *Java Settings* dialog box, ensure that the ***src\test\java*** folder **is highlighted**. Then **check** on the box ***Allow output folders for source folders***.

   Click on the *Configure Output Folder Properties* in the Details list immediately below the list box.
   In the *Source Folder Output Location* dialog box that appears, check the radio button for ***Specific output folder (path relative to 'Lab_Structure')***, and in text box that appears, type ***testclasses***. Click [Ok].

7. Back in the *Java Settings* dialog box with the *Source* tab selected, select ***src*** in the list box immediately below ***Lab_Structure***. In the **Details** list immediately below the list box, select the option ***Remove source folder 'src' from build path***. You should see ***src*** moving to the bottom below the two other directories *src\test\java* and *src\main\java* in the list box. Both these directories are shown with an open folder icon containing a small crossed rectangle (📂). This indicates that only source code files placed within these two root directories will actually be compiled.

8. Next, select the *Libraries* tab in the *Java Settings* dialog box. Add the ***JUnit*** and ***JUnitParams*** libraries. Click [Finish] when you are done.

9. Select ***src\main\java*** in the Package Explorer and create a new package named ***my.edu.utar***. Click [Finish] when you are done.

10. Create a new class in the *my.edu.utar* package named ***MyFirstClass*** and click [Finish].
    Insert the code for the Student class in *MyFirstClass.java* as shown below.

```java
package my.edu.utar;

class Student {

   public int addTwoNumbers(int a, int b) {
      return a + b;
   }
}

public class MyFirstClass {

}
```

11. Now create a test class with a test method to access the `Student` class in order to test its `addTwoNumbers()` method.

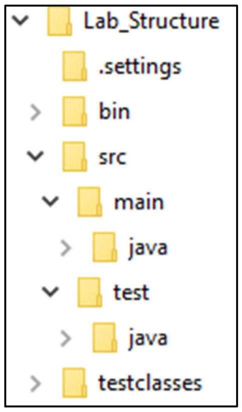
    Select *src\test\java* and create a new package named ***my.edu.utar***. Click [Finish] when you are done.

12. Create a new class in the *my.edu.utar* package that now appears below ***src\test\java***. Name the class as *MyFirstClassTest* and select [Finish].

```
package my.edu.utar;

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

public class MyFirstClassTest {

   @Test
   public void testAddTwoNumbers() {
      Student s = new Student();
      int result = s.addTwoNumbers(10, 15);
      assertEquals(25, result);
   }
}
```

13. Notice now that we no longer need to specify an `import` statement for the `Student` class since `MyFirstClassTest` is placed in the same package (`my.edu.utar`) as `MyFirstClass`.

14. Browse through the directory structure of the project in your Eclipse workspace directory to verify that its contents match the package and folder structure shown in the *Package Explorer*. Notice that the byte code for the application code is stored in the *bin* subfolder, while the byte code for the test code is stored in the *testclasses* subfolder; just as we had designated earlier.



UECS2354 Software Testing

Lab 04: Structuring Tests

Part B: Tests based on class state and testing exceptions

1. In **myFirstlass.java**, the class `Student` has two instance variables, `name` and `age`, which are both labelled with the `private` access modifier. This is in line with object orientation principles of information hiding, which state that the instance variables of a class should ideally be accessible only within that class. External access to these variables should occur through public methods of that class. These public methods are known as **accessors** (or getter methods) if they return the value of these private instance variables, or **mutators** (or setter methods) if they change the value of these private instance variables based on parameters in their method signature. For the case of the `Student` class, `getAge`() and `getName`() are accessor methods, while `setAge`(`int age`) and `setName`(`String name`) are mutator methods.

```
class Student {
   private String name;
   private int age;

   public Student(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }

   public boolean setAgeV2(int age) {
      if (age < 0 || age > 120)
         return false;
      else {
         this.age = age;
         return true;
      }
   }

   public void setAgeV3(int age) {
      if (age < 0 || age > 120)
         System.out.println ("Invalid age range");
      else {
         this.age = age;
      }
   }

   public void setAgeV4(int age) throws IllegalArgumentException {
      if (age < 0 || age > 120)
         throw new IllegalArgumentException("Invalid age range");
      else {
         this.age = age;
      }
   }
}
```

2. Notice that the `setAge`() method does not return a value. Up to this point in time, we have studied testing methods that return values, where the returned value is used to verify that the method has accomplished its required functionality. For the case of `setAge`(), the functionality of the mutator is to set the instance variable `age` to the value contained in the parameter `age`. Thus the state of the object (which is given by the values of its instance variables) is changed as result of invoking `setAge`(). To verify that `setAge`() functions properly, we will need to check the value of the instance variable `age` after it is invoked.

3. In the `testSetAge`() method, we first instantiate an object from the `Student` class using a constructor that sets the value of `age` to 25. Next, call the `setAge`() method on this object to change `age` to a different value (30). Then, to verify that `age` has been changed correctly, we retrieve it using the `getAge`() accessor in the `assertsEqual()` method call. Run as a JUnit test to verify that the tests pass.

```
package my.edu.utar;

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

import java.util.ArrayList;

import junitparams.JUnitParamsRunner;
import junitparams.Parameters;

@RunWith(JUnitParamsRunner.class)
public class DemoExceptionsTest {

   @Test
   public void testSetAge() {
      Student s = new Student("Peter", 25);
      s.setAge(30);
      assertEquals(30, s.getAge());
   }

   @Test
   @Parameters({"Peter, 25, 30, true","Peter, 25, -5, false", "Peter, 25,
   180, false"})
   public void testSetAgeV2(String name, int age, int change, boolean
   expectedResult) {
      Student s = new Student(name, age);
      boolean result = s.setAgeV2(change);
      if (expectedResult) {
         assertEquals(change, s.getAge());
         assertTrue(result);
      }
      else {
         assertEquals(age, s.getAge());
         assertFalse(result);
      }
   }
```

```
    @Test
    @Parameters({"Peter, 25, 30", "Peter, 25, 70"})
    public void testSetAgeV4Valid(String name, int age, int change) {
       Student s = new Student(name, age);
       s.setAgeV4(change);
       assertEquals(change, s.getAge());
    }

    @Test(expected=IllegalArgumentException.class)
    @Parameters({"Peter, 25, -5", "Peter, 25, 180"})
    public void testSetAgeV4Invalid(String name, int age, int change) {
       Student s = new Student(name, age);
       s.setAgeV4(change);
    }

    @Test(timeout=100)
    @Parameters({"10000", "50000", "100000"})
    public void testCreateMultipleObjects(int numObjects) {

       ArrayList<Student> sList = new ArrayList<Student>();
       for (int i = 0; i < numObjects; i++) {
          sList.add(new Student("Peter",25));
       }
    }
}
```

4. The 2nd version of the mutator, `setAgeV2(int age)`, does a range check on the parameter that it receives to decide whether or not to store it in `age`. This is a form of input validation, which is a very important activity that should be performed in all applications; since invalid user input is the primary cause of failures or security flaws in programs. We want to ensure that there are no invalid values for the age of a typical student (for e.g. negative values, zero or an impossibly high age). This range is context-dependent, so we could easily have used another range such as 10 to 70. However, regardless of what we consider to be a valid range for a student age, this validity check should be performed in the mutator for this class. In this example, the method additionally returns a false if the parameter it receives is out of range, or true otherwise. Thus it returns a result as well as changing the value of an instance variable.

5. The 2nd test method (`testSetAgeV2`) is a parameterised test with 3 combinations of input parameter values. The parameters in the signature of a parameterised test method can be roughly classified into 3 categories based on the intended usage of these values:

   - values that are used to instantiate an object from the class containing the method to be tested
   - values that are passed as parameters to the method being tested
   - values that are used in the `assertXXX` comparisons

   In this case, the `Student` object needs to be instantiated with a constructor that receives two arguments which will be used to initialise its instance variables. The first two parameters in `testSetAgeV2()` are used to receive these values. The next parameter `change` is used to accept the value that will be subsequently passed to the method being tested (`setAgeV2`). Finally, the last parameter `expectedResult` represents the expected result from calling the method being tested.

6. The first combination of input parameters checks for an attempt to change the `age` instance variable to a valid value (30), while the next two checks for attempts to change the `age` instance variable to invalid values (-5 and 180).

   For the case when `expectedResult` is true, it means that we expect `setAgeV2` to confirm the new value (as given by the parameter `change`) as valid and set `age` to it. Therefore, we check that `age` is now having this new value, and that the result returned is true. Alternatively, when `expectedResult` is false, it means that we expect `setAgeV2` to reject the new value (as given by the parameter `change`). Therefore, we check that the instance variable `age` still retains its original value (as given by the parameter `age`) when the object was first instantiated, and that the result returned is false.

   Please read this explanation carefully to ensure you clearly understand why the test method is structured in this manner. Remember that we are testing BOTH the result returned from the method AS WELL as the change (if any) that is affected on the instance variables of the object.

7. You may also have noticed that the constructor of `Student` which provides the initial value for the `age` instance variable should also incorporate a range check. We have omitted this check here to keep the example slightly simpler.

8. The 3rd version of the mutator, `setAgeV3(int age)`, does not return any kind of value in order to indicate the success or otherwise of its validity range check. Instead, it uses `System.out.println` to display an error message to the console when the range check fails.

   This approach is useless except in the simplest cases, as it is very easy for a developer to miss this output message when executing code that uses this method. A developer may accidentally write code that calls this method with an invalid argument and assume that the value of the instance variable age has being changed (when in fact, it hasn't). Later on, any other code that the developer writes based on this assumption will not function correctly, and it may be difficult to trace the source of the error.

9. To avoid the occurrence of this situation, the recommended approach to handling invalid parameter values is for the method to throw an **`Exception`**. Code must be explicitly written to handle certain types of exceptions. For example, exceptions of class types that derive from the `Exception` class (http://docs.oracle.com/javase/7/docs/api/index.html?java/lang/Exception.html) in the Java standard API (excluding `RuntimeException` subclasses) are known as checked exceptions, as the compiler will explicitly check that code is written to handle them.

   This involves creating a `try-catch` block to handle the thrown exception, with the method that can throw the exception being placed in the `try` block, and the code that handles any thrown exceptions being placed in the `catch` block. If an exception is actually thrown but is not caught and handled in a `try-catch` block, the application will prematurely terminate. Regardless of whether a `try-catch` block is created, or whether an application crashes prematurely, the end result is that the developer will immediately be alerted to the invalid parameter value.

10. Run `DemoExceptions.java` as a **Java application**. You will see in the console view that the program terminates with a stack trace shown at the point in the code where the called method threw an exception. All remaining statements in the program are not executed.

```
package my.edu.utar;

public class DemoExceptions {

   public static void main(String[] args) {

      Student s = new Student("Muthu", 20);

      System.out.println ("Calling setAgeV4 with an invalid parameter
   value");
      s.setAgeV4(150);

      System.out.println ("Calling setAgeV4 again with an invalid parameter
   value");
      try {
         s.setAgeV4(150);
      }
      catch (IllegalArgumentException iae) {
         System.out.println ("The code to handle the case of exception");
         s.setAgeV4(25);
      }
   }
}
```

11. Comment out the first **`s.setAgeV4(150);`** (without the try-catch block), save and run `DemoExceptions` again. This time the exception thrown from the method call using an invalid parameter is caught and handled in the `catch` block. The premise is that the code in the `catch` block will perform some appropriate action to handle that particular exception.

    In this example, since we know that the exception only occurs when an attempt is made to use an invalid parameter in the call to `setAgeV4`, one possible solution is to call `setAgeV4` with some value that we know is definitely valid.

    However, this is not the only possible solution, we could also just simply print a stack trace (`iae.printStackTrace();`) and exit the method. The point again to be made here is that regardless of whether a `try-catch` block handles an exception properly; or whether an application crashes prematurely because the exception is not handled, the desired end result is that the developer will immediately be alerted to the possibility of an invalid parameter value.

12. The next thing to consider is how to write tests to check a method that is implemented to throw an exception whenever it receives invalid input parameters. Returning to **`DemoExceptionsTest.java`**, the first test method for `setAgeV4`() (`testSetAgeV4Valid`) is a parameterised test involving 2 combinations of valid parameter values for the call to `setAgeV4`().

    The second test method for `setAgeV4`() (`testSetAgeV4Invalid`) is also a parameterised test, but involving 2 combinations of invalid input parameter values instead. The `@Test` annotation includes the statement (`expected=IllegalArgumentException.class`). This indicates that the test method will expect an exception of this class (or from a derived class) to be thrown when the method is executed. The `IllegalArgumentException` is derived from `RuntimeException` so we could also write (`expected=RuntimeException.class`) as well (http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html)

If this expected exception is not thrown, the test method will fail. Run `DemoExceptionsTest` as a JUnit test again to check that all the test pass.

Then change one of the invalid parameters (either -5 or 180) passed to `testSetAgeV4Invalid` to a valid value (between 0 and 120). Run `DemoExceptionsTest` and confirm that this test method fails.

13. Note that in the `Student` class, we label the different mutators for the age instance variable with postfixes such as `V2`, `V3`, etc. for illustration purposes only. In practice, we will usually only have one mutator for each instance variable in a class. If there are more than one mutator, than it will be an overloaded method (i.e. a method with the same name, but different parameter list). It is bad practice to use postfixes such as `V2`, `V3`, etc. to distinguish between different methods that accomplish nearly identical functionality in real-life production code.

Part C: Testing time outs and skipping tests

1. So far, we have looked at testing functional aspects of application code; which essentially means verifying that the code performs the tasks that it was designed to accomplish. However, non-functional aspects of an application such as usability, response time, security and so on are equally important to verify as well. For example, we may wish to verify that an application is actually secure to use in an online environment or that it responds to user input requests in an acceptable time frame. Response time is one of the key non-functional characteristics of code that is often tested; particularly in application areas like e-commerce or real time systems.

2. JUnit provides a way to measure response time of a particular method.

```
@Test(timeout=100)
@Parameters({"10000", "50000", "100000"})
public void testCreateMultipleObjects(int numObjects) {

   ArrayList<Student> sList = new ArrayList<Student>();
   for (int i = 0; i < numObjects; i++) {
         sList.add(new Student("Peter",25));
   }
}
```

   `@Test` annotation specifies `timeout=x`, where **x is the time in milliseconds**. This indicates that the test method will fail if it takes longer than x milliseconds to complete execution. We pass this parameterised test different values for `numObjects`, which is the total number of new `Student` objects that are instantiated and added to `ArrayList`. The time that the method takes to complete increases proportionally with the number of `Student` objects that are created.

   Currently, the timeout value is set to 100; which assures the test method will pass for all runs. Change this value to 1 and run it again. It is likely that one or more of the test runs with a higher value for the parameter `numObjects` will fail. Experiment with different values for the `timeout` and `numObject` parameters to cause a test run to fail; these values will differ depending on the computer system that you run this test on. Note that the JUnit view includes the time taken for each run of the parameterised `testCreateMultipleObjects()` method, so we will be able to verify which run has caused the test method to fail.

3. Sometimes, we may wish to skip running certain tests, particularly when we have a lot of tests to run and some of these tests take a long time to complete. The simplest way to do this is to comment out the test methods that we want to skip.

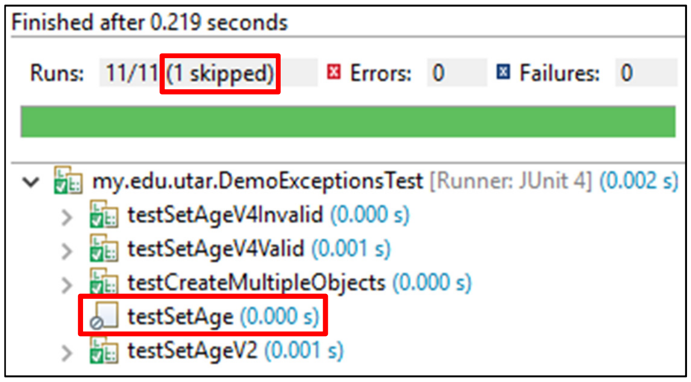
   The problem with this approach is that when they are commented out, the test method names no longer appear in the JUnit view when the test class is run. As a result, there is a very strong chance that we may forget these tests exist and fail to uncomment them to make them active again when we need to run all the tests.

   A better approach is to mark the tests as inactive so that JUnit does not run them, but still provides an indicator of these inactive tests in the JUnit view. This way we are always provided with a visual reminder that we have skipped some tests, and will not forget to include these tests in again when they are required.

To do this, add the `@Ignore` annotation, with optional message as a parameter, to the top of the `@Test` annotation of test method we wish to skip. For example:

```
@Ignore("Temporarily skipped this test")
@Test
public void testSetAge() {
   Student s = new Student("Peter", 25);
   s.setAge(30);
   assertEquals(30, s.getAge());
}
```

Run the test again. Notice in the JUnit view, the number of tests skipped is shown and if we expand the view for all the tests, the methods that were skipped are marked with a different icon from the rest of the tests. Remove the `@Ignore` annotation to make the tests active again.



```
Finished after 0.219 seconds
Runs: 11/11 (1 skipped)    Errors: 0    Failures: 0

my.edu.utar.DemoExceptionsTest [Runner: JUnit 4] (0.002 s)
  testSetAgeV4Invalid (0.000 s)
  testSetAgeV4Valid (0.001 s)
  testCreateMultipleObjects (0.000 s)
  testSetAge (0.000 s)
  testSetAgeV2 (0.001 s)
```

Part D: Setting up test fixtures

1. The term **test fixture** refers to the concept of a well understood and fixed environment in which tests are run so that the test results are repeatable. The steps required to set up such an environment differ, depending on the types of test and tool used. For a unit test, test fixtures usually include the instantiation of objects from the classes to be tested, and the creation of the input data to be supplied as parameters to the methods in these classes. For unit tests, setting up the test fixture is reasonably straightforward but for integration and system testing, the setup is more complicated because it involves the installation of additional software and creation of database tables.

2. The test class `SimpleClassTestV1` provides a series of test methods to test the 3 methods in the application class `SimpleClass`. Notice that the statement for the instantiation of the `SimpleClass` object is identical and **duplicated in all 3 test methods**.

```
public class SimpleClassTestV1 {

   @Test
   public void testGetSumNumbers() {
      SimpleClass sc = new SimpleClass(5,3);
      sc.initializeArray();
      // add 3, 4, 5, 6, 7
      int result = sc.getSumNumbers();
      assertEquals(25, result);
   }

   @Test
   public void testFindLargestNumber() {
      SimpleClass sc = new SimpleClass(5,3);
      sc.initializeArray();
      // find largest number in 3, 4, 5, 6, 7
      int result = sc.findLargestNumber();
      assertEquals(7, result);
   }

   @Test
   public void testFindSmallestNumber() {
      SimpleClass sc = new SimpleClass(5,3);
      sc.initializeArray();
      // find largest number in 3, 4, 5, 6, 7
      int result = sc.findSmallestNumber();
      assertEquals(3, result);
   }
}
```

We can simplify the test fixture by making this an instance variable of the test class and instantiating it when we declare it. This is shown in test class `SimpleClassTestV2`.

JUnit creates a new instance of a test class before executing any test method marked with the `@Test` annotation. This means that the instance variable `sc` will be created anew before the execution of any test method:
`testGetSumNumbers()`, `testFindLargestNumber()`, or `testFindSmallestNumber()`

```
public class SimpleClassTestV2 {
   SimpleClass sc = new SimpleClass(5,3);

   @Test
   public void testGetSumNumbers() {
      sc.initializeArray();
      int result = sc.getSumNumbers();
      assertEquals(25, result);
   }

   @Test
   public void testFindLargestNumber() {
      sc.initializeArray();
      int result = sc.findLargestNumber();
      assertEquals(7, result);
   }

   @Test
   public void testFindSmallestNumber() {
      sc.initializeArray();
      int result = sc.findSmallestNumber();
      assertEquals(3, result);
   }
}
```

3. However, there are still duplications of call to `initializeArray()` in all the 3 test methods. We cannot place this statement in the class body after the declaration of `sc`, since the class body can only contain declaration and initialization statements.
Therefore, we need a special method that will be called before every test method is executed. The `sc.initializeArray()` statement can then be removed from all test methods to minimise code duplication. JUnit provides a method with such capability, and marks it with the **`@Before`** annotation, as in the class `SimpleClassTestV3`.

```
public class SimpleClassTestV3 {
   SimpleClass sc = new SimpleClass(5,3);

   @Before
   public void setup() {
      sc.initializeArray();
   }

   @Test
   public void testGetSumNumbers() {
      int result = sc.getSumNumbers();
      assertEquals(25, result);
   }

   @Test
   public void testFindLargestNumber() {
      int result = sc.findLargestNumber();
      assertEquals(7, result);
   }

   @Test
   public void testFindSmallestNumber() {
      int result = sc.findSmallestNumber();
      assertEquals(3, result);
   }
}
```

4. To facilitate the extraction of common code used in setting up tests as well as releasing system resources (such as closing files that were opened) when tests are completed, JUnit provides a variety of annotations similar to `@Before`.

   The **`@Before`** and **`@After`** annotated methods are executed right before or right after the execution of each one of the `@Test` methods and regardless of whether the test passes or not. We can have as many of these methods as we want, but if we have more than one of them, the order of their execution is not defined.

   JUnit also provides the **`@BeforeClass`** and **`@AfterClass`** annotations. Methods marked with these annotations are only executed once; the `@BeforeClass` is the first to execute before any other method, and the `@AfterClass` method is run after all other methods have completed execution. All these annotated methods must be public. In addition, the `@BeforeClass/@AfterClass` annotated methods must be **public** and **static**.

5. `DemoSetupTest` uses all these methods. `System.out.println` statements are inserted in all the annotated methods to trace the execution sequence. Run and confirm that the outputs follow the sequence described in the previous paragraph.

```
public class DemoSetupTest {

   static int x = 0;
   int y = 0;

   @BeforeClass
   public static void setupClass() {
      x = 5;
      System.out.println ("setup class");
   }

   @AfterClass
   public static void endClass() {
      System.out.println ("end class");
   }

   @Before
   public void setupMethod() {
      y += 10;
      System.out.println ("setup method. x = " + x + " y = " + y);
   }

   @After
   public void endMethod() {
      y += 20;
      System.out.println ("end method. x = " + x + " y = " + y);
   }

   @Test
   public void thirdTest() {
      System.out.println ("third Test. x = " + x + " y = " + y);
      x += 1;
   }

   @Test
   public void secondTest() {
      System.out.println ("second Test. x = " + x + " y = " + y);
      x += 1;
   }
   @Test
   public void firstTest() {
      System.out.println ("first Test. x = " + x + " y = " + y);
      x += 1;
   }
}
```

6. In the test class `DemoSetupTest`, **x is a static** (or class) variable, which means that there is one single copy of `x` that belongs to the class and is **shared by all objects** instantiated from the class. **y is an instance** (or non-static) variable, which means that **each object from the class has its own copy** of `y`.

   As JUnit creates a new instance of a test class before executing any test method, the value of `y` is always reset back to 10 when the `setupMethod()` is called after a new instance of the class is called. On the other hand, since there is only one copy of `x` shared among all the instances created, any change to `x` (from the `x += 1` in each test method) is persisted among all the instances.

   Note that an instance variable can only be accessed from within an instance (or non-static) method. Therefore, attempting to access the variable y from within the static `setupClass`() and `endClass`() methods will result in a compile time error. On the other hand, static variables can be accessed in both static and instance methods.

Part E: Reading test values from an input file

1. Up to this point of time, we have been specifying all the input parameter values that use for test methods directly in the code itself. Hardcoding the values in this way provides a quick and simple way to create the test.

   However, if these values need to be changed frequently, the need to modify the source code of the test and recompile may become unnecessarily tedious. In such a situation, it might be helpful to have all the test values stored in an external text file and read in by the test class at the point when the tests are to be run.

   This allows us to change the values without needing to access the test source code. Having the test values stored in an external text file also provides a central location from where we can get a quick overview of all these values; as opposed to the situation where they are distributed throughout the source code.

2. The test class `TextFileInputDemo` provides a simple example of reading lines of text from an input file and storing it into an internal data structure.

   We use an `ArrayList` whose individual elements are String arrays for this purpose. The file used for this example is specified as `values.txt`. The project folder is the default location for files specified without an explicit directory path in a Java Eclipse project. It consists of a series of lines, where each line contains several strings or numbers separated by spaces.

   The operation to open the file, `inputStream = new Scanner(new File(fileName));`, is nested within a `try-catch` block, as the failure of this operation will automatically throw an `Exception`.

The `while` loop reads lines from the input file one at a time, breaks each line into an array of Strings (using the space between the strings as a delimiter), and then stores this array as an element into the `ArrayList`.

The `for` loop iterates through the `ArrayList`; and for each element of the `ArrayList` (which is a String array), it prints out each element within the array. We can experiment with this program by changing the values in `values.txt`.

```java
public class TextFileInputDemo {

   public static void main(String[] args) {

      ArrayList<String[]> linesRead = new ArrayList<String[]>();
      String fileName = "values.txt";
      Scanner inputStream = null;
      System.out.println("The file " + fileName +
      "\ncontains the following lines:\n");

      try {
         inputStream = new Scanner(new File(fileName));
      }

      catch(FileNotFoundException e)
      {
         System.out.println("Error opening the file " + fileName);
         System.exit(0);
      }

      while (inputStream.hasNextLine())
      {
         String singleLine = inputStream.nextLine();
         String[] tokens = singleLine.split(" ");
         linesRead.add(tokens);
      }

      int lineNum = 1;
      for (String[] strArray : linesRead) {
         System.out.print("Line " + lineNum + " : ");
         for (int i = 0; i < strArray.length; i++)
            System.out.print(strArray[i]+ " ");
         lineNum++;
         System.out.println();
      }

      inputStream.close();
   }
}
```

UECS2354 Software Testing

Lab 04: Structuring Tests

Exercise

The class `BasicClass` has two methods, `addStringsToList()` and `getTotalStringLength()` whose functionality is specified in comments, DO NOT use parameterised tests.

Write some tests for these methods where the parameter values used for testing are read from a text file. You can reuse the code from `TextFileInputDemo`.

```
package my.edu.utar;

import java.util.ArrayList;

public class BasicClass {

   private ArrayList<String> strList;
   private int strLimit = 0;

   public void initializeList() {
      strList = new ArrayList<String>();
   }

   public void setStrLimit(int strLimit) {
      this.strLimit = strLimit;
   }

   // Return strList as an array of Strings
   public String[] getArrayList() {
      String [] returnStr = new String[strList.size()];
      returnStr = strList.toArray(returnStr);
      return returnStr;
   }

   // Takes an array strToAdd, and for every element in this array whose
   // length is more than strLimit, add that element to strList

   public void addStringsToList(String[] strToAdd) {
      for (int i = 0; i < strToAdd.length; i++) {
         if (strToAdd[i].length() > strLimit)
            strList.add(strToAdd[i]);
      }
   }

   // returns the total length of all the strings contained in strList

   public int getTotalStringLength() {
      int totalCount = 0;
      for (String str : strList)
         totalCount += str.length();
      return totalCount;
   }
}
```

To help you towards this end, structure your test using the `@BeforeClass`, `@Before` and `@AfterClass` methods. Since the values from the input file only needs to be read once, the code that does this can be placed in the `@BeforeClass` method. The `@Before` can contain method calls that you need to initialise the `BasicClass` object before using it in the tests.

For this exercise, DO NOT use parameterised tests. Write the tests as simply as possible to ensure that you understand clearly what is involved in writing tests in this way.

When you are done, create a second version of your test using parameterised tests. Note that parameterised tests DO NOT work well with `@BeforeClass`, `@Before` and `@AfterClass` annotations, as the JUnit runner runs the parameterised tests before any of these methods are called.

Any code that needs to run before the parameterised tests start (for example, code that reads and stores test values from the input text file, or code that initialises the `BasicClass` objects that will be used in the test) should be placed somewhere else. Two possible options are to place them in a static block or to place them in the constructor for the test class. A `static` block is conventionally used in Java to initialise all the static variables of a class, and will thus run before any object is instantiated from the class. The constructor will run every time JUnit instantiates an object from the test class, which it will do before it calls any test method in the class.

Part F: Combining test classes using a Test Suite

1. So far we have seen the creation of test classes containing one or more test methods within them. We can then run these test classes as a JUnit test. This is fine for a small number of test classes, but as the size of the application grows, the number of test classes will grow correspondingly as well. Manually selecting each test class to run when there are dozens or hundreds of test classes is slow, tedious and error-prone (we may easily skip one test class).

2. A more logical approach is to combine related test classes into a larger group, which we can then select and run as a single unit. JUnit provides a structure for this purpose called the **test suite**, which is a container used to group related tests together so that they can all be run together.

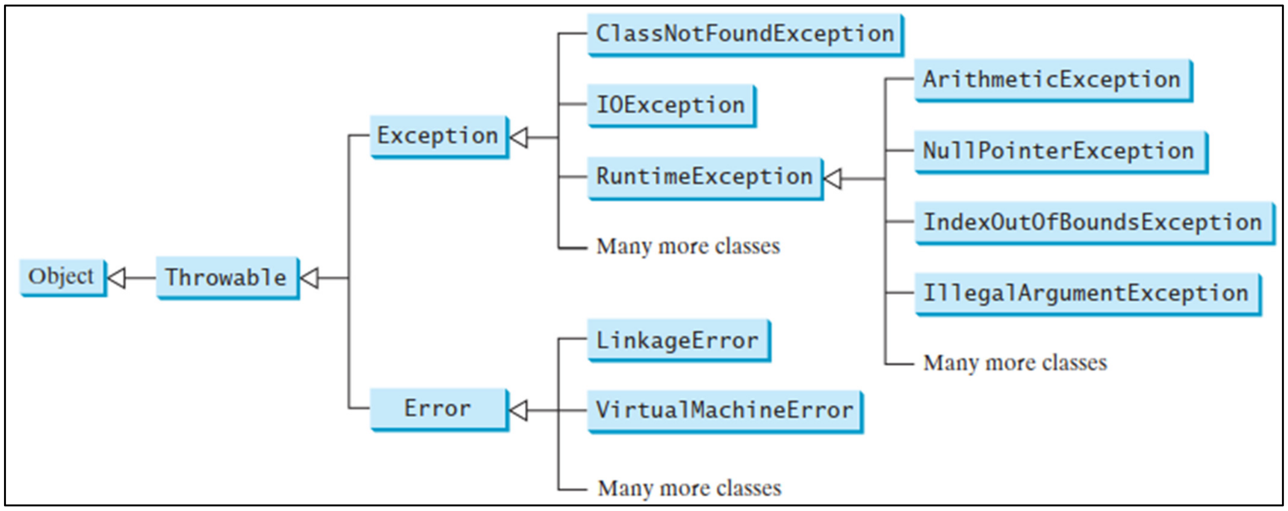
3. The class in the following example is a Test Suite which identifies all the other test classes that are grouped within this suite in the **`@SuiteClasses`** annotation. As we can see, these are classes that we have been working with. Right click `DemoTestSuite` in the *Package Explorer*, and select *Run as JUnit test*. Expand the `DemoTestSuite` icon in the JUnit view to verify that all the test methods in all the specified test classes have being run.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(value = Suite.class)
@SuiteClasses(value = {SimpleClassTestV1.class,
   SimpleClassTestV2.class,
   SimpleClassTestV3.class,
   DemoExceptionsTest.class,
   DemoSetupTest.class
})
public class DemoTestSuite {

}
```

4. To build further upon this grouping facility, we can create Test Suites which are themselves composed of other Test Suites. Simply list the names of these Test Suite classes in the `@SuiteClasses` annotation. This is left as an exercise to work with.

UECS2354 Software Testing
Lab 04: Structuring Tests

Java Exception

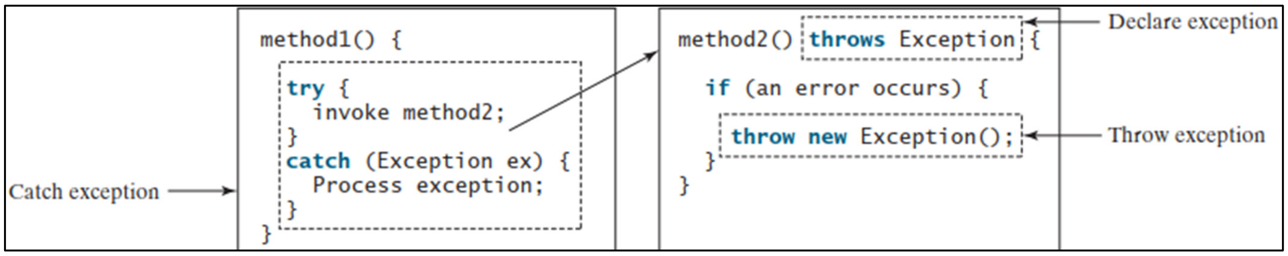


Exception classes can be categorised into 3 major types:

- Exceptions (https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html)
- System Errors (https://docs.oracle.com/javase/7/docs/api/java/lang/Error.html)
- Runtime exceptions (https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html)

Some important exceptions

Exception	**Description**
Arithmetic Exception	Arithmetic error occurs, such as divide-by-zero.
ArrayIndexOutOfBoundException	Array index is out-of-bounds (either negative or greater than or equal to the size of the array).
FileNotFoundException	File is not accessible or does not open.
IOException	Input-output operation failed or interrupted.
NullPointerException	Referring to the members of a null object.
RuntimeException	Any exception which occurs during runtime.
StringIndexOutOfBoundsException	An index is either negative or greater than the size of the string.
NumberFormatException	A method could not convert a string into a numeric format.
IllegalArgumentException	A method has been passed an illegal or inappropriate argument.



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```

Exception handling in Java

Every method must declare types of exceptions it might throw. However, Java does not require **Error** and **Runtime Exception** to be declared in a method (because system errors and runtime errors can happen to any code). Example:

```
public void getMyFile() throws IOException
```

For a method that throws multiple exceptions, declare a list of exceptions, separated by commas.

```
public void myMethod() throws Exception1, Exception2, Exception3
```