

# COM1027 Programming Fundamentals

## Lab 8

### Defensive Code: Introduction to Exceptions and File Handling

#### Purpose

The purpose of this lab is to develop your skills and ability to:

- Differentiate between the three main different kinds of exceptions.
- Read text from files using the FileReader and BufferedReader.
- Split up lines of text which have been read in from a file.
- Use the text from files to create and/or initialise various data structures.

#### Lab Structure

Labs are a mixture of step by step instructions that enable you to learn new skills, and exercises so that you can define your own examples.

We will use **Concept Reminders** in front of text when we are reminding you of concepts you have already learnt previously in labs or lectures.

In using this document it is possible to paste text directly into Eclipse. That means when defining pieces of code contained in lab sheets in Eclipse their value can be copied directly from here, which will remove the possibility of mistyping values. If you are reading a PDF version of the document, then clicking on the 'Select' icon activates text selection. Before running the code, ensure that the code has been copied correctly, and make the relevant changes to make the code fully-functional.

These lab exercises will be **assessed**. They are marked, and contribute to your final grade. This lab exercise has 10 marks to earn which will contribute towards your final grade. Every place you have to add code is indicated as an 'Exercise' with instructions.

A total of 10 labs will be assessed, that will correspond to the first 20% of your final grade. Please submit your completed work using GitLab before 4pm on Friday Xth Month 2021.

You must comply with the University's academic misconduct procedures: <https://www.surrey.ac.uk/office-student-complaints-appeals-and-regulation/academic-misconduct>

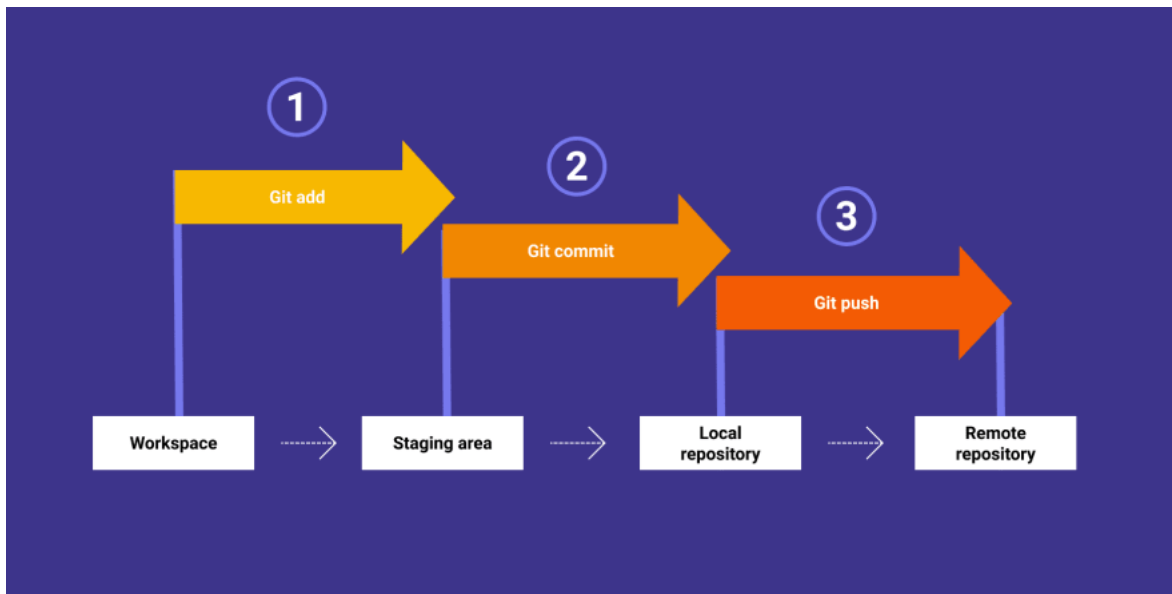
**Please ask us lots of questions during the labs, or use the discussion forums on SurreyLearn.  
All the demonstrators are here to help you get a better understanding of the Java  
programming concepts.**

## Instructions

For this module, we will be using automated assessment tools for all lab activities and coursework. To achieve this, we will be using the Faculty of Engineering and Physical Sciences (FEPS) GitLab platform to upload completed code.

In last week's lab activity, you completed the following:

- Downloaded a project from SurreyLearn and imported it into your workspace
- Made changes to the project by creating new classes
- Used the UML diagrams to convert the structured English language to Java code
- Committed the changes to your local repository and
- Pushed those changes onto your remote repository on GitLab



*Screenshot taken from <https://docs.gitlab.com/>*

In this lab, you will download a new project from SurreyLearn, make substantial changes to the project, commit and push those changes to your remote repository, just like in previous weeks.

**Important Note:** Do not push any buggy code to the repository unless you have to. Use the 'Run As > Java Application' or 'Run As > JUnit Test' as indicated in the instructions below.

**Reminder:** You can access your GitLab repository via <https://gitlab.eps.surrey.ac.uk/>, using your username and password.

## Example

In this example, we will use the pre-defined methods that make use of different types of Exceptions. Read the Example classes carefully to get a better understanding on how to catch and/or throw exceptions.

### 1. Import an existing Eclipse project and set your workspace

From SurreyLearn, download the project called `COM1027_Lab08.zip`. Extract the contents of the project in the Downloads folder. Copy the uncompressed (unzipped) Lab08 folder to your local repository. **Note:** Do not copy the `COM1027_Lab08` folder. Navigate in the folder, and only copy Lab08 folder.

If you are using the computer labs, then your local git directory would look like this:

```
/user/HS223/[username]/git/com1027[username]/
```

For example:

```
/user/HS223/sk0041/git/com1027sk0041/
```

where sk0041 shows a sample username.

Start Eclipse and open the same lab workspace as last time. Import the project in Eclipse, by clicking on `File > Import > General > Project from Folder or Archive`. Select `Directory` to locate Lab08 from your local git repository. Once the files are loaded, select the Eclipse project and click `Finish`.

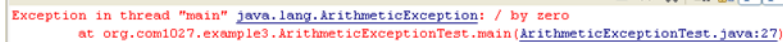
## 2. Unchecked Exceptions

Once imported, look at the `Example1` class in the `lab8` package.

The `Example1` class consists of a main method which defines three local variables: `dontDivideMeByZero`, `x` and `result`. The method is responsible for applying division between the `dontDivideMeByZero` and `x` and displaying the result on the console (value temporarily stored in `result`). An invalid calculation could cause an unchecked exception. Java does not verify unchecked exceptions at compile-time. Furthermore, we don't have to declare unchecked exceptions in a method with the `throws` keyword.

Since unchecked exceptions are caused by a logic error, they do not normally get checked for in a try/catch statement. Catching unchecked exceptions simply masks a problem that should be fixed by a developer. Catching unchecked exceptions should only be done in rare cases where the logic error cannot be fixed and the system must not be allowed to terminate prematurely. It is recommended that conditions such as this be clearly documented and the caught exceptions be logged to a file.

Run the `Example1` code as a Java Application (right click on the class file, and then 'Run As > Java Application'). The following should appear on your console:

A screenshot of a Java IDE console window showing an exception. The text is: "Exception in thread \"main\" java.lang.ArithmeticException: / by zero at org.com1027.example3.ArithmeticExceptionTest.main(ArithmeticExceptionTest.java:27)". The text is color-coded: "Exception in thread \"main\"" is black, "java.lang.ArithmeticException:" is blue, "/ by zero" is red, and the stack trace line is black.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at org.com1027.example3.ArithmeticExceptionTest.main(ArithmeticExceptionTest.java:27)
```

An `ArithmeticException` is used to signal an arithmetic problem. Typically this is dividing by zero.

**Concept Reminder:**

**Exception:** `ArithmeticException`

**Purpose:** To illustrate an error in calculation.

**Hierarchy:** Extends `RuntimeException`

Add a defensive line of code (for example a conditional statement - IF) which checks that `x` is not 0, in order to make the code safe.

Continue on next page

### 3. Throwing Exceptions

Look at the `Person` class in the `lab8` package. This is the same `Person` class we have used in a several lab activities. It represents the attributes and behaviour of a person object. Each `Person` object consists of a `forename`, `surname` and an `age`.

Run the `Example2` class as a Java Application. This creates a `Person` object with an age of `-1`.

The output should be:



The current version of the `Person` class does not check any of the parameters that are passed into the constructor when creating a `Person` object. This can easily be amended. We should make sure that the `forename`, `surname` and `age` are valid.

Change the `Person` constructor so that it looks like this:

```

5 public Person(String forename, String surname, int age) {
    super();
    this.forename = forename;
    this.surname = surname;

    if (age <= 0) {
        throw new IllegalArgumentException("Age too young");
    }
10    this.age = age;
    }

```

**Explanation:** We have added a check on the `age` parameter. If the age is 0 or less then we throw an `IllegalArgumentException`. This exception indicates that a method (or a constructor in this case) has

passed an illegal or inappropriate argument.

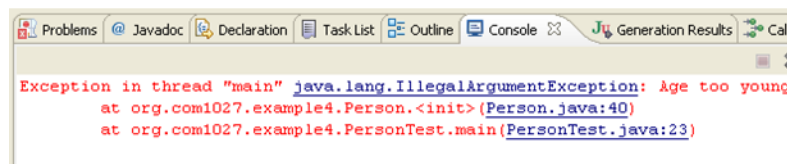
#### Concept Reminder:

**Exception:** `IllegalArgumentException`

**Purpose:** To illustrate an error in the argument(s).

**Hierarchy:** Extends `RuntimeException`

Run the code again. You should see this:



By throwing an unchecked exception the program has terminated when an invalid age is used.

- How can the rest of the parameters get validated?
- What would happen if the `forename` is left blank?
- What if instead of entering a `String` value for the `surname` parameter, an integer value is passed?

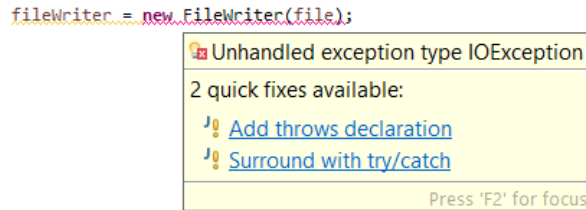
#### 4. Checked Exceptions

Look at the Example3 file.

There is a compilation error in the code which occurs on line 28:

```
FileWriter fileWriter = new FileWriter(file);
```

The compilation error is:



The compilation error is saying that the new `FileWriter(file)` does not handle `IOException`.

**Concept Reminder:**

**Exception:** `IOException`

**Purpose:** This exception should be recovered from in a well-written application

**Hierarchy:** Extends `Exception` class

An `IOException` is used to signal that a problem has occurred in either reading or writing to an input/output stream. For example, when trying to read from a file and something goes wrong. This is a common error to occur and hence Java requires you to handle an `IOException`. This is a checked exception, which means that Java expects you to catch it.

Hover the mouse of the `FileWriter` code and select the **Surround with try/catch** option. Eclipse will insert a try/catch block around the line of code:

```
try {
    fileWriter = new FileWriter(file);
}
catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Note that the `IOException` class has been explicitly imported at the top of the class definition automatically when you do the try/catch.

In order to confirm that the try block is being executed add the following two lines after the `fileWriter` object creation statement. These must be within the try block:

```
fileWriter.write("Adding this line to the file");
fileWriter.close();
```

**Explanation:** What these two lines do is write a piece of text to the file and then close the stream. The methods `write` and `close` can also throw an `IOException`, hence these lines of code must also be within a try block.

Run the code as a Java Application by right-clicking on the class file and then ‘Run As > Java Application’. You will not see anything appearing in the console window. However, if you look in the folder where your workspace code is stored you will find a file called `test1.txt` which contains the line of text included in the code.

Now change:

```
File file = new File("test1.txt");
to be:
File file = new File("");
```

Now run the code again as a Java Application. This time you will see an exception raised:

```
java.io.FileNotFoundException: (No such file or directory)
at java.io.FileOutputStream.open(Native Method)
at java.io.FileOutputStream.<init>(FileOutputStream.java:212)
at java.io.FileOutputStream.<init>(FileOutputStream.java:165)
at java.io.FileWriter.<init>(FileWriter.java:90)
at org.com1027.example1.IOExceptionTest.main(IOExceptionTest.java:29)
```

This exception occurs because the “ ” for the file name to open is invalid. The line that is causing this exception is:

```
fileWriter = new FileWriter(file);
```

which is trying to open a file name which is incorrectly defined. What is displayed is called a “Stack Trace” and it is output because we have put the line (by default):

```
e.printStackTrace();
```

in the catch block. Replace the catch block:

```
catch (IOException e) {
    e.printStackTrace();
}
```

with the following:

```
catch (FileNotFoundException e) {
    System.out.println("Could not find file " + file);
}
catch (IOException e) {
    System.out.println("Something went wrong");
}
```

Run the code again. We should now get the following output:

```
Could not find file
```

**Explanation:** We are now explicitly catching a `FileNotFoundException`. It turns out that a `FileNotFoundException` is-a `IOException`, and hence we were catching it before correctly. However, now we are distinguishing between a generic `IOException` and the specific `FileNotFoundException`.

Put back the correct filename:

```
File file = new File("test1.txt");
```

so that the program works correctly again.

There is one more change we should make to make our code more robust.

An `IOException` might be raised either by the new `FileWriter(file)`, the `write()`, or the `close()`. If the exception is raised on the `write()`, then the catch block will be executed and we will not have closed the file.

Leaving a file open is not a good idea when a program terminates. We should therefore close the file if we have managed to open it.

To achieve this we can use a finally block. A finally block is always executed after a try/catch, even if no exception was raised.

Remove the following line of code from within the try block:

```
fileWriter.close();
```

and add the following code to the end of the try/catch block:

```
5  finally {
    if (fileWriter != null) {
        try {
            fileWriter.close();
        }
        catch (IOException e) {
            System.out.println("Could not close file");
        }
    }
10 }
```

**Explanation:** We are now closing the file if it has been opened, even if a problem occurred. We have removed the close from within the try block because we cannot close the file twice (if no exception was raised). In a finally block we may need to nest another try/catch block since the `close()` method can also raise an `IOException`.

Note: For more information on how to perform file handling (i.e. reading and writing in files), visit the recorded content on SurreyLearn. There are coding demonstrations that explain how the try/catch block of code can be used, as well as explanations on the different types of exceptions.



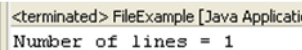
## 5. Throwing instead of catching

In the previous example we caught the checked exceptions that had been raised explicitly. This is the best option. However, occasionally we want to force the calling application to handle the exceptions for us.

Look at the `Example4` file in the `lab8` package.

**Explanation:** `Example4` contains a main method plus two other static methods.

`calculateNumberOfLinesCatch()` counts the number of lines in a file. This method includes the required try/catch blocks to handle `IOException` locally. `calculateNumberOfThrow()` also counts the number of lines in the file but this time it does not include any try/catch blocks and instead throws `IOException`. Run the program as a Java Application. You should see:



```
<terminated> FileExample [Java Applicati
Number of lines = 1
```

Here the main method is calling the `calculateNumberOfLinesCatch()` method. Add the following code to the main method:

```
count = calculateNumberOfLinesThrow("test1.txt");
System.out.println("Number of lines = " + count);
```

This code will not compile. Why?

**Explanation:** The `calculateNumberOfThrow()` is not handling the checked `IOException`. It achieves this by adding the throws `IOException`. Instead the calling method must handle the checked exceptions. All that has happened is that the exception handling has been postponed.

Fix the code so that it compiles you will need a try/catch block. Try the quick fix by hovering over the red line.

## 6. Defining your own exceptions

Java provides checked and unchecked exceptions that we can use. Often, however, it is better to define our own checked exceptions so that the reason why they are being raised is clear.

For example, by throwing `IllegalArgumentException` in the previous examples, we are using an unchecked exception. This means we do not have to catch the exception and the result is that the code compiles but does not run. It is better if we catch this exception and even better if we define our own exception to make clear what the problem is.

When we define our own exceptions, they should all extend from the `Exception` class.

Open the `InvalidAgeException` class. This is our own exception that is used to signal a problem with the age. You will see it just consists of two constructors.

In the `Person` constructor, replace `IllegalArgumentException` with `InvalidAgeException`.

The `Person` class will now not compile because `InvalidAgeException` is a checked exception. We do not want to handle the exception in the constructor using a try/catch block but instead want the calling method to handle it. Use quick fix to add a throws declaration to the constructor.

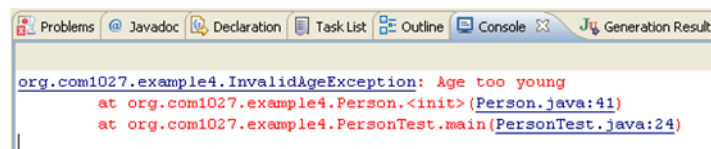
Once we have done this we will see that `Example2` does not compile. In `Example2` we now need to handle the `InvalidAgeException`.

Quick fix the line:

```
Person person = new Person("Joe", "Bloggs", -1);
```

so that it is surrounded by a try/catch block.

Run the code. You should see:



## 7. Reading a character at a time from a textfile

This example will make use of `words.txt` and the `Example5` class. Open `words.txt` to see that it contains three lines of text. Open the `Example5` class file. You will see that the code has the following structure (but with a lot more detail):

```

5      try {
        // Open the text file for reading.
        // Read a single character from the file.

10     // Loop while we can read characters from the file.
        while (not end of file) {
            // Do something with data...
            // Read in the next character.
        }
15    }
        catch (FileNotFoundException e) {
            // Executed if the words.txt file does not exist.
        }
        catch (IOException e) {
            // Executed if something goes wrong while reading the file.
        }
        finally {
            // Close the file.
20    }
    }

```

Run `Example5`.

The output you will see is as follows:

```

<terminated> Example. [Java Application]
116
104
101
32
99
97
116
32
115
97
116
32

```

**Explanation:** A file reader object is created given a String which is the name and location of the file:

```
in = new FileReader("words.txt");
```

Then a single character is read in:

```
int data = in.read();
```

The type of the data variable may not be what we expect. Read the comments in the code to understand more.

We then start a while loop. A while loop allows us to loop while a condition is true. This is similar to a for loop except that we are not looping for a fixed number of times but instead continuously looping until we identify when we want to stop. In this case we stop the loop when we have run out of characters in the file (when data is -1).

If the data character is does not signify the end of the file, then the character is printed. Then the next character is read in, and then the execution goes round the while loop again.

When the end of the file has been reached the execution jumps out of the while loop and we must close the file. This is done in the finally block which follows the try and catch blocks.

**We will look at try, catch and finally in more detail later in the module, so do not worry about them now but understand that they are needed to read files.**

**(Check out <http://docs.oracle.com/javase/tutorial/essential/exceptions/> if you want to learn more.)**

In the output what does 116, 104, mean? Look at <http://www.asciitable.com/> and see if you can work out what each number means in relation to the text file.

Change the line:

```
System.out.println(data);
```

to the following:

```
System.out.println((char) data);
```

Run Example5 again as a Java Application and this time the output should be different because each integer has been converted to its corresponding character.

## 8. Reading a line at a time from a textfile

Reading one character at a time from a text file is tedious. There is an easier way to read lines from a file using a `BufferedReader`.

Look at `Example6` class. It contains the following changes to the code:

```
// Wrap the FileReader inside a BufferedReader.
buffer = new BufferedReader(in);

// Read a line from the file.
5 String line = buffer.readLine();

// While we have not reached the end of the file.
while (line != null) {
    // Do something with data...
10    System.out.println(line);

    // Read in the next line.
    line = buffer.readLine();
}
```

**Explanation:** We now create a `BufferedReader` object which allows us to buffer a character-input stream that uses a default-sized input buffer.

Using the buffer, we read in a line at a time. Notice that now the return type is `String` (and not `int` as in `Example5`). So this makes it easier to do something with the text from the file.

Notice that closing the input stream is now done on the buffered reader object and not the file handler. This is because the file handler is wrapped inside the `BufferedReader`.

Run the `Example6` program as a Java Application. You will see the lines of the text file displayed on the console.

## 9. Extracting words from a String

Now we have lines of text which we can read from a file, we are going to start processing the text to do something useful with it. First we need to understand how to split the line up into useful chunks.

Look at the `Example7` class and run it as a Java Application. It will take a string and print out the number of separate words in the string and print each word on a separate line:

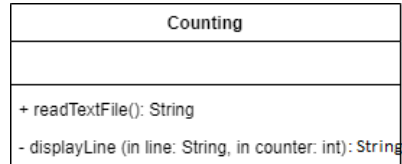
```
<terminated> Example
Extracted 7 tokens from the line
What
on
earth
is
going
on
here?
```

## Exercise 1 (2 Marks)

**Note:**

Each exercise builds on the activities covered in the above examples.

This exercise requires you to create a simple class called `Counting` as follows:



For this exercise, you will be using the `lab8.exercise1` package and you are expected to demonstrate your understanding of reading files. Create a new class with the name `Counting` and add the required methods as specified in the UML diagram above. *Note: The `Example4` and `Example6` classes might come in handy in defining this class.*

- Define a default constructor for the class.
- Add the `readTextFile` method which should return a `String` value that represents the content of the text file (`words.txt`) and a counter. Change your code so that in between each of your sentences you will store the number of lines that have been read in. The return value of this method should look like the following (if you were to display it on the console):

```
the cat sat on the mat : 1
the sat mat on the cat : 2
the mat cat on the sat : 3
```
- To achieve the above output, define a private method called `displayLine` which is responsible for formatting the output so that it structures the `String` as `[line read] + [colon] + [number of line]`.
- Test your code by right-clicking the project name and then `Run As > JUnit Test`. `JUnit` tests have been predefined in the `src/test/java` source folder. Do not modify them as this would cause your Maven Test to fail!

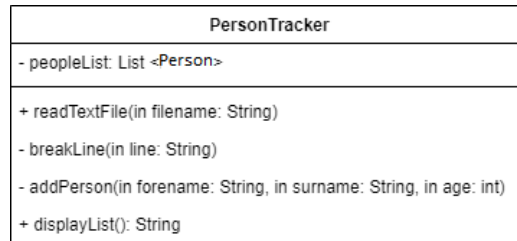
**Go to next page**

## Exercise 2 (5 Marks)

This exercise requires you to define a new class that read a specified text file and stores the information in a data structure, prior to then displaying them on the console.

For this exercise, you will be using the `lab8_exercise2` package. Copy the `Person` class from the examples (see package `lab8`). Then create a new class called `PersonTracker`.

The aim of this exercise is to assign person objects in an array list using the information provided in the text file. The following class diagram shows the structure of the `PersonTracker` class. Convert the class diagram into Java code:



- Define a `PersonTracker` class. Each `PersonTracker` object comprises of a list of people.
- Define a default constructor for the class and define the `peopleList` in the body of the constructor
- Add the following a simple display method `displayList` that loops through each value of the list and displays the `toString` representation of the `Person` objects.
- Create a method called `readTextFile()`. This method is similar with the `readTextFile()` method from `Exercise1`. (*Note: This is defined as a procedure not a function, unlike Exercise 1*). This method is responsible for reading the file specified in its parameters. If the file name is valid, then this method should enable you to read the file line by line.
- Define the `breakLine` method as a private procedure. This methods is only called within the `readTextFile` method and it is responsible for extracting the values from each line read into three sections: forename, surname and age.
- Define the `addPerson` method. This is also a private method which is only called in the `breakLine` method. The values extracted from the text file are passed as arguments in the `addPerson` method in order to initialise a new `Person` object and add it to the `peopleList`.
- Before moving to Exercise 3, test your code by right-clicking the project name and then `Run As > JUnit Test`. Once all the tests for the `Person` and `PersonTracker` classes pass, you can start working on Exercise 3.

Note: Prior to running the JUnit tests, remove the exceptions raised in the `Person` class. The changes include the modification of the constructor's signature and the removal of the conditional statement that throws an exception. We will be using this exception again in Exercise 3, but not in this exercise.

## Stretch & Challenge Exercise (3 Marks)

This exercise requires you to define your own exceptions, and raise them where required.

For this exercise, you will be using the `lab8_exercise3` package. Copy the classes `Person` and `PersonTracker` from the package `lab8_exercise2`. Create a new class called `InvalidSurnameException`. You can use the `InvalidAgeException` as a template to define your exception. Add both exception classes in your package as the `Person` class is responsible for validating a person's age and surname.

- Define a `InvalidSurnameException` class.
- Define a default constructor for the class that allows an object to be created. In the `Person` constructor, raise an exception if the surname is not a string which consists of the letters a to z only (a-z), with the first letter being a capital. **Hint: use a regular expression to achieve this check.**
- Introduce a conditional statement in the constructor to validate the age too. Throw an `InvalidAgeException` if the age is less than or equals to 0 (zero). You can refer back to the previous examples and either copy the code, or re-write it as expected.

If the conversion fails, the exception will be raised that would cause the program to terminate. In the `PersonTracker` class, catch the `InvalidSurnameException` and `InvalidAgeException` where required. Use the pre-defined JUnit test cases in the `lab8_exercise3` test package to guide you.

- You can test your code, by creating a `Test` class with multiple objects or by running the pre-defined JUnit tests.

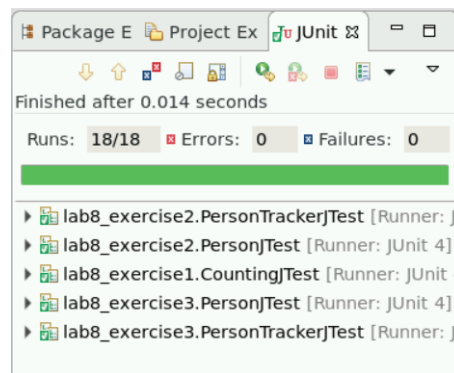


## Final Steps

The final step of the lab activity is to check on whether the code is fully functional. To do this, right-click on the project name and select **Run As > JUnit Test**.

If any errors occur, select the relevant file and check the error.

By selecting **Run As > JUnit Test** all the pre-defined JUnit tests will be called. You should get the following in order to proceed:



It is now time to test the structure of your Maven project. To do this, right-click on the project name and select **Run As > Maven Test**.

This should return the following message on the console:

```
Results :
Tests run: 18, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 4.008 s
```

The test passes - now what?

You can now commit and push all the changes made to the remote repository. To do this, select the **Git Staging** view via the menu **Window > Show View > Other...** and then **Git > Git Staging**. Add all the unstaged changes to your local repository, and commit and push the changes to the remote repository.

