

Lab: Working with Git

The aim of this lab is to give you a chance to work with Git and Git Hub.

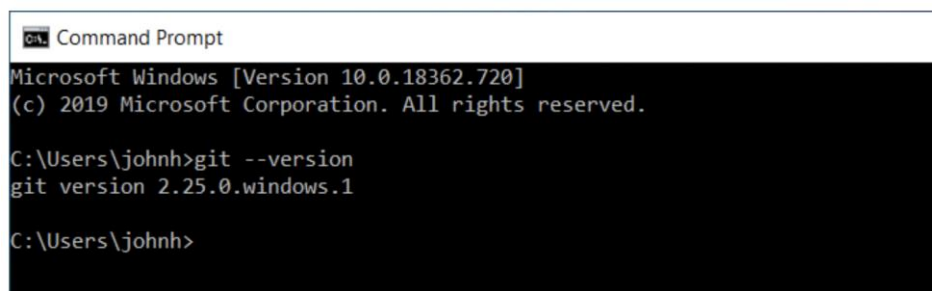
Note: Please work on your R: Drive.

Step 1: Git Client

In this first step we will first confirm that you have the git command available. To do this open a Command Window on a Windows PC or a Terminal on a Mac or Linux box. Once you have done this at the prompt enter the command

```
> git --version
```

If you have the Git client installed then you should see the version number of the git client displayed, for example:



```
Command Prompt
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\johnh>git --version
git version 2.25.0.windows.1

C:\Users\johnh>
```

In this case the version of Git installed is version 2.2.5.0 for Windows.

If you do not have the Git client installed, then you will be told that the command cannot be found. In this case you need to install the Git client.

You can do this by going to the following web site and downloading the correct version for your operating system:

1. <https://git-scm.com/downloads>

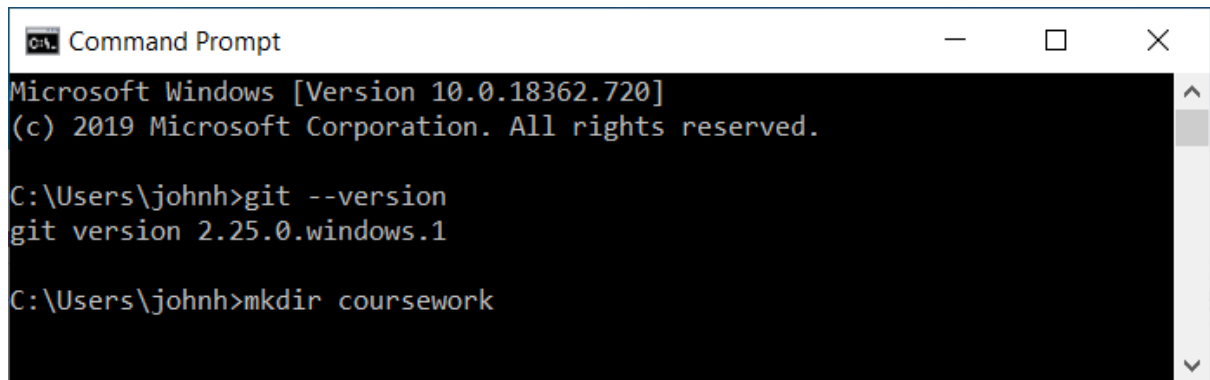
That is, you need to select one of:



Step 2: Create a directory to work in

We will handle working with our labs using a specific directory. If you have not already done so, then create a directory for course work called something like coursework. On a Mac you might put this under / on Windows PC this might be in R:\.

For example:



```
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\johnh>git --version
git version 2.25.0.windows.1

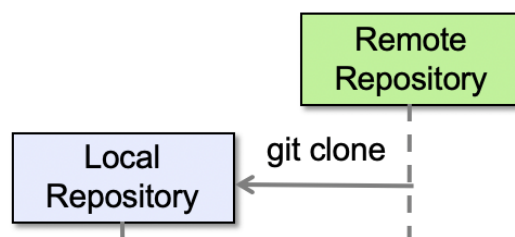
C:\Users\johnh>mkdir coursework
```

Now change directory into this newly created directory, for example use

```
> cd coursework
```

Step 3: Git Clone a Repository

We will now *clone* an existing Git project from a GitHub hosted repository. Remember to work with an existing repository in Git, you need to obtain a clone (copy) of that repository or project locally. This will copy the report project locally to your machine and allow you to make changes to that project.



To do this you can issue the git clone command from within a Terminal or Command Window with an appropriate git project web URL.

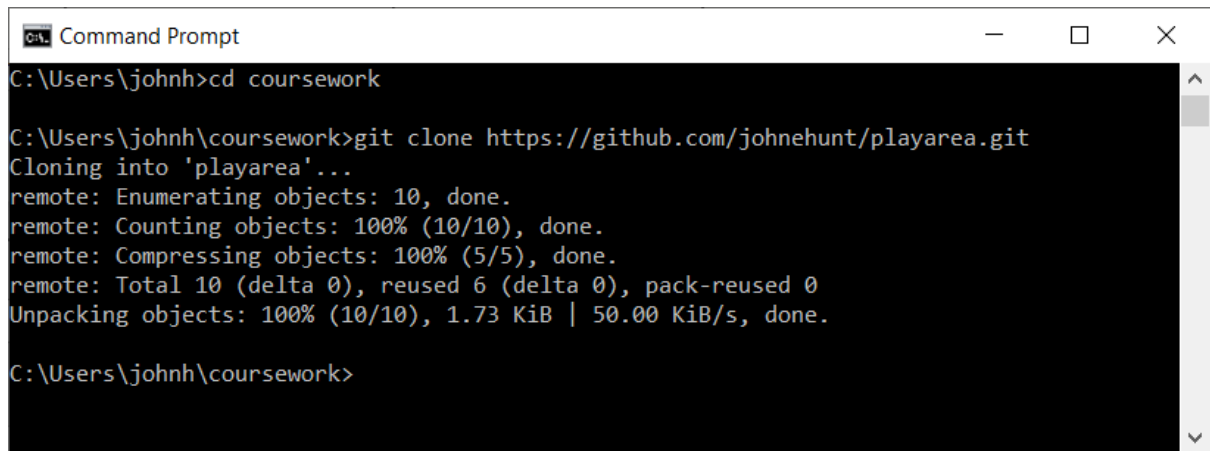
Return to your Terminal / Command Window and ensure that you have moved into the whatever directory you have been using for your coursework directory (using cd coursework). For example:

```
> cd coursework
```

Next issue the git clone command as follows:

```
> git clone <git repo URL>
```

This will clone to your machine a copy of the project from the GitHub repository.

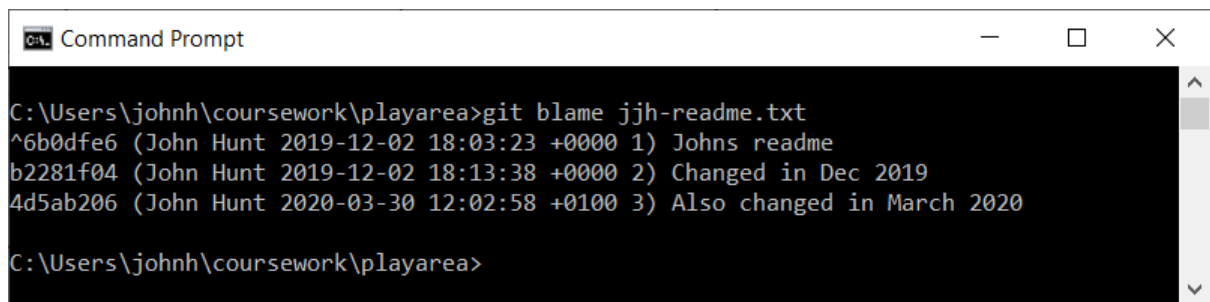


```
Command Prompt
C:\Users\johnh>cd coursework

C:\Users\johnh\coursework>git clone https://github.com/johnehunt/playarea.git
Cloning into 'playarea'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 10 (delta 0), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), 1.73 KiB | 50.00 KiB/s, done.

C:\Users\johnh\coursework>
```

You should now change directory into the cloned repo directory; in the above case this is the playarea directory.



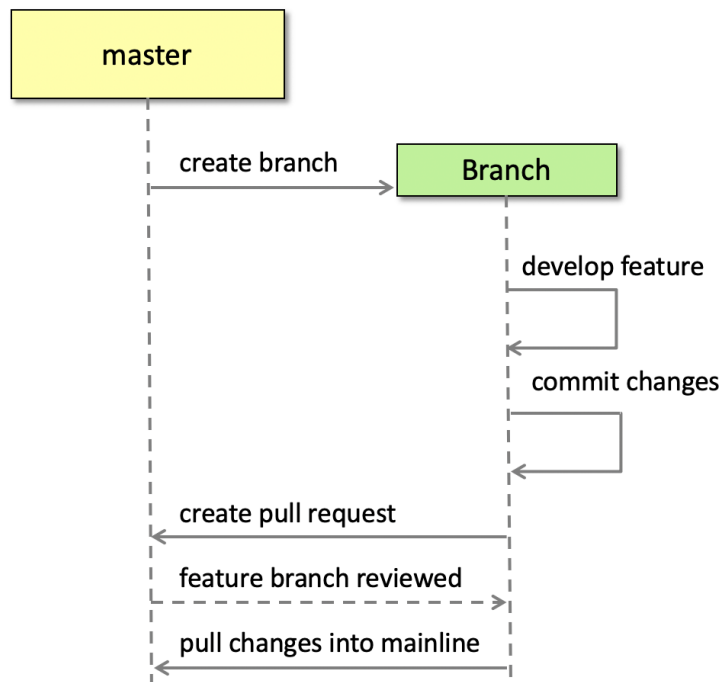
```
Command Prompt
C:\Users\johnh\coursework\playarea>git blame jjh-readme.txt
^6b0dfe6 (John Hunt 2019-12-02 18:03:23 +0000 1) Johns readme
b2281f04 (John Hunt 2019-12-02 18:13:38 +0000 2) Changed in Dec 2019
4d5ab206 (John Hunt 2020-03-30 12:02:58 +0100 3) Also changed in March 2020

C:\Users\johnh\coursework\playarea>
```

Step 4. Create a branch

When working with Git it is usual to work in a development *branch*.

That is all changes are made to a branch and then this branch can be merged back into the main *master* branch via a Pull Request (or PR). This process is illustrated below:



In the next few steps we will work through this process.

The first thing we will do is to create a new branch to work in. You can use the following commands to create a branch and change into that branch:

```
git branch <new-branch-name>
git checkout <new-branch-name>
```

There is a shorthand form for this which is

```
> git checkout -b <new-branch-name>
```

This creates the new branch and switches to that branch.

Any changes you now make are applied to the branch you created.

We will now create a branch; - name the branch after yourself, for example, if you are John Reginald Smith then call your branch jrsbranch, for example:

```
Command Prompt
C:\Users\johnh\coursework\playarea>git checkout -b jrsbranch
Switched to a new branch 'jrsbranch'

C:\Users\johnh\coursework\playarea>
```

The current branch is now the development branch you created.

You can obtain a list of call branches using the command

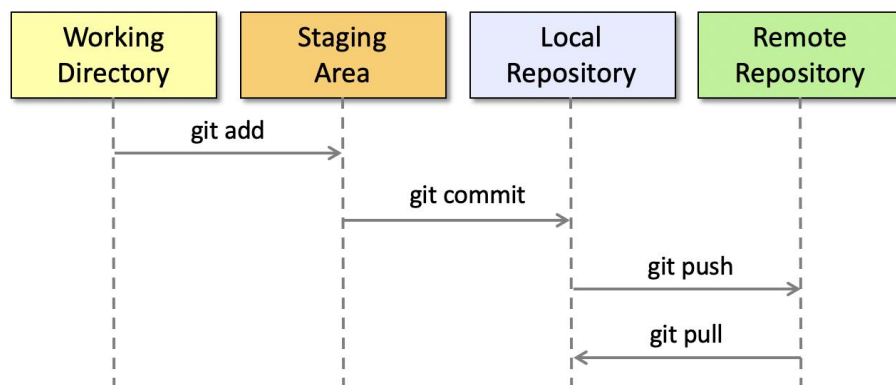
```
> git branch
```

For example:

```
Command Prompt
C:\Users\johnh\coursework\playarea>git branch
* jrsbranch
  master
C:\Users\johnh\coursework\playarea>
```

Step 6: Adding a file

We will now create a file and add it to current branch. The steps we will follow are illustrated below:

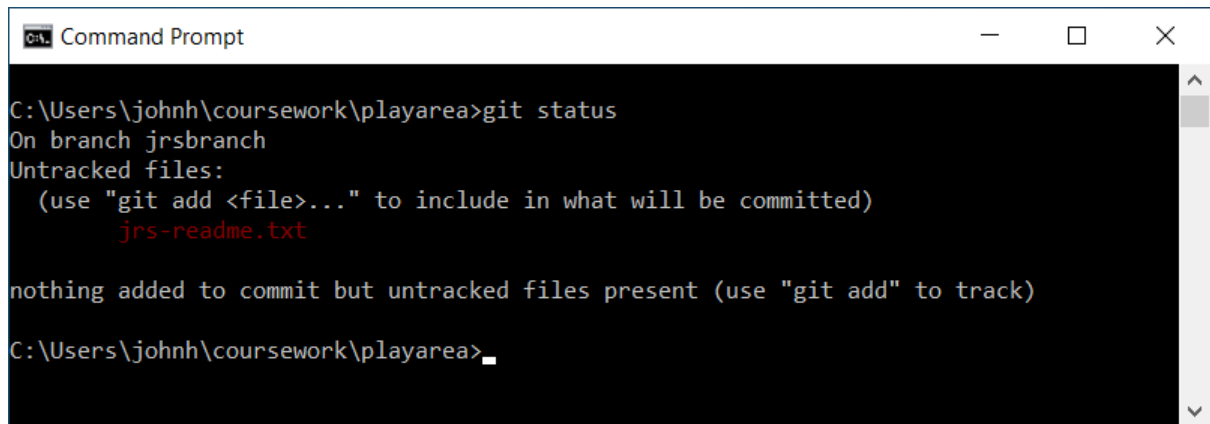


First of all, create a file in your branch again named after yourself, for example `jrs-readme.txt` within the `playarea` directory. You can do this in any way you wish.

Once you have created the file use the `git` command to see the status of your branch:

```
> git status
```

For example:



```
C:\Users\johnh\coursework\playarea>git status
On branch jrsbranch
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        jrs-readme.txt

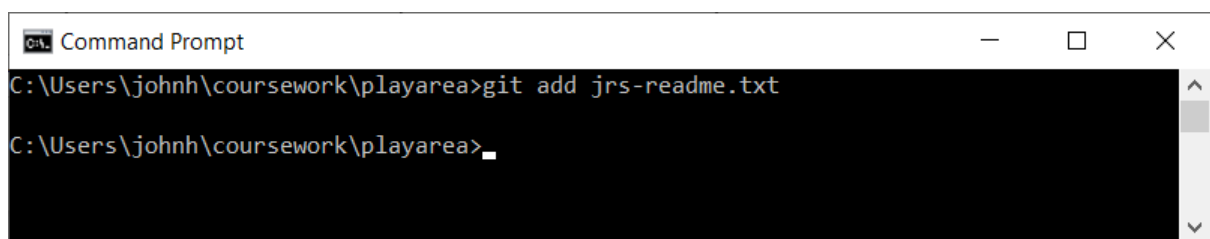
nothing added to commit but untracked files present (use "git add" to track)
C:\Users\johnh\coursework\playarea>
```

From this you can see that there is one new file but it is currently not being tracked by Git (i.e. Git knows nothing about it).

We will now add the file to the branch. To do this you use the git add command:

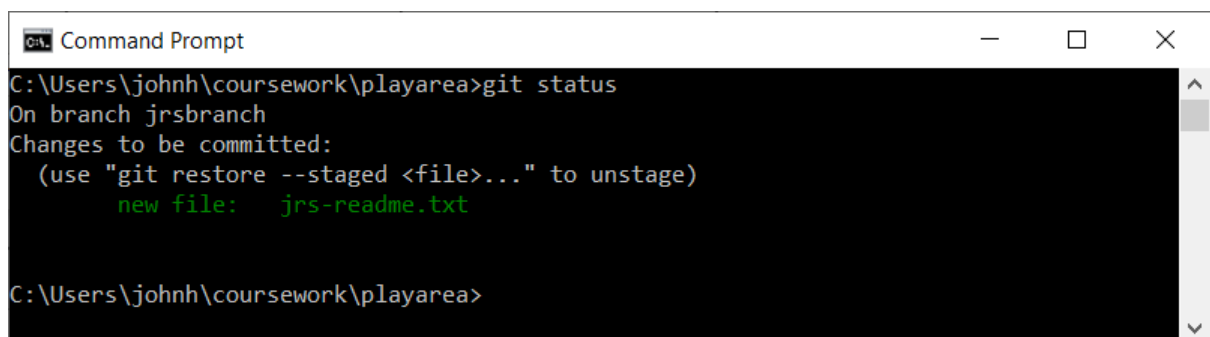
```
> git add <filename>
```

For example:



```
C:\Users\johnh\coursework\playarea>git add jrs-readme.txt
C:\Users\johnh\coursework\playarea>
```

Once you have done this re-run the git status command, you should now see:



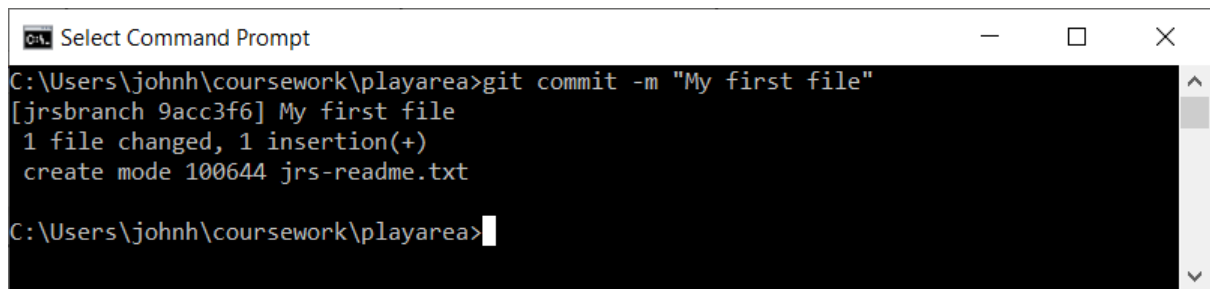
```
C:\Users\johnh\coursework\playarea>git status
On branch jrsbranch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   jrs-readme.txt

C:\Users\johnh\coursework\playarea>
```

Notice that Git now knows about the new file; but the changes have not yet been committed (added to the branch).

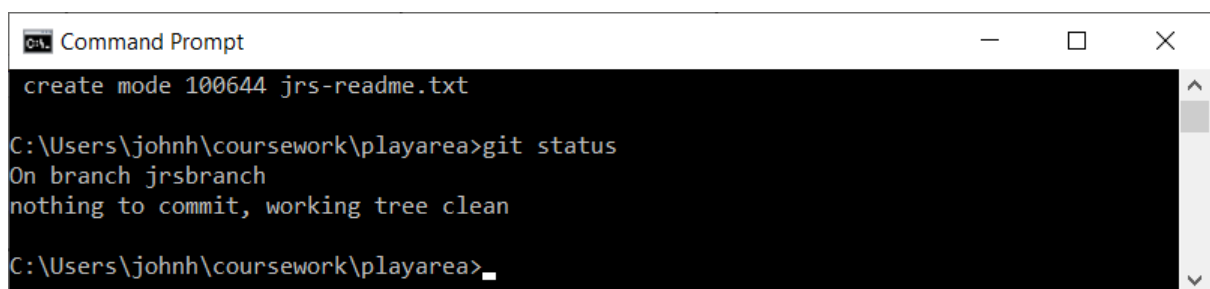
Step 8: Commit changes

Next we will commit the changes we have just made to the branch. To do this use the git commit command. This command needs a description for the commit you are making, we can provide this using the -m option, for example:



```
C:\Users\johnh\coursework\playarea>git commit -m "My first file"
[jrsbranch 9acc3f6] My first file
1 file changed, 1 insertion(+)
create mode 100644 jrs-readme.txt
C:\Users\johnh\coursework\playarea>
```

Now use git status to see what the status is:



```
create mode 100644 jrs-readme.txt
C:\Users\johnh\coursework\playarea>git status
On branch jrsbranch
nothing to commit, working tree clean
C:\Users\johnh\coursework\playarea>
```

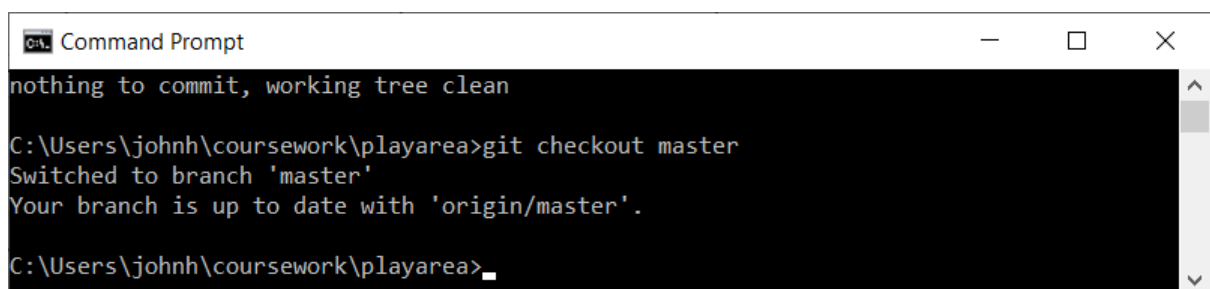
Step 8: Switching branches

Next we will see the effect of switch between the master branch and the branch you have been working in.

From the command window / Terminal enter the command

```
> git checkout master
```

This will switch the current branch back to the master branch.



```
nothing to commit, working tree clean
C:\Users\johnh\coursework\playarea>git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
C:\Users\johnh\coursework\playarea>
```

Now look at the contents of your directory. Can you see the file you created?

```
Command Prompt

C:\Users\johnh\coursework\playarea>dir
Volume in drive C is OS
Volume Serial Number is 64DE-F663

Directory of C:\Users\johnh\coursework\playarea

30/03/2020  15:12    <DIR>          .
30/03/2020  15:12    <DIR>          ..
30/03/2020  14:59                63 jjh-readme.txt
               1 File(s)                63 bytes
               2 Dir(s)  376,131,387,392 bytes free

C:\Users\johnh\coursework\playarea>
```

It's not there; why not?

Because it is not part of the master branch.

Next issue the git status command; what do you see?

Finally switch back to the branch you were working in, for example:

```
> git checkout jrsbranch
```

(Remember to replace jrsbranch with whatever you called your own branch).

Now look at the contents of your directory; what is the difference?

```
Command Prompt

C:\Users\johnh\coursework\playarea>dir
Volume in drive C is OS
Volume Serial Number is 64DE-F663

Directory of C:\Users\johnh\coursework\playarea

30/03/2020  15:13    <DIR>          .
30/03/2020  15:13    <DIR>          ..
30/03/2020  14:59                63 jjh-readme.txt
30/03/2020  15:13                19 jrs-readme.txt
               2 File(s)                82 bytes
               2 Dir(s)  376,130,600,960 bytes free

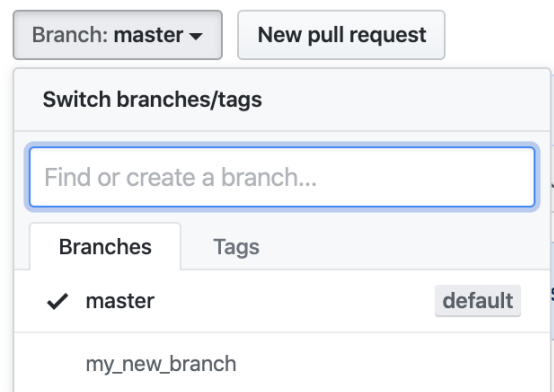
C:\Users\johnh\coursework\playarea>
```

Your file is back again; because it is part of this branch.

Step 9: Push changes

All the changes we have made so far have been limited to your local machine. If you go to the online GitLab repository which is centrally hosting this project you will see that none of your branches are available there.

Next select the 'Branch: master' drop down button to see the list of branches available. Initially it looks something like:



Your own branch is not listed.

We will now push your branch to the GitHub repository. As this is the first time we have done this; and the remote Git repository does not know anything about our repository we need to use an extended form of the git push command:

```
> git push --set-upstream origin jrsbranch
```

To do this you will need a GitLab account and depending on the configuration on your machine you may need to supply your credentials at this point.

Once you have done this you can push a file to the project within the repository using git push command:

```
> git push --set-upstream origin jrsbranch
```

For example:

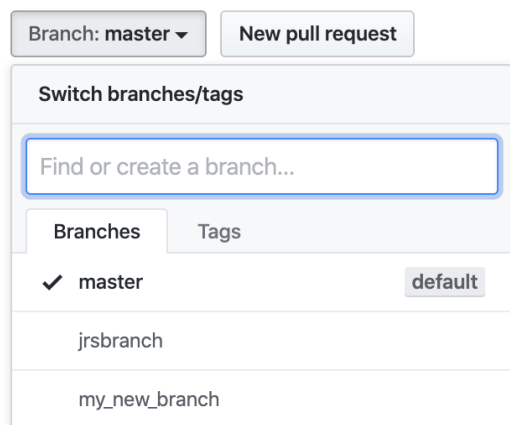
```
Command Prompt
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 294 bytes | 58.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'jrsbranch' on GitHub by visiting:
remote:   https://github.com/johnehunt/playarea/pull/new/jrsbranch
remote:
To https://github.com/johnehunt/playarea.git
* [new branch]      jrsbranch -> jrsbranch
Branch 'jrsbranch' set up to track remote branch 'jrsbranch' from 'origin'.
C:\Users\johnh\coursework\playarea>
```

To see any other changes made to this project you can use the git pull command:

```
> git pull
```

If others have added their branches to the project when you do a git pull followed by a git branch you should see them listed.

If you now return to the GitHub web page you should now see your branch listed in the drop down 'Branch: master' menu.



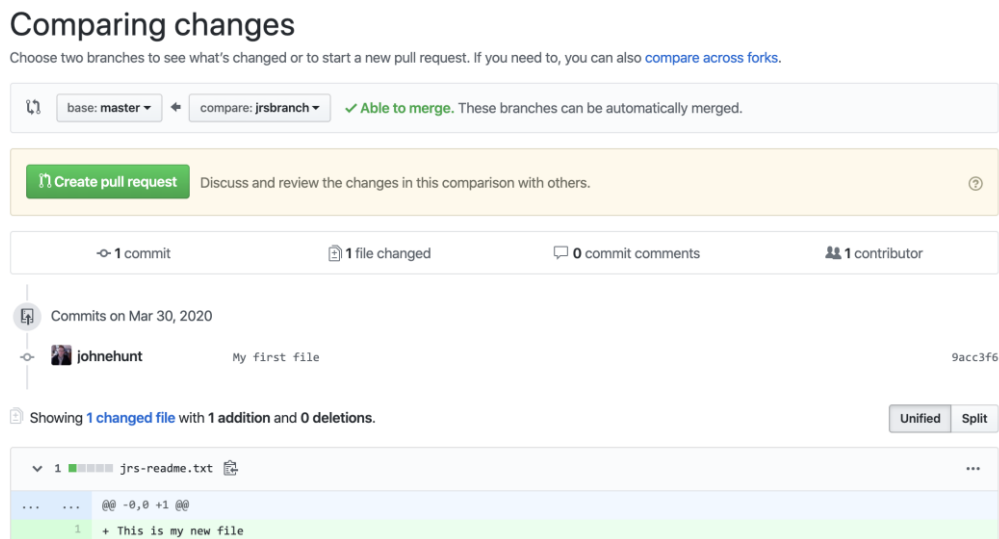
Note you may need to refresh the web page to pick up the new branch. Select your branch now.

Step 10: Pull Request

You can now create a new Pull Request (PR) aka a Merge Request (on GitLab). A Merge Request is a request to merge the changes you have made in your development or feature branch into the main master branch.

You can start a Pull Request by selecting the 'New pull request' button which is next to the Branch drop down button.

You should then see a display similar to:



Next click on the 'Create pull request' button.

Next provide a comment for this Pull Request and select one of the other people on the course to act as a reviewer and as an assignee.

Pull Requests are reviewed by one or more other team members. It is those team members who then accept the changes and merge them into the master branch.

Lab: Hello Bookshop

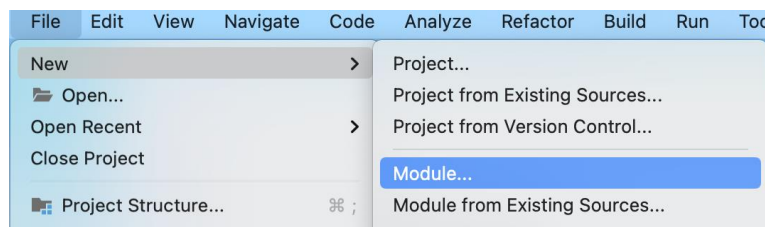
The aim of this lab is to create an IntelliJ project with a Java module within which to develop your Java code.

Part 1: Set up working environment

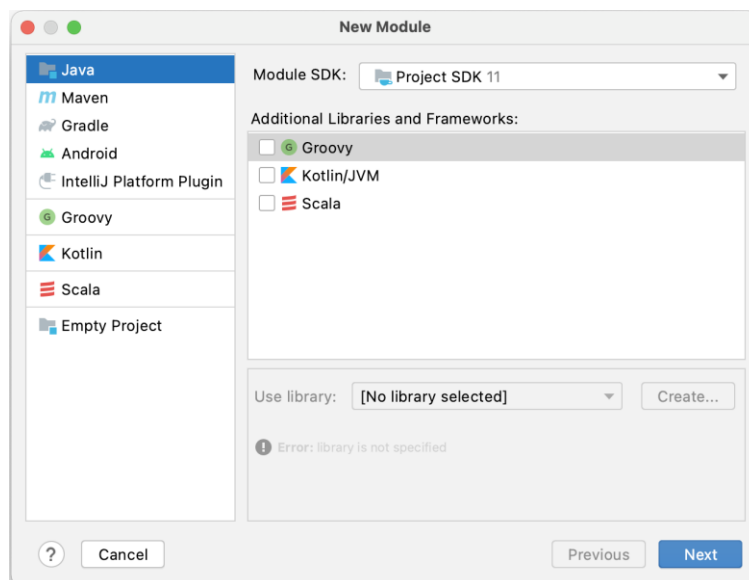
Step 1: Create a new Module

We are going to create a new Java module within our IDE. In the next lab we will create a new Maven project but for now we will just limit ourselves to a Java project.

To do this select the File>New>Module option, which should look similar to:



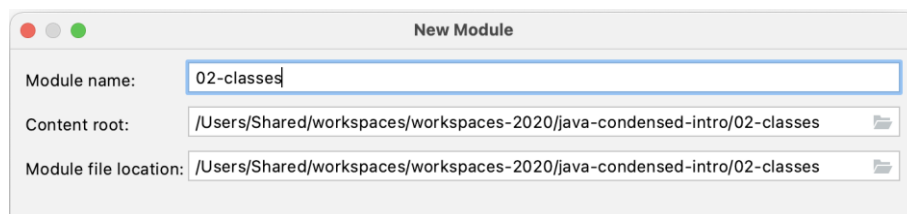
You should now see the 'New Module' dialog. The exact set of options displayed to you on the left-hand side may differ from that below as it depends on the plug-ins installed in IntelliJ. In this case this IntelliJ instance is set up not just for Java but also for Kotlin and Scala development.



Select the Java option in the left-hand option list. You may also need to select an appropriate Java SDK (JDK) for the module. In this example the version for Java to be used for the module is Java 11 (any version from Java 8 onwards would be acceptable but it is most common to use those versions with Long term Support (LTS) such as Java 8 and a Java 11).

Now click on the 'Next' button.

The next dialog displayed will allow you to provide a name for the module, and the location in which you wish to store the code and configuration files for the module.



Make sure the data is appropriate for your environment. For example, if you are on a Windows machine, make sure you set the root directory for the module on an appropriate drive.

It is recommended that you use a directory on your **R: drive**.

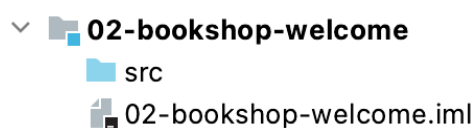
Now click on Finish.

You should now see the new project listed in the left-hand project window of the IntelliJ IDE. You may need to wait a minute for this to happen as IntelliJ will rebuild its internal structures before displaying the new module.

Part 2: Writing a Simple Application

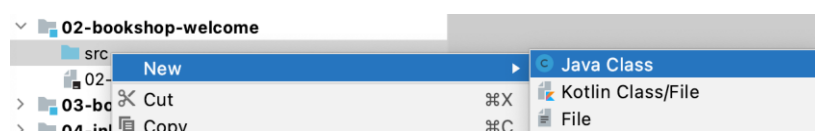
Step 1: Add a Main Method Class

Open the module you just created in the Project View and expand the tree if necessary, to find the src directory:



We will now add a class under the **src** directory.

To do this select the blue src node in the project tree and then from the right mouse menu select New>Java Class



In the 'New Java Class' dialog enter the name of the class; use something such as BookshopApp as shown below:



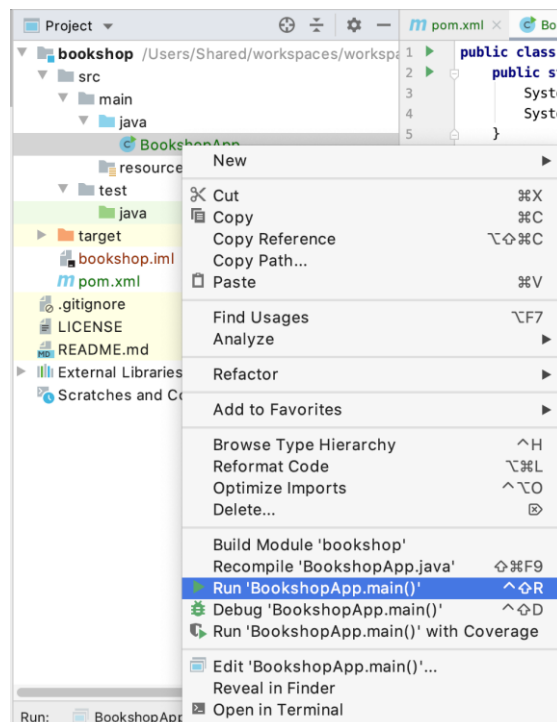
You will need to add the main method to the BookshopApp class as shown below:

```
public class BookshopApp {  
    public static void main(String[] args) {  
        System.out.println("Welcome to the Java Bookshop");  
        System.out.println("=====");  
    }  
}
```

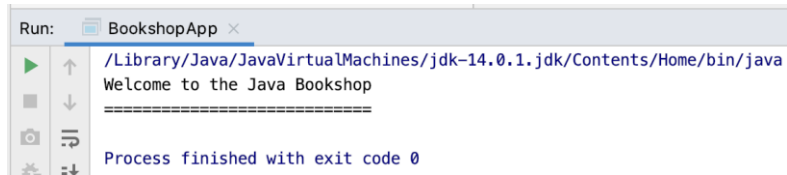
Save the file once you have made the change.

Step 2: Run the Program

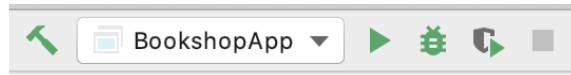
Next select the class in the Project view and from the right mouse menu select the Run option with the name of your class and main() on it, for example:



You should now see the output of your application shown in the Run window at the bottom of the IDE:



You should now see the BookshopApp appear as an option in the top tool bar of your IDE alongside the green run arrow:



This shows that the Application can be rerun. Now click on the green arrow. The program will run again.

Lab: Hello Bookshop Maven Application

The aim of this lab is to create a new module within your IntelliJ project that is configured using Maven. This will allow you to manage the version of Java used with the module, the dependencies the module has on other libraries and tests associated with the module.

The Lab has two parts:

1. Setting up a Maven managed module
2. implementing a basic Hello (BookShop) Program (again)

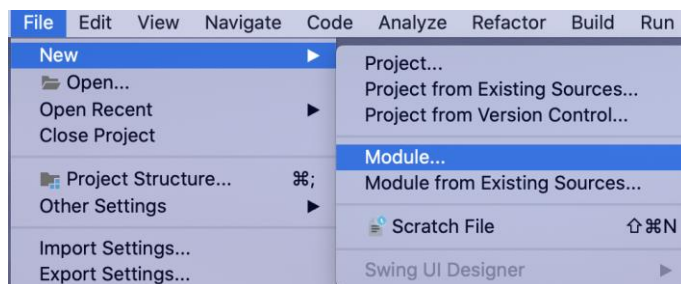
Part 1: Set up Module

Step 1: Create a Module

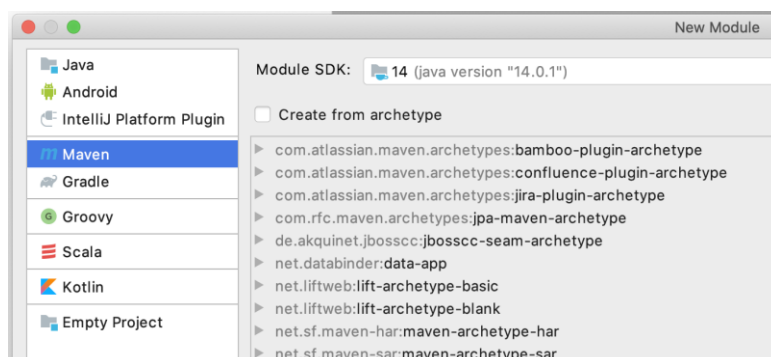
As previously mentioned IntelliJ projects are comprised of 1 or more modules; each of which can have their own configuration, their own Maven POM file etc.

We will now create a new module to work within and to be displayed within the IDE.

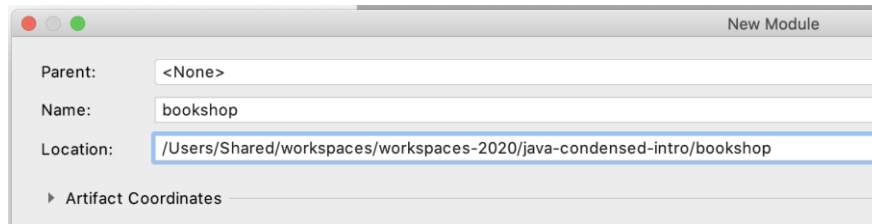
To do this use the File>New>Module... menu option again; for example:



In the new Module dialog this time you should select the 'Maven' option:



In the next dialog enter a name for your module and ensure that you select the appropriate location for the module – it should be inside the previous created IntelliJ project, for example:



Step 2: Update the Module pom.xml file

When you created the module this time you will have used the default archetype. This archetype creates a default directory structure and a default POM file.

In this step we will update the pom.xml file.

We will now update the POM file with some of the dependencies that we will be using later on as well as specify the version of Java we want to use. To do this add the following to the POM file after the version tag but within the project element (note you can copy this from within the PDF):

```
<packaging>jar</packaging>

<name>bookshop-app</name>
<url>http://midmarsh.co.uk</url>

<properties>
  <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
  <maven.compiler.target>11</maven.compiler.target>
  <maven.compiler.source>11</maven.compiler.source>
</properties>

<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
```

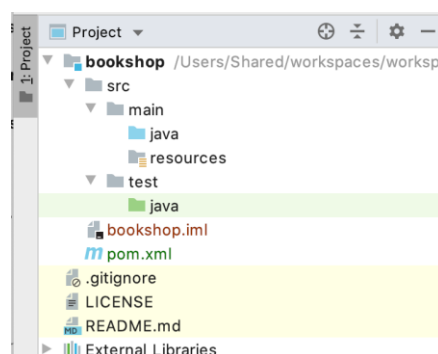
```
</dependency>  
</dependencies>
```

IntelliJ will now notify you that the Maven file has changed and ask if you want to import the changes to your project. Select the Auto-enable option and each time you change the POM the project will be updated automatically.

Part 2: Writing a Simple Application

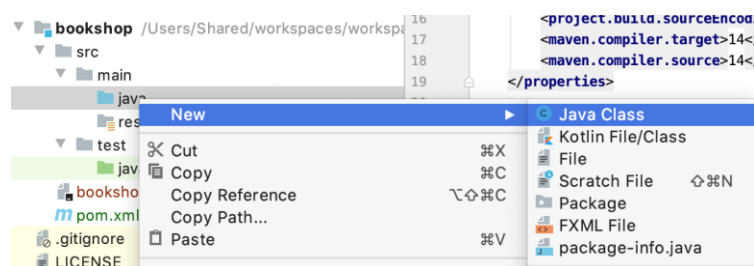
Step 1: Add a Main Method Class

Open the module you just created in the Project View and expand the tree if necessary, so that you can see the structure created below:

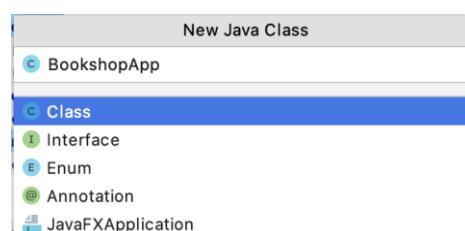


We will now add a class under the **src/main/java** path.

To do this select the blue Java node in the project tree and then from the right mouse menu select New>Java Class



In the 'New Java Class' dialog enter the name of the class; use something such as BookshopApp as shown below:



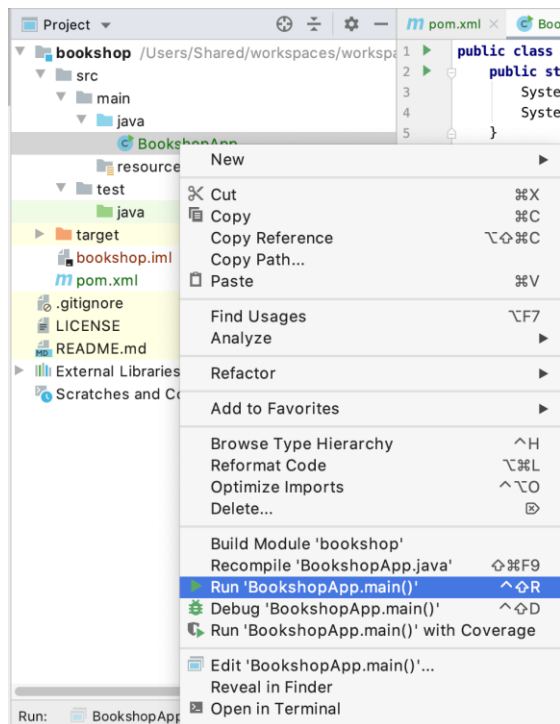
The actual application itself is exactly the same as for the previous lab so you can either retype that code or copy it into this new class.:

```
public class BookshopApp {  
    public static void main(String[] args) {  
        System.out.println("Welcome to the Java Bookshop");  
        System.out.println("=====");  
    }  
}
```

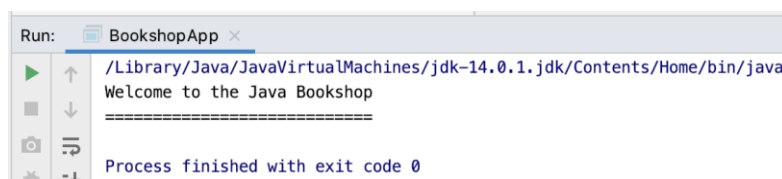
Save the file once you have made the change.

Step 2: Run the Program

Next select the class in the Project view and from the right mouse menu select the Run option with the name of your class and main() on it. This is done is exactly the same way as the last lab for example:



You should now see the output of your application shown in the Run window at the bottom of the IDE:



Lab: Bookshop Classes

The aim of this lab is to define a set of classes that will be used to represent

- Authors
- Publishers
- Addresses
- Books

For the moment we will assume that books are only written by a single author.

The properties for each class are given below:

- Author – name, address
- Address – number, street, city, county, postcode
- Publisher – name, address
- Book – title, price, author, publisher

Make sure that each class has appropriate setters and getters for the instance variables (properties) they hold.

The `Book` class should also allow a sale *discount* to be set and for a sale *price* to be *calculated* based on this discount.

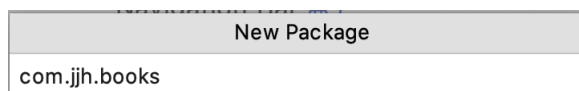
You should also consider overriding the `toString()` method.

Note that this means that we will be using multiple classes. For this to work within a Module inside IntelliJ we need to create a package to hold all the classes.

To do this select the `src/main/java` node in the Project view and from the right mouse menu select `New>Package`:

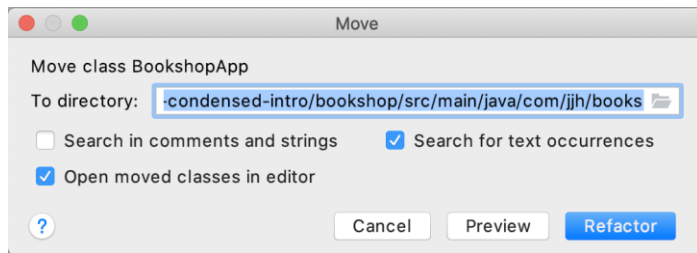


This will display the New Package dialog. Enter a package name of the form `com.<domain>.<project>.domain`, for example:



Then press enter. You should now have a new package listed under the blue `java` node in the project view.

Now drag the BookshopApp class into this package. When you do this you will see the Move dialog displayed; click on the Refactor button:



An example of the code that these classes should support is given below:

```
package com.jjh.books;
```

```
public class BookshopApp {
    public static void main(String[] args) {
        System.out.println("Welcome to the Java Bookshop");
        System.out.println("=====");

        Address authorAddress = new Address(10, "High Street",
            "Any Town", "Somerset",
            "SA1 23Z");
        Author author = new Author("Pete Smith", authorAddress);

        Address publisherAddress = new Address(1, "Main Street",
            "Some City", "Kent",
            "KA1 43A");
        Publisher publisher =
            new Publisher("Tech Books Publishing Ltd.",
                publisherAddress);

        Book book = new Book("Java Unleashed",
            author, publisher, 15.95);
        System.out.println("Book: " + book);

        System.out.println("Calculating the Sales Discount price");
        book.setSaleDiscount(10.0);
        System.out.println("Sale price of book: " +
            book.calculateSalePrice());
    }
}
```

A sample of the type of output generated by the above is given blow:

```
Welcome to the Java Bookshop
```

```
=====
```

```
Book: Book [title=Java Unleashed, author=Author [name=Pete Smith, address=Address
[number=10, street=High Street, city=Any Town, county=Somerset, postcode=SA1 23Z]],
publisher=Publisher [name=Tech Books Publishing Ltd., address=Address [number=1,
street=Main Street, city=Some City, county=Kent, postcode=KA1 43A]], price=15.95]
```

Calculating the Sales Discount price

Sale price of book: 14.354999999999999

Don't forget to add these classes to Git – you can do this as you are working; you don't need to wait until the end.

Lab: Inheritance

The aim of this lab is to explore inheritance.

The Bookshop will eventually sell a number of different things such as books, DVDs, CDs, Games etc.

All of these items are considered *products* that the shop can sell.

All these *products* have a *title* and a *price*; however, they may have other attributes such as authors, artists, directors, producers, etc.

Create a new *root* class, called `Product`, that will hold the information and behaviour common to all items sold in the bookshop. As a hint of other things to consider; all items may be put on sale.

Now modify the `Book` class to extend the `Product` class and extend as appropriate.

Consider the `toString` method of the `Book` class, how might you modify that?

The external interface to the `Book` class should not change from the developer's point of view.

Now look at the `Author` and `Publisher` classes. Add a genre to the author class such that an `Author` writes for a particular Genre, we would now create an `Author` using:

```
Author author = new Author("Pete Smith", authorAddress, "Technical");
```

Where `Technical` is a genre.

For the publisher, change the publisher class such that you create a `Publisher` using:

```
Publisher publisher = new Publisher("Bill Smith", publisherAddress, "Tech Books Publishing Ltd.");
```

So that a publisher has an individual name and a organisational name.

Consider the other classes you have created; is there potential for any other examples of inheritance; if not why not?

For example, you could create a `Person` class that has a name and address – how would you use that with inheritance?

Lab: Further Inheritance

The aim of this lab is to explore *abstract* classes and class side behaviour to keep a count of the number of books created.

Step 1: Create Abstract Classes

Examine your classes and identify any that might be considered abstract; these classes might include `Product` and any class you might have used for common aspects of Authors and Publishers.

Make these classes abstract. This will stop anyone making instances of a `Product` etc.

Now add an abstract method `printer()` to the `Product`. It will then be used to *print* a type of product. However, this may mean different things for different types of product; we will therefore require each subclass of `Product` to determine what *printer* means for it and implement its own version.

The signature for this method should be:

```
public abstract void printer();
```

You will now need to implement this method in the class `Book`.

Are there any other class you would want to make abstract – as this would stop anybody creating unwanted or unintended instances of those classes.

Step 2: introduce some class side (static) behaviour

Next add class side (static) behaviour to the `Book` class so that you can keep track of the number of books instantiated. You should provide a method called `getCount()` that will return a count of the number of instances created.

Your application should now look like:

```
public class BookshopApp {  
  
    public static void main(String[] args) {  
        System.out.println("Welcome to the Java Bookshop");  
  
        Address authorAddress = new Address(10, "High Street",  
                                             "Any Town",  
                                             "Somerset",  
                                             "SA1 23Z");  
        Author author = new Author("Pete Smith", authorAddress);  
  
        Address publisherAddress = new Address(1, "Main Street",  
                                                "Some City", "Kent",  
                                                "KA1 43A");  
        Publisher publisher =
```



```

        new Publisher("Tech Books Publishing Ltd.",
            publisherAddress);

        Book book = new Book("Java Unleashed",
            author, publisher, 15.95);
        System.out.println("Book: " + book);

        System.out.println("\nCalculating the Sales Discount price");
        book.setSaleDiscount(10.0);
        System.out.println("Sale price of book: " +
            book.calculateSalePrice());

        Book book2 = new Book("Java For Professionals", author,
            publisher, 12.55);
        System.out.println("\nBook2: " + book2);

        System.out.println("Book Instance Count: " + Book.getCount());

    }
}

```

The output from this in the sample solution is:

Welcome to the Java Bookshop

Book: Book[title=Java Unleashed, price=15.95, author=Author [name=Pete Smith, address=Address [number=10, street=High Street, city=Any Town, county=Somerset, postcode=SA1 23Z]], publisher=Publisher [name=Tech Books Publishing Ltd., address=Address [number=1, street=Main Street, city=Some City, county=Kent, postcode=KA1 43A]]]

Calculating the Sales Discount price

Sale price of book: 14.354999999999999

Book2: Book[title=Java For Professionals, price=12.55, author=Author [name=Pete Smith, address=Address [number=10, street=High Street, city=Any Town, county=Somerset, postcode=SA1 23Z]], publisher=Publisher [name=Tech Books Publishing Ltd., address=Address [number=1, street=Main Street, city=Some City, county=Kent, postcode=KA1 43A]]]

Book Instance Count: 2

Lab: Interfaces

The aim of this lab is to work with Java Interfaces.

For this lab we will create an interface Sales.

This interface will encapsulate all the behaviour associated with a *sales item*; this includes the ability to set the discount and to calculate the discount.

You should make the Product implement the interface – this should be achieved by the methods it already implements.

As before this is a refactoring and the external API to the classes you have been working with should not change.

You should now be able to add the following to the end of your application main method code:

```
Sales salesProduct = book;
salesProduct.setSaleDiscount(10.0);
System.out.println("Sale price of book: " +
                    salesProduct.calculateSalePrice());
```

Once the book is treated as a Sales item; it should not be possible to access any of the Book specific behaviour.

Now define a second interface PrettyPrinter.

This might look like:

```
public interface PrettyPrinter {

    void prettyPrint();

}
```

Now modify the Author and Publisher classes to implement this interface.

This means that you will have to implement this method in those classes.

Now add the following lines to the end of your main method:

```
author.prettyPrint();
publisher.prettyPrint();
```

Lab 8: Collections

The aim of this lab is to work with some of the collections classes.

In this lab we will modify the Bookshop application such that we create a Bookshop class that can hold a list of books.

This class is separate from the BookshopApp you have already created. It should be added to your data package and might have an API that allows books to be obtained from it, for example:

```
Bookshop bookshop = new Bookshop();

for (Book book : bookshop.getBooks()) {
    System.out.println("Book: " + book);
}
```

The books list can be initialised when the class is created; in a real application this might be done by reading information from a database; in this application they will be hard coded.

You can use the code from the previous version of the main method to set up two books in your book list.

Now update the main application class such that the main method looks something like that presented below. Note that the Bookshop class now encapsulates much of the functionality and behaviour of the book shop from the application class that runs the whole thing:

```
package com.jjh.bookshop.app;

import com.jjh.bookshop.data.Book;
import com.jjh.bookshop.data.Bookshop;
import com.jjh.bookshop.data.Sales;

public class BookshopApp {

    public static void main(String[] args) {
        System.out.println("Welcome to the Java Bookshop");

        Bookshop bookshop = new Bookshop();

        for (Book book : bookshop.getBooks()) {
            System.out.println("Book: " + book);
        }

        // if (bookshop.isEmpty())
        if (bookshop.getBooks().size() > 0) {
            // bookshop.getBookAtIndex(0);
            Book book1 = bookshop.getBooks().get(0);
            book1.setSaleDiscount(10.0);
            System.out.println("Sale price of book: " + book1.calculateSalePrice());
        }
    }
}
```

```
        book1.getAuthor().prettyPrint();
        book1.getPublisher().prettyPrint();
    }

    if (bookshop.getBooks().size() > 1) {
        Sales salesProduct = bookshop.getBooks().get(1);
        salesProduct.setSaleDiscount(10.0);
        System.out.println("Sale price of book: " + salesProduct.calculateSalePrice());
    }

}

}
```

You could now add methods to the Bookshop class if you want that will return the number of books or obtain a book at a specific index etc.

Lab: Functions and Streams

The aim of this lab is to use higher order functions and streams to apply `forEach` and `filter` to the books collection.

There are two parts to this lab:

1. Use `forEach` with your list of books to print out each book in turn.
2. Use `filter` to select all books in your bookstore that are over 13.00 in price. You should then print out the results of this filter operation.

Lab: Exception Handling

The aim of this lab is to add *exception handling* to the Bookshop application.

Part One

The first thing we will do is to create a custom exception to be used with the book shop classes. Call this exception `BookshopException` and extend the `RuntimeException` class. This exception class should be able to take an optional message.

Next add validation code to your abstract `Product` class to verify that the `Price` is a positive value; negative values as well as Zero are illegal in the bookshop.

If the validation code fails, then you should throw the `BookshopException`.

Try out your code and validate the you cannot create a book with a negative `Price`.

Part Two

Once you are happy that your validation code works, now add some exception handling code to your main method application code so that it can cope with an exception being thrown.

Lab: JUnit testing

The aim of this lab is to write some JUnit tests for the Book class.

You should create a test class for the Book class.

This test class should at least do the following:

1. Validate that when a book is created it can return the correct title, price, author and publisher.
2. When a discount amount is set that it returns the correct discounted price.
3. Setting a price for the book to be -1.0

If you have time take this further and define a more complete set of tests for the Book class.

Source Code

For examples used in course notes

See java-condensed-intro-master.zip

Alternatively, the GitHub repository for the sample for the course can be obtained using

git clone <https://github.com/johnehunt/java-condensed-intro.git>

For the web page see <https://github.com/johnehunt/java-condensed-intro>

For sample solutions for the labs

See java-condensed-intro-labs-master.zip

Alternatively, the GitHub repository for the sample lab solutions can be obtained using

git clone <https://github.com/johnehunt/java-condensed-intro-labs.git>

For the web page see <https://github.com/johnehunt/java-condensed-intro-labs>