COM1027 Programming Fundamentals

Lab 9

Introduction to Inheritance

Purpose

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

- Define an implementation class based on an Interface.
- Develop classes with inherited constructors.
- Override methods.
- Develop abstract classes.

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`

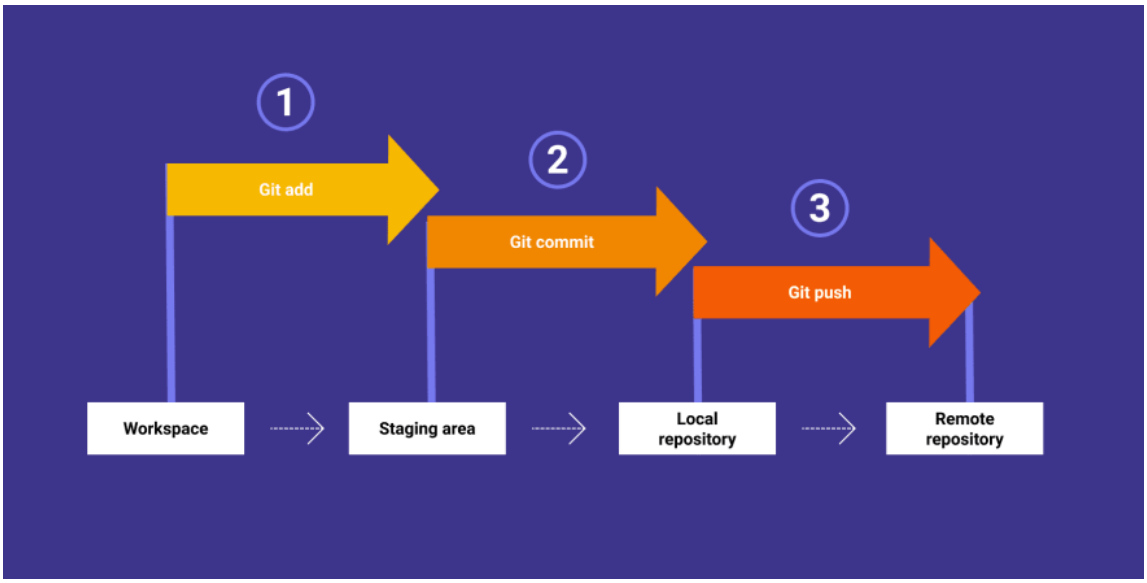
Please ask us lots of questions during the labs, or use the discussion forums on SurreyLearn. All the demonstrators are here to help you get a better understanding of the Java programming concepts.

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
- Commited 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 be creating a hierarchy of classes, that share common attributes and behaviour.

1. Import an existing Eclipse project and set your workspace
From SurreyLearn, download the project called `COM1027_Lab09.zip`. Extract the contents of the project in the Downloads folder. Copy the uncompressed (unzipped) `Lab09` folder to your local repository. **Note:** Do not copy the `COM1027_Lab09` folder. Navigate in the folder, and only copy `Lab09` 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 `Lab09` from your local git repository. Once the files are loaded, select the Eclipse project and click `Finish`.

2. Demonstrating the use of interfaces

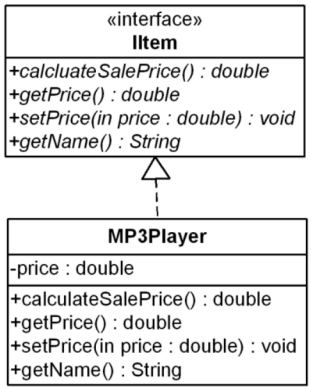
Once imported, look at the `IItem` file in the package `lab9`.

`IItem` is an interface (prefixed with I as a style convention). This was done using "`New`" > "`Interface`" (not "`New`" > "`Class`").

Concept Reminder:
The `interface` defines the behaviour we expect for any class which defines an item in a shop.

Note: The interface **does not** say ***how*** the behaviour is implemented and hence it does not specify what information should be stored within an object which implements `IItem`. Thus, on implementation of the interface, all of its methods must be overridden.

We have defined the class `MP3Player` to be an implementation of the `IItem` interface.



`MP3Player` is a typical concrete class with attributes and instance methods. However, by implementing `IItem`, the class is guaranteed to include the four interface methods. The interface methods are written in *italics* within the UML diagram, and in this example, they are also provided in the concrete class:

- calculateSalePrice
- getPrice
- setPrice
- getName

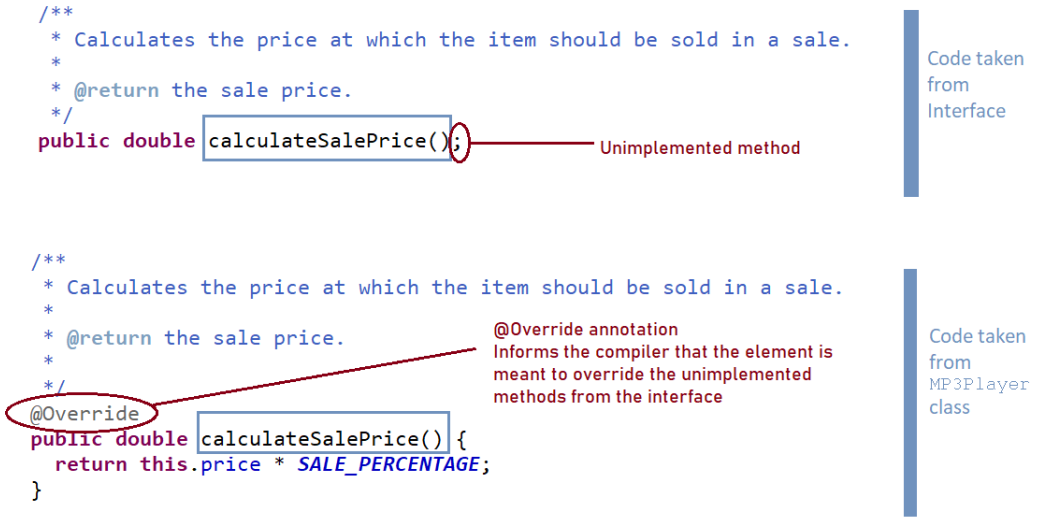
These methods are structured in a similar way as any other user-defined method, with the addition of the `@Override` annotation.

```
  @Override
public double calculateSalePrice() {
  return this.price * SALE_PERCENTAGE;
}

@Override
public double getPrice() {
  return this.price;
}
```

```java
@Override
public void setPrice(double price) {
  this.price = price;
}

@Override
public String getName() {
  return "MP3 Player";
}
```

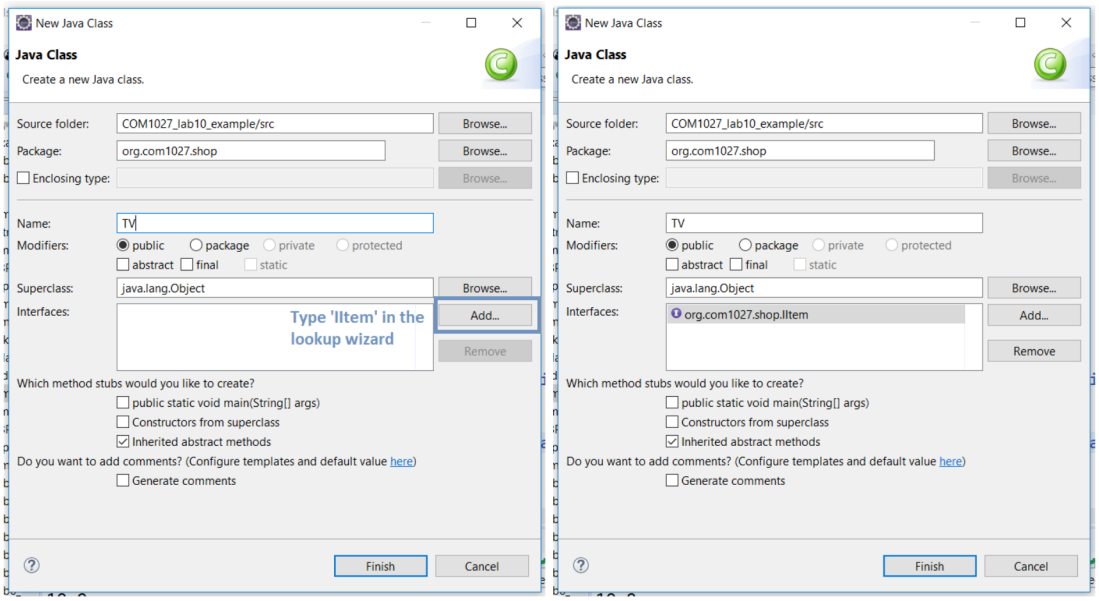
Inspect the code in the `MP3Player` class and in the `IItem` interface.

The interface is a completely "abstract class" and its methods do not have an implementation (no method body). To access the interface methods, the `MP3Player` class `implements` the interface, by overriding the unimplemented methods and providing context.

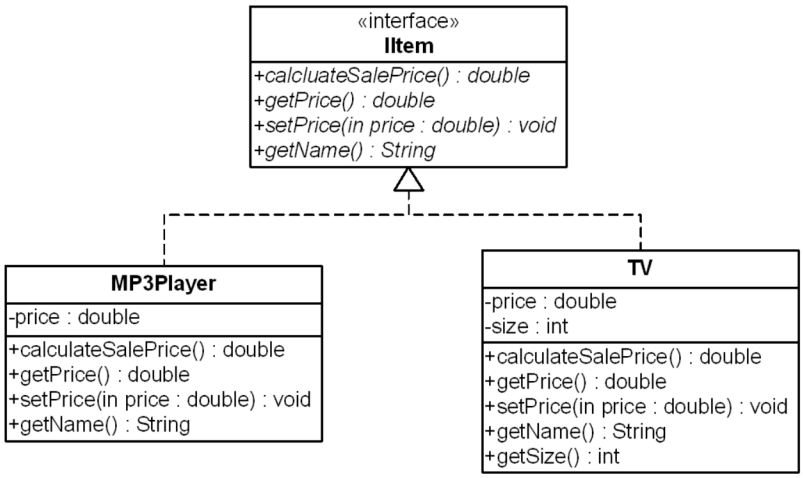


3. Defining your own class based on an interface

Using the structure provided in the `IItem` interface and `MP3Player` class, define a `TV` class which implements the `IItem` interface as per the revised class diagram. Eclipse can help you fill in the required methods from the interface if you enter the implementing interfaces in the New Java Class wizard via the "Add" button.



In the new `TV` class you will see 4 skeleton methods which have been automatically generated to match the `IItem` interface. The UML class diagram for these relations will look like:



Add the required attributes and fill in the method detail. Note that a TV is discounted to 80% of its normal price when it is in a sale. The name of the TV set should include its size, such as "TV (26")". To add a speech-mark in your text you can use the escape sequence (**\"**).

Provide a parameterised constructor which sets the size and price. There is no need to provide a setter for the size (because it will not change). However, you will need the `getSize` method.

Explanation: A class which implements an interface can have more attributes and methods than those required by the interface. **However, the class must provide the interface methods.**

4. Demonstrating the use of super constructors
In this example we will look at the impact of inheritance on constructors. Look at the code in the `lab9` package for the `Animal`, `Duck`, `Mallard`, and `Pond` classes:

- `Animal` which is the superclass.
- `Duck` which is a subclass of Animal.
- `Mallard` which is a subclass of Duck.
- `Pond` which just contains a main method.

`Duck` and `Mallard` are subclasses. Note the use of the extends keyword in the definition of the classes:

```
public class Duck extends Animal{
public class Mallard extends Duck {
```

Run the `main` method in the `Pond` class. What do you see on the console? You should see three messages which convey which constructors have been executed in a specific order, followed by the name of the duck.

Why are the constructors executed in a specific order?

In the next series of steps we will make subtle changes to each class constructor and ask you to run the code again after each change.

Comment out `super(name);` in the `Mallard` class. The class should fail to compile.

> **Concept Reminder**: The reason for this is that if you do not call any `super` constructor (a parameterised one or a default one) explicitly then the Java compiler will call the default `super()` implicitly. If a default constructor does not exist in the superclass (`Duck`), you will get a compilation error as is the case here.

Now try putting `super(name)` after the `System.out.println("In Mallard constructor");` line of code. You should get another compilation error.

Explanation: Note that in a constructor any call to `super` must be the first line of code in a constructor method. **Why?**

Now put the code back as you found it originally, which is as follows:

```
public Mallard(String name) {
    super(name);
    System.out.println("In Mallard constructor");
  }
```

Replace `super(name);` in the Mallard class with `super();`. You should get another compilation error complaining about the lack of a default constructor in `Duck`. Do not remove the empty call to `super();`

Add a default constructor to Duck:

```
  public Duck() {
    super();
  }
```

You should get another compilation error, this time complaining about the lack of a default constructor in `Animal`. Do not delete the default constructor from the Duck class.

Add a default constructor to `Animal`:

```
public Animal() {
  super();
}
```

Everything should now compile.

Run the main method in the `Pond` class. **What do you see on the console?** You should see a different set of messages from before. **What is the name of the duck? Why is the name not correct?**

Explanation: The `Duck` default constructor is called from the parameterised constructor in `Mallard`. The string value input by `Mallard` will be ignored and the name will be assigned with the value null.

Next comment out the `super();` in the `Mallard` constructor.

Run the code again as a Java Application. **Do you still get the same behaviour?**

Explanation: The answer is yes. Since there is no explicit call to `super` the compiler will put one in and since there are now default constructors in the underlying classes they will now be called automatically.

Remove all of the default constructors we have just added to `Duck` and `Animal` and make sure the `Mallard` constructor looks like this:

```
public Mallard(String name) {
  super(name);
  System.out.println("In Mallard constructor");
}
```

Everything should be back as it was.

5. Using other constructors and super methods

In this example, we will be working with the same classes as before (`Mallard`). Add a more sensible default constructor to `Mallard`:

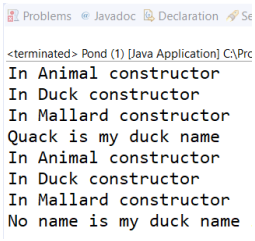
```
public Mallard() {
  this("No name");
}
```

Explanation: This constructor does not call a default or parameterised super constructor. Instead it calls the existing `Mallard` parameterised constructor with a default name parameter.

Add the following to the main method in Pond:

```
Mallard nameless = new Mallard();
System.out.println(nameless.getName());
```

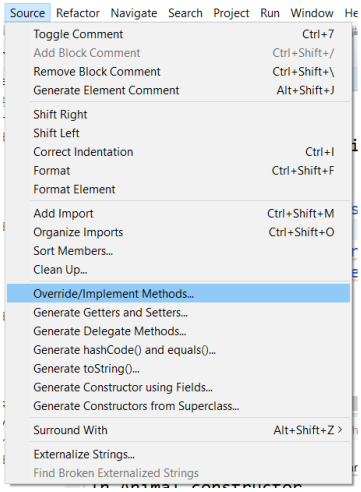
You should get the following output when you run the code:



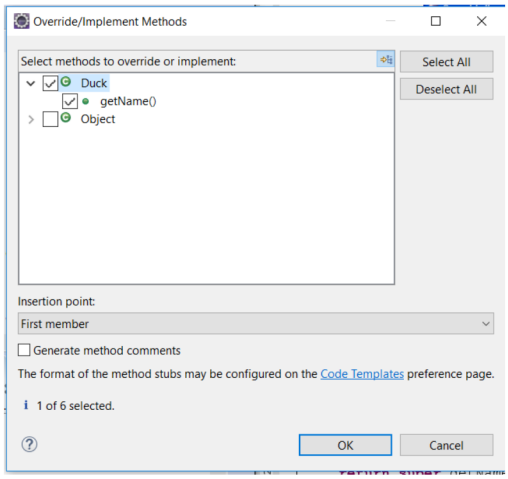
Explanation: When an object is created it is given the name "No name". Initially, the `Mallard` default constructor is called which in turn calls the parameterised `Mallard` constructor which then cascades through the hierarchy of constructors. **Note:** There is no `getName` method in Mallard and so it looks for the one in the `Duck` class.

We are now going to add a method to the `Mallard` class to demonstrate how overriding methods works.

In the Mallard class, select "Source", "Override/Implement Methods":



Select the method `getName` and press "OK":



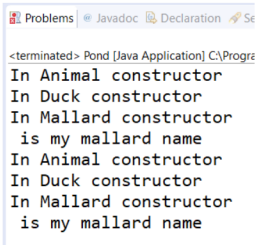
Eclipse will add the following code to the `Mallard` class:

```
  /**
   * @return
   *
   */
5 @Override
  public String getName() {
    // TODO Auto-generated method stub
    return super.getName();
  }
```

Replace the body of the method so that it looks like this:

```
@Override
  public String getName() {
    return " is my mallard name";
  }
```

Run the main method in the Pond class. You should see the following:



Explanation: When the mallards name is being displayed via the `getName` call we are not retrieving the name of the animal from the `Animal` class. This means that the call to:

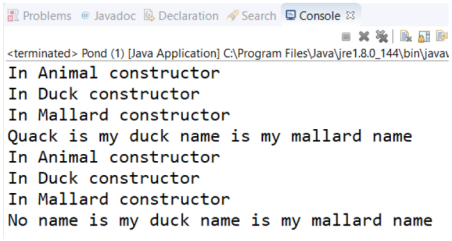
```
mallard.getName()
```

is calling the `getName` method in `Mallard`, but this method is not calling the super method `getName` in `Duck` which in turn is not calling the super method `getName` in `Animal`. **Phew.**

To correct this we need to make sure that the super method is called in our overridden `getName` method. Change the method to read:

```
return super.getName() + " is my mallard name";
```

You should see the name restored to its rightful place:



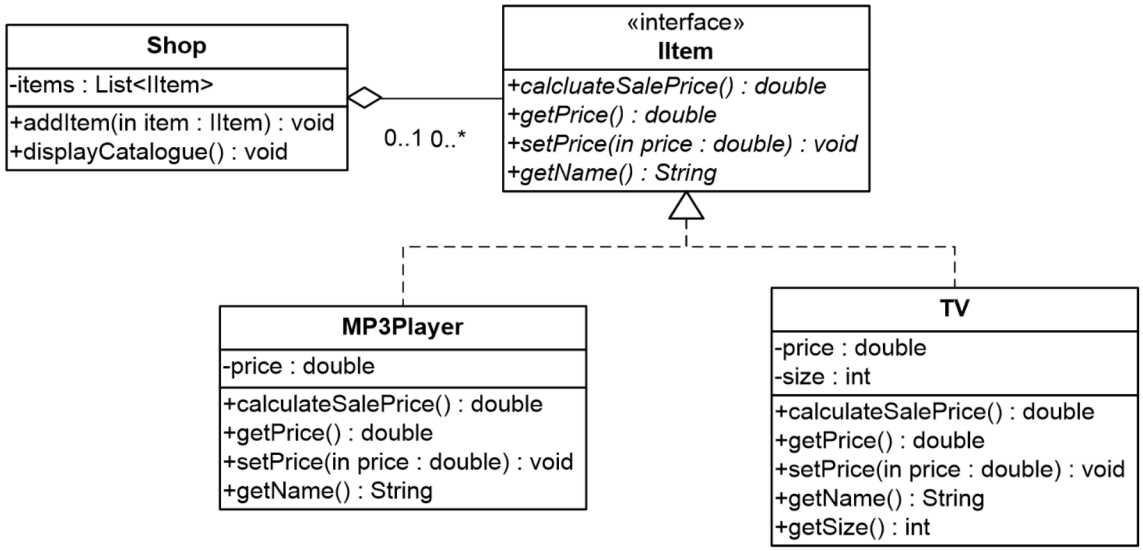
Explanation: By adding in the `super.getName()` we are making sure that the first thing done in the method is everything defined in the super `getName` method. This creates the return String as follows:

1. In the `Animal getName` method:
   `return this.name;`
   `Quack`
2. In the `Duck getName` method:
   `return super.getName() + `` is my duck name";`
   `Quack is my duck name`
3. In the `Mallard getName` method:
   `return super.getName() + `` is my mallard name";`
   `Quack is my duck name is my mallard name`

Concept Reminder: We do not need to call a super method in an overridden method. We could choose to ignore any super methods, but this is bad practice. The super method call could be placed in any part of the overridden method. Unlike super constructors, it does not matter when we call the super method.

6. Abstract classes

We are now going to look at an improved version of our first example. The final class diagram from the first example looked like this:



Concept Reminder: Inheritance is designed to reduce duplication. Using class hierarchies, duplicated code can be placed in a superclass and used by a subclass.

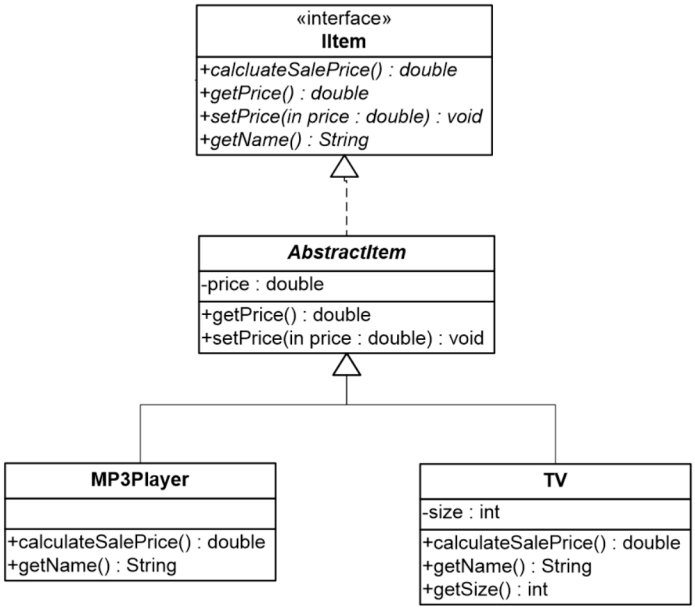
In this class diagram we have duplicated code because both `MP3Player` and `TV` provide implementations of the price field and `calculateSalePrice`, `getPrice`, `setPrice` and `getName` methods.

If we want to remove duplicated code, where do we put it?

We cannot put it into the `IItem` interface because we are not allowed to put implementation (method bodies) into an interface. We could put it into a concrete class `Item` instead of having an `IItem` but this would mean that we would have to supply all of the implementation.

However, we could argue that an `Item` class cannot have methods `calculateSalePrice` and `getName` because they can only be defined within the classes `MP3Player` and `TV`.

The answer is an abstract class which is half way between an interface and a concrete class.



Explanation: In this new class diagram we have added an `AbstractItem` class which implements the `IItem` interface. However, you will note that it only implements `getPrice` and `setPrice`. The concrete classes `MP3Player` and `TV` now extend (**not implement**) from `AbstractItem` and they complete the implementation of the `IItem` interface by providing implementations of the `calculateSalePrice` and `getName` methods. The `TV` class is not provided, as you will be required to define it in one of the exercises.

Look at the code in the package `lab9_abstract`. You can note that the `IItem` interface has not changed.

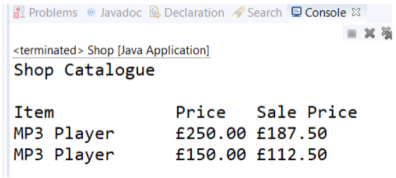
The `AbstractItem` class is abstract and defines a single field price plus two of the interface methods `getPrice` and `setPrice`.

It also defines a single parameterised constructor which allows a price to be specified.

The `MP3Player` class is now much simpler and just defines a constructor (which delegates setting the price to the super constructor in `AbstractItem`) and the two missing interface methods.

Note: Now that the price field is private to `AbstractItem`, we can no longer directly access price in `MP3Player`. Therefore, in the `calculateSalePrice` method the getter `this.getPrice()` is used to obtain the price.

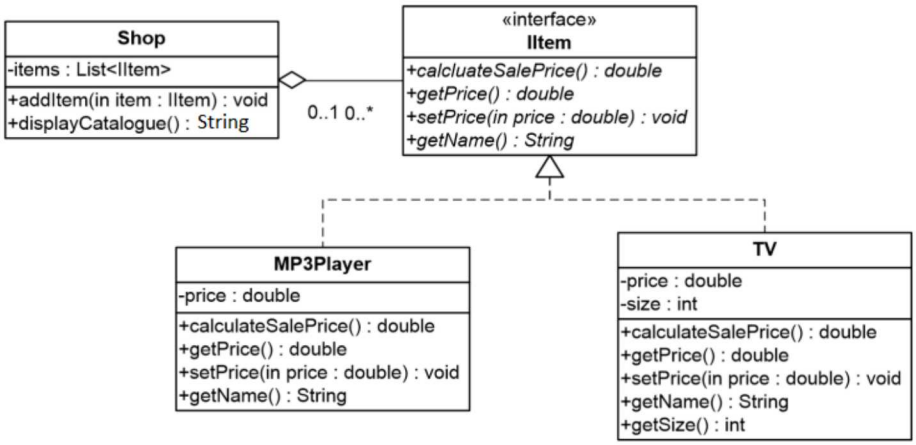
Run the code in the `Check` class as a Java Application. You should see the following output:



Concept Reminder: We are using inheritance in this example to improve our implementation by reducing duplicated code. The interface to the classes and the function of the classes remains unchanged and therefore `Shop` does not need to change.

Exercise 1 (3 Marks)

This exercise requires you to modify a class called `Shop` as follows:



The `Shop` class defines a shop which can contain items of type `IItem` for sale. The Shop class therefore contains an attribute which is a list of `items`, a method `addItem` to add an item to the shop and a method `displayCatalogue` to display the current shop catalogue. For testing the output, a `main` method has been added and defined at the end of the class.

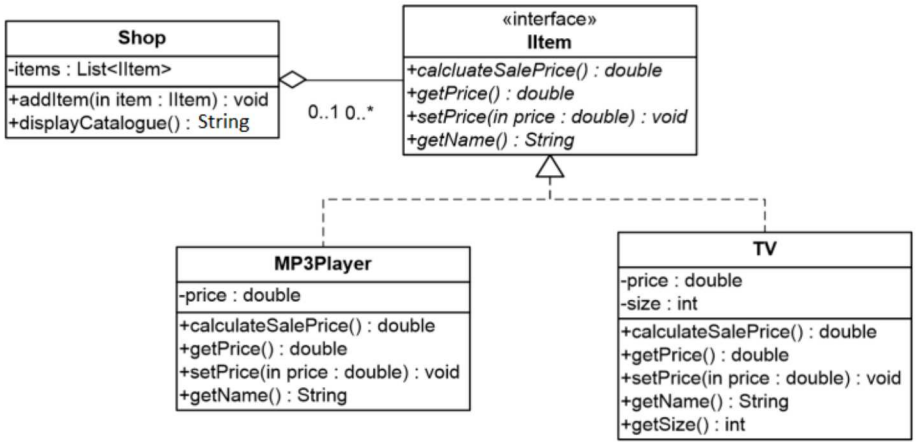
For this exercise, you will be using the `lab9_exercise1` package and you are expected to demonstrate your understanding of constructors, getters and lists. You will need to copy the `IItem`, `TV` and `MP3Player` files from the `lab9` package.

- Create the field `items` and set it to its default value. *Note: The field makes use of the `IItem` interface.*
- Define the list as an `ArrayList` within a default constructor
- Define the `addItem` method with appropriate validation. Use the pre-defined JUnit tests to support you in identifying an appropriate exception.
- Define the `displayCatalogue` method as a function that returns the output in this format:
  `Item\t\tPrice \tSale Price`
  where "\t" is the escape sequence for the tab space.

After completing the functionality of the class, you can check your work by creating a `main` method in the `Shop` class, to return the following output:

- Add two TVs to the shop's catalogue (by adding them in your `main` method and run the program again as a Java Application. You should get something like this:

Explanation: The `Shop` class relies upon the `IItem` interface to specify what items can be added to the `Shop`. This means that we can create any class which implements `IItem` and add it to the `Shop`.



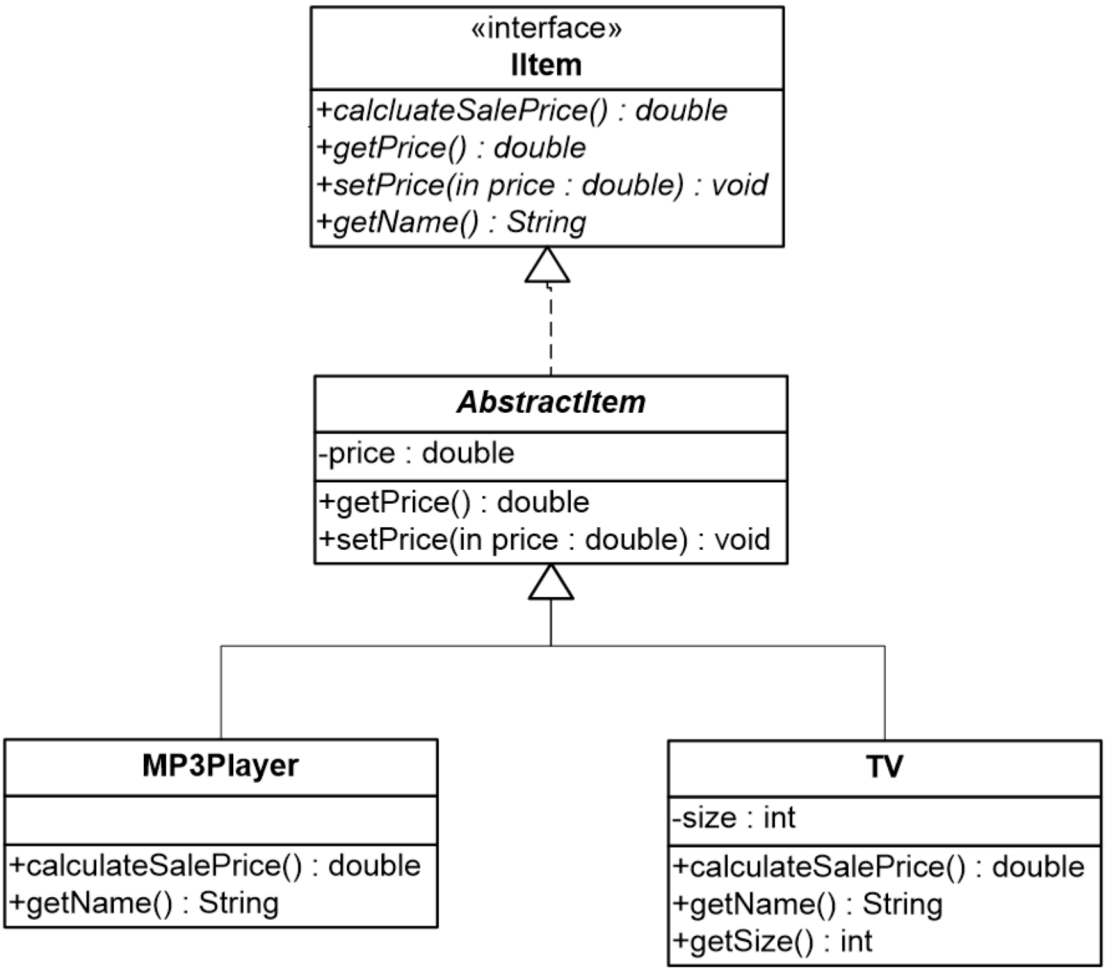
To test the functionality of the code, and whether it meets the expectations of this exercise, right-click on the project name and select `Run As > JUnit Test`.

Go to next page

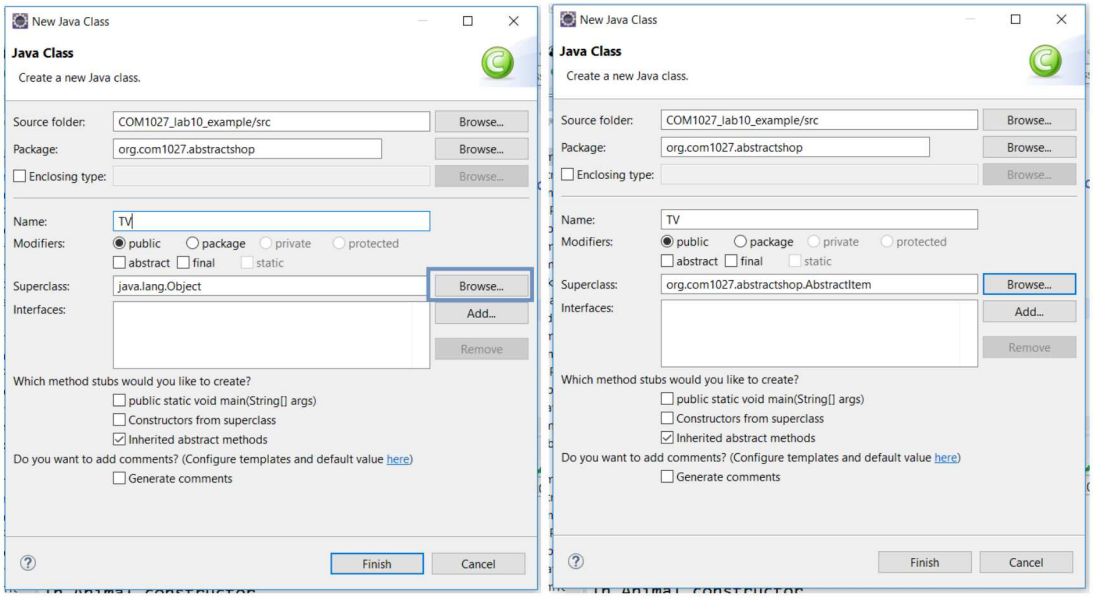
Exercise 2 (2 Marks)

This exercise requires you to create a new class using your understanding of abstract classes, and their methods.

For this exercise, you will be using the `lab3_exercise2` package. Copy the `AbstractItem`, `MP3Player` classes from the examples (`lab9 abstract`) as well as the `IItem` interface. Then create a new class called `TV` that follows this UML diagram:



- Define a `TV` class which extends `AbstractItem` as per the revised class diagram. Eclipse can help you fill in the required methods from the class if you enter the superclass in the New Java Class wizard via the "Browse" button.



Note: The `TV` class does not need to implement the `IItem` interface because `AbstractItem` already implements the interface.

Therefore, when you press "Finish", Eclipse will provide you with a template for the new class with the two missing `calculateSalePrice` and `getName` methods.

- Add the required size attribute and getter and fill in the method detail. Note that a TV is discounted to 80% of its normal price when it is in a sale. The name of the TV set should include its size, such as "TV (26")".
- Provide a parameterised constructor which sets the size and price. There is no need to provide a setter for the size (because it will not change). However, you will need the `getSize` method.

  Hint: Use the super constructor to set the price first before setting the size field.
- Create a `Check` class similar to the one in the examples. Add a `main` method within that class to test your code. In the `main` method, add two MP3Players and two TVs to the shop's catalogue and run the program again as a Java Application. You should get something like this:

```
Shop Catalogue

Item            Price   Sale Price
MP3 Player      £250.0  £187.5
MP3 Player      £150.0  £112.5
TV (26")        £499.0  £399.2
TV (42")        £999.0  £799.2
```

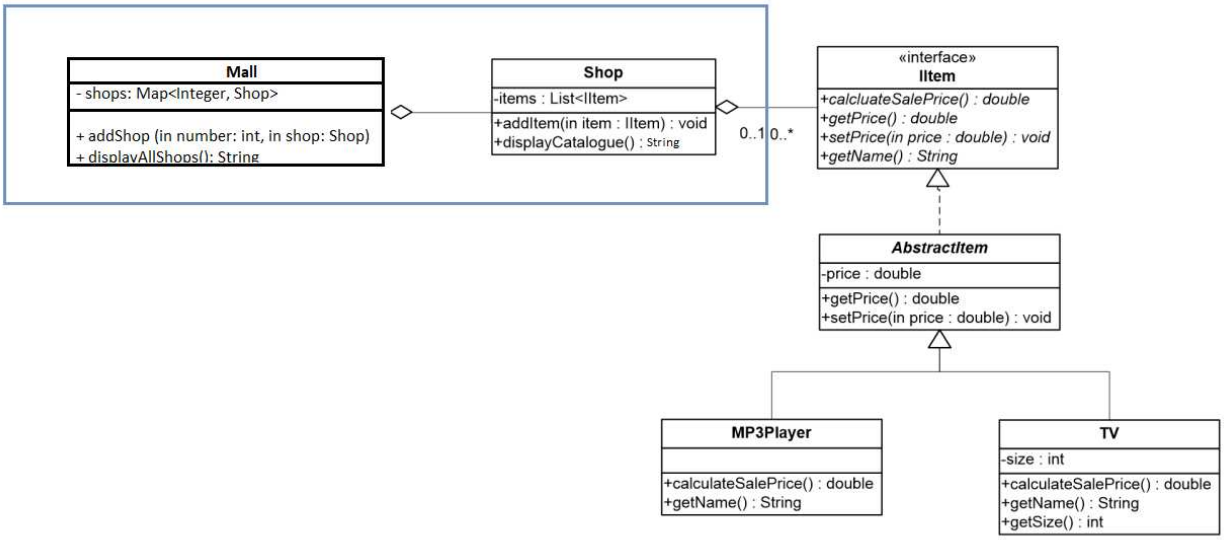
Before moving to Exercise 3, test your code by right-clicking the project name and then `Run As > JUnit Test`. Once all the tests for the `TV` class pass, you can start working on Exercise 3.

Stretch & Challenge Exercise (5 Marks)

This exercise requires you to use the `TV`, `MP3Player`, `AbstractItem` and `IItem` files with minimum support (use the JUnit tests to guide you instead). The new class will require the use of lists and maps, as well as repetition statements.

For this exercise, you will be using the `lab9_exercise3` package. Copy all the relevant classes from `lab9_exercise2` to the `lab9_exercise3` package.

- Create a new class called `Shop` as per the UML diagram. You may be able to use the previously-defined `Shop` class from Exercise 1 (with minor changes).
- Define a new class called `Mall` that has a map of all the shops. This are stored with unique numbers that only get assigned to the shops when added into the Mall's system.



- When a new shop is added in the map, the number should be validated. The number can only be zero or above. Add a validation to check if the parameter holds a value below 0 (zero).

- Some of the methods should throw exceptions. How do we identify those methods? Check the JUnit files to identify what type of exceptions are required.

Note: No exceptions should be caught in a try/catch statement.

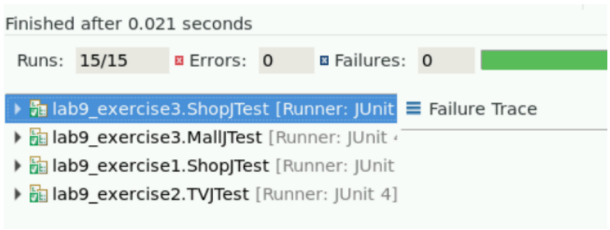
- You can test your code, by creating a `Test` class with multiple objects or run the predefined JUnit tests.

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:

```
Running lab9_exercise3.ShopJTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running lab9_exercise2.TVJTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running lab9_exercise1.ShopJTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Results :

Tests run: 15, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time: 4.303 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.