Exercise 8: Inheritance

Lab Objectives

- Learn how to:
    - Extend your classes using inheritance
    - Override methods appropriately
    - Make use of polymorphism
    - Override equals(...) in your own inheritance hierarchies

Associated Reading:

Horstmann,

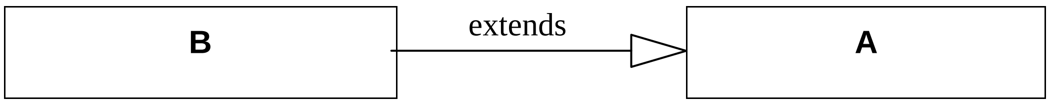
Chapter 9, Sections 9.1 - 9.4: Inheritance

Extending classes using inheritance

In OO programming, there are two fundamental ways of reusing functionality of existing classes; the first is by composition/aggregation where a new class is made up of existing classes, the second is by inheritance, where a new class is based on an existing class, with modifications or extensions.

Inheritance allows code to be shared between classes and services to be provided that can be used by multiple classes. By organising related classes that share common behaviour into a hierarchy, designers can avoid duplication and reduce redundancy.

- In inheritance, we say that a one class **extends** another class. The subclass extends the superclass. "B extends A", "B inherits from A", "B is an A".

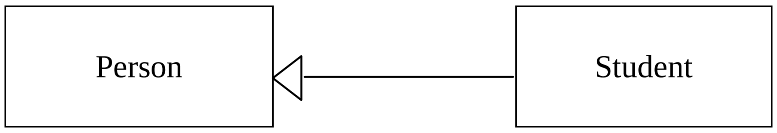


- The subclass inherits all of the superclass' fields and methods.
- The subclass can add more fields and methods. So it becomes more specialized.
- The subclass can override (i.e. redefine) the inherited methods, or leave them unchanged.
- In Java a class can only have one superclass, (but it can have many behaviours by implementing many interfaces - not yet covered).
- The first line of a subclass' constructor is always a call to the superclass' constructor (because the superclass' fields must be initialized). If the programmer does not explicitly call the superclass' constructor then its no-argument constructor is automatically called.
- Classes that might be extended should always have a no-argument constructor.
- **super()** is the code for calling the no-argument constructor of a superclass.

- **super** is a keyword that refers to the superclass object (as opposed to **this**, which refers to 'self'). It can be used to access the superclass' methods, e.g. **super.toString()**.
- A superclass' ***private*** fields **cannot** be accessed directly by a subclass.
- ***protected*** fields can be accessed directly by subclasses, although it is not usually recommended using this scope.

Notes - The Inheritance association:

- *Inheritance* (Extending and specializing a class) **"is-a"**
- Illustrated in UML class diagrams using the following association symbol:

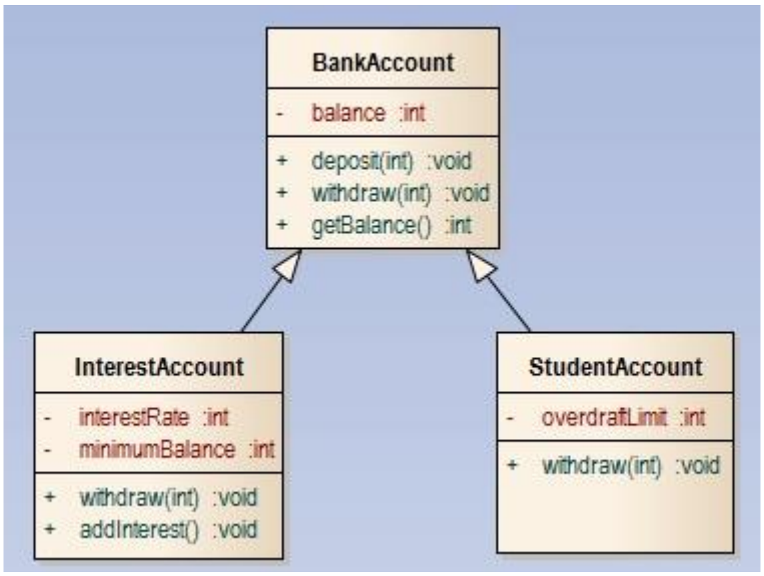


Inheritance
("is-a" or "extends").
A student "**is a"** person.

Please continue to the next page...

Question 8.1 To get started with this lab exercise, inside your "EclipseWorkspaces" folder create a new folder called "**LabExercise8**", then open up Eclipse, selecting this as your workspace. Following this, create a new Java Project called "**bankaccounts**" and then ensure the classes "**BankAccount**", "**InterestAccount**" and "**BankAccountTest**" are in it - you may download them from blackboard.

Using the following specification, see how you can create your first inheritance hierarchy:



a) Study the code within the **BankAccount** superclass and the **InterestAccount** class which extends it. In InterestAccount take note of how the constructors firstly call a parent constructor, the overridden withdraw(...) method and how there are get and set methods and a toString() that are not listed in the UML class diagram.

b) Implement the **StudentAccount** class ensuring it extends BankAccount and overrides its withdraw(...) method, such that money is only withdrawn if the *overdraftLimit* is not exceeded. For example if the overdraft was set to 1000 then the balance should never go below -1000. You should also have a get and set method for the overdraft field and a toString() method. **Hint**: Look at InterestAccount to help guide you.

c) Add some more code to the **BankAccountTest** program so that a StudentAccount object is created and its methods called, then add at least one instance to the ArrayList so that it is processed within the for-each loop.

Notes - Using the super keyword:

- The `super` keyword is most obviously used to call the constructor of a superclass. It can also be used to invoke any method within a superclass. For example `super.toString()` will invoke the toString() method in a superclass.
- When you want to call a method of the superclass, that has been overridden in the subclass, it is necessary to use the `super` keyword, e.g. `super.withdraw(amount)`, this is typically done when a subclass method overrides the superclass implementation by adding functionality, but then also invokes the original method as well. The functionality for the withdraw() method in InterestAccount can reuse that in BankAccount but also add more specialised behaviour.

- There are other occasions when it is optional to use the `super` keyword, for example InterestAccount does not override the deposit(...) method specified in BankAccount. When it needs to use it within the addInterest() method, either `super.deposit(amount)` or simply `deposit(amount)` would be acceptable. This is because public methods of the superclass are automatically inherited. In this circumstance, some developers like to use `super`, to make it explicitly clear that the method belongs to the superclass. The only issue that could arise here is if at a later stage, the deposit(...) method was overridden, each occasion where it was called within that class would need to be checked to ensure either the overridden version or that of the superclass was called as appropriate.

Notes - Inheritance and the toString() method:

- A toString() representation should consist of the class name and the names and values of the instance variables. If you want it to be usable by subclasses of your class, you should use the getClass().getName() methods rather than hardcoding the class name, e.g.

```
return this.getClass().getName() + ":[balance=" + balance + "]";
```

- In a subclass, which will also override toString(), the parent's toString() method should firstly be called, followed by the subclass instance variables, e.g.

```
return super.toString() + "[interestRate=" + interestRate + ", minimumBalance=" + minimumBalance + "]";
```

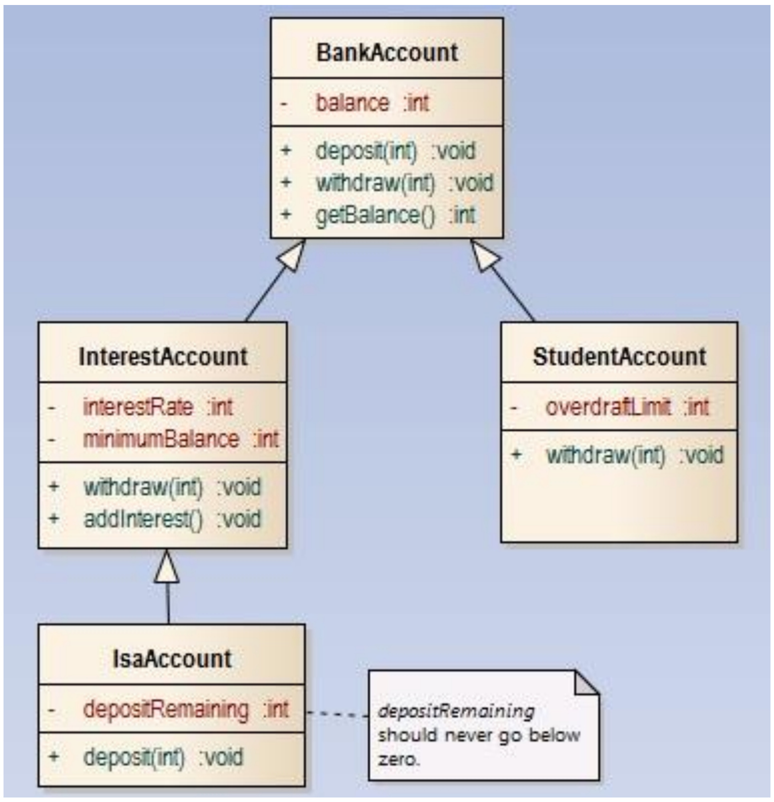
- The result is that the correct class name is output, matching that of the object for which the method is invoked, e.g.

```
OUTPUT:
BankAccount:[balance=100]
InterestAccount:[balance=3000][interestRate=5, minimumBalance=1000]
```

- The brackets are also helpful in showing which variables belong to the superclass.

Please continue to the next page...

Question 8.2 [✸Portfolio✸] Continue with the application you created in the previous question by implementing an "**IsaAccount**" class that extends InterestAccount. An IsaAccount 'is-a' InterestAccount and therefore also 'is-a' BankAccount. It shares their features but adds its own specialized functionality, in that there is a *depositRemaining*. This instance variable is initialized with an amount and any deposits will result in it being reduced. It should never go below zero.



a) Implement the **IsaAccount** class ensuring it extends InterestAccount and overrides the **deposit(...)** method (that it would automatically inherit from BankAccount), such that every deposit is deducted from the *depositRemaining* and is only carried out if the resultant value is zero or greater. For example if *depositRemaining* was set to 1000 then 600 was deposited, it would reduce to 400. If a user then attempts to deposit 401, it should not be allowed as this would result in the limit being exceeded.

b) Add some more methods **getDepositRemaining()**, **resetDepositRemaining()** that resets it to a sensible fixed amount (e.g. 5000) and a suitable **toString()**.

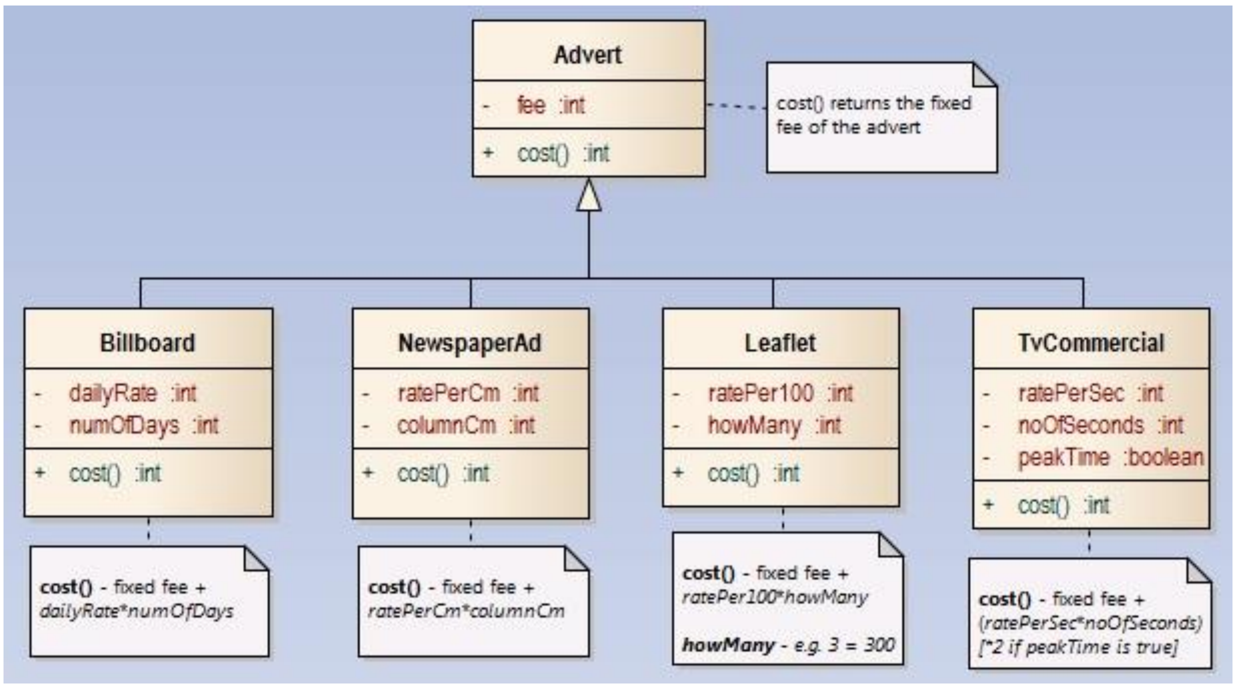
c) Add some more code to the **BankAccountTest** program so that an IsaAccount object is tested properly, then add an instance to the ArrayList so that it is processed within the for-each loop.

d) As this is a portfolio question, you should fully document the IsaAccount class with Javadoc, i.e. class header, constructor/method description and tags where appropriate.

Question 8.3 Create a new project called "**advertising**" and then using the combination of the following written specification and UML class diagram, build the inheritance hierarchy to store details of advertising campaigns.

An advertising campaign consists of a set of advertisements. Adverts can be of several types, for example: a roadside Billboard, a Newspaper Ad, TV Commercial, or Leaflet. Each type of advert has a cost associated with it: a fixed fee to cover materials, production staff and media costs, and then variable costs for buying advertising time and space depending on the type of advert. An advertising campaign consists of one or more advert types, their respective costs are calculated as follows:

- **Billboard**: A poster is hired for a number of days. There is a daily rate.
- **Newspaper Ad**: An ad column has a size in cm. There is a rate per cm.
- **Leaflets**: A leaflet batch is ordered in multiples of 100. There is a rate per 100.
- **TV Commercial**: A commercial lasts a number of seconds. There is a rate per second. The rate is doubled during peak times.



You will notice there are no 'get' or 'set' methods or toString() methods listed in the UML class diagram - this is common and for the purpose of this example you may leave out the 'get' and 'set' methods, and use constructors to set your adverts' values. Therefore each class should have a default no argument constructor and then a custom constructor that allows all fields to be initialised. You should however develop a **toString()** method for each class.

a) Start off by implementing the **Advert** superclass, this is a simple class that has a single attribute to store the fixed fee of an advert and a cost() method, which returns that fee.

b) Next, implement one of the subclasses, **Billboard**. This class overrides the cost() method by returning the fixed fee specified in its superclass, i.e. super.cost(), added to the cost of the billboard advertisement, based upon its two fields.

c) Create a test program called **AdTest** and within the main() method, create an ArrayList of type Advert and add some Advert and Billboard objects, initialised with sensible values. The within a for-each loop output the toString() and accumulate the cost of each advertisement, such that following the loop, the total cost of all advertisements is output. **Note**: You will be using the technique of polymorphism enabled by dynamic method lookup at runtime that will ensure the appropriate cost() and toString() methods are invoked.

d) Implement the other 3 subclasses: **NewspaperAd**, **Leaflet** and **TvCommercial**. They are largely similar in design to Billboard with the exception of TvCommercial, that has its rate doubled during peak time.

e) Add some more code to the **AdTest** program so that these newly implemented subclasses are added to the ArrayList and processed within the for-each loop.

Notes - Polymorphism and inheritance:

- When multiple classes share common features, they may all extend the same superclass. They may then either inherit or override a particular method from the superclass. It is possible to process these objects with different behaviour in a uniform way - a technique called polymorphism.
- This is because each object is guaranteed to either inherit or override the method. But how does the Java virtual machine know which method to invoke? This is achieved via dynamic method lookup, which finds the exact class type of the object instance and then either executes the overridden version of the method or that inherited by its superclass.
- The technique of polymorphism is also relevant to and can be used with Interfaces, which you will be working with in the coming weeks.

Question 8.4 For each of the classes you have developed during this lab exercise, you have not yet overridden the equals(...) method. In the previous lab exercise, you learnt how to override the equals(...) method to test the equality of objects. The principle is very similar when doing so in your own inheritance hierarchy, with a slight difference:

a) Go to your **BankAccount** class and copy in the following equals(...) method:

```
@Override
public boolean equals(Object obj) {
      if ((this.getClass() != obj.getClass()) || (obj == null) )
            return false;

      BankAccount other = (BankAccount) obj;

      return this.balance == other.balance;
}
```

b) The logic within the above implementation of the method should look identical to that you have tried before. Now let's override the equals(...) method in a subclass. Go to your **InterestAccount** class and copy in the following code:

```
@Override
public boolean equals(Object obj) {
      if (!super.equals(obj))
            return false;

      InterestAccount other = (InterestAccount) obj;
```

```
                return this.interestRate == other.interestRate
                        && this.minimumBalance == other.minimumBalance;
        }
```

c) **Note above**: In your subclass, you firstly call equals(...) within the superclass, this will cause the class type and null checks to occur, as well as checking the instance variable(s) of the superclass, in this case *balance*. It is therefore not necessary to check the class type and object reference (for null) again in the subclass, as this is done by the superclass. Instead, simply cast the object into the subclass type (e.g. InterestAccount) and then check the instance variables of the subclass.

d) Implement a copy of the equals(...) method within your **StudentAccount** class.

e) Implement a copy of the equals(...) method within your **IsaAccount** class. **Note**: This method will call the equals method within its direct superclass, i.e. InterestAccount, which will in turn call that within BankAccount, such that all instance variables of the subclass and its superclasses are tested for equality.

f) Test each of your equals(...) implementations work as expected in your **BankAccountTest** program.

Question 8.5 (*optional challenge*) Override the equals(...) method for each of the classes in your **advertising** project, e.g. Advert, Billboard, etc. Then test your implementations work within the AdTest program.