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
- Understand the purpose and use of Abstract classes

Associated Reading:

Horstmann,

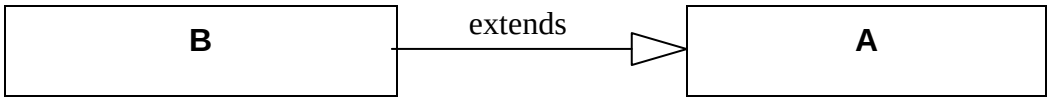
Chapter 9, Sections 9.1 - 9.4: Inheritance

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".

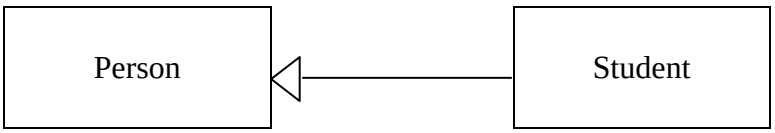


- The subclass inherits all of the superclass' fields and methods.
- The subclass can add more fields and methods. So it becomes more specialized.
- The subclass can override (i.e. redefine) the inherited methods, or leave them unchanged.
- In Java a class can only have one superclass, (but it can have 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.

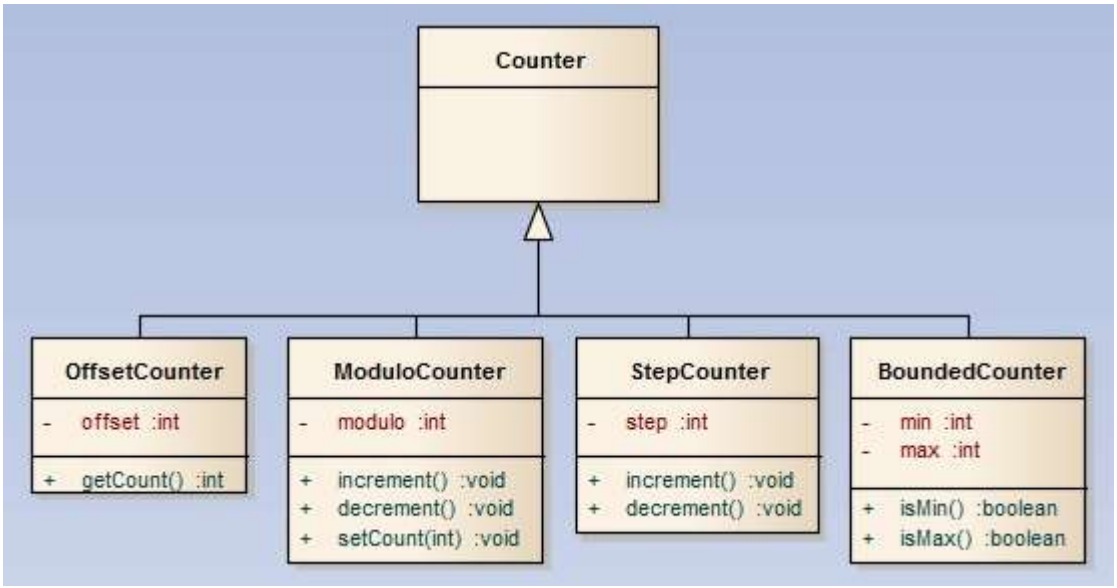
Please continue to the next page...

Question 8.1 Reopen your Eclipse workspace called "**CTEC2905_OO_Design**" then download the existing Java Project that can be extracted from the "**Inheritance.zip**" file, and imported into Eclipse. In Eclipse press Ctrl+F10 and select Package Presentation > Flat.

Earlier in the module you may recall implementing two types of Counter. A 'modulo' counter behaved in the same way as a standard counter except that its value stayed in the range 0 to (modulo-1). When the value of the counter reached 'modulo' it wrapped back to zero. For example, the seconds counter on a watch behaves like a modulo 60 counter.

The ModuloCounter '**is-a**' Counter, and therefore can naturally be associated with it through inheritance so as to reuse common implementation. We could easily define additional variants of the Counter class, e.g.

- **StepCounter**: increment and decrement by a **step** amount. E.g. count up and down in steps of 2.
- **OffsetCounter**: The internal representation of the count value is the same as in Counter, but `getCount` returns the count+**offset**.
- **BoundedCounte**r: The counter value must always be in some given range **min..max** inclusive. There are additional methods `isMin` and `isMax`.



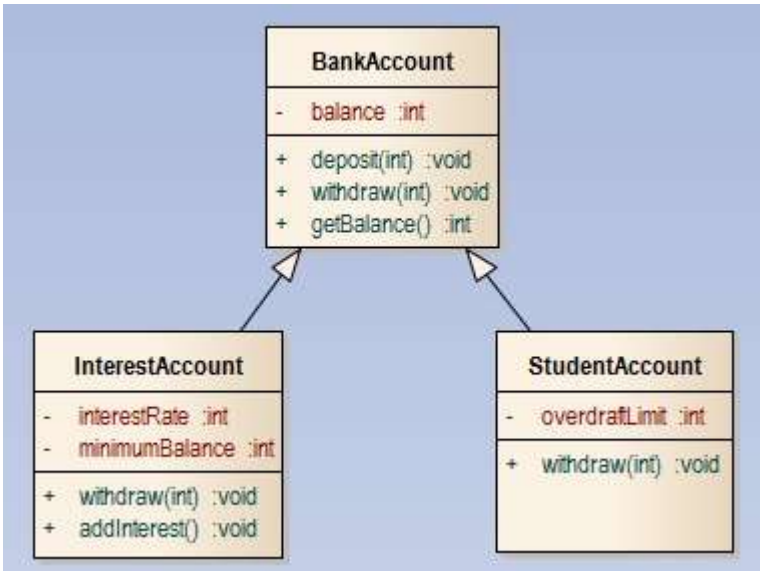
a) In the lib.counters package, study the code within the classes you have been provided, especially paying attention to how the inheritance relationships have been defined. Additionally, look in the main package at the CountersDemo program and how different types of counter can be processed in a uniform way, i.e. *polymorphism* - that you have already seen when using interface types.

b) Implement the **StepCounter** class ensuring it extends Counter and overrides the increment(), decrement() and toString() methods. It should define a default and custom constructor, as well as get/set methods for its *step* field.

c) Add some more code to the **CountersDemo** program so that a StepCounter object is created and added to the ArrayList so that it is processed within the for-each loop.

Question 8.2 Continue working in the same project and open the lib.bankaccounts package.

The following UML specification shows how you can create an inheritance hierarchy for bank accounts:



a) Study the code within the **BankAccount** superclass and the **InterestAccount** subclass 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** subclass 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 **BankAccountDemo** program (in the main package) 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 - 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 (also relevant to interface types).
- 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.

Please continue to the next page...

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.

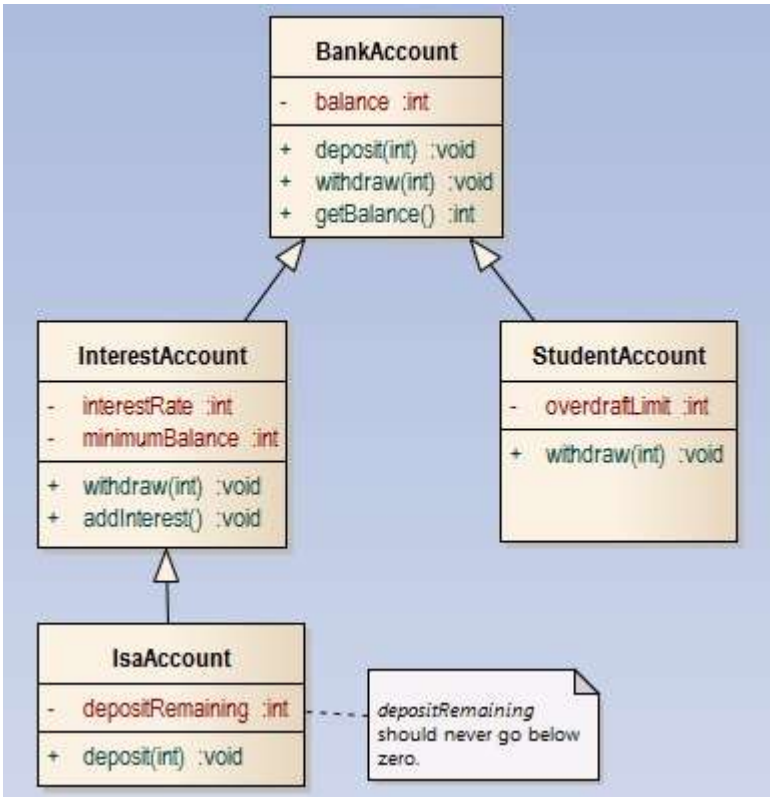
```
return super.toString() + "[interestRate=" + interestRate + ",
minimumBalance=" + minimumBalance + "]";
```

- The result is that the correct class name is output, matching that of the object for which the method is invoked, and the brackets are helpful in showing which variables belong to the superclass.

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

Please turn over...

Question 8.3 Continue working in the same package as the previous question and develop 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 **BankAccountDemo** program (in the main package) 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. You can also experiment further with the different bank accounts if you wish.

Please turn over...

Question 8.4 For each of the classes you have developed during this lab exercise, you have not yet overridden the equals(...) method. In previous lab exercises, you have learnt how to override the equals(...) method to test objects for equality. 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 (obj == null || this.getClass() != obj.getClass())
		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 **BankAccountDemo** program.

Please turn over...

Abstract classes

Although superclasses and interfaces share some common principles such as substitution and polymorphism, they differ most significantly in the way of sharing implementation. A subclass inherits all fields and methods of the superclass and can optionally override methods. An interface however usually defines no implementation and therefore shares none with its derived classes. Instead it simply specifies method headers that must be overridden.

Abstract classes sit between standard superclasses and interfaces, in that as with a superclass, you can extend them and inherit fields and methods, optionally overriding the latter. They also cannot be instantiated and can provide abstract methods that have no implementation and force subclasses to override, as with interfaces. Depending on your design scenario, a concrete superclass, an abstract class or an interface may be most appropriate and it is most important to understand their differences. Abstract classes are frequently present within the Java library and in the right circumstances can provide the best design solution, most notably when wanting to provide a partial implementation for subclasses to extend and complete.

Notes - The Abstract Inheritance association:

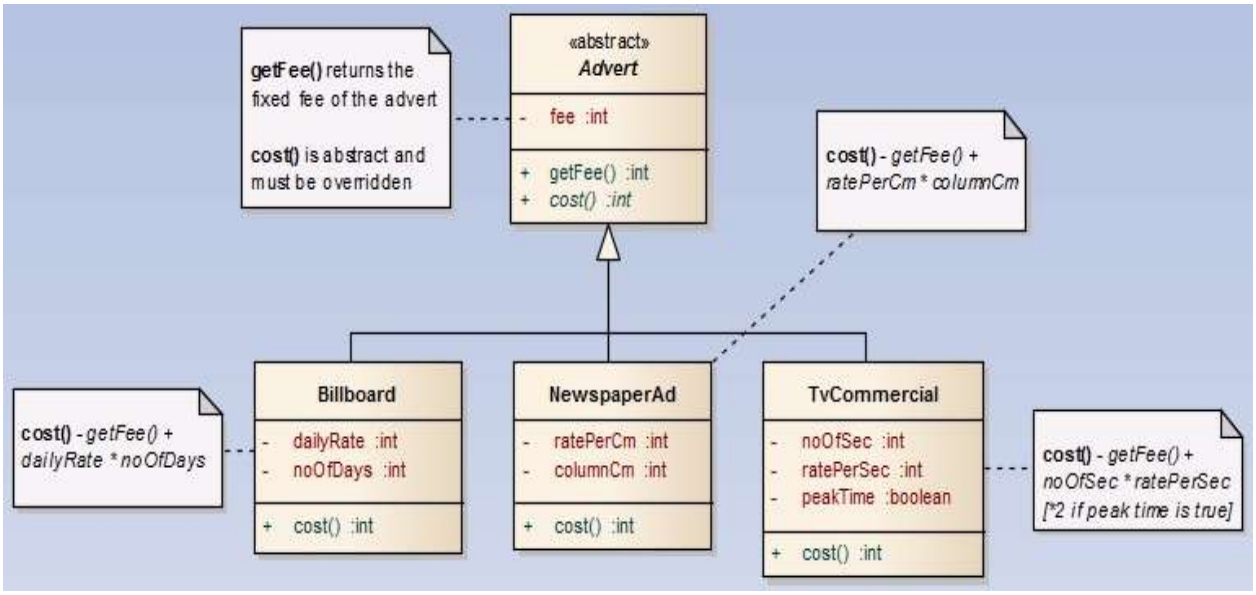
- UML diagrams use the same symbol for extending an abstract class as they do for extending a concrete superclass.
- Abstract classes are often differentiated in diagrams within the class box, by way of using italics for the class name (and any abstract methods) and optionally placing <<abstract>> above of it.

Question 8.5 Download the existing Java Project that can be extracted from the "**AbstractClasses.zip**" file, and imported into Eclipse.

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.
- **TV Commercial**: A commercial lasts a number of seconds. There is a rate per second. The rate is doubled during peak times.

The UML specification overleaf introduces an abstract class Advert. In this scenario it makes no sense to create advert objects, but each specific type of advert can extend this partial abstract implementation, and override the abstract `cost()` method.



a) Study the code for each of the classes in the lib.adverts package, taking note of the abstract class Advert and its subclasses that have overridden the cost method.

b) If you uncomment the two lines of code within AdDemo (in the main package) where it attempts to instantiate an Advert, it will give a compilation error. By making classes abstract you ensure they can never be instantiated. In this case, there is no real purpose to have an instance of Advert, but you can still store different types of adverts in a collection.

   Also note how the technique of polymorphism is applied within the for-each loop as every object stored within the collection is guaranteed to have a cost method. Dynamic method lookup will determine which specific cost method to invoke at runtime.

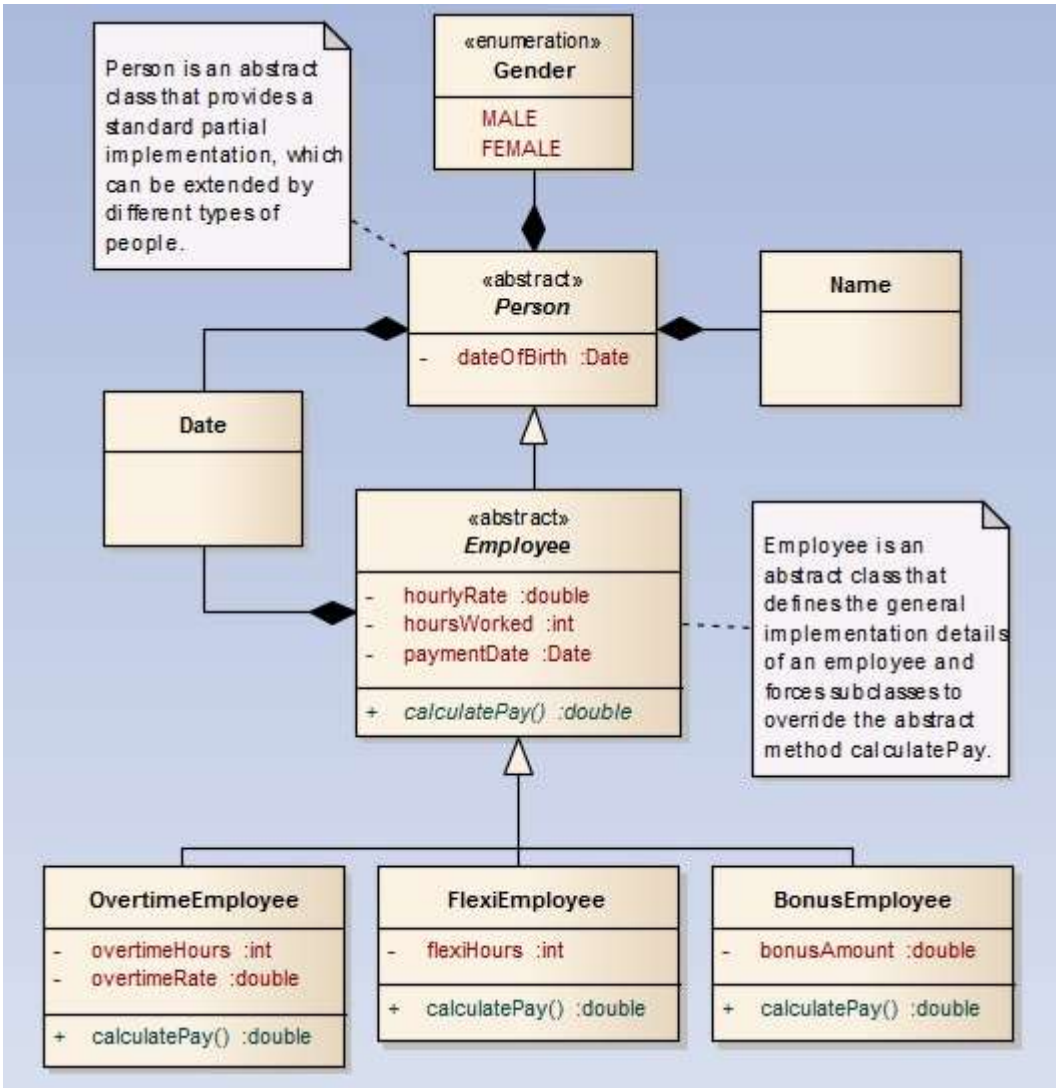
c) Attempt to create a **TvCommercial** subclass that extends Advert, holds three fields representing number of seconds, rate per second, and whether it is peak time or not, and overrides the cost method using the logic stated in the corresponding bullet point above the UML diagram.

d) Add some more code to the **AdDemo** program so that these newly implemented subclass is added to the ArrayList and processed within the for-each loop.

Notes – Abstract classes; did you know:

- All methods within an interface are implicitly abstract, so the abstract modifier is not necessary as it would be redundant.
- An abstract class, which solely defines abstract methods, is in-fact equivalent to an interface with the same implementation. In this case, prefer using an interface and "implementing" rather than "extending" – you can implement many interfaces but only extend one class.
- Placing an abstract method within a class forces subclasses to override it. In this case the class also must become abstract. However, when a class is declared abstract it does not require abstract methods, it simply cannot be instantiated.
- When extending an abstract class or implementing an interface, if the derived class does not provide implementations of each abstract method, then it will itself become abstract.

Question 8.6 Open the lib.employees package in the AbstractClasses project and then study the following UML diagram and scenario.



An employee is a person, both of which are abstract. Every employee therefore has a name, gender and date of birth (inherited from Person), in addition to a hourly rate, amount of hours worked, and a payment date. An employee requires its subclasses to provide a means of calculating their pay, and defines an abstract method to ensure they override this behaviour.

There are three specific types of employee that work within this organisation:

- **Overtime Employee** – has an overtime rate and an amount of overtime hours worked. Pay is calculated with the following formula:
  *(((hoursWorked - overtimeHours) * hourlyRate) + (overtimeHours * overtimeRate))*

- **Flexi Employee** – has an amount of flexi hours worked, which is either positive or negative. Pay is calculated with the following formula:
  *((hoursWorked + flexiHours) * hourlyRate)*

- **Bonus Employee** – has a bonus amount that they get every time they are paid. Pay is calculated with the following formula:
  *((hoursWorked * hourlyRate) + bonusAmount)*

a) Study the classes you have been given, taking particular note of how FlexiEmployee and BonusEmployee have been implemented.

b) Create a new subclass called **OvertimeEmployee**, which extends Employee. It should have two fields: `overtimeHours`, `overtimeRate`. It should then override the `calculatePay()` method using the formula on the previous page. You should also ensure it adheres to standard Java conventions, i.e. get/set, toString().

c) Create an **AbstractEmployeeDemo** program in the main package and informally test your OvertimeEmployee class works as expected by using each of its constructors and methods (also remembering those it inherits).

d) Create a list of type Employee and add instances of each of its subclasses, e.g.

```
List<Employee> employees = new ArrayList<>();
employees.add(new OvertimeEmployee(65, 7.5, 5, 15));
employees.add(new FlexiEmployee(65, 7.5, -5));
employees.add(new BonusEmployee(65, 7.5, 50));
```

Now write a for-each loop to process each employee and output their salary by calling the `calculatePay()` method. **Note**: In the example above, each of the employees has the same hours worked and hourly rate, but differ in their specialised values. Also, whilst the constructors used above are legitimate, you should have a standard default and custom constructor, with the latter needing to accept values for each of the fields of the class (including those that are inherited).