3. Inheritance and Method Overriding

Introduction

This chapter will discuss the essential concepts of Inheritance, method overriding and the appropriate use of ‘Super’.

Objectives

By the end of this chapter you will be able to….

- Appreciate the importance of an Inheritance hierarchy,
- Understand how to use Abstract classes to factor out common characteristics
- Override methods (including those in the ‘Object’ class),
- Explain how to use ‘Super’ to invoke methods that are in the process of being overridden,
- Document an inheritance hierarchy using UML and
- Implement inheritance and method overriding in Java programs.

All of the material covered in this chapter will be developed and expanded on in later chapters of this book. While this chapter will focus on understanding the application and documentation of an inheritance hierarchy, Chapter 6 will focus on developing the analytical skills required to define your own inheritance hierarchies.

This chapter consists of twelve sections :-

1) Object Families
2) Generalisation and Specialisation
3) Inheritance
4) Implementing Inheritance in Java
5) Constructors
6) Constructor Rules
7) Access Control
8) Abstract Classes
9) Overriding Methods
10) The ‘Object’ Class
11) Overriding toString() defined in ‘Object’
12) Summary

3.1 Object Families

Many kinds of things in the world fall into related groups of ‘families’. ‘Inheritance’ is the idea ‘passing down’ characteristics from parent to child, and plays an important part in Object Oriented design and programming.

While you are probably already familiar with constructors and access control (public/private), and there are particular issues in relating these to inheritance.

Additionally we need to consider the use of Abstract classes and method overriding as these are important concepts in the context of inheritance.

Finally we will look at the ‘Object’ class which has a special role in relation to all other classes in Java.

3.2 Generalisation and Specialisation

Classes are a generalized form from which objects with differing details can be created. Objects are thus ‘instances’ of their class. For example Student 051234567 is an instance of class Student. More concisely, 051234567 **is a** Student.

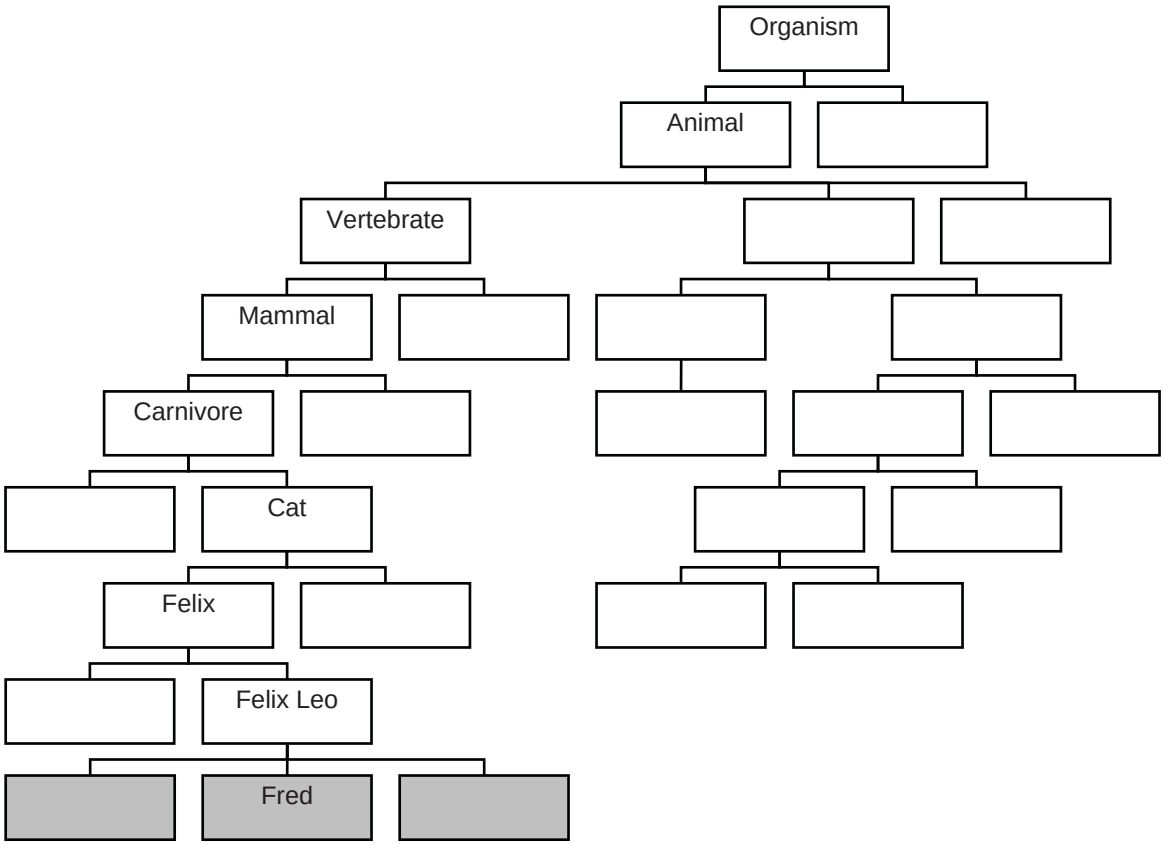
Classes themselves can often be organised by a similar kind of relationship.

One hierarchy, that we all have some familiarity with, is that which describes the animal kingdom :-

- Kingdom (e.g. animals)
- Phylum (e.g. vertebrates)
- Class (e.g. mammal)
- Order (e.g. carnivore)
- Family (e.g. cat)

- Genus (e.g. felix)
- Species (e.g. felix leo)

We can represent this hierarchy graphically ….



Of course to draw the complete diagram would take more time and space than we have available.

Here we can see one specific animal shown here :-'Fred'. Fred is not a class of animal but an actual animal.

Fred **is a** felix leo **is a** felix **is a** cat **is a** carnivore

Carnivores eat meat so Fred has the characteristic 'eats meat'.

Fred **is a** felix leo **is a** felix **is a** cat **is a** carnivore **is a** mammal **is a** vertebrate

Vertebrates have a backbone so Fred has the characteristic 'has a backbone'.

The 'is a' relationship links an individual to a hierarchy of characteristics. This sort of relationship applies to many real world entities, e.g. BonusSuperSaver is a SavingsAccount **is a** BankAccount.

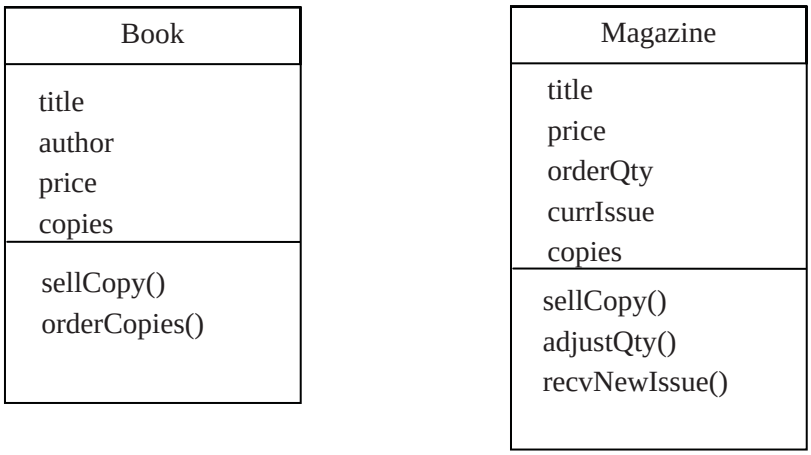
3.3 Inheritance

We specify the general characteristics high up in the hierarchy and more specific characteristics lower down. An important principle in OO – we call this **generalization** and **specialization**.

All the characteristics from classes above a class/object in the hierarchy are automatically featured in it – we call this **inheritance**.

Consider books and magazines - both specific types of publication.

We can show classes to represent these on a UML class diagram. In doing so we can see some of the instance variables and methods these classes may have.



Attributes ‘title’, ‘author’ and ‘price’ are obvious. Less obvious is ‘copies’ this is how many are currently in stock.

For books, orderCopies() takes a parameter specifying how many copies are added to stock.

For magazines, orderQty is the number of copies received of each new issue and currIssue is the date/period of the current issue (e.g. “January 2009”, “Fri 6 Jan”, “Spring 2009” etc.) When a newIssue is received the old are discarded and orderQty copies are placed in stock. Therefore recvNewIssue() sets currIssue to date of new issue and restores copies to orderQty. adjustQty() modifies orderQty to alter how many copies of subsequent issues will be stocked.

Activity 1

Look at the ‘Book’ and ‘Magazine’ classes defined above and identify the commonalities and differences between two classes.

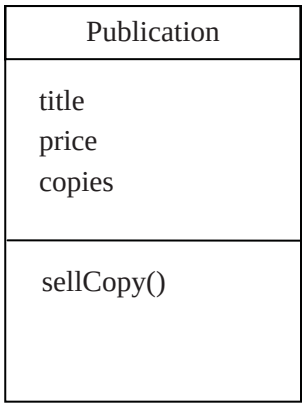
Feedback 1

These classes have three instance variables in common: title, price, copies.
They also have in common the method sellCopy().

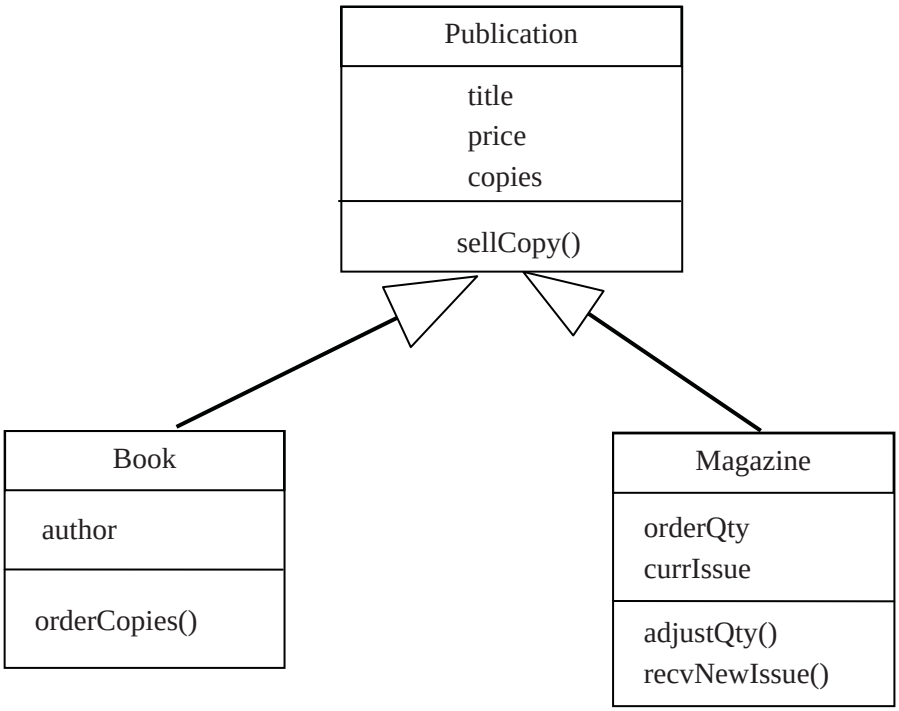
The differences are as follows…

Book additionally has author, and orderCopies().
Magazine additionally has orderQty, currIssue, adjustQty() and recvNewIssue().

We can separate out ('factor out') these common members of the classes into a superclass called Publication.



The differences will need to be specified as additional members for the ‘subclasses’ Book and Magazine.



In this is a UML Class Diagram.

The hollow-centred arrow denotes inheritance.

Note the Subclass has the generalized superclass characteristics + additional specialized characteristics. Thus the Book class has four instance variables (title, price, copies and author) it also has two methods (sellCopy() and orderCopies()).

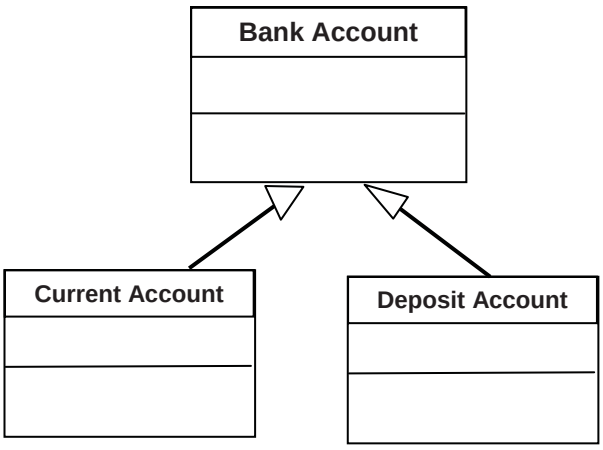
The inherited characteristics are NOT listed in subclasses. The arrow shows they are acquired from superclass.

Activity 2

Arrange the following classes into a suitable hierarchy and draw these on a class diagram…

a current account
a deposit account
a bank account
Simon’s deposit account

Feedback 2



The most general class goes at the top of the inheritance hierarchy with the other classes then inheriting the attributes and methods of this class.

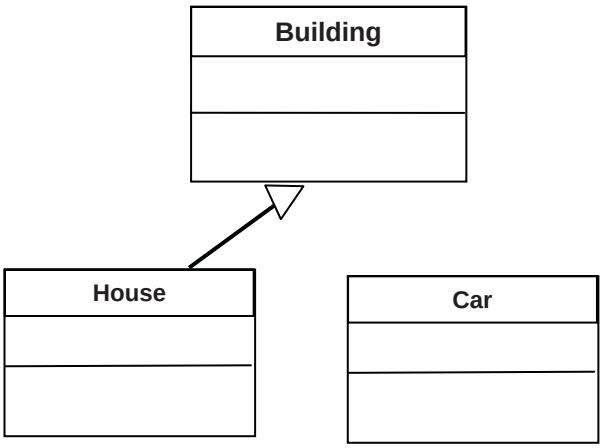
Simon's deposit account should not be shown on a class diagram as this is a specific instance of a class i.e. it is an object.

Activity 3

Arrange the following classes into a suitable hierarchy and draw these on a class diagram…

a building
a house
a car

Feedback 3



A house is a type of building and can therefore inherit the attributes of building however this is not true of a car. We cannot place two classes in an inheritance hierarchy unless we can use the term **is a**.

Note class names, as always, begin in uppercase.

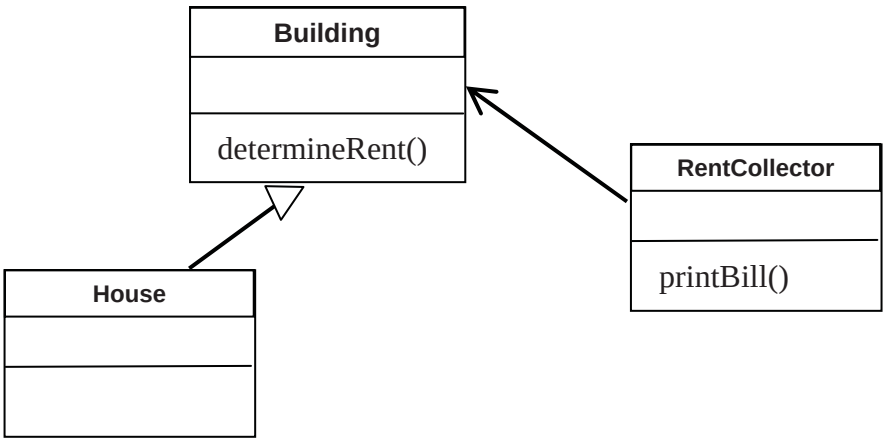
Activity 4

Describe the following using a suitable class diagram showing ANY sensible relationship…

a building for rent
this will have a method to determine the rent
a house for rent
this will inherit the determine rent method
a rent collector (person)
this person will use the determine rent method to print out a bill

HINT: You may wish to use the following arrow 

Feedback 4



Note: RentCollector does not inherit from Building as a RentCollector is a person not a type of Building. However there is a relationship (an association) between RentCollector and Building ie. a RentCollector needs to determine the rent for a Building in order to print out the bill.

Activity 5

Looking at the feedback from Activity 4 and determine if a RentCollector can print out a bill for the rent due on a house (or can they just print a bill for buildings?).

Feedback 5

Firstly to print out a bill a RentCollector would need to know the rent due. There is no method determineRent() defined for a house – but this does not mean it does not exist.

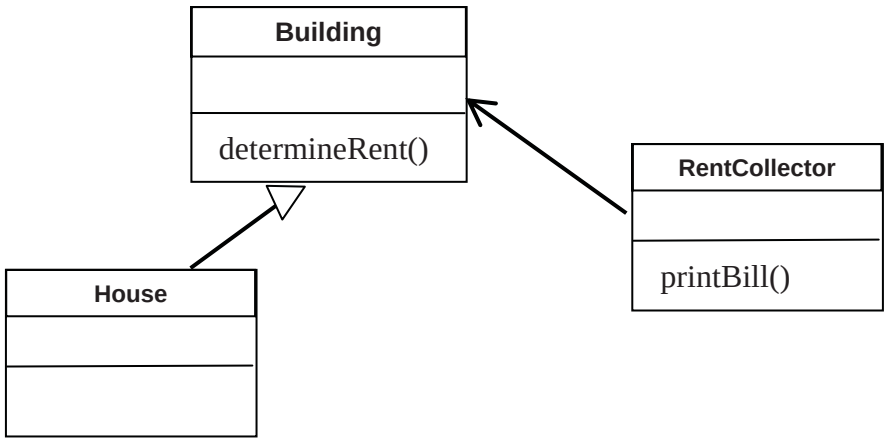
It must exist as House inherits the properties of Building!

We only show methods in subclasses if they are either additional methods or methods that have been overridden.

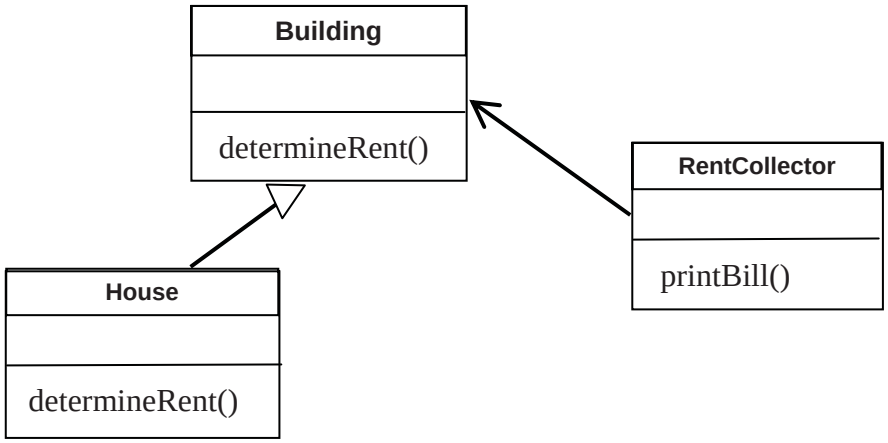
A rent collector requires a building but a House **is a** type of a Building. So, while no association is shown between the RentCollector and House, a Rentcollector could print a bill for a house. Wherever a Building object is required we could substitute a House object as this is a type of Building. This is an example of polymorphism and we will see other examples of this in Chapter 4.

Activity 6

Modify this UML diagram to show that determineRent() is overridden in House.



Feedback 6



By showing determineRent() in House we are showing that this method is overriding the one defined in the superclass (Building).

Interestingly Java will use the most correct determineRent() method depending upon which type of object the method is invoked on. Thus RentCollector will invoke the method defined in House if printing a bill for a house but will use the method defined in Building for any other type of building. This is automatic – the code in the RentCollector class does not distinguish between different types of Building.

Overriding will be discussed in more detail later in this chapter.

3.4 Implementing Inheritance in Java

No special features are required to create a superclass. Thus any class can be a superclass unless specifically prevented.

A subclass specifies it is inheriting features from a superclass using the keyword **extends**. For example….

```
class MySubclass extends MySuperclass
{
      // additional instance variables and
      // additional methods
}
```

3.5 Constructors

Each class (whether sub or super) should encapsulate its own initialization, usually relating to setting the initial state of its instance variables.

A constructor for a superclass should deal with general initialization.

Each subclass can have its own constructor for specialised initialization but it must often invoke the behaviour of the superclass constructor. It does this using the keyword **super**.

```
class MySubClass extends MySuperClass
{
      public MySubClass (sub-parameters)
      {
                  super(super-parameters);
                  // other initialization
      }
```

If **super** is called, ie. the superclass constructor, then this must be the first statement in the constructor.

Usually some of the parameters passed to MySubClass will be initializer values for superclass instance variables, and these will simply be passed on to the superclass constructor as parameters. In other words *super-parameters* will be some (or all) of *sub-parameters*.

Shown below are two constructors, one for the Publication class and one for Book. The book constructor requires four parameters three of which are immediately passed on to the superclass constructor to initialize its instance variables.

```
public Publication (String pTitle, double pPrice, int pCopies)
{
      title = pTitle;
      // etc.
}
```

```
public Book (String pTitle, String pAuthor, double pPrice,
                                                    int pCopies)
{
      super(pTitle, pPrice, pCopies);
      author = pAuthor;
      //etc.
}
```

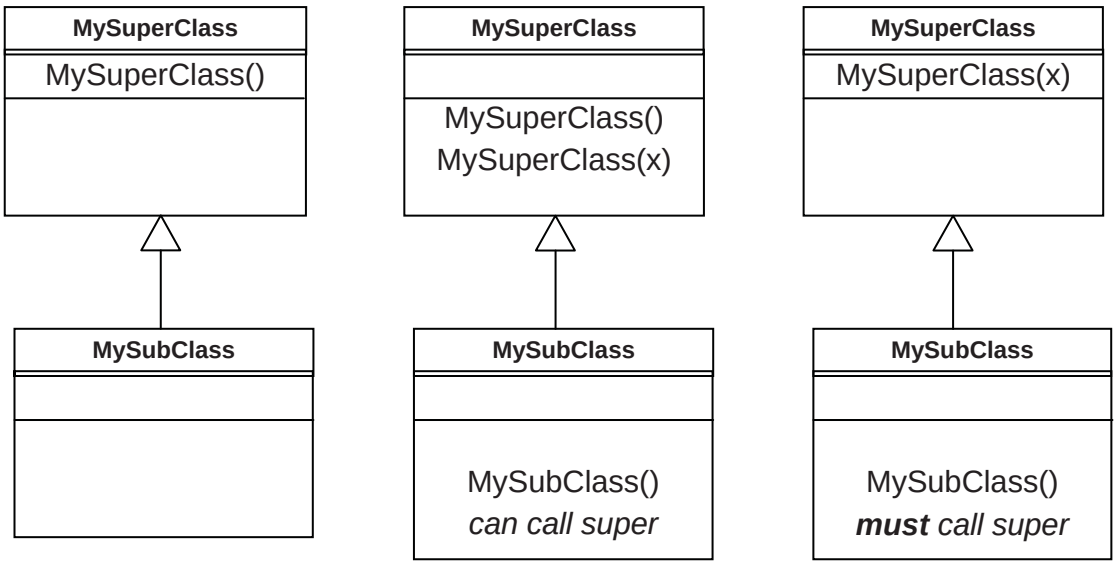
3.6 Constructor Rules

Rules exist that govern the invocation of a superconstructor.

If the superclass has a parameterless (or default) constructor this will be called automatically if no explicit call to super is made in the subclass constructor though an explicit call is still better style for reasons of clarity.

However if the superclass has no parameterless constructor but does have a parameterized one, this **must** be called explicitly using super.

To illustrate this….



On the left above:- it is legal, though bad practice, to have a subclass with no constructor because superclass has a parameterless constructor.

In the centre:- if subclass constructor doesn't call super, the parameterless superclass constructor will be called.

On the right:- because superclass has no paramterless constructor, subclass **must** have a constructor and it **must** call super. This is simply because a (super) class with only a parameterized constructor can only be initialized by providing the required parameter(s).

3.7 Access Control

To enforce encapsulation we normally make instance variables **private** and provide accessor/mutator methods as necessary.

The sellCopy() method of Publication needs to alter the value of the variable 'copies' it can do this even if 'copies' is a private variable. However Book and Magazine both need to alter 'copies'.

There are two ways we can do this …

1) make 'copies' 'protected' rather than 'private' – this makes it visible to subclasses, **or**
2) create accessor and mutator methods.

For variables we generally prefer to create accessors/mutators rather than compromise encapsulation though **protected** may be useful to allow subclasses to use methods (e.g. accessors and mutators) which we would not want generally available to other classes.

Thus in the superclass Publication we define ‘copies’ as a variable private but create two methods that can set and access the value ‘copies’. As these accessor methods are public or protected they can be used within a subclass when access to ‘copies’ is required.

In the superclass Publication we would therefore have….

```
private int copies;

public int getCopies ()
{
    return copies;
}

public void setCopies(int pCopies)
{
      copies = pCopies;
}
```

These methods allow superclass to control access to private instance variables.

As currently written they don’t actually impose any restrictions, but suppose for example we wanted to make sure ‘copies’ is not set to a negative value.

(a) If ‘copies’ is **private**, we can put the validation (i.e. an if statement) within the setCopies method here and know for sure that the rule can never be compromised

(b) If ‘copies’ is partially exposed as **protected**, we would have to look at every occasion where a subclass method changed the instance variable and do the validation at each separate place.

We might even consider making these *methods* **protected** rather than **public** themselves so their use is restricted to subclasses only and other classes cannot interfere with the value of ‘copies’.

Making use of these methods in the sublasses Book and Magazine we have ..

```
// in Book
public void orderCopies(int pCopies)
{
   setCopies(getCopies() + pCopies);
}
```

```
// and in Magazine
public void recvNewIssue(String pNewIssue)
{
      setCopies(orderQty);
      currIssue = pNewIssue;
}
```

These statements are equivalent to

mCopies = mCopies + pCopies

and

mCopies = mOrderQty

3.8 Abstract Classes

The idea of a Publication which is not a Book or a Magazine is meaningless, just like the idea of a Person who is neither a MalePerson nor a FemalePerson. Thus while we are happy to create Book or Magazine objects we may want to prevent the creation of objects of type Publication.

If we want to deal with a new type of Publication which is genuinely neither Book nor Magazine – e.g. a Calendar – it would naturally become another new subclass of Publication.

As Publication will never be instantiated ie. we will never create objects of this type the only purpose of the class exists is to gather together the generalized features of its subclasses in one place for them to inherit.

We can enforce the fact that Publication is non-instantiable by declaring it ‘abstract’:-

```
abstract class Publication
{
// etc.
```

3.9 Overriding Methods

A subclass inherits the methods of its superclass and must therefore always provide at least that set of methods, and often more. However, the implementation of a method can be changed in a subclass.

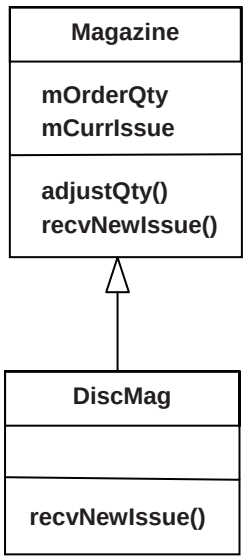
This is overriding the method.

To do this we write a new version in the subclass which replaces the inherited one.

The new method should essentially perform the same functionality as the method that it is replacing however by changing the functionality we can improve the method and make its function more appropriate to a specific subclass.

For example, imagine a special category of magazine which has a disc attached to each copy – we can call this a DiscMag and we would create a subclass of Magazine to deal with DiscMags. When a new issue of a DiscMag arrives not only do we want to update the current stock but we want to check that the discs are correctly attached. Therefore we want some additional functionality in the recvNewIssue() method to remind us to do this. We achieve this by redefining recvNewIssue() in the DiscMag subclass.

Note: when a new issue of Magazine arrives, as these don't have a disc we want to invoke the original recNewIssue() method defined in the Magazine class.



- The definition of **recvNewIssue()** in DiscMag overrides the inherited one.
- Magazine is not affected – it retains its original definition of **recvNewIssue()**
- By showing **recvNewIssue()** in DiscMag we are stating that the inherited method is being overridden (ie. replaced) as we do not show in inherited methods in subclasses.

When we call the **recvNewIssue()** method on a DiscMag object Java automatically selects the new overriding version – the caller doesn't need to specify this, or even know that it is an overriden method at all. When we call the **recvNewIssue()** method on a Magazine it is the method in the superclass that is invoked.

Implementing DiscMag

To implement DiscMag we must create a subclass of Magazine using extends. No additional instance variables or methods are required though it is possible to create some if there was a need. The constructor for DiscMag simply passes ALL its parameters directly on to the superclass and a version of newIssue() is defined in discMag to overrides the one inherited from Magazine (see code below).

```
public class DiscMag extends Magazine
{     // the constructor
      public DiscMag (String pTitle, double pPrice, int pOrderQt,
                                    String pCurrIssue, int pCopies)
      {
            super(pTitle, pPrice, pOrderQty, pCurrIssue, pCopies);
      }

      // the overridden method
      public void recvNewIssue(String pNewIssue)
      {
            super.recvNewIssue(pNewIssue);
            System.out.println("Check discs attached to this
                                                      magazine");
      }
}
```

Note the user of the **super** keyword to call a method of the superclass, thus re-using the existing functionality as part of the replacement, just as we do with constructors. It then additionally displays the required message for the user.

Operations

Formally, ‘recvNewIssue()’ is an operation. This one operation is implemented by two different methods, one in Magazine and the overriding one in DiscMag. However this distinction is an important part of ‘polymorphism’ which we will meet in Chapter 4.

3.10 The ‘Object’ Class

In Java all objects are (direct or indirect) subclasses of a class called ‘Object’. Object is the ‘root’ of the inheritance hierarchy in Java. Thus this class exists in every Java program ever created.

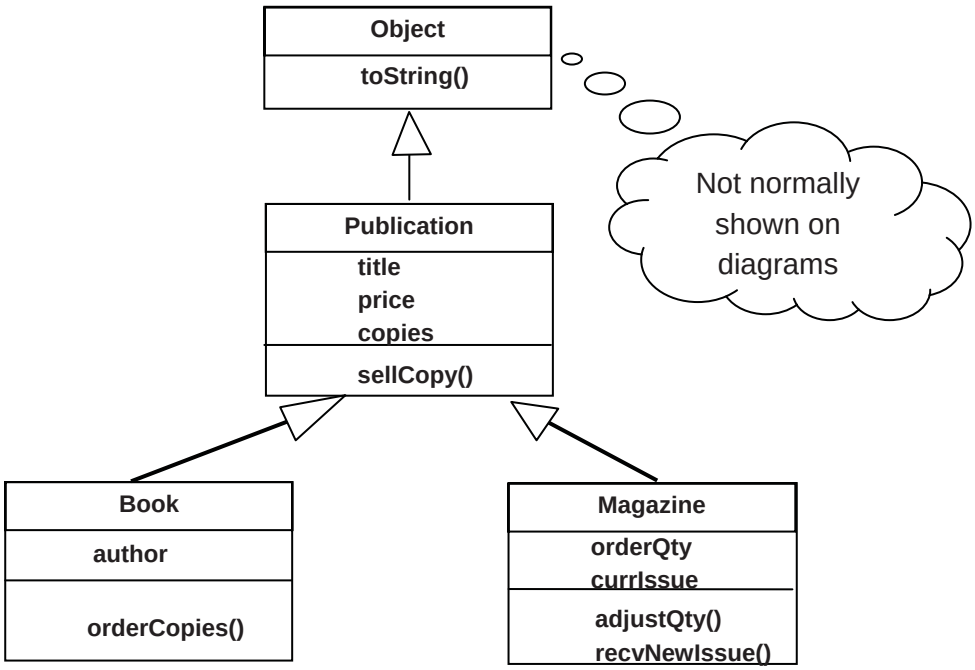
If a class is not declared to extend another then it implicitly extends Object.

Object defines no instance variables but several methods. Generally these methods will be overridden by new classes to make them useful. An example is the **toString()** method.

Thus when we define our own classes, by default they are direct subclasses of Object.

If our classes are organised into a hierarchy then the topmost superclass in the hierarchy is a direct subclass of object, and all others are indirect subclasses.

Thus directly, or indirectly, all classes created in Java inherit toString().



3.11 Overriding toString() defined in ‘Object’

The Object class defines a toString() method, one of several useful methods.

toString() has the signature

public String toString()

Its purpose is to return a string value that represents the current object. The version of toString() defined by Object produces output like: "Book@11671b2". This is the class name and the “hash code” of the object. However to be generally useful we need to override this to give a more meaningful string.

In Publication

```
public String toString()
{
    return mTitle;
}
```

In Book

```
public String toString()
{
    return super.toString() + " by " + mAuthor;
}
```

In Magazine

```
public String toString()
{
    return super.toString() + " (" + mCurrIssue + ")";
}
```

In the code above toString() originally defined in Object has been completely replaced, ie. overridden, so that Publication.toString() returns just the title.

The toString() method has been overridden again in Book such that Book.toString() returns title (via superclass toString() method) and author. Ie. this overridden version uses the version defined in Publication. Thus if Publication.toString() was rewritten to return the title and ISBN number then Book.toString() would automatically return the title, ISBN number and author.

Magazine.toString() returns title (via superclass toString() method) and issue

We will not further override the method in DiscMag because the version it inherits from Magazine is OK.

We could choose to provide more data (i.e. more, or even all, of the instance variable values) in these strings. The design judgement here is that these will be the most generally useful printable representation of objects of these classes. In this case title and author for a book, or title and current issue for a magazine, serve well to uniquely identify a particular publication.

3.12 Summary

Inheritance allows us to factor out common attributes and behaviour. We model the commonalities in a superclass.

Subclasses are used to model specialized attributes and behaviour.

Code in a superclass is inherited to all subclasses. If we amend or improve code for a superclass it impacts on all subclasses. This reduces the code we need to write in our programs.

Special rules apply to constructors for subclasses.

A superclass can be declared **abstract** to prevent it being instantiated (i.e. objects created).

We can ‘override’ inherited methods so a subclass implements an operation differently from its superclass.

In Java all classes descend from the class ‘Object’

‘Object’ defines some universal operations which can usefully be overriden in our own classes.