

# 4 Object Roles and the Importance of Polymorphism

## Introduction

Through the use of worked examples this chapter will explain the concept of polymorphism and the impact this has on OO software design.

## Objectives

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

- Understand how polymorphism allows us to handle related classes in a generalized way
- Employ polymorphism in Java programs
- Understand the implications of polymorphism with overridden methods
- Define interfaces to extend polymorphism beyond inheritance hierarchies
- Appreciate the scope for extensibility which polymorphism provides

This chapter consists of eight sections:-

- 1) Class Types
- 2) Substitutability
- 3) Polymorphism
- 4) Extensibility
- 5) Interfaces
- 6) Extensibility Again
- 7) Distinguishing Subclasses
- 8) Summary

## 4.1 Class Types

Within hierarchical classification of animals

Pinky is a pig (species *sus scrofa*)

Pinky is (also, more generally) a mammal

Pinky is (also, even more generally) an animal

We can specify the type of thing an organism is at different levels of detail:

higher level = less specific

lower level = more specific

If you were asked to give someone a pig you could give them Pinky or any other pig. If you were asked to give someone a mammal you could give them Pinky, any other pig or any other mammal (e.g. any lion, or any mouse, or any human being!). If you were asked to give someone an animal you could give them Pinky, any other pig, any other mammal or any other animal (bird, fish, insect etc).

The idea here is that an object in a classification hierarchy has an 'is a' relationship with every class from which it is descended and each classification represents a type of animal.

This is true in object oriented programs as well. Every time we define a class we create a new 'type'. Types determine compatibility between variables, parameters etc.

A subclass type is a subtype of the superclass type and we can substitute a subtype wherever a 'supertype' is expected. Following this we can substitute objects of a subtype whenever objects of a supertype are required (as in the example above).

The class diagram below shows a hierarchical relationship of types of object – or classes.

Excellent Economics and Business programmes at:



university of  
 groningen

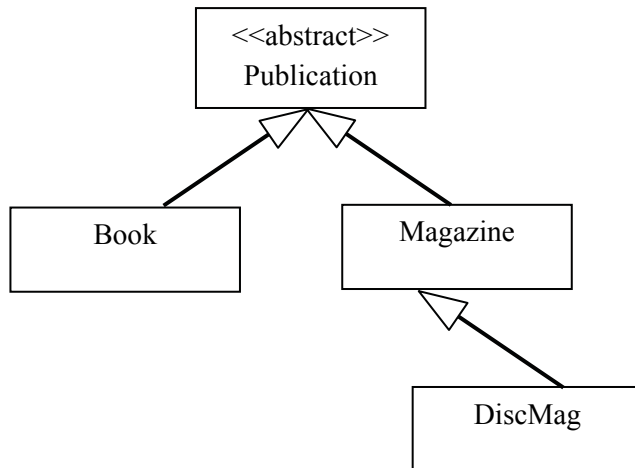


“The perfect start  
 of a successful,  
 international career.”

**CLICK HERE**  
 to discover why both socially  
 and academically the University  
 of Groningen is one of the best  
 places for a student to be

[www.rug.nl/feb/education](http://www.rug.nl/feb/education)





- If we want 'a DiscMag', it must be an object of class DiscMag.
- If we want 'a Magazine', it could be an object of class Magazine or an object of class DiscMag
- If we want 'a Publication' it could be a Book, Magazine or DiscMag.

In other words we can 'substitute' an object of any subclass where an object of a superclass is required. This is NOT true in reverse!

#### Activity 1

Look at the class diagram above and decide which of the following lines of code would be legal in a Java program where these classes had been implemented: -

```

Publication p = new Book(...);
Publication p = new DiscMag(...);
Magazine m = new DiscMag(...);
DiscMag dm = new Magazine(...);
Publication p = new Publication(...);
  
```

#### Feedback 1

```
Publication p = new Book(...);
```

Here we are defining a variable p of the general type of 'Publication' we are then invoking the constructor for the Book class and assigning the result to 'p' this is OK because Book is a subclass of Publication i.e. a Book **is a** Publication.

```
Publication p = new DiscMag(...);
```

This is OK because DiscMag is a subclass of Magazine which is a subclass of Publication i.e. DiscMag is an indirect subclass of Publication.

```
Magazine m = new DiscMag(...);
```

This is OK because DiscMag is a subclass of Magazine

```
DiscMag dm = new Magazine(...);
```

This is illegal because Magazine is a SUPERclass of DiscMag

```
Publication p = new Publication(...);
```

This is illegal because Publication is an abstract class and therefore cannot be instantiated.

## 4.2 Substitutability

When designing class/type hierarchies, the type mechanism allows us to place a subclass object where a superclass is specified. However this has implications for the design of subclasses – we need to make sure they are genuinely substitutable for the superclass. If a subclass object is substitutable then clearly it must implement all of the methods of the superclass – this is easy to guarantee as all of the methods defined in the superclass are inherited by the subclass. Thus while a subclass may have additional methods it must at least have all of the methods defined in the superclass and should therefore be substitutable. However what happens if a method is overridden in the subclass?

When overriding methods we must ensure that they are still substitutable for the method being replaced. Therefore when overriding methods, while it is perfectly acceptable to tailor the method to the needs of the subclass a method should not be overridden with functionality which performs an inherently different operation.

For example, `recvNewIssue()` in `DiscMag` overrides `recvNewIssue()` from `Magazine` but does the same basic job (“fulfils the contract”) as the inherited version with respect to updating the number of copies and the current issue. However, it extends that functionality in a way specifically relevant to `DiscMags` by displaying a reminder to check the cover discs.

What do we know about a ‘Publication’?

Answer: It’s an object which supports (at least) the operations:

```
void sellCopy()
double getPrice()
int getCopies()
void setCopies(int pCopies)
String toString()
```

Inheritance guarantees that objects of any subclass of `Publications` provides at least these.

Note that a subclass can never remove an operation inherited from its superclass(es) – this would break the guarantee. Because subclasses **extend** the capabilities of their superclasses, the superclass functionality can be assumed.

It is quite likely that we would choose to override the `toString()` method (initially defined within ‘Object’) within `Publication` and override it again within `Magazine` so that the `String` returned provides a better description of `Publications` and `Magazines`. However we should not override the `toString()` method in order to return the price – this would be changing the functionality of the method so that the method performs an inherently different function. Doing this would break the substitutability principle.

### 4.3 Polymorphism

Because an instance of a subclass is an instance of its superclass we can handle subclass objects as if they were superclass objects. Furthermore because a superclass guarantees certain operations in its subclasses we can invoke those operations without caring which subclass the actual object is an instance of.

This characteristic is termed ‘polymorphism’, originally meaning ‘having multiple shapes.’

Thus a Publication comes in various shapes...it could be a Book, Magazine or DiscMag. We can invoke the sellCopy() method on any of these Publications irrespective of their specific details.

Polymorphism is a fancy name for a common idea. Someone who knows how to drive can get into and drive most cars because they have a set of shared key characteristics – steering wheel, gear stick, pedals for clutch, brake and accelerator etc – which the driver knows how to use. There will be lots of differences between any two cars, but you can think of them as subclasses of a superclass which defines these crucial shared ‘operations’.

If ‘p’ ‘is a’ Publication, it might be a Book or a Magazine or a DiscMag.

Whichever, it has a sellCopy() method.



**REGENT'S**  
UNIVERSITY LONDON

## Enhance your career opportunities

We offer practical, industry-relevant undergraduate and postgraduate degrees in central London

- › Accounting and finance
- › Business, management and leadership
- › Oil and gas trade management
- › Global banking and finance
- › Luxury brand management
- › Media communications and marketing

**Contact us to arrange a visit**  
**Apply direct for January or September entry**

**T** +44 (0)20 7487 7505   **E** [exrel@regents.ac.uk](mailto:exrel@regents.ac.uk)   **W** [regents.ac.uk](http://regents.ac.uk)



So we can invoke `p.sellCopy()` without worrying about what exactly 'p' is.

This can make life a lot simpler when we are manipulating objects within an inheritance hierarchy. We can create new types of `Publication` e.g. a `Newspaper` and invoke `p.sellCopy()` on a `Newspaper` without have to create any functionality within the new class – all the functionality required is already defined in `Publication`.

Polymorphism makes it very easy to extend the functionality of our programs as we will see in Chapter 11.

#### 4.4 Extensibility

Huge sums of money are spent annually creating new computer programs but over the years even more is spent changing and adapting those programs to meet the changing needs of an organisation. Thus as professional software engineers we have a duty to facilitate this and help to make those programs easier to maintain and adapt. Of course the application of good programming standards, commenting and layout etc, have a part to play here but also polymorphism can help as it allows programs to be made that are easily extended.

##### **CashTill class**

Imagine we want to develop a class `CashTill` which processes a sequence of items being sold. Without polymorphism we would need separate methods for each type of item:

```
sellBook (Book pBook)
sellMagazine (Magazine pMagazine)
sellDiscMag (DiscMag pDiscMag)
```

With polymorphism we need only

```
sellItem (Publication pPub)
```

Every subclass is 'type-compatible' with its superclass. Therefore any subclass object can be passed as a `Publication` parameter.

This also has important implications for extensibility of systems. We can later introduce further subclasses of `Publication` and these will also be acceptable by the `sellItem()` method of a `CashTill` object, even through these subtypes were unknown when the `CashTill` was implemented.

**Publications sell themselves!**

Without polymorphism we would need to check for each item 'p' so we were calling the right method to sell a copy of that subtype

```

if 'p' is a Book call sellCopy() method for Book
else if 'p' is a Magazine call sellCopy() method for Magazine
else if 'p' is a DiscMag call sellCopy() method for DiscMag

```

Instead we trust the Java virtual machine to look at the object 'p' at run time, to determine its 'type' and its own method for selling itself. Thus we can call:-

```
p.sellCopy()
```

and if the object is a Book it will invoke the sellCopy() method for a Book. If 'p' is a Magazine, again at runtime Java will determine this and invoke the sellCopy() method for a Magazine.

Polymorphism often allows us to avoid conditional 'if' statements – instead the 'decision' is made implicitly according to which type of subclass object is actually present.

**Implementing CashTill**

The code below shows how CashTill can be implemented to make use of Polymorphism.

```

class CashTill
{
    private double runningTotal;
CashTill ()
    {
        runningTotal = 0;
    }
    public void sellItem (Publication pPub)
    {
        runningTotal = runningTotal + pPub.getPrice();
        pPub.sellCopy();
        System.out.println("Sold " + pPub + " @ " +
                            pPub.getPrice() + "\nSubtotal = " +
                            runningTotal);
    }
    public void showTotal()
    {
        System.out.println ("GRAND TOTAL: " + runningTotal);
    }
}

```

Download free eBooks at [bookboon.com](http://bookboon.com)

The CashTill has one instance variable – a double to hold the running total of the transaction. The constructor simply initializes this to zero.

The sellItem() method is the key feature of CashTill. It takes a Publication parameter, which may be a Book, Magazine or DiscMag. First the price of the publication is added to the running total using the getPrice() accessor. Then the sellCopy() operation is invoked on the publication.


Finally a message is constructed and displayed to the user, e.g.

```
Sold Windowcleaning Weekly (Sept 2005) @ 2.75
Subtotal = 2.75
```

Note that when **pPub** appears in conjunction with the string concatenation operator '+'. This implicitly invokes the **toString()** method for the subclass of this object, and remember that **toString()** is different for books and magazines.

This is polymorphism in action – using the **toString()** operation to invoke the appropriate toString() method for the relevant class!



.....Alcatel-Lucent 

[www.alcatel-lucent.com/careers](http://www.alcatel-lucent.com/careers)

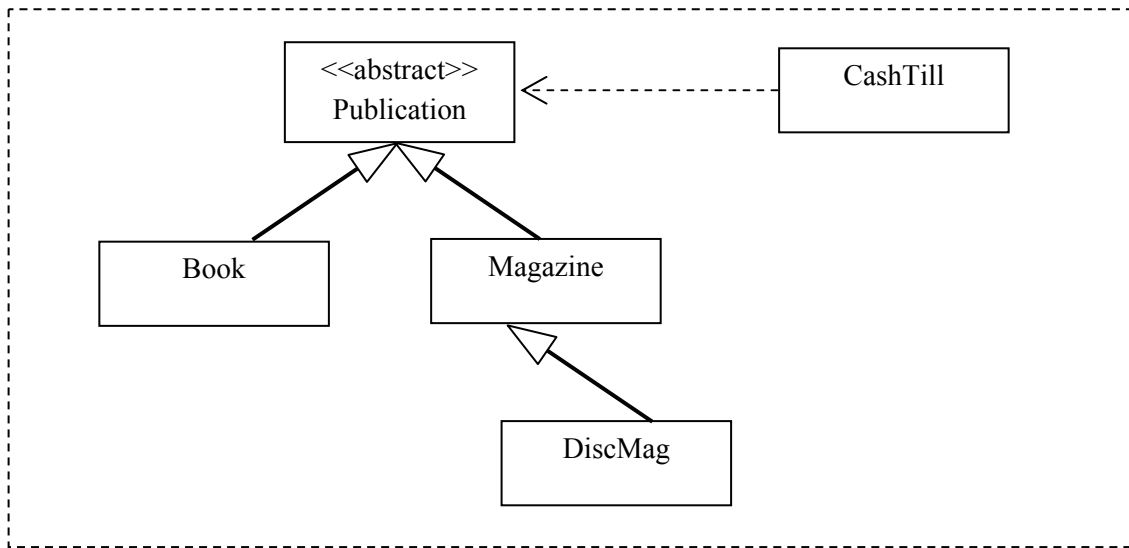
What if you could build your future and create the future?

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".





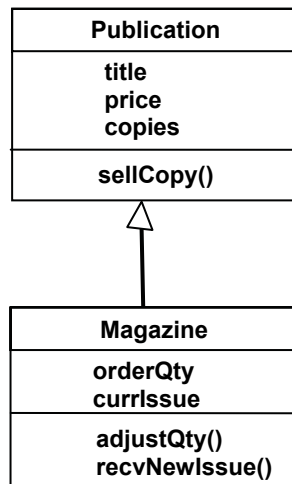
We can show CashTill on a class diagram as below:-



Note that CashTill has a dependency on Publication because the sale() method is passed a parameter of type Publication. What is actually passed will of course be an object of one of the concrete types descended from Publication.

**Activity 2**

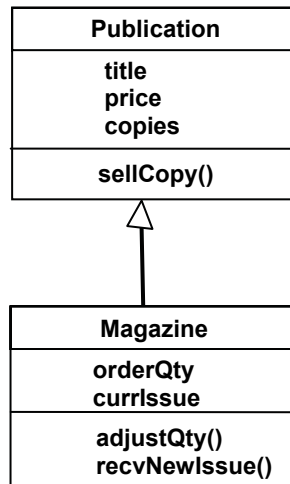
Look at the diagram below and, assuming Publication is not an abstract type, decide which of the pairs of operations shown are legal.



- a) Publication p = new Publication(...); p.sellCopy();
- b) Publication p = new Publication(...); p.recvNewIssue();
- c) Publication p = new Magazine(...); p.sellCopy();
- d) Publication p = new Magazine(...); p.recvNewIssue();
- e) Magazine m = new Magazine(...); m.recvNewIssue();

**Activity 2**

Look at the diagram below and, assuming Publication is not an abstract type, decide which of the pairs of operations shown are legal.



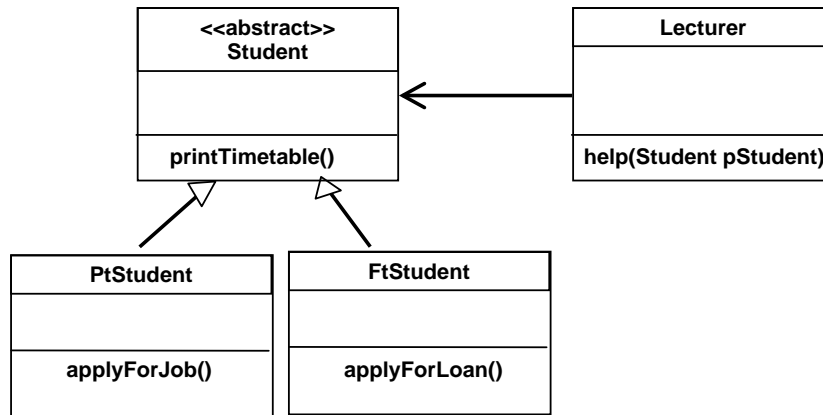
- a) `Publication p = new Publication(...); p.sellCopy();`
- b) `Publication p = new Publication(...); p.recvNewIssue();`
- c) `Publication p = new Magazine(...); p.sellCopy();`
- d) `Publication p = new Magazine(...); p.recvNewIssue();`
- e) `Magazine m = new Magazine(...); m.recvNewIssue();`

**Feedback 2**

- a) Legal – you can invoke `sellCopy()` on a publication
- b) Illegal – the `recvNewIssue()` method does not exist in publications
- c) Legal – Magazine is a type of Publication and therefore you can assign an object of type Magazine to a variable of type Publication (you can always substitute subtypes where a supertype is requested). Also you can invoke `sellCopy()` on a publication. The publication happens to be a magazine but this is irrelevant as far as the compiler knows in this instance 'p' is just a publication.
- d) Illegal – while we can invoke `recvNewIssue` on a magazine the compiler does not know that 'p' is a magazine...only that it is a publication.
- e) Legal – m is a magazine and we can invoke this method on magazines.

**Activity 3**

Look at the diagram below and, noting that Student is an abstract class, decide which of the following code segments are valid...



Note FtStudent is short for Full Time Student and PtStudent is short for Part Time Student.

- a) `Student s = new Student(); Lecturer l = new Lecturer(); l.help(s);`
- b) `Student s = new FtStudent(); Lecturer l = new Lecturer(); l.help(s);`

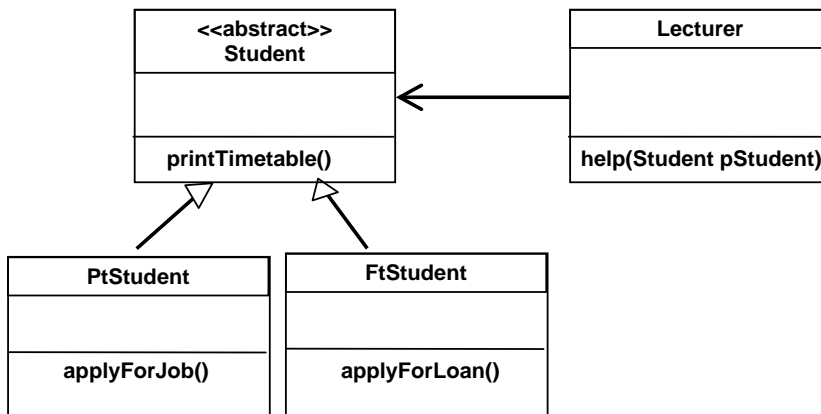
**Feedback 3**

- a) This is not valid as class Student is abstract and cannot be instantiated
- b) This is valid. FtStudent is a type of Student and can be assigned to variable of type Student. This can then be passed as a parameter to l.help()

**Activity 4**

Taking the same diagram and having invoked the code directly below decide which of the following lines (a) or (b) would be valid inside the method help(Student pStudent)...

```
Student s = new FtStudent();
Lecturer l = new Lecturer();
l.help(s);
```



- a) pStudent.printTimetable();
- b) pStudent.applyForLoan();



**Maastricht University** *Leading in Learning!*

**Join the best at  
the Maastricht University  
School of Business and  
Economics!**

**Top master's programmes**

- 33<sup>rd</sup> place Financial Times worldwide ranking: MSc International Business
- 1<sup>st</sup> place: MSc International Business
- 1<sup>st</sup> place: MSc Financial Economics
- 2<sup>nd</sup> place: MSc Management of Learning
- 2<sup>nd</sup> place: MSc Economics
- 2<sup>nd</sup> place: MSc Econometrics and Operations Research
- 2<sup>nd</sup> place: MSc Global Supply Chain Management and Change

Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012

**Maastricht University is the best specialist university in the Netherlands**  
(Elsevier)

**Visit us and find out why we are the best!**  
**Master's Open Day: 22 February 2014**

[www.mastersopenday.nl](http://www.mastersopenday.nl)



**Feedback 4**

- a) This is valid – we can invoke this method on a Student object and also on an FtStudent object (as the method is inherited).
- b) Not Valid! While we can invoke this method on a FtStudent object, and we are passing an FtStudent object as a parameter to the help() method, the help() method cannot know that the object passed will be a FtStudent (it could be any object of type Student). Therefore there is no guarantee that the object passed will support this method. Hence this line of code would generate a compiler error.

## 4.5 Interfaces

There are two aspects to inheritance:

- the subclass inherits the interface (i.e. access to public members) of its superclass – this makes polymorphism possible
- the subclass inherits the implementation of its superclass (i.e. instance variables and method implementations) – this saves us copying the superclass details in the subclass definition

In Java, the **extends** keyword automatically applies both these aspects.

A subclass is a subtype. Its interface must include all of the interface of its superclass, though the implementation of this can be different (though overriding) and the interface of the subclass may be more extensive with additional features being added.

However, sometimes we may want two classes to share a common interface without putting them in an inheritance hierarchy. This might be because:-

- they aren't really related by a true 'is a' relationship
- we want a class to have interfaces shared with more than one would-be superclass, but Java does not allow such 'multiple inheritance'
- we want to create a 'plug and socket' arrangement between software components, some of which might not even be created at the current time.

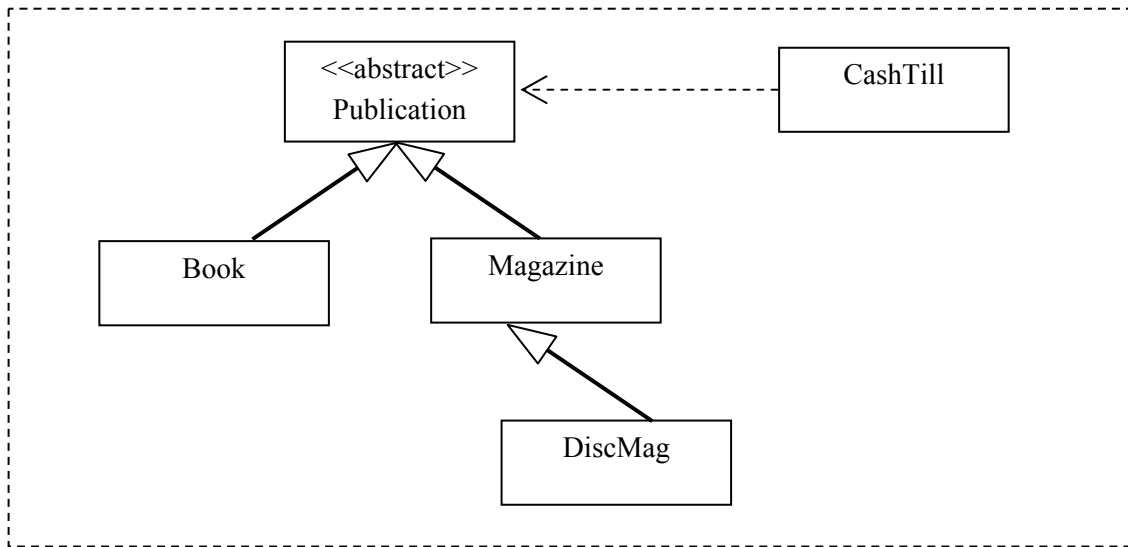
This is like making sure that two cars have controls that work in exactly the same way, but leaving it to different engineers to design engines which 'implement' the functionality of the car, possibly in quite different ways.

Be careful of the term 'interface' – in Java programming it has at least three meanings:

- 1) the public members of a class – the meaning used above
- 2) the "user interface" of a program, often a "Graphical User Interface" – an essentially unrelated meaning
- 3) a specific Java construct which we are about to meet

Download free eBooks at [bookboon.com](http://bookboon.com)

Recall how the subclasses of Publication provide additional and revised behaviour while retaining the set of operations – i.e. the interface – which it defined.



This is why the CashTill class can deal with a ‘Publication’ without worrying of which specific subclass it is an instance. (Remember that Publication is an abstract class – a ‘Publication’ is in reality **always** a subclass.)

### Tickets

Now consider the possibility that in addition to books and magazines, we now want to sell tickets, e.g. for entertainment events, public transport, etc. These are not like Publications because:-

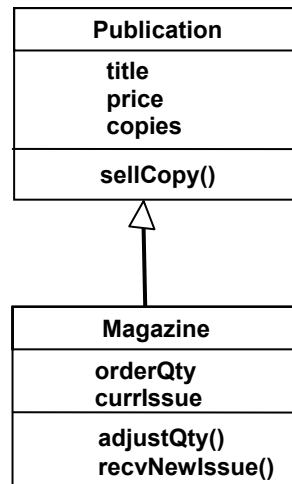
- we don’t have a finite ‘stock’ but print them on demand at the till
- tickets consist simply of a description, price and client (for whom they are being sold)
- these sales are really a service rather than a product

Tickets seem to have little in common with Publications – they share a small **interface** associated with being sold, but even for this the underlying **implementation** will be different because we will not be decrementing them from a current stock

For these reasons Ticket and Publication do not seem closely related and thus we do not want to put them in an inheritance hierarchy. However we do want to make them both acceptable to CashTill as things to sell and we need a mechanism for doing this.

Without putting them in an inheritance hierarchy what we want is a more general way of saying “things of this class can be sold” which can be applied to whatever (present and future) classes we wish, thus making the system readily extensible to Tickets and anything else.

While the Ticket class is sufficiently different from a Publication that we don't want to put it in an inheritance hierarchy it does have some similarities – namely it has a `getPrice()` method and a `sellCopy()` method – both needed by `CashTill`.



However the `sellCopy()` method is very different from the `sellCopy()` method defined in `Publication`. To sell a publication the stock had to be reduced by 1 – with a ticket we just need to print it.

**> Apply now**

REDEFINE YOUR FUTURE  
**AXA GLOBAL GRADUATE PROGRAM 2014**

redefining / standards 

agence c&g - © Photonstop



```
public void sellCopy()  
{  
    System.out.println("*****");  
    System.out.println(" TICKET VOUCHER ");  
    System.out.println(toString());  
    System.out.println("*****");  
    System.out.println();  
}
```

As the `sellCopy()` method is so different we do not want to inherit its implementation details therefore we don't feel that `Ticket` belongs in an inheritance hierarchy with `Publications`. But we do want to be able to check tickets through the till as we can with publications.

Just like publications, tickets provide the operations which `CashTill` needs:

```
sellCopy()  
getPrice()
```

and thus the `CashTill` can sell a `Ticket`. In fact `CashTill` can sell anything that has these methods not just `Publications`. To enable this to happen we will define this set of operations as an 'Interface' called `SaleableItem`.

```
interface SaleableItem  
{  
    public void sellCopy ();  
    public double getPrice ();  
}
```

Note that the interface defines purely the signatures of operations without their implementations. All the methods are implicitly public even if this is not stated, and there can be no instance variables or constructors.



In other words, an interface defines the **availability** of specified operations without saying anything about their implementation. That is left to classes which **implement** the interface.

An interface is a sort of contract. The **SaleableItem** interface says “I undertake to provide, at least, methods with these signatures:

```
public void sellCopy ();  
public double getPrice ();
```

though I might include other things as well”

Where more than one class implements an interface it provides a guaranteed area of commonality which polymorphism can exploit.

Think of a car and a driving game in an arcade. They certainly are not related by any “is a” relationship – they are entirely different kinds of things, one a vehicle, the other an arcade game. But they both implement what we could call a “SteeringWheel interface” which we can use in exactly the same way, even though the implementation (mechanical linkage in the car, video electronics in the game) are very different.

We now need to state that both Publication (and all its subclasses) and Ticket both offer the operations defined by this interface:

```
class Publication implements SaleableItem  
{  
    [...class details...]  
}
```

```
class Ticket implements SaleableItem  
{  
    [...class details...]  
}
```

Contrast **implements** with **extends**.

- **extends** causes the inheritance of both interface and implementation from a superclass.
- **implements** gives a guarantee that the operations specified by an interface will be provided – this is enough to allow polymorphic handling of all classes which implement a given interface

### The Polymorphic CashTill

The CashTill class already employs polymorphism: the sale method accepts a parameter of type Publication which allows any of its subclasses to be passed:

```
public void sellItem (Publication pPub)
```

We now want to broaden this further by accepting anything which implements the SaleableItem interface:

```
public void sellItem (SaleableItem pSelb)
```

When the type of a variable or parameter is defined as an interface, this works just like a superclass type. Any class which **implements** the interface is acceptable for assignment to the variable/parameter because the interface is a **type** and all classes implementing it are subtypes of that type.

This is now shown below...



The image shows the BI Norwegian Business School logo, which is a central blue square with 'BI' in white, surrounded by a colorful, multi-colored sunburst of lines. The lines are labeled with various business programs: Business, Strategic Marketing Management, International Business, Leadership & Organisational Psychology, Shipping Management, and Financial Economics. Below the logo is the text 'BI NORWEGIAN BUSINESS SCHOOL' and the EFMD EQUIS ACCREDITED logo.

## Empowering People. Improving Business.

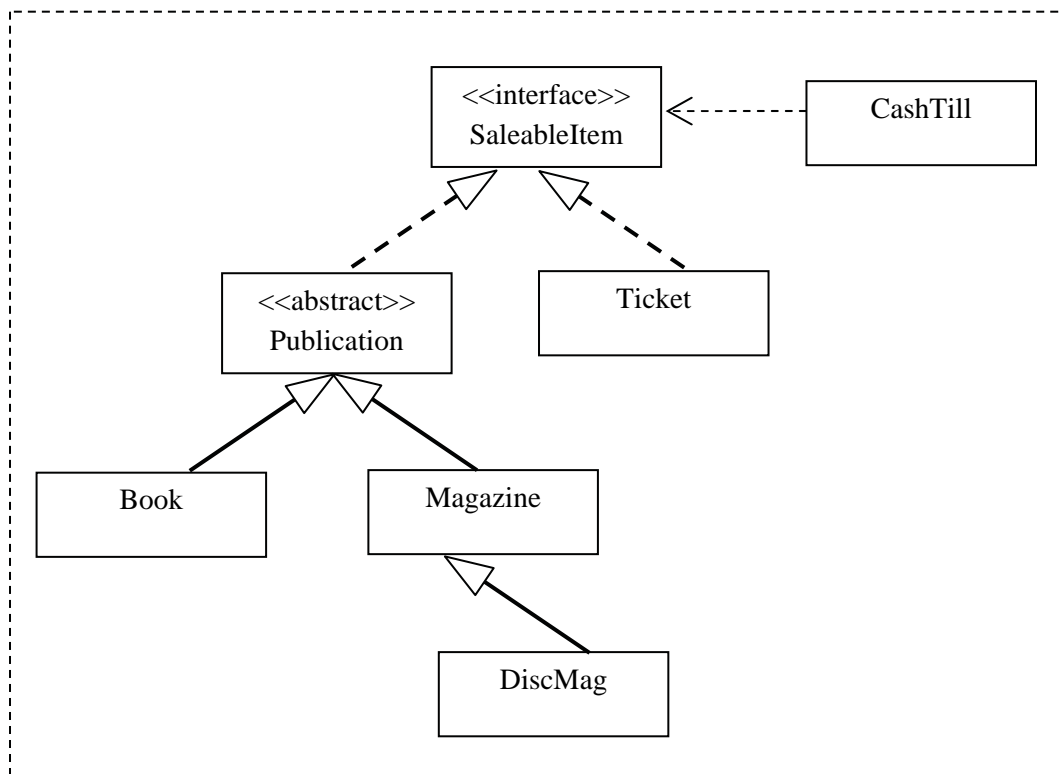
BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

[www.bi.edu/master](http://www.bi.edu/master)





CashTill is no longer directly dependent on class Publication – instead it is dependent on the interface SaleableItem.

The relationships from Publication and Ticket to SaleableItem are like inheritance arrows except that the lines are **dotted** – this shows that each class **implements** the interface.

#### 4.6 Extensibility Again

Polymorphism allows objects to be handled without regard for their precise class. This can assist in making systems extensible without compromising the encapsulation of the existing design.

For example, we could create new classes for more products or services and so long as they implement the SaleableItem interface the CashTill will be able to process them **without a single change to its code!**

An example could be ‘Sweets’. We could define a class Sweets to represent sweets in a jar. We can define the price of the sweets depending upon the weight and then sell the sweets by subtracting this weight from our total stock. This is not like selling a Publication, where we always subtract 1 from the stock, nor it this like selling tickets, where we just print them.

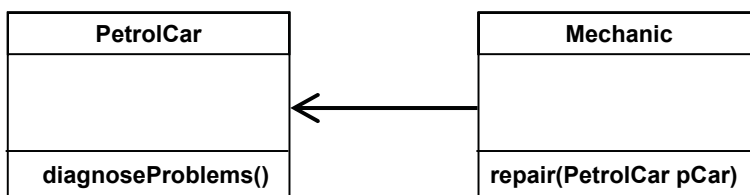
However if we create a class ‘Sweets’ that implements the SaleableItem interface our enhanced polymorphic cash till can sell them because it a sell any SaleableItem.

In this case, without polymorphism we would need to add an additional “sale” method to CashTill to handle Tickets, Sweets and further new methods for every new type of product to be sold. By defining the SaleableItem interface can introduce additional products without affecting CashTill at all. Poymorphism makes it easy to extend our programs and this is very important.

Interfaces allow software components to plug together more flexibly and extensibly, just as many other kinds of plugs and sockets enable audio, video, power and data connections in the everyday world. Think of the number of different electrical appliances which can be lugged into a standard power socket – and imagine how inconvenient it would be if instead you had to call out an electrician to wire up each new one you bought!

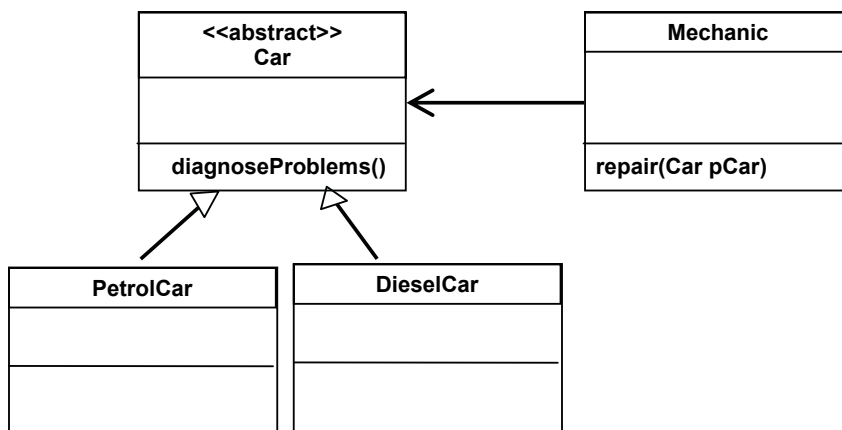
**Activity 5**

Adapt the following diagram by adding a class for Diesel cars in such a way that it can be used to illustrate polymorphism.



**Feedback 5**

This is one solution to this exercise...there are of course others.



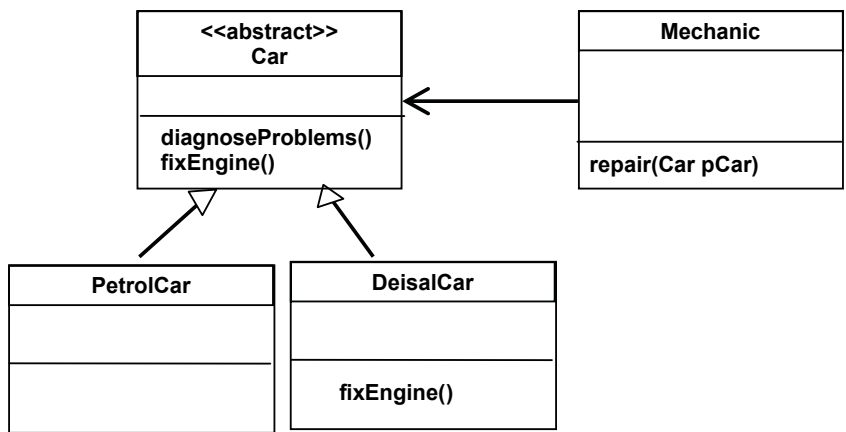
Here Mechanic is directly interacting with Car. In doing so it can interact with any subtype of Car e.g. Petrol, Diesel or any other type of Car developed in the future e.g. (Electric). These are all different (different shapes) and yet Mechanic can still interact with them as they are all Cars. This is polymorphic.

If an ElectricCar class was added Mechanic would still be able to work with them without making any changes to the Mechanic class.

**Activity 6**

Assume Car has a fixEngine() method that is overridden in DieselCar but not overridden in PetrolCar (as shown on the diagram below).

Look at this diagram and answer the following questions...



- a) Would the following line of code be valid inside the repair() method ? pCar.  
fixEngine();
- b) If a DieselCar object was passed to the repair() method which actual method  
would be invoked by pCar.fixEngine(); ?

# Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

**Get Help Now**



Go to [www.helpmyassignment.co.uk](http://www.helpmyassignment.co.uk) for more info



**Click on the ad to read more**

**Feedback 6**

- a) Yes! We can apply the method fixEngine() to any Car object as it is defined in the class Car.
- b) This would invoke the overridden method. The method must be defined in the class Car else the compiler will complain at compile time. However at run time the Java Runtime Environment (JRE) will identify the object passed as being of the subtype DieselCar and will invoke the overridden method. Clever stuff given that the repairCar() method is unaware of which type of car is actually passed.

## 4.7 Distinguishing Subclasses

What if we have an object handled polymorphically but need to check which subtype it **actually** is? The **instanceof** operator can do this:

*object instanceof class*

This test is **true** if the object is of the specified class (or a subclass), **false** otherwise.

Note that (**myDiscMag instanceof Magazine**) would be TRUE because a DiscMag is a Magazine

**instanceof** can also be used with an interface name on the right, in which case it tests whether the class implements the interface.

Strictly **instanceof** is testing whether the item on the left is of the **type**, or a subtype of, the type specified on the right. Doing this we could extend the CashTill class such that it displays a specific message depending upon the object sold.

```
public void saleType (SaleableItem pSelb)
{
    if (pSelb instanceof Publication)
    {
        System.out.println("This is a Publication");
    }
    else if (pSelb instanceof Ticket)
    {
        System.out.println("This is a Ticket");
    }
    else
    {
        System.out.println("This is a an unknown sale type");
    }
}
```

**pSelb instanceof Publication** will be true if pSelb is any subclass of Publication (i.e. a Book, Magazine or DiscMag). If we wished to we could equally test for a more specific subtype, e.g. **pSelb instanceof Book**

Notice that once we compromise the polymorphism by checking for subtypes we also compromise the extensibility of the system – new classes (e.g. Sweets) implementing the SaleableItem interface may also require new clauses adding to this if statement, so the change ripples through the system with the consequence that it becomes more costly and error-prone to maintain.

Instead of doing this we should try to package different behaviours into the subclasses themselves, e.g. we could define a **describeSelf()** method in the interface SaleableItem this would then need to be implemented in each class that implements the SaleableItem interface. Thus each subtype would display a message giving the type of item being sold. The if statement above, in CashTill, can then be replaced with **pSelb.describeSelf()**. Thus when we add new classes to the system we would not need to change the CashTill class.

## 4.8 Summary

Polymorphism allows us to refer to objects according to a superclass rather than their actual class.

Polymorphism makes it easy to extend our programs by adding additional classes without needing to change other classes.

We can manipulate objects by invoking operations defined for the superclass without worrying about which subclass is involved in any specific case.

Java ensures that the appropriate method for the actual class of the object is invoked at run-time.

Sometimes we want to employ polymorphism without all the classes concerned having to be in an inheritance hierarchy. The ‘interface’ construct allows us to provide shared interfaces (i.e. collections of operations) in this situation. When doing this there is no inherited implementation – each class must implement ALL the operations defined by the Interface.

Any number of classes can implement a particular interface.