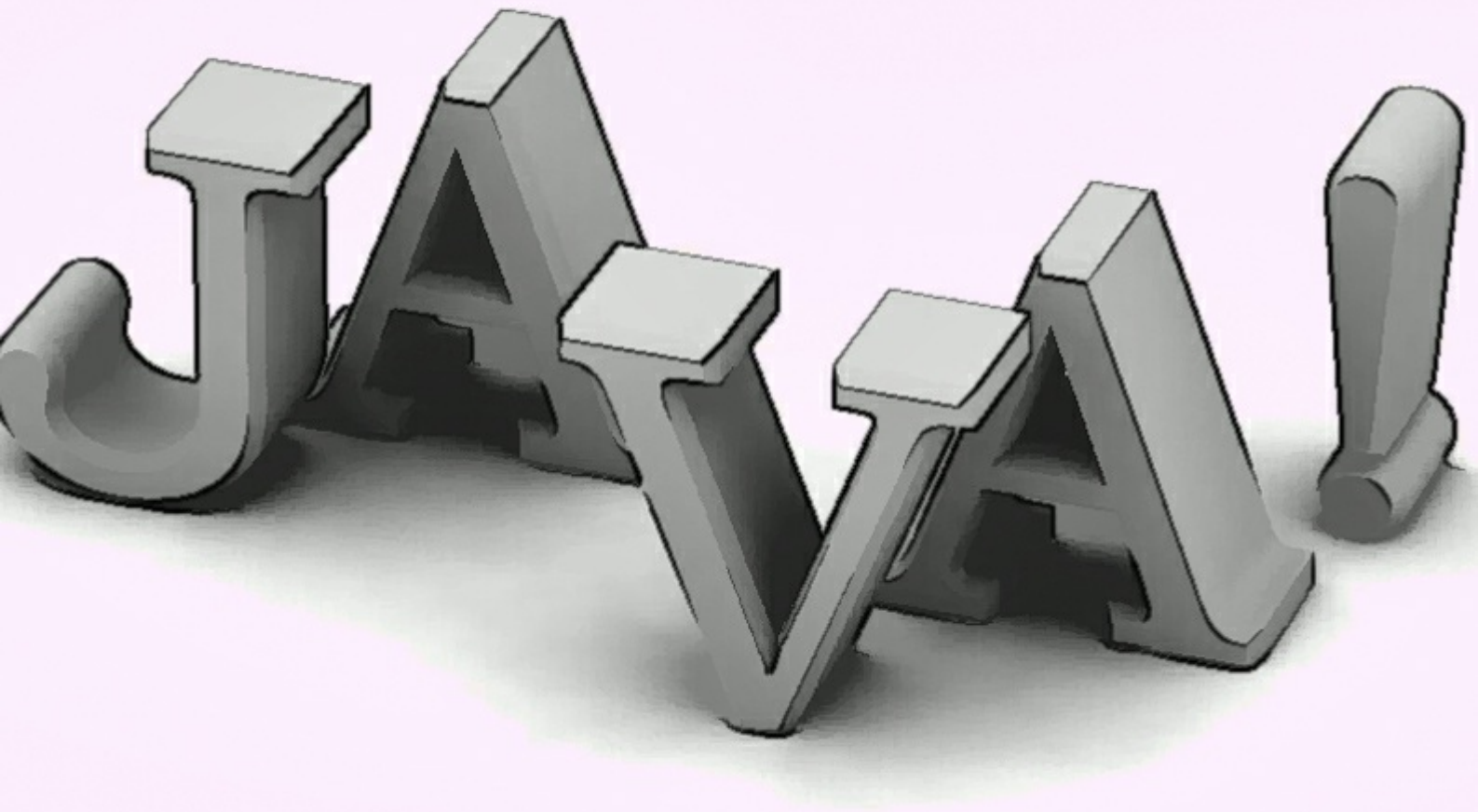
bookboon
The eBook company

Object Oriented Programming using Java

Simon Kendal



Simon Kendal

Object Oriented Programming Using Java

"To my wife Janice and daughter Cara, without whom life would be no fun at all!"

Download free eBooks at bookboon.com

Object Oriented Programming Using Java
1st edition
© 2009 Simon Kendal & bookboon.com
ISBN 978-87-7681-501-1

Download free eBooks at bookboon.com

Contents

	Foreword	**11**
1	**An Introduction to Object Orientated Programming**	**12**
1.1	A Brief History of Computing	12
1.2	Different Programming Paradigms	13
1.3	Why use the Object Orientation Paradigm?	15
1.4	Object Oriented Principles	16
1.5	What Exactly is Object Oriented Programming?	20
1.6	The Benefits of the Object Oriented Programming Approach	23
1.7	Summary	23
2	**The Unified Modelling Language (UML)**	**24**
2.1	An Introduction to UML	24
2.2	UML Class diagrams	25
2.3	UML Syntax	29
2.4	UML Package Diagrams	41



Click on the ad to read more

Download free eBooks at bookboon.com

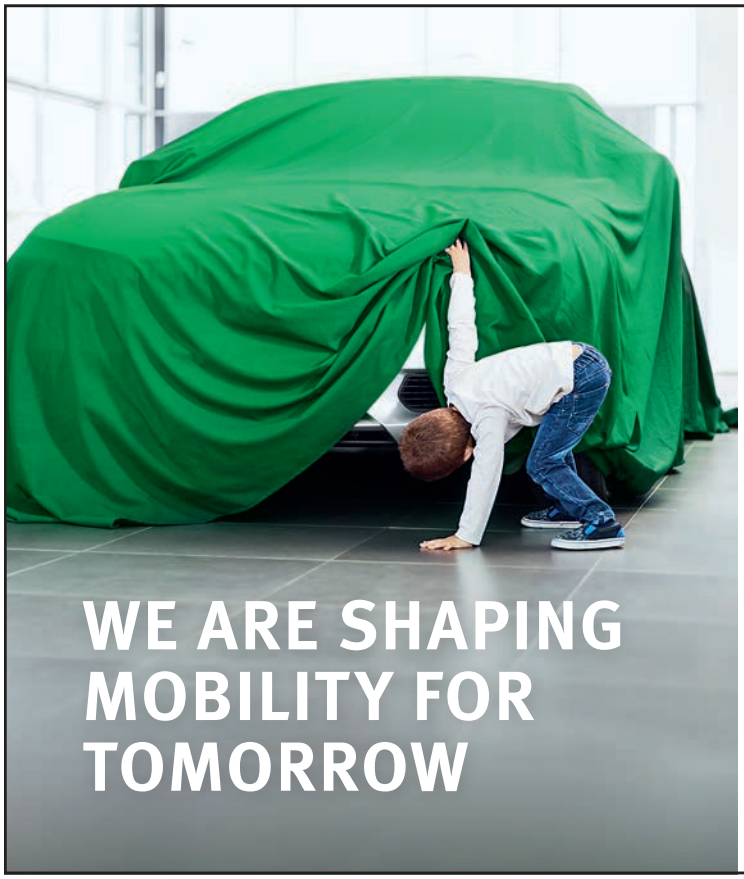
2.5	UML Object Diagrams	47
2.6	UML Sequence Diagrams	48
2.7	Summary	50
3	**Inheritance and Method Overriding**	**51**
3.1	Object Families	52
3.2	Generalisation and Specialisation	52
3.3	Inheritance	54
3.4	Implementing Inheritance in Java	61
3.5	Constructors	61
3.6	Constructor Rules	62
3.7	Access Control	64
3.8	Abstract Classes	65
3.9	Overriding Methods	66
3.10	The 'Object' Class	68
3.11	Overriding toString() defined in 'Object'	70
3.12	Summary	71



Click on the ad to read more

Download free eBooks at bookboon.com

4	**Object Roles and the Importance of Polymorphism**	**72**
4.1	Class Types	72
4.2	Substitutability	75
4.3	Polymorphism	76
4.4	Extensibility	77
4.5	Interfaces	84
4.6	Extensibility Again	90
4.7	Distinguishing Subclasses	93
4.8	Summary	94
5	**Overloading**	**95**
5.1	Overloading	96
5.2	Overloading To Aid Flexibility	96
5.3	Summary	99
6	**Object Oriented Software Analysis and Design**	**100**
6.1	Requirements Analysis	101
6.2	The Problem	102
6.3	Listing Nouns and Verbs	104
6.4	Identifying Things Outside The Scope of The System	105



How will people travel in the future, and how will goods be transported? What resources will we use, and how many will we need? The passenger and freight traffic sector is developing rapidly, and we provide the impetus for innovation and movement. We develop components and systems for internal combustion engines that operate more cleanly and more efficiently than ever before. We are also pushing forward technologies that are bringing hybrid vehicles and alternative drives into a new dimension – for private, corporate, and public use. The challenges are great. We deliver the solutions and offer challenging jobs.

www.schaeffler.com/careers

SCHAEFFLER

Click on the ad to read more

Download free eBooks at bookboon.com

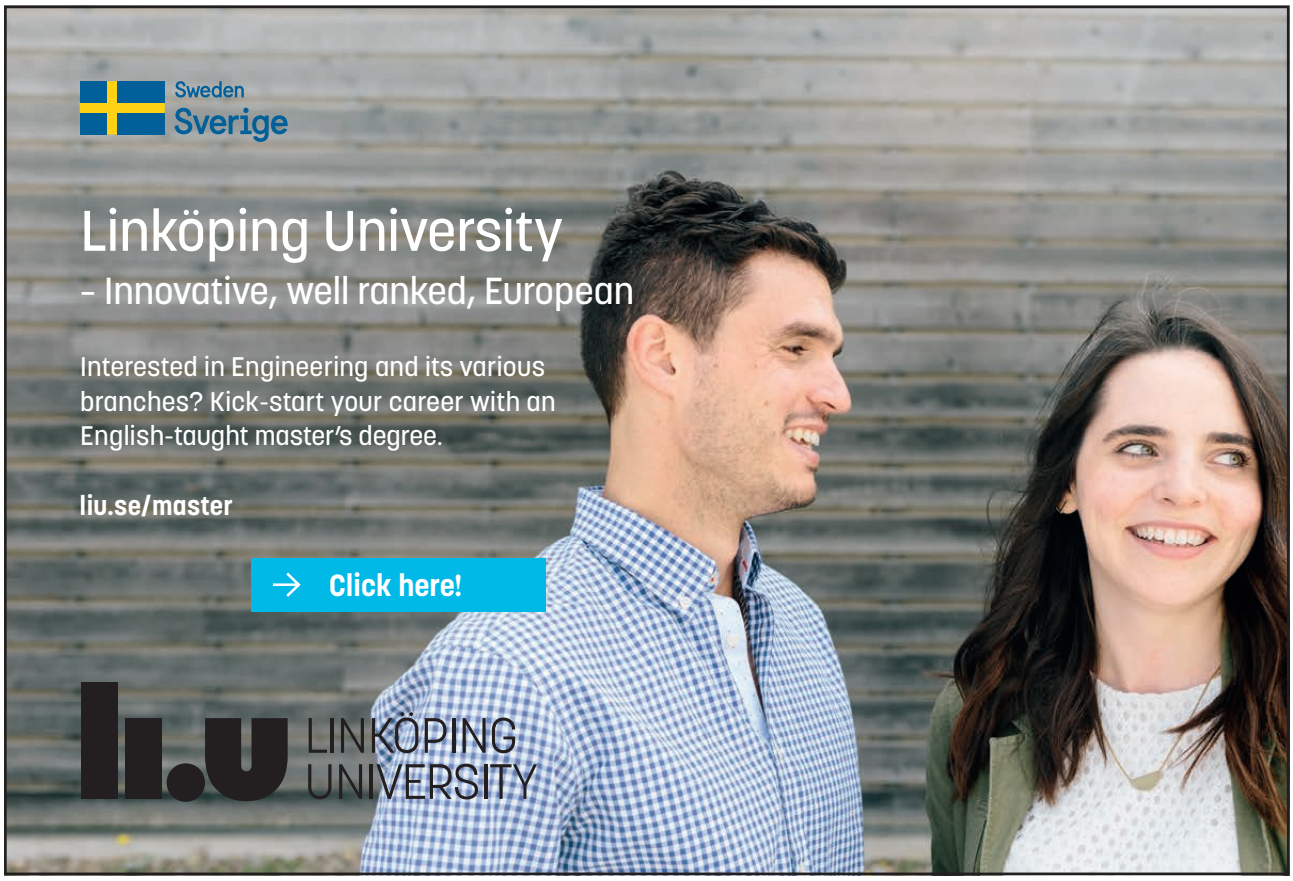
6.5	Identifying Synonyms	106
6.6	Identifying Potential Classes	107
6.7	Identifying Potential Attributes	108
6.8	Identifying Potential Methods	108
6.9	Identifying Common Characteristics	109
6.10	Refining Our Design using CRC Cards	110
6.11	Elaborating Classes	112
6.12	Summary	113
7	**The Collections Framework**	**114**
7.1	An Introduction to Collections	115
7.2	Collection Interfaces	115
7.3	Old and New Collections	116
7.4	Lists	116
7.5	Sets	117
7.6	Maps	118
7.7	Collection Implementations	119
7.8	Overview of the Collections Framework	121
7.9	An Example Using Un-typed Collections	123



Click on the ad to read more

Download free eBooks at bookboon.com

7.10	An Example Using Typed Collections	125
7.11	A Note About Sets	127
7.12	Summary	132
8	**Java Development Tools**	**133**
8.1	Software Implementation	134
8.2	The JRE	137
8.3	Java Programs	138
8.4	The JDK	139
8.5	Eclipse	140
8.6	Eclipse Architecture	140
8.7	Eclipse Features	142
8.8	NetBeans	142
8.9	Developing Graphical Interfaces Using NetBeans	144
8.10	Applying Layout Managers Using NetBeans	145
8.11	Adding Action Listeners	148
8.12	The Javadoc Tool	149
8.13	Summary	153



Click on the ad to read more
Download free eBooks at bookboon.com

9	**Creating And Using Exceptions**	**154**
9.1	Understanding the Importance of Exceptions	154
9.2	Kinds of Exception	157
9.3	Extending the Exception Class	158
9.4	Throwing Exceptions	160
9.5	Catching Exceptions	161
9.6	Summary	161
10	**Agile Programming**	**162**
10.1	Agile Approaches	163
10.2	Refactoring	163
10.3	Examples of Refactoring	164
10.4	Support for Refactoring	164
10.5	Unit Testing	166
10.6	Automated Unit Testing	167
10.7	Regression Testing	167
10.8	JUnit	168
10.9	Examples of Assertions	168
10.10	Several Test Examples	168
10.11	Running Tests	172

STUDY FOR YOUR MASTER'S DEGREE
IN THE CRADLE OF SWEDISH ENGINEERING

Chalmers University of Technology conducts research and education in engineering and natural sciences, architecture, technology-related mathematical sciences and nautical sciences. Behind all that Chalmers accomplishes, the aim persists for contributing to a sustainable future – both nationally and globally.

Visit us on **Chalmers.se** or **Next Stop Chalmers** on facebook.



Click on the ad to read more

Download free eBooks at bookboon.com

10.12	Test Driven Development (TDD)	173
10.13	TDD Cycles	174
10.14	Claims for TDD	174
10.15	Summary	174
11	**Case Study**	**175**
11.1	The Problem	176
11.2	Preliminary Analysis	177
11.3	Further Analysis	183
11.4	Documenting the design using UML	188
11.5	Prototyping the Interface	193
11.6	Revising the Design to Accommodate Changing Requirements	193
11.7	Packaging the Classes	197
11.8	Programming the Message Classes	199
11.9	Programming the Client Classes	205
11.10	Creating and Handling UnknownClientException	206
11.11	Programming the Main classes	208
11.12	Programming the Interface	209
11.13	Using Test Driven Development and Extending the System	210
11.14	Generating Javadoc	212
11.15	Running the System and Potential Compiler Warnings	213
11.16	The Finished System…	214
11.17	Summary	216

Download free eBooks at bookboon.com

Foreword

This book aims to instil the reader with an understanding of the Object Oriented approach to programming and aims to develop some practical skills along the way. These practical skills will be developed by small exercises that the reader will be invited to undertake and the feedback that will be provided.

The concepts that will be explained and skills developed are in common use among programmers using many modern object oriented languages and are thus transferrable from one language to another. However for practical purposes these concepts are explored and demonstrated using the Java programming language.

While the Java programming language is used to highlight and demonstrate the application of fundamental object oriented principles and modelling techniques this book is not an introduction to Java programming. The reader will be expected to have an understanding of basic programming concepts and their implementation in Java (inc. the use of loops, selection statements, performing calculations, arrays, data types and a basic understanding of file handling).

This text is designed not as a theoretical textbook but as a learning tool to aid in understanding theoretical concepts and learning the practical skills required to implement these. To this end each chapter will incorporate small exercises with solutions and feedback provided.

At the end of the book one larger case study will be described – this will be used to illustrate the application of the techniques explored in the earlier chapters. This case study will culminate in the development of a complete Java program that can be downloaded with this book.

Download free eBooks at bookboon.com

1 An Introduction to Object Orientated Programming

Introduction

This chapter will discuss different programming paradigms and the advantages of the Object Oriented approach to software development and modelling. The concepts on which object orientation depend (abstraction, encapsulation, inheritance and polymorphism) will be explained.

Objectives

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

- Explain what Object Oriented Programming is,
- Describe the benefits of the Object Oriented programming approach and
- Understand and the basic concepts of abstraction, encapsulation, generalisation and polymorphism on which object oriented programming relies.

All of these issues will be explored in much more detail in later chapters of this book.

This chapter consists of six sections:-

1) A Brief History of Computing
2) Different Programming Paradigms
3) Why use the Object Oriented Paradigm
4) Object Oriented Principles
5) What Exactly is Object Oriented Programming?
6) The Benefits of the Object Oriented Programming Approach.

1.1 A Brief History of Computing

Computing is a constantly changing our world and our environment. In the 1960s large machines called mainframes were created to manage large volumes of data (numbers) efficiently. Bank account and payroll programs changed the way organisations worked and made parts of these organisations much more efficient. In the 1980s personal computers became common and changed the way many individuals worked. People started to own their own computers and many used word processors and spreadsheets applications (to write letters and to manage home accounts). In the 1990s email became common and the world wide web was born. These technologies revolutionised communications allowing individuals to publish information that could easily be accessed on a global scale. The ramifications of these new technologies are still not fully understood as society is adapting to opportunities of internet commerce, new social networking technologies (twitter, facebook, myspace, online gaming etc) and the challenges of internet related crime.

Download free eBooks at bookboon.com

Just as new computing technologies are changing our world so to are new techniques and ideas changing the way we develop computer systems. In the 1950s the use machine code (unsophisticated, complex and machine specific) languages were common.

In the 1960s high level languages, which made programming simpler, became common. However these led to the development of large complex programs that were difficult to manage and maintain.

In the 1970s the structured programming paradigm became the accepted standard for large complex computer programs. The structured programming paradigm proposed methods to logically structure the programs developed into separate smaller, more manageable components. Furthermore methods for analysing data were proposed that allowed large databases to be created that were efficient, preventing needless duplication of data and protected us against the risks associated with data becoming out of sync. However significant problems still persisted in a) understanding the systems we need to create and b) changing existing software as users requirements changed.

In the 1980s 'modular' languages, such as Modula-2 and ADA were developed that became the precursor to modern Object Oriented languages.

In the 1990s the Object Oriented paradigm and component-based software development ideas were developed and Object Oriented languages became the norm from 2000 onwards.

The object oriented paradigm is based on many of the ideas developed over the previous 30 years of abstraction, encapsulation, generalisation and polymorphism and led to the development of software components where the operation of the software and the data it operates on are modelled together. Proponents of the Object Oriented software development paradigm argue that this leads to the development of software components that can be re-used in different applications thus saving significant development time and cost savings but more importantly allow better software models to be produced that make systems more maintainable and easier to understand.

It should perhaps be noted that software development ideas are still evolving and new agile methods of working are being proposed and tested. Where these will lead us in 2020 and beyond remains to be seen.

1.2 Different Programming Paradigms

The structured programming paradigm proposed that programs could be developed in sensible blocks that make the program more understandable and easier to maintain.

Download free eBooks at bookboon.com

Activity 1

Assume you undertake the following activities on a daily basis. Arrange this list into a sensible order then split this list into three blocks of related activities and give each block a heading to summarise the activities carried out in that block.

Get out of bed
Eat breakfast
Park the car
Get dressed
Get the car out of the garage
Drive to work
Find out what your boss wants you to do today
Feedback to the boss on today's results.
Do what the boss wants you to do

Feedback 1

You should have been able to organise these into groups of related activities and give each group a title that summarises those activities.

Get up:-
- Get out of bed
- Get dressed
- Eat breakfast

Go to Work:-
- Get the car out of the garage
- Drive to work
- Park the car

Do your job:-
- Find out what your boss wants you to do today
- Do what the boss wants you to do
- Feedback to the boss on today's results.

By structuring our list of instructions and considering the overall structure of the day (Get up, go to work, do your job) we can change and improve one section of the instructions without changing the other parts. For example we could improve the instructions for going to work…

- Listen to the local traffic and weather report
- Decide whether to go by bus or by car
- If going by car, get the car and drive to work.
- Else walk to the bus station and catch the bus

without worrying about any potential impact this may have on 'getting up' or 'doing your job'. In the same way structuring computer programs can make each part more understandable and make large programs easier to maintain.

The Object Oriented paradigms suggest we should model instructions in a computer program with the data they manipulate and store these as components together. One advantage of doing this is we get reusable software components.

Download free eBooks at bookboon.com

Activity 2

Imagine and a personal address book with some data stored about your friends

Name,
Address,
Telephone Number.

List three things that you may do to this address book.

Next identify someone else who may use an identical address book for some purpose other than storing a list of friends.

Feedback 2

With an address book we would want to be able to perform the following actions:- find out details of a friend i.e. their telephone number, add an address to the address book and, of course, delete an address.

We can create a simple software component to store the data in the address book (i.e. list of names etc) and the operations we can perform (i.e add address, find telephone number etc).

By creating a simple software component to store and manage addresses of friends we can reuse this in another software system i.e. it could be used by a business manager to store and find details of customers. It could also become part of a library system to be used by a librarian to store and retrieve details of the users of the library.

Thus in object oriented programming we can create re-usable software components (in this case an address book).

The Object Oriented paradigm builds upon and extends the ideas behind the structured programming paradigm of the 1970s.

1.3 Why use the Object Orientation Paradigm?

While we can focus our attention on the actual program code we are writing, whatever development methodology is adopted, it is not the creation of the code that is generally the source of most problems. Most problems arise from:-

- poor analysis and design: the computer system we create doesn't do the right thing.
- poor maintainability: the system is hard to understand and revise when, as is inevitable, requests for change arise.

Statistics show 70% of the cost of software is not incurred during its initial development phase but is incurred during subsequent years as the software is amended to meet the ever changing needs of the organisation for which it was developed. For this reason it is essential that software engineers to everything possible to ensure that software is easy to maintain during the years after its initial creation.

Download free eBooks at bookboon.com

The Object Oriented programming paradigm aims to help overcome these problems by helping with the analysis and design tasks during the initial software development phase (see chapter 6 for more details on this) and by ensuring software is robust and maintainable (see chapters 9–11 for information on the support Object Orientation and Java provides).

1.4 Object Oriented Principles

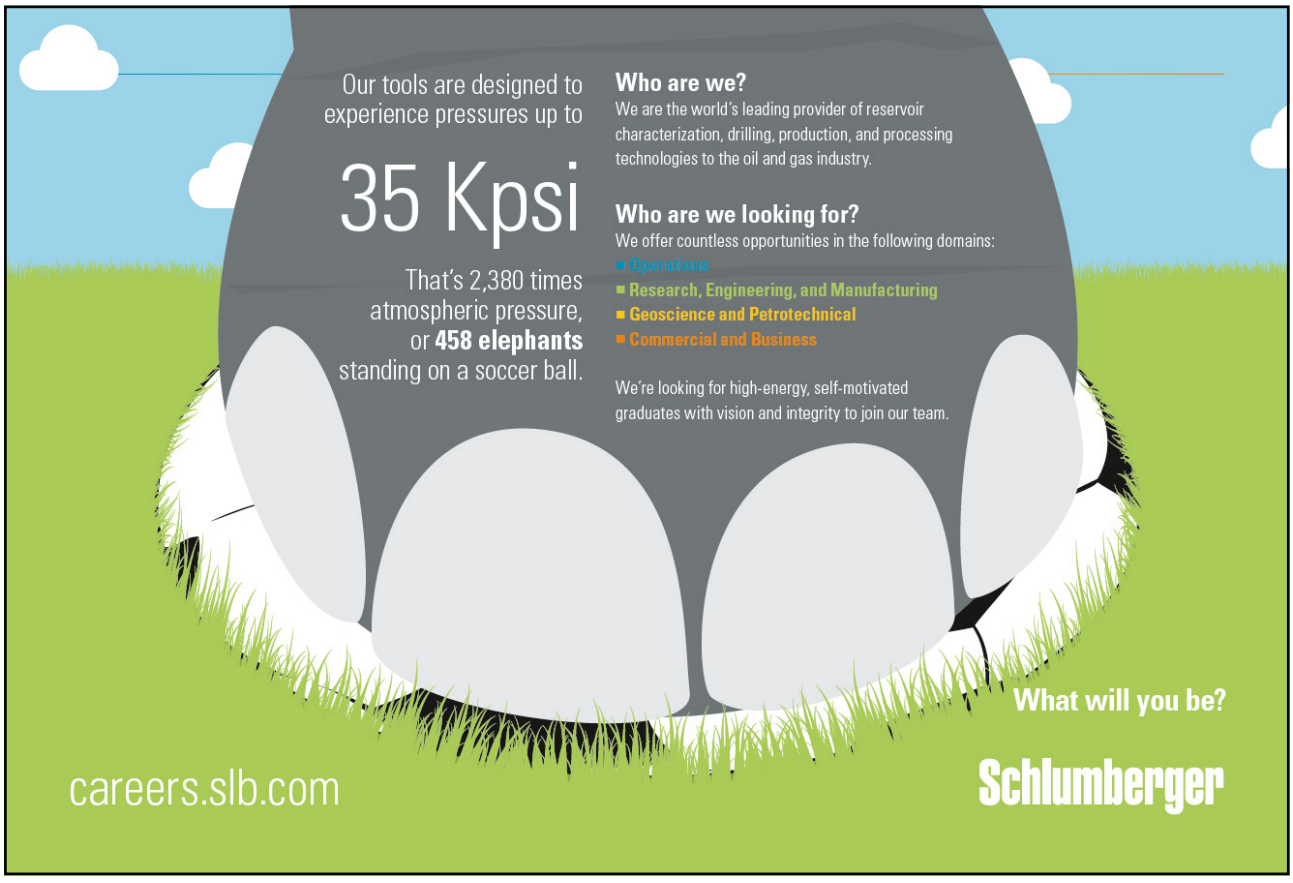
Abstraction and encapsulation are fundamental principles that underlie the Object Oriented approach to software development. Abstraction allows us to consider complex ideas while ignoring irrelevant detail that would confuse us. Encapsulation allows us to focus on what something does without considering the complexities of how it works.

Activity 3

Consider your home and imagine you were going to swap your home for a week with a new friend.

Write down three essential things you would tell them about your home and that you would want to know about their home.

Now list three irrelevant details that you would not tell your friend.



Click on the ad to read more

Download free eBooks at bookboon.com

Feedback 3

You presumably would tell them the address, give them a basic list of rooms and facilities (e.g. number of bedrooms) and tell them how to get in (i.e which key would operate the front door and how to switch off the burglar alarm (if you have one).

You would not tell them irrelevant details (such as the colour of the walls, seats etc) as this would overload them with useless information.

Abstraction allows us to consider the important high level details of your home, e.g. the address, without becoming bogged down in detail.

Activity 4

Consider your home and write down one item, such as a television, that you use on a daily basis (and briefly describe how you operate this item).

Now consider how difficult it would be to describe the internal components of this item and give full technical details of how it works.

Feedback 4

Describing how to operate a television is much easier than describing its internal components and explaining in detail exactly how it works. Most people do not even know all the components of the appliances they use or how they work – but this does not stop them from using appliances every day.

You may not know the technical details such as how the light switches are wired together and how they work internally but you can still switch the lights on and off in your home (and in any new building you enter).

Encapsulation allows us to consider what a light switch does, and how we operate it, without needing to worry about the technical detail of how it actually works.

Two other fundamental principles of Object Orientation are Generalization/specialization (which allows us to make use of inheritance) and polymorphism.

Generalisation allows us to consider general categories of objects which have common properties and then define specialised sub classes that inherit the properties of the general categories.

Activity 5

Consider the people who work in a hospital – list three common occupations of people you would expect to be employed there.

Now for each of these common occupations list two or three specific categories of staff.

Download free eBooks at bookboon.com

Feedback 5

Depending upon your knowledge of the medical profession you may have listed three very general occupations (e.g. doctor, nurse, cleaner) or you may have listed more specific occupations such as radiologist, surgeon etc.

Whatever your initial list you probably would have been able to specify more specialised categories of these occupations e.g.

Doctor:-
- Trainee doctor,
- Junior doctor,
- Surgeon,
- Radiologist,
- etc.

Nurse:-
- Triage nurse,
- Midwife,
- Ward sister

Cleaner:-
- General cleaner
- Cleaning supervisor

Now we have specified some general categories and some more specialised categories of staff we can consider the general things that are true for all doctors, all nurses etc.

Activity 6

Make one statement about doctors that you would consider to be true for all doctors and make one statement about surgeons that would **not** be true for all doctors.

Feedback 6

You could make a statement that all doctors have a knowledge of drugs, can diagnose medical conditions and can prescribe appropriate medication.

For surgeons you could say that they know how to use scalpels and other specialised pieces of equipment and they can perform operations.

According to our list above all surgeons are doctors and therefore still have a knowledge of medical conditions and can prescribe appropriate medication. However not all doctors are surgeons and therefore not all doctors can perform operations.

What ever we specify as true for doctors is also true for trainee doctors, junior doctors etc – these specialised categories (or classes) can inherit the attributes and behaviours associated with the more general class of 'doctor'.

Download free eBooks at bookboon.com

Generalisation / specialisation allows us to define general characteristics and operations of an object and allows us to create more specialised versions of this object. The specialised versions of this object will automatically inherit all of the characteristics of the more generalised object.

The final principle underlying Object Orientation is Polymorphism which is the ability to interact with a object as its generalized category regardless of its more specialised category.

Activity 7

Make one statement about how a hospital manager may interact with all doctors employed at their hospital irrespective of what type of doctor they are.

Feedback 7

You may have considered that a hospital manager could pay all doctors (presumably this will be done automatically at the end of every month) and could discipline any doctor guilty of misconduct – of course this would be true for other staff as well. More specifically a manager could check that a doctors medical registration is still current. This would be something that management would need to do for all doctors irrespective of what their specialism is.

Furthermore if the hospital employed new specialist doctor (e.g. a Neurologist), without knowing anything specific about this specialism, hospital management would still know that a) these staff needed to be paid and b) their medical registration must be checked. i.e. they are still doctors and need to be treated as such.



Click on the ad to read more

Download free eBooks at bookboon.com

Using the same idea polymorphism allows computer systems to be extended, with new specialised objects being created, while allowing current parts of the system to interact with new object without concern for the specific properties of the new objects.

1.5 What Exactly is Object Oriented Programming?

Activity 8

Think of an object you possess. Describe its current state and list two or three things you can do with that object.

Feedback 8

You probably thought about an entirely physical object such as a watch, a pen, or a car.

Objects have a current status. A watch has a time (represented internally by wheels and cogs or in an electronic component). A pen has a certain amount of ink in it and has its lid on or off. A car has a current speed and has a certain amount of fuel inside it.

Specific behaviour can also be associated with each object (things that you can do with it):- a watch can be checked to find out its time, its time can also be set. A pen can be used to write with and a car can be driven.

You can also think of other non physical things as objects:- such as a bank account. A bank account is not something that can be physically touched but intellectually we can consider a bank account to be an object. It also has a current status (the amount of money in it) and it also has behaviour associated with it (most obviously deposit money and withdraw money).

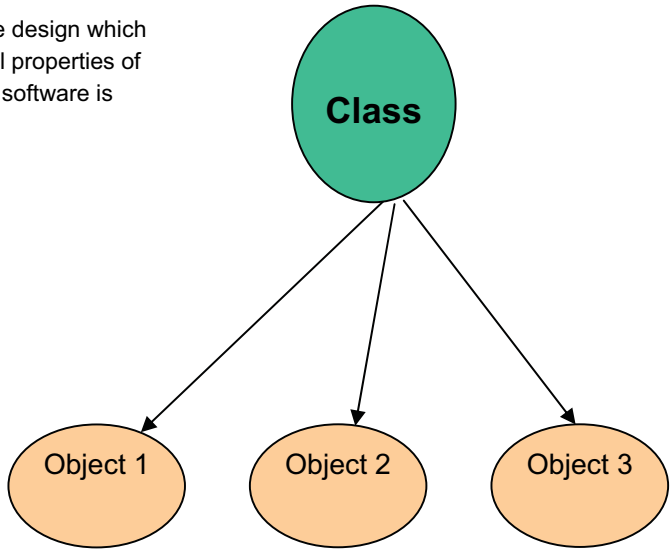
Object oriented programming it a method of programming that involves the creation of intellectuals objects that model a business problem we are trying to solve (e.g. a bank account, a bank customer and a bank manager – could all be objects in a computerised banking system). With each object we model that data associated with it (i.e. it status at any particular point in time) and the behaviour associated with it (what our computer program should allow that object to do).

In creating a object oriented program we define the properties of a class of objects (e.g. all bank accounts) and then create individual objects from this class (e.g. your bank account).

Download free eBooks at bookboon.com

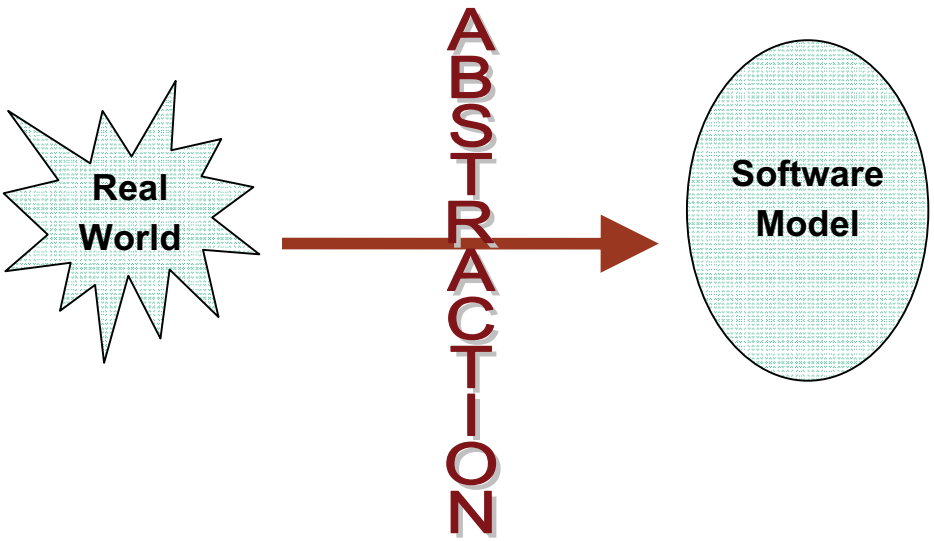
A 'class' is a software design which describes the general properties of something which the software is modelling.

Individual 'objects' are created from the class design for each actual thing



However deciding just what classes we should create in our system is not a trivial task as the real world is complex and messy (see chapter 6 for more advice on how to go about this). In essence we need to create an abstract model of the real world that focuses on the essential aspects of a problem and ignores irrelevant complexities. For example in the real world bank account holders sometimes need to borrow money and occasionally their money may get stolen by a pick pocket. If we were to create a bank account system should we allow customers to borrow money? Should we acknowledge that their cash may get stolen and build in some method of them getting an immediate loan – or is this an irrelevant detail that would just add complexity to the system and provide no real benefit to the bank?

Using object oriented analysis and design techniques our job would be to look at the real world and come up with a simplified abstract model that we could turn into a computer system. How good our final system is will depend upon how good our software model is.



Download free eBooks at bookboon.com

Activity 9

Consider a computer system that will allow items to be reserved from a library. Imagine one such item that you may like to reserve and list two or three things that a librarian may want to know about this item.

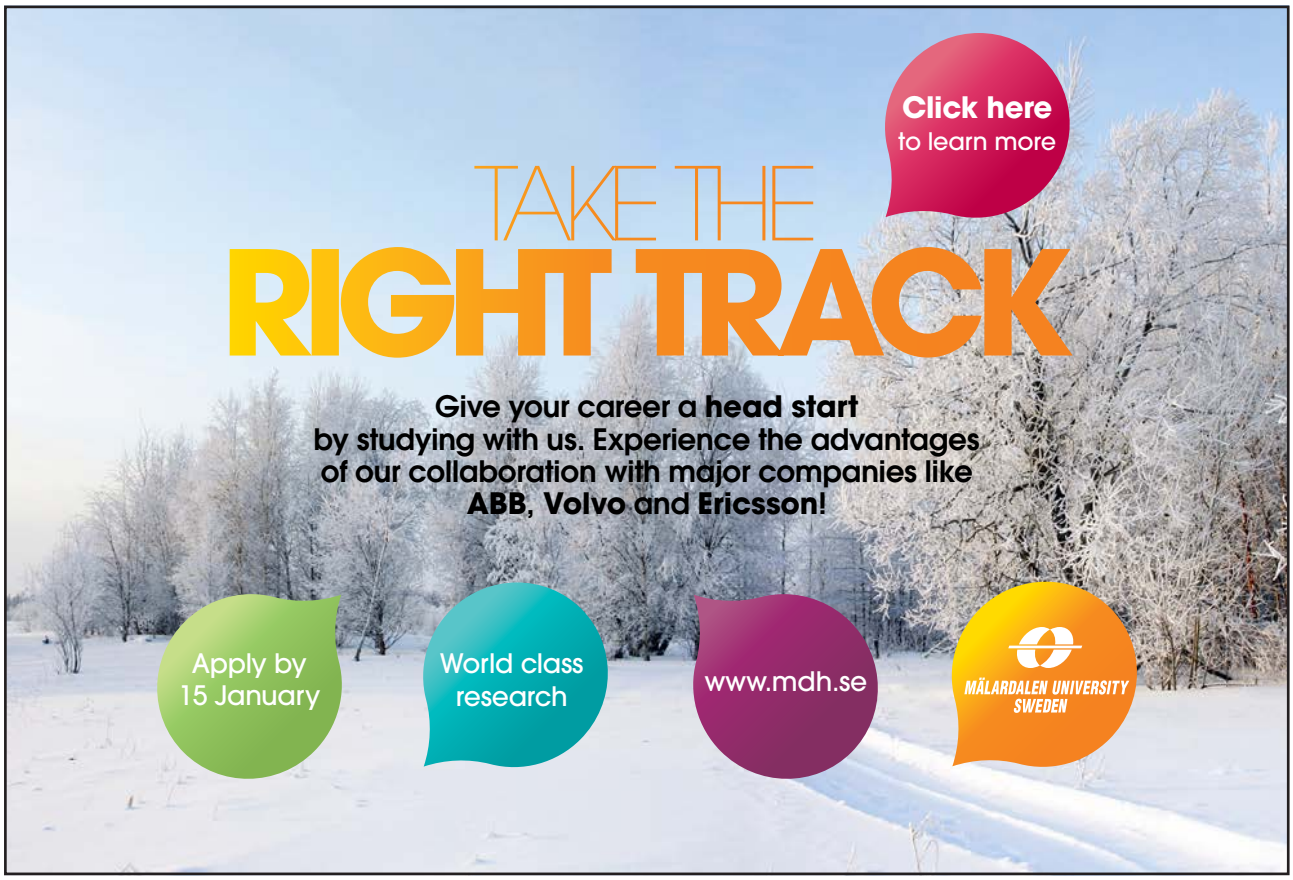
Feedback 9

You may have thought of a book you wish to reserve in which case the librarian may need to know the title of the book and its author.

Thus for every object we create in a system we need to define the attributes of that object i.e. the things we need to know about it.

Activity 10

Note that we can consider a reservation to be is an intellectual object (where the actual item is a physical object). Considering this intellectual object (item reservation) list two or three actions the librarian may need to perform on this object.



Click on the ad to read more

Download free eBooks at bookboon.com

Feedback 10

The librarian may need to cancel this reservation (if you change your mind) they may also need to tell you if the item has arrived in stock for you to collect.

Thus for each object we need to define the operations that will be performed on that object (as well as its attributes).

Activity 11

Considering the most general category of object that can be borrowed from a library, a 'loan item', list two or three more specific subcategory of object a library can lend out.

Feedback 11

Having defined the most general category of object (we call this a class) – something that can be borrowed – we may want to define more specialised sub-classes (e.g. books, magazines, audio/visual material). These will share the attributes defined for the general class but will have specific differences (for example there could be a charge for borrowing audio/visual items).

1.6 The Benefits of the Object Oriented Programming Approach

Whether or not you develop programs in an object oriented way, before you write the software you must first develop a model of what that software must be able to do and how it should work. Object oriented modelling is based on the ideas of abstraction, encapsulation, inheritance and polymorphism.

The general proponents of the object oriented approach claims that this model provides:

- better abstractions (modelling information and behaviour together)
- better maintainability (more comprehensible, less fragile)
- better reusability (classes as encapsulated components)

We will return to look at these claims later in Chapter 11 as we see a case study showing in detail how object oriented analysis works and how the resultant models can be implemented in an object oriented programming language (i.e. Java).

1.7 Summary

Object oriented programming involves the creation of classes by modelling the real world. This allows more specialised classes to be created that inherit the behaviour of the generalised classes. Polymorphic behaviour means that systems can be changed, as business needs change, by adding new specialised classes and these classes can be accessed by the rest of the system without any regard to their specialised behaviour and without changing other parts of the current system.

Download free eBooks at bookboon.com

2 The Unified Modelling Language (UML)

Introduction

This chapter will introduce you to the roles of the Unified Modelling Language (UML) and explain the purpose of four of the most common diagrams (class diagrams, object diagrams, sequence diagrams and package diagrams). Particular emphasis will be placed on class diagrams as these are the most used part of the UML notation.

Objectives

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

- Explain what UML is and explain the role of four of the most common diagrams,
- Draw class diagrams, object diagrams, sequence diagrams and package diagrams.

The material covered in this chapter will be expanded on throughout later chapters of the book and the skills developed here will be used in later exercises (particularly regarding class diagrams.

This chapter consists of six sections:-

1) An introduction to UML
2) UML Class Diagrams
3) UML Syntax
4) UML Package Diagrams
5) UML Object diagrams
6) UML Sequence Diagrams

2.1 An Introduction to UML

The Unified Modelling Language, UML, is sometimes described as though it was a methodology. It is not!

A methodology is a system of processes in order to achieve a particular outcome e.g. an organised sequence of activities in order to gather user requirements. UML on the other hand a precise diagramming notation that will allow program designs to be represented and discussed. As it is graphical in nature it becomes easy to visualise, understand and discuss the information presented in the diagram. However as the diagrams represent technical information they must be precise and clear – in order for them to work therefore there is a precise notation that must be followed.

Download free eBooks at bookboon.com

As UML is not a methodology it is left to the user to follow whatever processes they deem appropriate in order to generate the designs described by the diagrams. UML does not constrain this – it merely allows those designs to be expressed in an easy to use, but precise, graphical notation.

2.2 UML Class diagrams

Classes are the basic components of any object oriented software system and UML class diagrams provide an easy way to represent these. As well as showing individual classes, in detail, class diagrams show multiple classes and how they are related to each other. Thus a class diagram shows the architecture of a system.

A class consists of:-

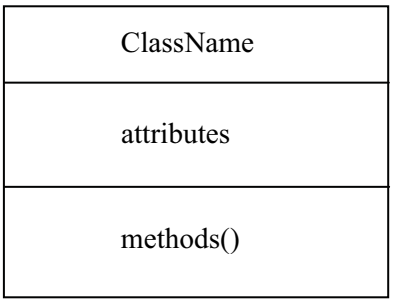
- a unique name (conventionally starting with an uppercase letter)
- a list of attributes (int, double, boolean, String etc)
- a list of methods



Click on the ad to read more

Download free eBooks at bookboon.com

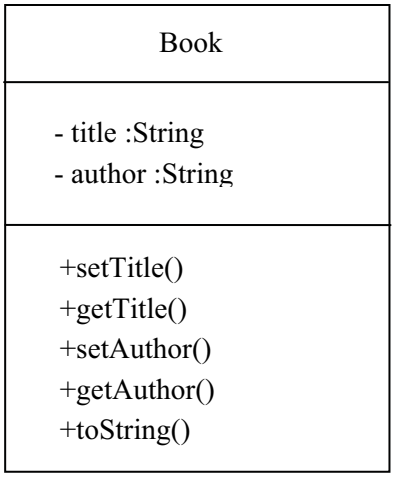
This is shown in a simple box structure…



For attributes and methods visibility modifiers are shown (+ for public access, – for private access). Attributes normally being kept private and methods normally made public.

Note: String shown above is not a primitive data type but is itself a class. Hence it starts with a capital letter.

Thus a class Book, with String attributes of title and author, and the following methods setTitle(), getTitle(), setAuthor(), getAuthor() and toString() would be shown as…

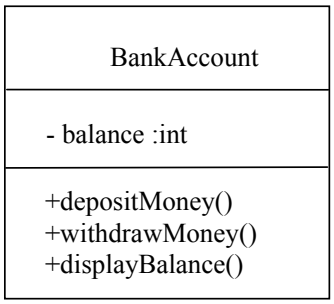


Activity 1

Draw a diagram to represent a class called 'BankAccount' with the attribute balance (of type int) and methods depositMoney() and withdrawMoney(). Show appropriate visibility modifiers.

Download free eBooks at bookboon.com

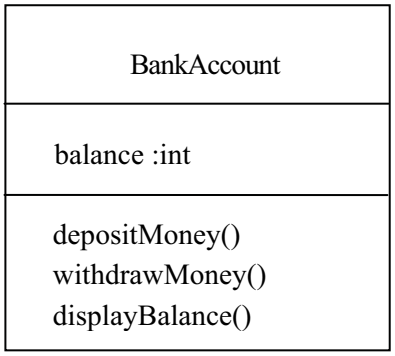
Feedback 1



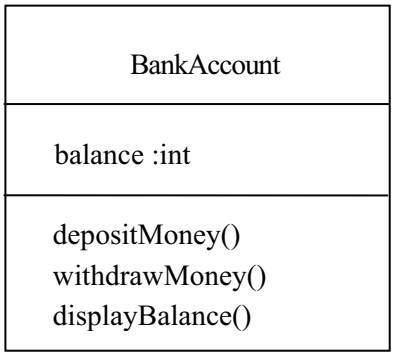
The diagram above shows this information

UML allows us to suppress any information we do not wish to highlight in our diagrams – this allows us to suppress irrelevant detail and bring to the readers attention just the information we wish to focus on. Therefore the following are all valid class diagrams…

Firstly with the access modifiers not shown…



Secondly with the access modifiers and the data type not shown…



Download free eBooks at bookboon.com

And finally with the attributes and methods not shown…



i.e. there is a class called 'BankAccount' but the details of this are not being shown.

Of course virtually all Java programs will be made up of many classes and classes will relate to each other – some classes will make use of other classes. These relationships are shown by arrows. Different type of arrow indicate different relationships (including inheritance and aggregation relationships).

In addition to this class diagrams can make use of keywords, notes and comments.

As we will see in examples that follow, a class diagram can show the following information:-

- Classes
- attributes
- operations
- visibility



Click on the ad to read more

Download free eBooks at bookboon.com

- Relationships
- navigability
- multiplicity
- dependency
- aggregation
- composition

- Generalization / specialization
- inheritance
- interfaces

- Keywords
- Notes and Comments

2.3 UML Syntax

As UML diagrams convey precise information there is a precise syntax that should be followed.

Attributes should be shown as: *visibility name: type multiplicity*

Where visibility is one of:-

- '+' public
- '-' private
- '#' protected
- '~' package

and Multiplicity is one of:-

- 'n' exactly n
- '*' zero or more
- 'm..'n' between m and n

The following are examples of attributes correctly specified using UML:-

- **custRef: int** [**1**]

  a private attribute custRef is a single int value
  this would often be shown as – **custRef: int** However with no multiplicity shown we cannot safely assume a multiplicity of one was intended by the author.

Download free eBooks at bookboon.com

 # itemCodes: String [1..*]

a protected attribute itemCodes is one or more String values

validCard: boolean

an attribute validCard, of unspecified visibility, has unspecified multiplicity

Operations also have a precise syntax and should be shown as:

visibility name (par1: type1, par2: type2): *returntype*

where each parameter is shown (in parenthesis) and then the return type is specified.

An example would be

+ addName (newName: String): boolean

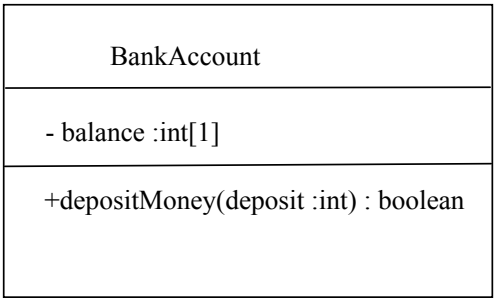
This denotes a public method 'addName' which takes one parameter 'newName' of type String and returns a boolean value.

Activity 2

Draw a diagram to represent a class called 'BankAccount' with a private attribute balance (this being a single integer) and a public method depositMoney() which takes an integer parameter, 'deposit' and returns a boolean value. Fully specify all of this information on a UML class diagram.

Feedback 2

The diagram below shows this information

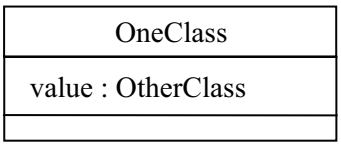


Denoting Relationships

As well as denoting individual classes, Class diagrams denote relationships between classes. One such relationships is called an 'Association'.

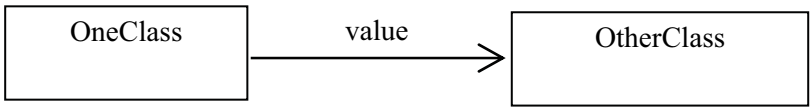
Download free eBooks at bookboon.com

In a class attributes will be defined. These could be primitive data types (int, boolean etc.) however attributes can also be complex objects as defined by other classes.



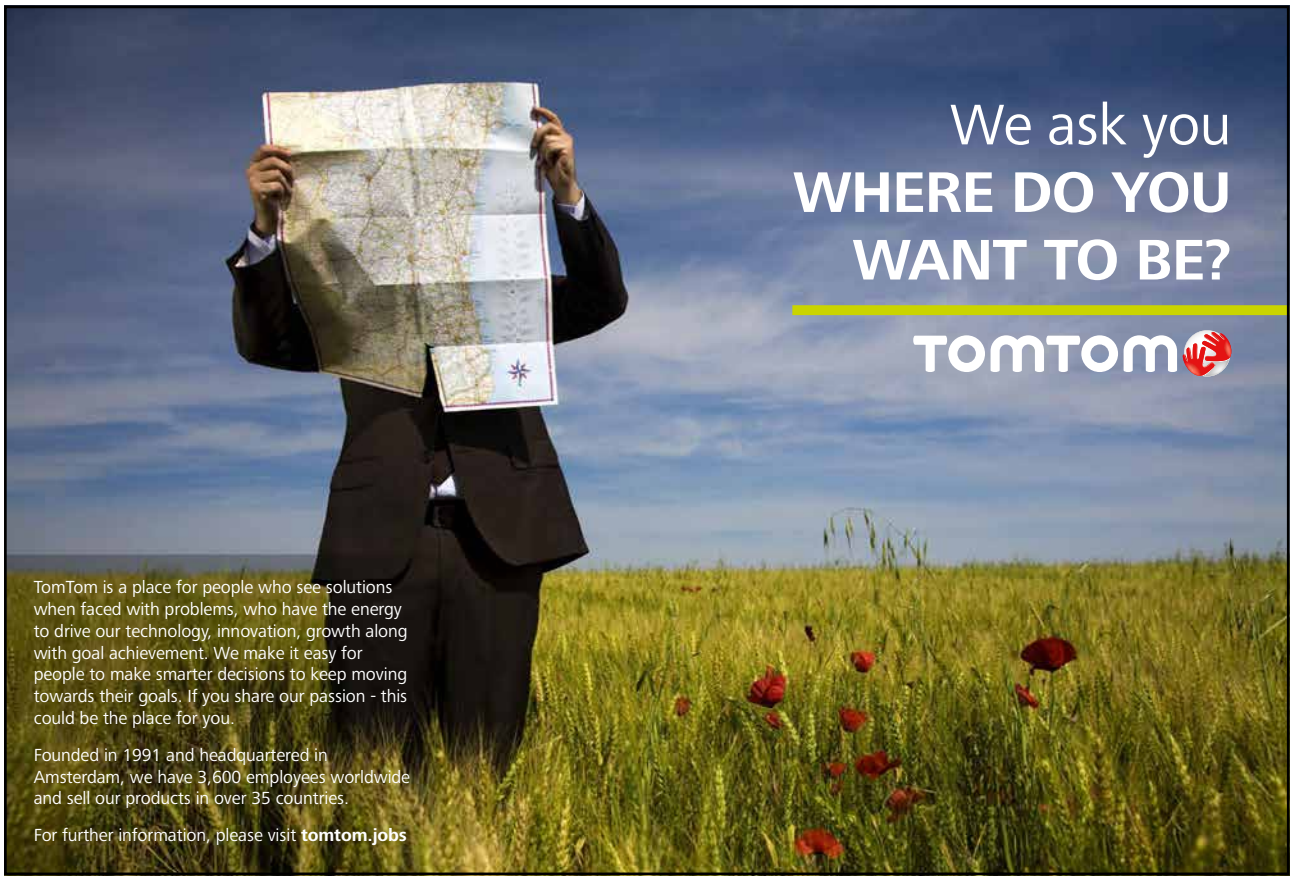
Thus the figure above shows a class 'OneClass' that has an attribute 'value'. This value is not a primitive data type but is an object of type defined by 'OtherClass'.

We could denote exactly the same information by the diagram below.



We use an association when we want to give two related classes, and their relationship, prominence on a class diagram

The 'source' class points to the 'target' class.

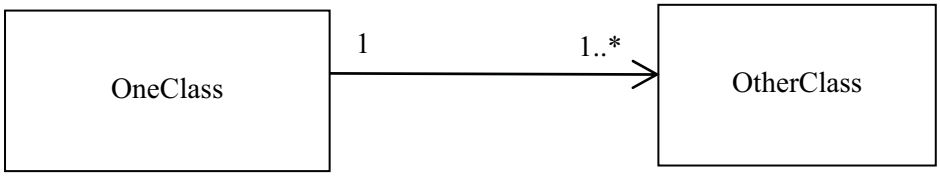


Click on the ad to read more

Download free eBooks at bookboon.com

Strictly we could use an association when a class we define has a String instance variable – but we would not do this because the String class is part of the Java platform and 'taken for granted' like an attribute of a primitive type. This would generally be true of all library classes unless we are drawing the diagram specifically to explain some aspect of the library class for the benefit of someone unfamiliar with its purpose and functionality.

Additionally we can show multiplicity at both ends of an association:



This implies that 'OneClass' maintains a collection of objects of type 'OtherClass'. Collections are an important part of the Java framework that we will look at the use of collections in Chapter 7.

Activity 3

Draw a diagram to represent a class called 'Catalogue' and a class called 'ItemForSale' as defined below:-

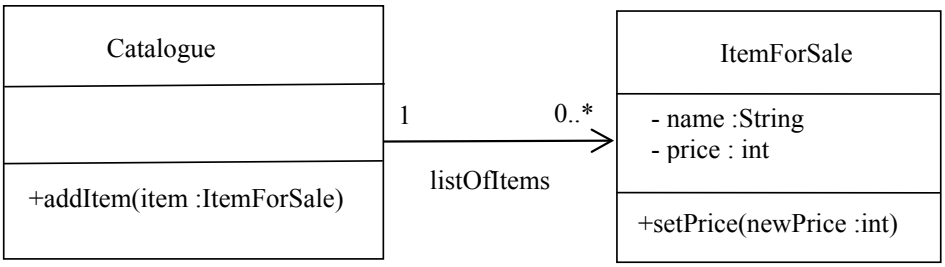
ItemForSale has an attribute 'name' of type string and an attribute 'price' of type int. It also has a method setPrice() which takes an integer parameter 'newPrice'.

'Catalogue' has an attribute 'listOfItems' i.e. the items currently held in the catalogue. As zero or more items can be stored in the catalogue 'listOfItems' will need to be an array or collection. 'Catalogue' also has one method addItem() which takes an 'item' as a parameter (of type ItemForSale) and adds this item to the 'listOfItems'.

Draw this on a class diagram showing appropriate visibility modifiers for attributes and methods.

Feedback 3

The diagram below shows this information



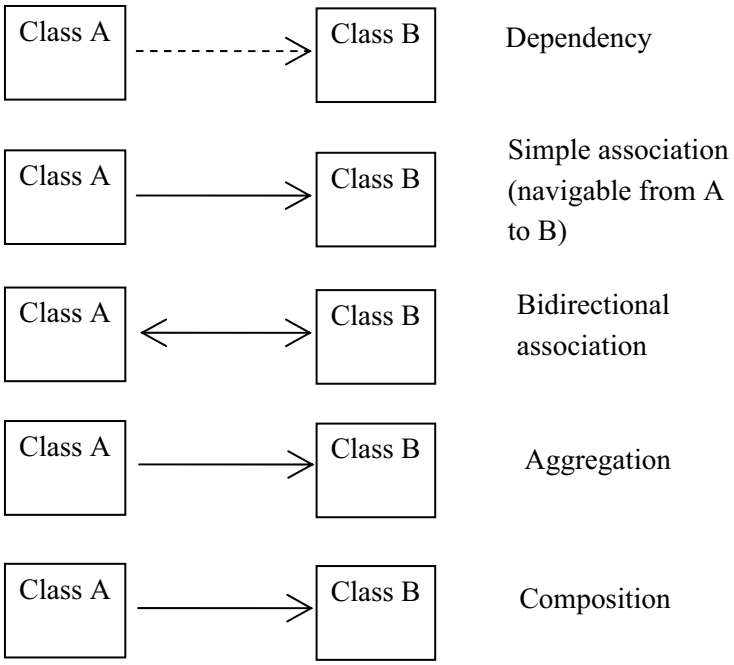
Download free eBooks at bookboon.com

Note: All class names begin in uppercase, attribute and method names begin in lowercase. Also note that the class ItemForSale describes a single item (not multiple items). 'listOfItems' however maintains a list of zero or more individual objects.

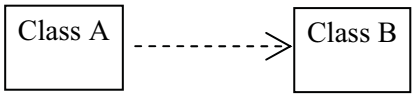
Types of Association

There are various different types of association denoted by different arrows:-

- Dependency,
- Simple association
- Bidirectional association
- Aggregation and
- Composition



Dependency

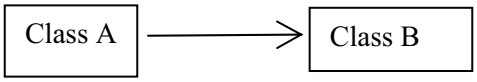


- Dependency is the most unspecific relationship between classes (not strictly an 'association')
- Class A in some way uses facilities defined by Class B
- Changes to Class B may affect Class A

Download free eBooks at bookboon.com

Typical use of dependency lines would be where Class A has a method which is passed a parameter object of Class B, or uses a local variable of that class, or calls 'static' methods in Class B.

Simple Association



- In an association Class A 'uses' objects of Class B
- Typically Class A has an attribute of Class B
- Navigability is from A to B:

  i.e. A Class A object can access the Class B object(s) with which it is associated. The reverse is not true – the Class B object doesn't 'know about' the Class A object

A simple association typically corresponds to an instance variable in Class A of the target class B type.

Example: the **Catalogue** above needs access to 0 or more **ItemsForSale** so items can be added or removed from a **Catalogue**. An **ItemForSale** does not need to access a **Catalogue** in order to set its price or perform some other method associated with the item itself.

INNOVATIVE
LIKE YOU.

If you're hoping for a truly modern education, one where you're encouraged to speak your mind and to think long-term, both when it comes to your own future and the future of the planet. Then the University of Gothenburg is the place for you.

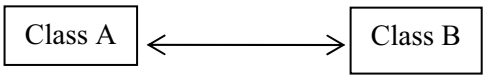
Study a Master's programme in Gothenburg, Sweden | www.gu.se/education

UNIVERSITY OF GOTHENBURG

Click on the ad to read more

Download free eBooks at bookboon.com

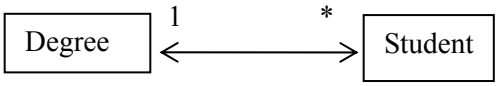
Bidirectional Association



- Bidirectional Association is when Classes A and B have a two-way association
- Each refers to the other class
- Navigability A to B and B to A:
  - A Class A object can access the Class B object(s) with which it is associated
  - Object(s) of Class B 'belong to' Class A
  - Implies reference from A to B
  - Also, a Class B object can access the Class A object(s) with which it is associated

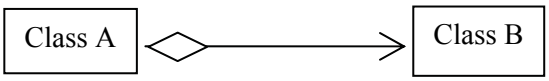
A bidirectional association is complicated because each object must have a reference to the other object(s) and generally bidirectional associations are much less common than unidirectional ones.

An example of a bidirectional association may between a 'Degree' and 'Student'. ie. given a Degree we may wish to know which Students are studying on that Degree. Alternatively starting with a student we may wish to know the Degree they are studying.



As many students study the same Degree at the same time, but students usually only take one Degree there is still a one to many relationship here.

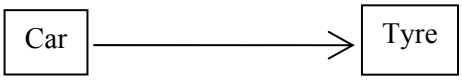
Aggregation



- Aggregation denotes a situation where Object(s) of Class B 'belong to' Class A
- Implies reference from A to B
- While aggregation implies that objects of Class B belong to objects of Class A it also implies that object of Class B retain an existence independent of Class A. Some designers believe there is no real distinction between aggregation and simple association

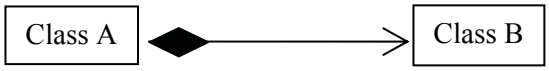
Download free eBooks at bookboon.com

An example of aggregation would be between a Class Car and a Class Tyre



We think of the tyres as belonging to the car they are on, but at the garage they may be removed and placed on a rack to be repaired. Their existence isn't dependent on the existence of a car with which they are associated.

Composition



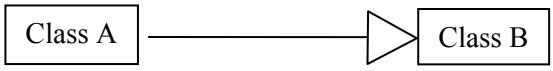
- Composition is similar to aggregation but implies a much stronger belonging relationship i.e. Object(s) of Class B are 'part of' a Class A object
- Again implies reference from A to B
- Much 'stronger' than aggregation in this case Class B objects are an integral part of Class A and in general objects of Class B never exist other than as part of Class A, i.e. they have the same 'lifetime'

An example of composition would be between Points, Lines and Shapes as elements of a Picture. These objects can only exist as part of a picture, and if the picture is deleted they are also deleted.

As well as denoting associations, class diagrams can denote:-

- Inheritance,
- Interfaces,
- Keywords and
- Notes

Inheritance



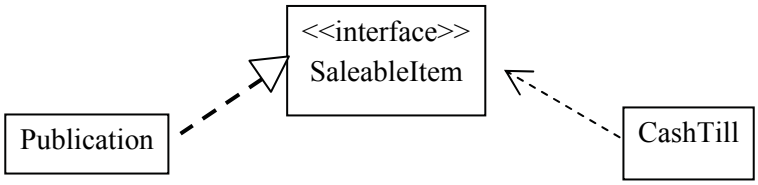
- Aside from associations, the other main modelling relationship is inheritance:
- Class A 'inherits' both the interface and implementation of Class B, though it may override implementation details and supplement both.

We will look at inheritance in detail in Chapter 3.

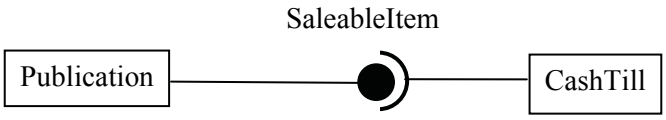
Download free eBooks at bookboon.com

Interfaces

- Interfaces are similar to inheritance however with interfaces only the interface is inherited. The methods defined by the interface must be implemented in every class that implements the interface.
- Interfaces can be represented using the <<interface>> keyword:



There is also a shorthand for this



In both cases these examples denote that the SaleableItem interface is **required by** CashTill and **implemented by** Publication.



Click on the ad to read more

Download free eBooks at bookboon.com

NB the dotted-line version of the inheritance line/arrow which shows that Publication 'implements' or 'realizes' the SaleableItem interface.

The "ball and socket" notation is new in UML 2 – it is a good visual way of representing how interfaces connect classes together.

We will look at the application of interfaces in more detail in Chapter 4.

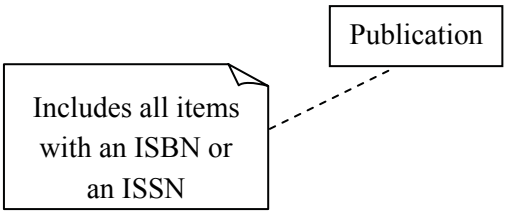
Keywords

UML defines keywords to refine the meaning of the graphical symbols

We have seen <<interface>> and we will also make use of <<abstract>> but there are many more.

An abstract class may alternatively be denoted by showing its name in *italics* though this is perhaps less obvious to a casual reader.

Notes



Finally we can add notes to comment on a diagram element. This gives us a 'catch all' facility for adding information not conveyed by the graphical notation.

Activity 4

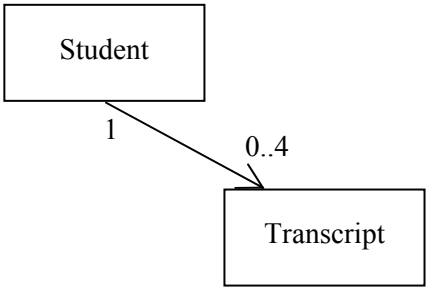
From your own experience, try to develop a model which illustrates the use of the following elements of UML Class Diagram notation:

- simple association
- bidirectional association
- aggregation (tricky!)
- composition
- association multiplicity
- generalization (inheritance)
- interfaces
- notes

For this exercise concentrate on the relationships between classes rather than the details of their members. Explain and discuss your model with other students and your tutor.

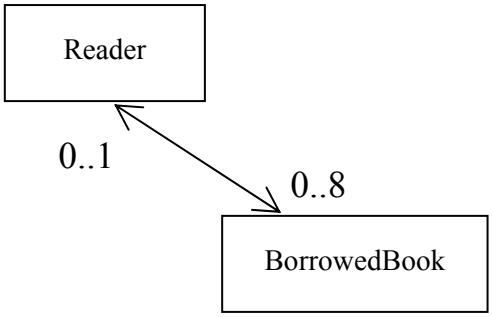
To help you get started some small examples are given below.

Download free eBooks at bookboon.com



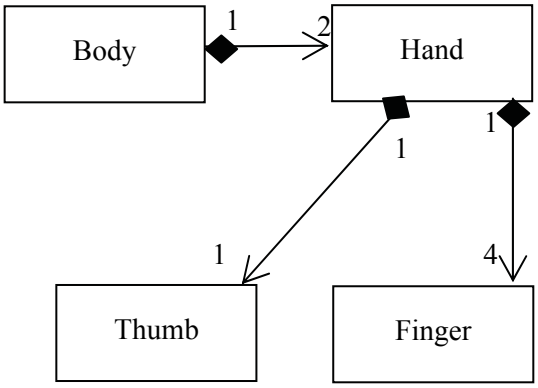
In a University administration system we might produce a transcript of results for each year the student has studied (including a possible placement year).

This association relationship is naturally unidirectional – given a student we might want to find their transcript(s), but it seems unlikely that we would have a transcript and need to find the student to whom it belonged.



In a library a reader can borrow up to eight books. A particular book can be borrowed by at most one reader.

We might want a bidirectional relationship as shown here because, in addition to being able to identify all the books which a particular reader has borrowed, we might want to find the reader who has borrowed a particular book (for example to recall it in the event of a reservation).

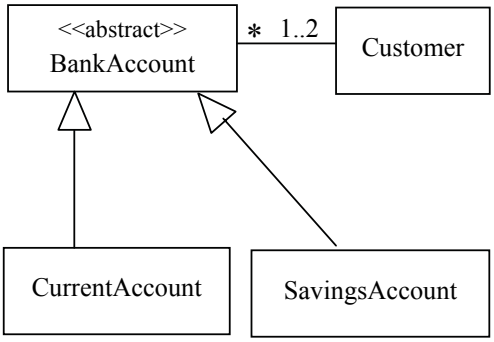


This might be part of the model for some kind of educational virtual anatomy program.

Composition – the "strong" relationship which shows that one object is (and has to be) part of another seems appropriate here.

The multiplicities would not always work for real people though – they might have lost a finger due to accident or disease, or have an extra one because of a genetic anomaly.

And what if we were modelling the "materials" in a medical school anatomy lab? A hand might not always be part of a body! Perhaps the "weaker" aggregation relationship would reflect this better.



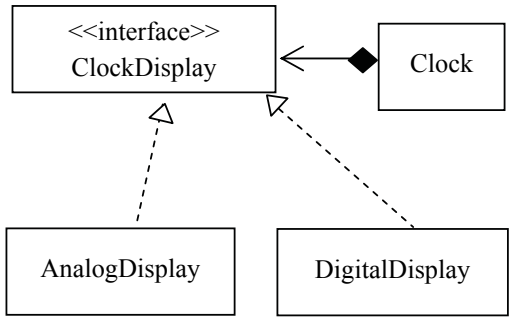
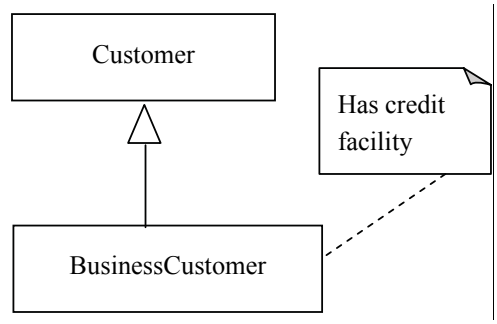
A customer can have any number of bank accounts, and a bank account can be held by one person or two people (a "joint account"). We have suppressed the navigability of this relationship, perhaps because we have not yet decided this issue.

A bank account **must** either be a current account or a savings account – hence BankAccount itself is abstract.

(We could have shown this using italics rather than the <<abstract>> keyword)

Bank Account

Download free eBooks at bookboon.com

	Part of a clock is a display to show the time. This might be an analogue display or a digital display. We could use a superclass and two subclasses, but since the implementation of the two displays will be entirely different it may be more appropriate to use an interface to define the operations which AnalogDisplay and DigitalDisplay must provide.
	A note is used here to add some information which cannot be conveyed with standard UML notation and may not be obvious simply from the names and relationships of the classes depicted.

Feedback 4

There is no specific feedback for this activity.



Download free eBooks at bookboon.com

2.4 UML Package Diagrams

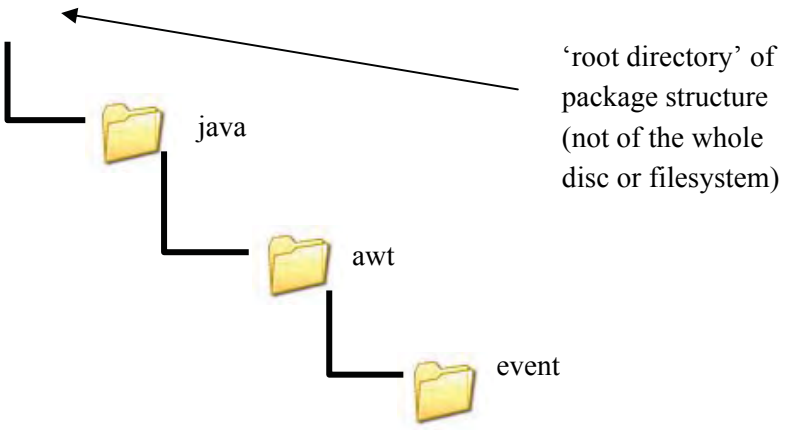
While class diagrams are the most commonly used diagram of those defined in UML notation, and we will make significant use of these throughout this book, there are other diagrams that denote different types of information. Here we will touch upon three of these:-

- Package Diagrams
- Object Diagrams and
- Sequence Diagrams

World maps, country maps and city maps all show spatial information, just on different scales and with differing levels of detail. Large OO systems can be made up of hundreds, or potentially thousands, of classes and thus if the class diagram was the only way to represent the architecture of a large system it would become overly large and complex. Thus, just as we need world maps, we need package diagrams to show the general architecture of a large system. Even modest systems can be broken down into a few basic components i.e. packages. We will see an example of packages in use in Chapter 11. For now we will just look at the package diagramming notation.

A package is not just a visual representation of a group of classes instead a 'package' is a directory containing a group of related classes (and interfaces). Packages allow us to provide a level of organisation and encapsulation above that of individual classes and all of the standard Java platform classes are arranged in a single large package hierarchy. Similarly we can also arrange our own classes using the Java package mechanism.

Packages are described as a series of dot-separated names, e.g. java.awt.event. The names correspond to a series of sub-directories in the file system, e.g.



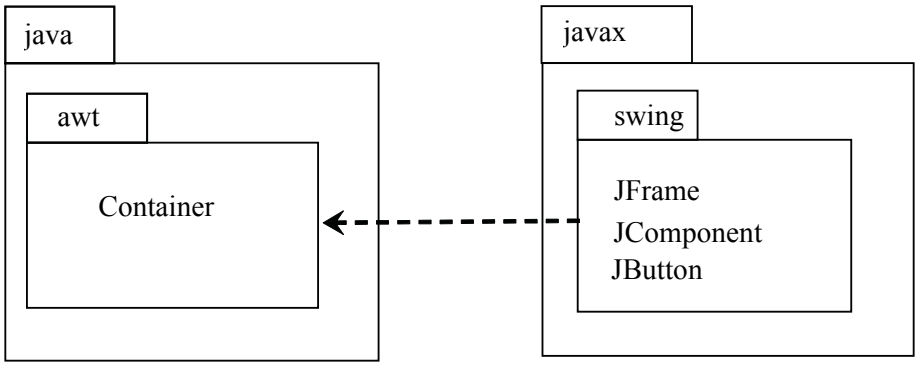
A large Java development should be split into suitable packages at the design stage UML provides a 'Package Diagram' to represent the relationships between classes and packages.

Download free eBooks at bookboon.com

We can depict

- classes within packages
- nesting of packages
- dependencies between packages

In the diagram below we see two packages:- 'java' and 'javax'

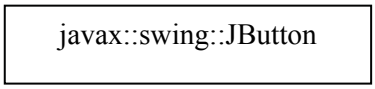


Looking at this more closely we can see that inside the 'java' package is another called 'awt' and inside 'javax' is a package called 'swing'.

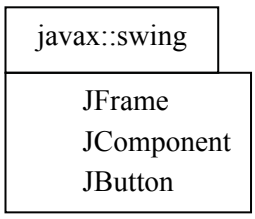
The package 'awt' contains a class 'Container' and 'javax' contains three classes 'JFame', 'JComponent' and 'JButton'. Finally we show that the javax.swing package has dependencies on the java.awt package.

Note that the normal UML principle of suppression applies here – both java.awt and javax.swing contain many more classes, and 'java' contains other sub-packages, but we simply choose not to show them.

In the diagram below we have an alternative way of indicating that a JButton is in the javax.
swing package.



And again below a form which shows all three classes more concisely than at the top.



Download free eBooks at bookboon.com

These different representations will be useful in different circumstances depending on what a package diagram is aiming to convey.

Package Naming

By convention, package names are normally in lowercase
For local individual projects packages could be named according to personal preference, e.g.

mysystem
mysystem.interface
mysystem.engine
mysystem.engine.util
mysystem.database

However, packages are often distributed and to enable this packages need globally unique names, thus a naming convention has been adopted based on URLs

uk.co.ebay.www.department.project.package



Click on the ad to read more

Download free eBooks at bookboon.com

Part based on organisation URL (e.g. ww.ebay.co.uk) reversed, though this does **not** specifically imply you can download the code there.

Part distinguishing the particular project and component or subsystem which this package contains.

Note on a package diagram each element is not separated by a '.' but by '::'.

Activity 5

You and a flatmate decide to go shopping together. For speed split the following shopping list into two halves – items to be collected by you and items to be collected by your flatmate.

Apples, Furniture polish, Pears, Carrots, Toilet Rolls, Potatoes, Floor cleaner. Matches, Grapes

Feedback 5

To make your shopping efficient you probably organised your list into two lists of items that are located in the same parts of the shop:-

List 1	List 2
Apples,	Furniture polish,
Pears,	Floor cleaner
Grapes	Matches
Carrots,	Toilet Rolls,
Potatoes	

Activity 6

You run a team of three programmers and are required to write a program in Java to monitor and control a network system. The system will be made up of seven classes as described below. Organise these classes into three packages. Each programmer will then be responsible for the code in one package. Give the packages any name you feel appropriate.

Main	this class starts the system
Monitor	this class monitors the network for performance and breaches in security
Interface	this is a visual interface for entire system
Reconfigure	this allows the network to be reconfigured
RecordStats	this stores data regarding the network in a database
RemoteControl	this allows some remote control over the system via telephone
PrintReports	this uses the data stored in the database to print management reports for the organisations management.

Download free eBooks at bookboon.com

Feedback 6

When organising a project into packages there is not always 'one correct answer' but if you organise your classes into appropriate packages (with classes that have related functionality) you improve the encapsulation of the system and improve the efficiency of your programmers. A suggested solution to activity 6 is given below.

interface
- Main
- Interface
- RemoteControl

network
- Monitor
- Reconfigure

database
- RecordStats
- PrintReports

Activity 7

Assume the URL of your organisation is 'www.myorg.com' and the three packages and seven classes shown below are all part of 'project1'. Draw a package diagram to convey this information.

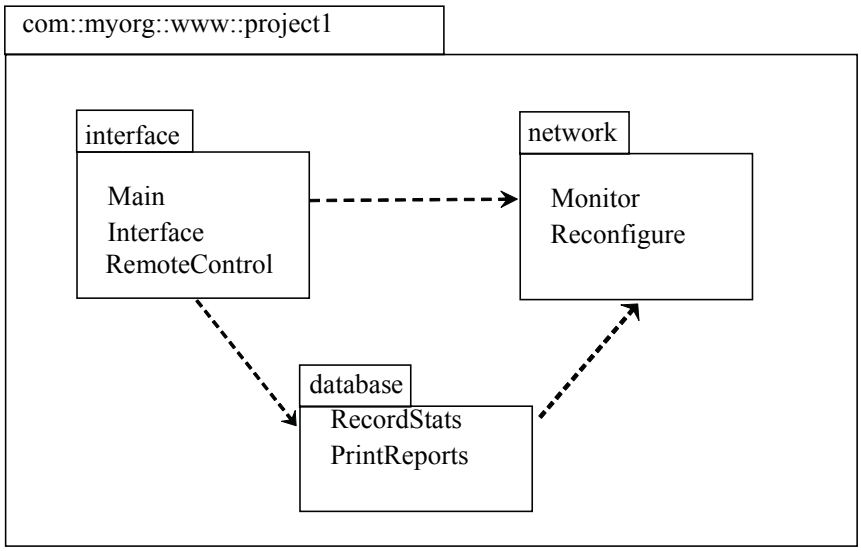
interface
- Main
- Interface
- RemoteControl

network
- Monitor
- Reconfigure

database
- RecordStats
- PrintReports

Download free eBooks at bookboon.com

Feedback 7



Note: Dependency arrows have been drawn to highlight relationships between packages. When more thought has been put into determining these relationships they may turn out to be associations (a much stronger relationship than a mere dependency).

SIMPLY CLEVER
ŠKODA
WE WILL TURN YOUR CV INTO AN OPPORTUNITY OF A LIFETIME
Do you like cars? Would you like to be a part of a successful brand?
As a constructer at ŠKODA AUTO you will put great things in motion. Things that will ease everyday lives of people all around Send us your CV. We will give it an entirely new new dimension.
Send us your CV on www.employerforlife.com
Click on the ad to read more

Download free eBooks at bookboon.com

2.5 UML Object Diagrams

Class diagrams and package diagrams allow us to visualise and discuss the architecture of a system however at times we wish to discuss the data a system processes. Object diagrams allow us to visual one instance of time and the data that a system may contain in that moment.

Object diagrams look superficially similar to class diagrams however the boxes represent specific instances of objects.

Boxes are titled with:-

objectName: ClassName

As each box describes a particular object at a specific moment in time the box contains attributes and their values (at that moment in time).

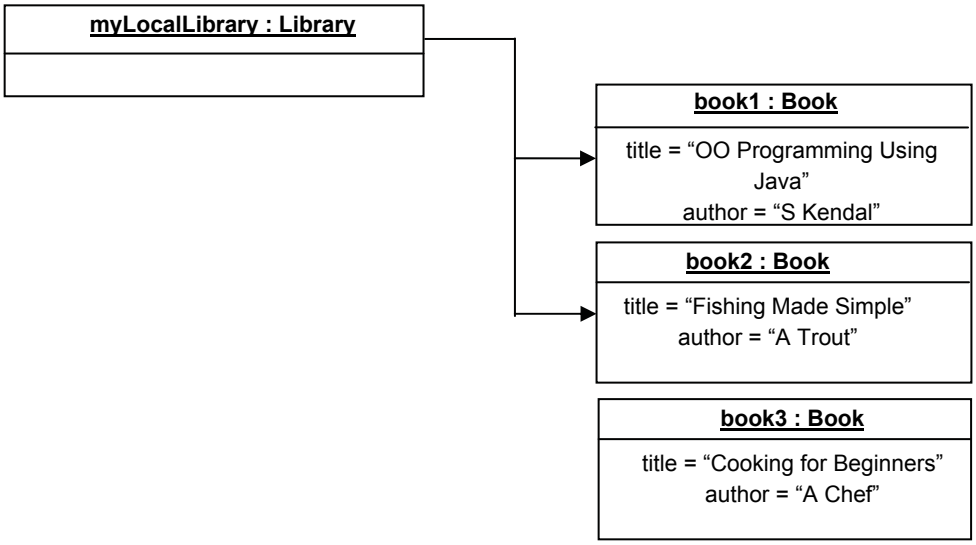
attribute = value

These diagrams are useful for illustrating particular 'snapshot' scenarios during design.

The object diagram below shows several object that may exist at a moment in time for a library catalogue system. The system contains two classes:-

Book, which store the details of a book and

Library, which maintains a collection of books. With books being added, searched for or removed as required.

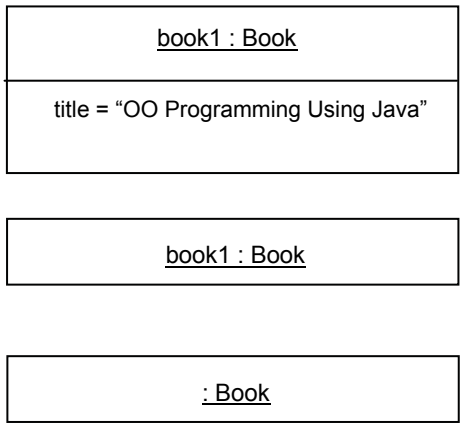


Looking at this diagram we can see that at a particular moment in time, while three books have been created only two have been added to the library. Thus if we were to search the library for 'Cooking for Beginners' we would not expect the book to be found.

As with class diagrams, elements can be freely suppressed on object diagrams.

Download free eBooks at bookboon.com

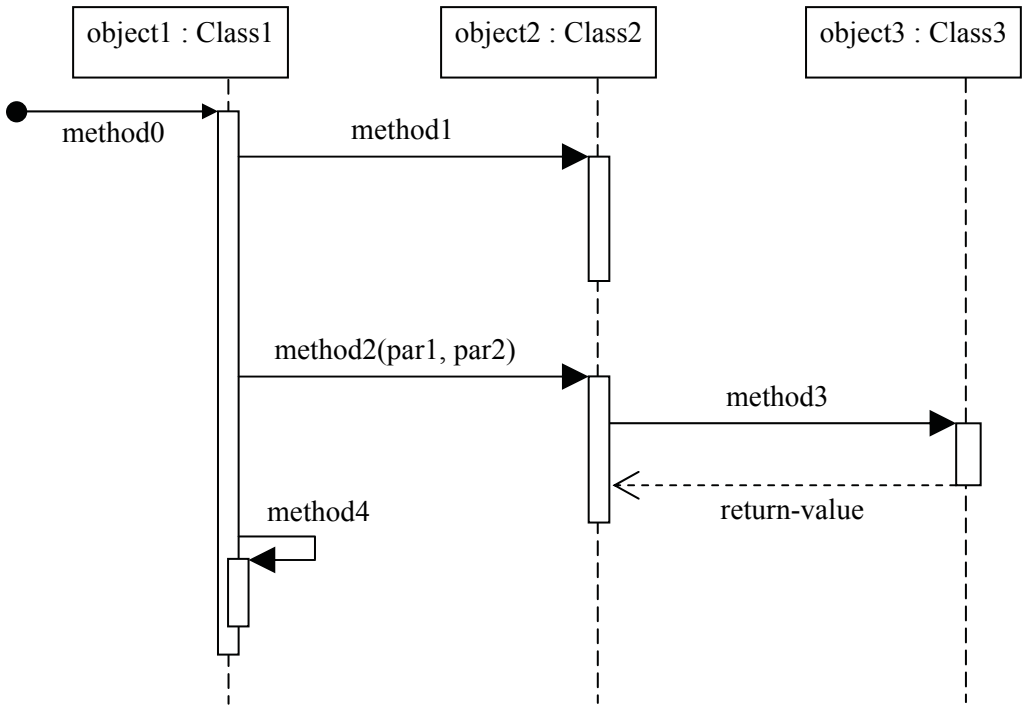
For example, all of these are legal:



2.6 UML Sequence Diagrams

Sequence diagrams are entirely different from class diagrams or object diagrams. Class diagrams describe the architecture of a system and object diagrams describe the state of a system at one moment in time. However sequence diagrams describe how the system works over a period of time. Sequence diagrams are 'dynamic' rather than 'static' representations of the system. They show the sequence of method invocations within and between objects over a period of time. They are useful for understanding how objects collaborate in a particular scenario.

See the example below:-



Download free eBooks at bookboon.com

We have three objects in this scenario. Time runs from top to bottom, and the vertical dashed lines (lifelines) indicate the objects' continued existence through time.

This diagram shows the following actions taking place:-

- Firstly a method call (often referred to in OO terminology as a message) to method0() comes to object1 from somewhere – this could be another class outside the diagram.
- object1 begins executing its method0() (as indicated by the vertical bar (called an activation bar) which starts at this point.
- object1.method0() invokes object2.method1() – the activation bar indicates that this executes for a period then returns control to method0()
- Subsequently object1.method0() invokes object2.method2() passing two parameters
- method2() subsequently invokes object3.method3(). When method3() ends it passes a return value back to method2()
- method2() completes and returns control to object1.method0()
- Finally method0() calls another method of the same object, method4()

Selection and Iteration

The logic of a scenario often depends on selection ('if') and iteration (loops).

There is a notation ('interaction frames') which allow ifs and loops to be represented in sequence diagrams however these tend to make the diagrams cluttered.

Sequence diagrams are generally best used for illustrating particular cases, with the full refinement reserved for the implementation code.

Fowler ("UML Distilled", 3rd Edn.) gives a brief treatment of these constructs.

Download free eBooks at bookboon.com

2.7 Summary

UML is not a methodology but a precise diagramming notation.

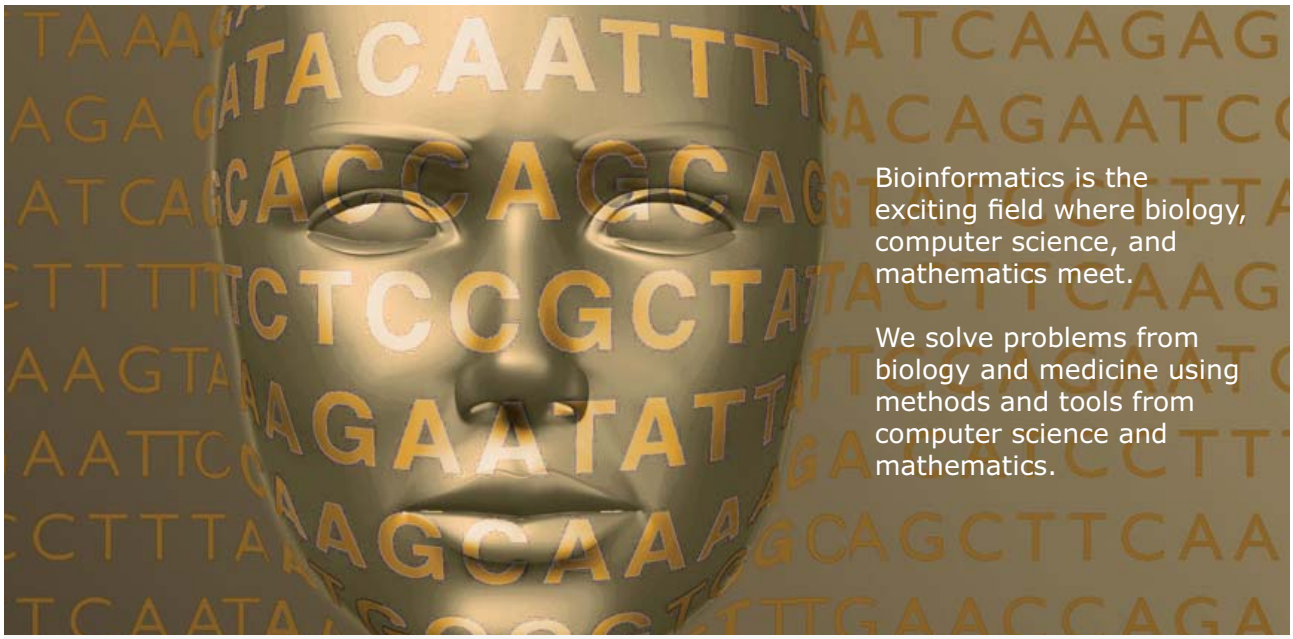
Class diagrams and package diagrams are good for describing the architecture of a system. Object diagrams describe the data within an application at one moment in time and sequence diagrams describe how a system works over a period of time.

UML gives different meaning to different arrows therefore one must be careful to use the notation precisely as specified.

With any UML diagram suppression is encouraged – thus the author of a diagram can suppress any details they wish in order to convey essential information to the reader.

UPPSALA UNIVERSITET

Develop the tools we need for Life Science
Masters Degree in Bioinformatics



Read more about this and our other international masters degree programmes at www.uu.se/master

Click on the ad to read more

Download free eBooks at bookboon.com

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

Download free eBooks at bookboon.com

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.

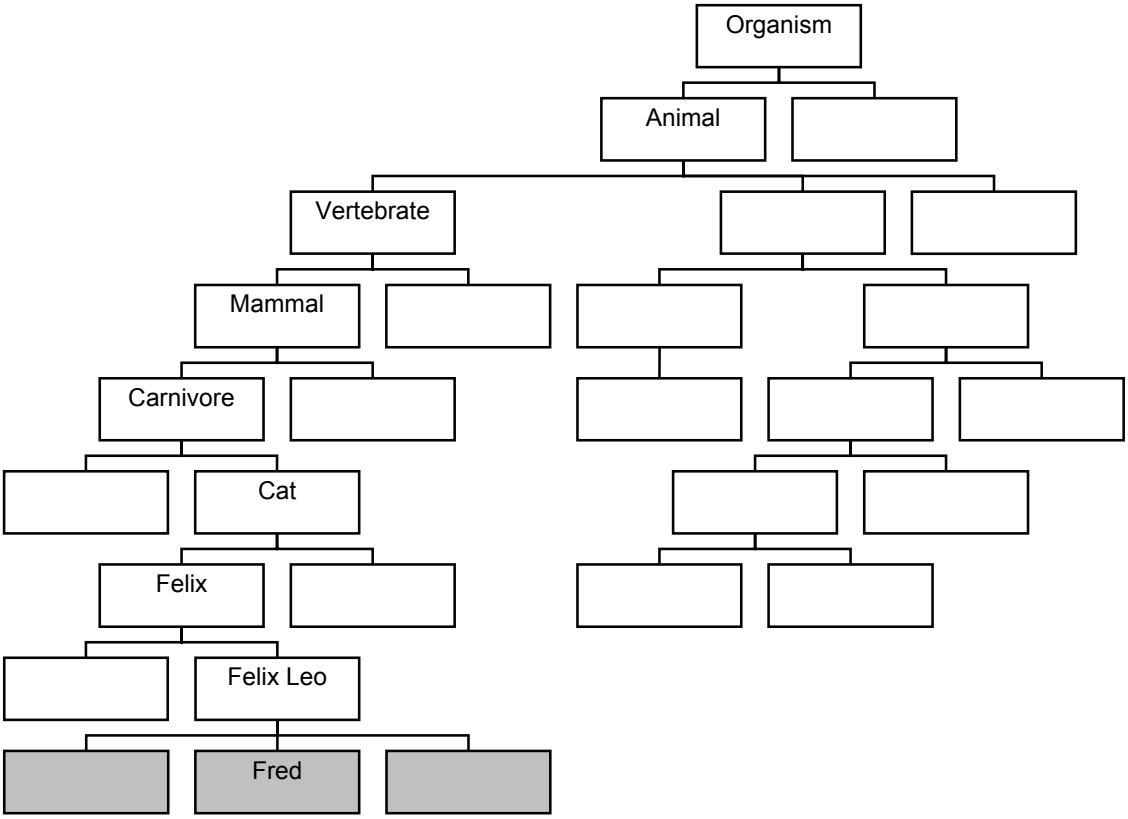
Classes themselves can often be organised by a similar kind of relationship.

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…

Download free eBooks at bookboon.com



UNIVERSITY OF COPENHAGEN

Copenhagen Master of Excellence

Copenhagen Master of Excellence are two-year master degrees taught in English at one of Europe's leading universities

Come to Copenhagen - *and aspire!*

Apply now at
www.come.ku.dk



Click on the ad to read more

Download free eBooks at bookboon.com

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.

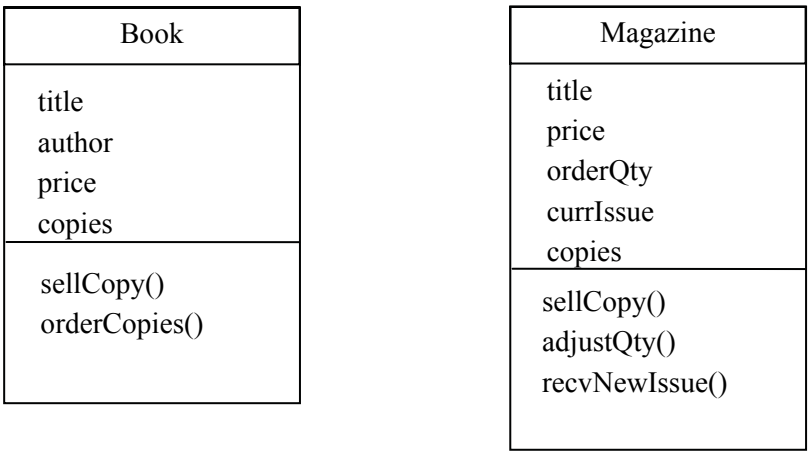
3.3 Inheritance

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.



Download free eBooks at bookboon.com

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.

Activity 1

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

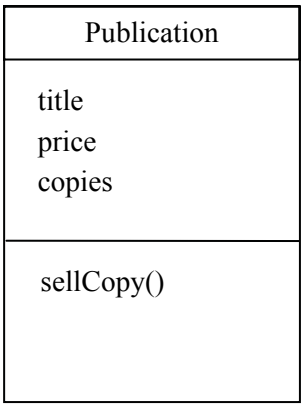
Feedback 1

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

The differences are as follows…

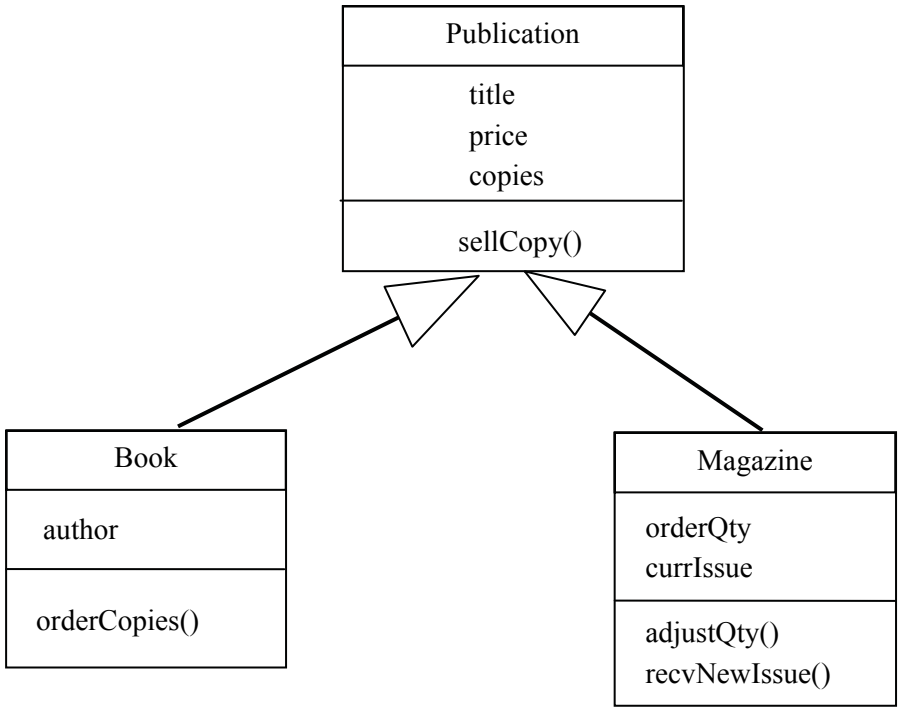
Book additionally has author, and orderCopies().
Magazine additionally has orderQty, currIssue, adjustQty() and recvNewIssue().

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



Download free eBooks at bookboon.com

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





Click on the ad to read more

Download free eBooks at bookboon.com

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()).

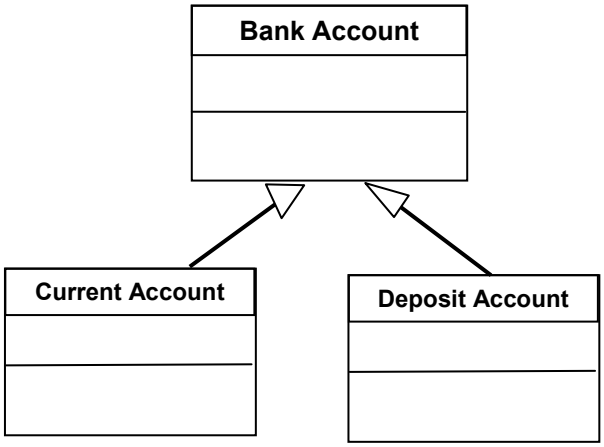
The inherited characteristics are NOT listed in subclasses. The arrow shows they are acquired from superclass.

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.

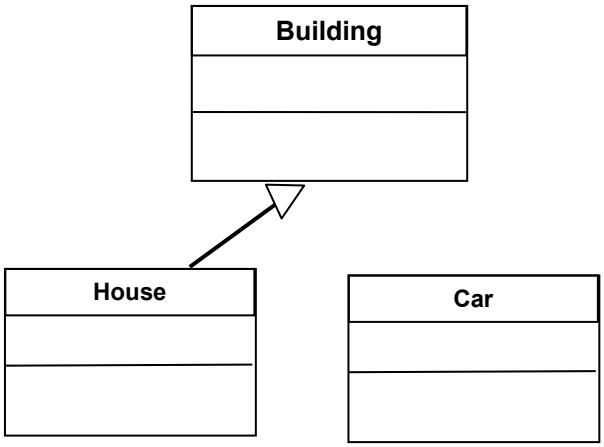
Download free eBooks at bookboon.com

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

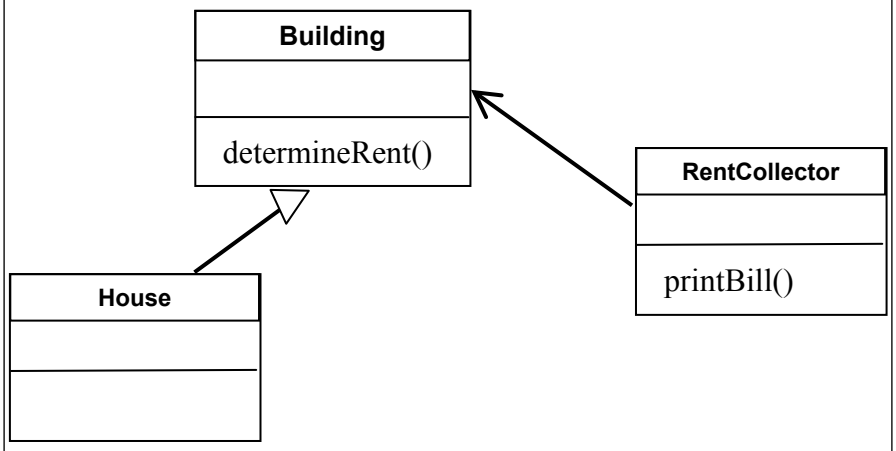
Describe the following using a suitable class diagram showing ANY sensible relationship…

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 

Download free eBooks at bookboon.com

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!

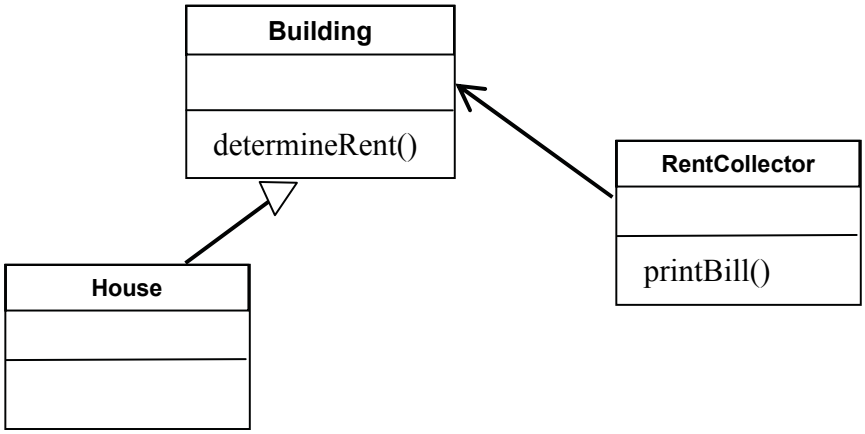
We only show methods in subclasses if they are either additional methods or methods that have been overridden.

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.

Download free eBooks at bookboon.com

Activity 6

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



Trust and responsibility

NNE and Pharmaplan have joined forces to create NNE Pharmaplan, the world's leading engineering and consultancy company focused entirely on the pharma and biotech industries.

Inés Aréizaga Esteva (Spain), 25 years old
Education: Chemical Engineer

– You have to be proactive and open-minded as a newcomer and make it clear to your colleagues what you are able to cope. The pharmaceutical field is new to me. But busy as they are, most of my colleagues find the time to teach me, and they also trust me. Even though it was a bit hard at first, I can feel over time that I am beginning to be taken seriously and that my contribution is appreciated.



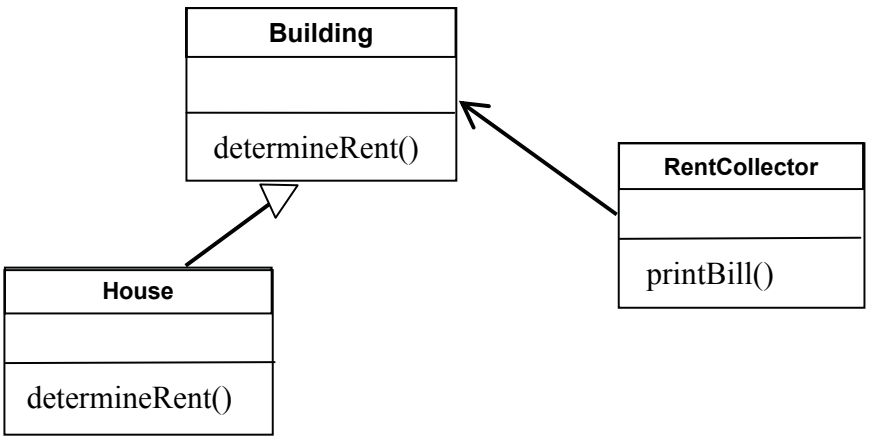
NNE Pharmaplan is the world's leading engineering and consultancy company focused entirely on the pharma and biotech industries. We employ more than 1500 people worldwide and offer global reach and local knowledge along with our all-encompassing list of services. **nnepharmaplan.com**

nne pharmaplan®

Click on the ad to read more

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

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

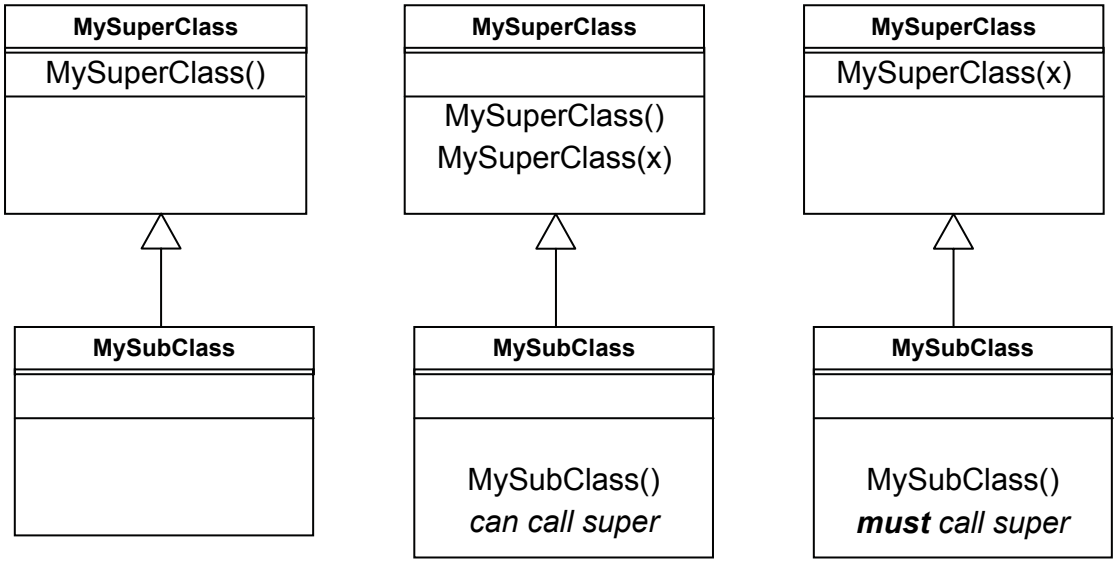
Rules exist that govern the invocation of a superconstructor.

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.

Download free eBooks at bookboon.com

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.

This e-book
is made with
SetaPDF
SETASIGN
PDF components for PHP developers
www.setasign.com
Click on the ad to read more

Download free eBooks at bookboon.com

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;
}
```


Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

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.

FOSS

Sharp Minds - Bright Ideas!

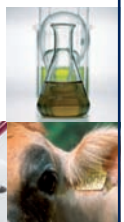
Employees at FOSS Analytical A/S are living proof of the company value - First - using new inventions to make dedicated solutions for our customers. With sharp minds and cross functional teamwork, we constantly strive to develop new unique products - Would you like to join our team?

FOSS works diligently with innovation and development as basis for its growth. It is reflected in the fact that more than 200 of the 1200 employees in FOSS work with Research & Development in Scandinavia and USA. Engineers at FOSS work in production, development and marketing, within a wide range of different fields, i.e. Chemistry, Electronics, Mechanics, Software, Optics, Microbiology, Chemometrics.

We offer
A challenging job in an international and innovative company that is leading in its field. You will get the opportunity to work with the most advanced technology together with highly skilled colleagues.

Read more about FOSS at www.foss.dk - or go directly to our student site www.foss.dk/sharpminds where you can learn more about your possibilities of working together with us on projects, your thesis etc.

The Family owned FOSS group is the world leader as supplier of dedicated, high-tech analytical solutions which measure and control the quality and production of agricultural, food, pharmaceutical and chemical products. Main activities are initiated from Denmark, Sweden and USA with headquarters domiciled in Hillerød, DK. The products are marketed globally by 23 sales companies and an extensive net of distributors. In line with the corevalue to be 'First', the company intends to expand its market position.



Dedicated Analytical Solutions
FOSS
Slangerupgade 69
3400 Hillerød
Tel. +45 70103370
www.foss.dk



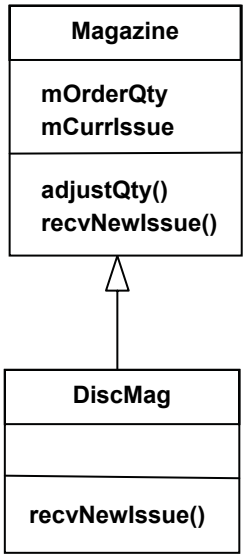
Click on the ad to read more

Download free eBooks at bookboon.com

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).

Download free eBooks at bookboon.com

```
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.

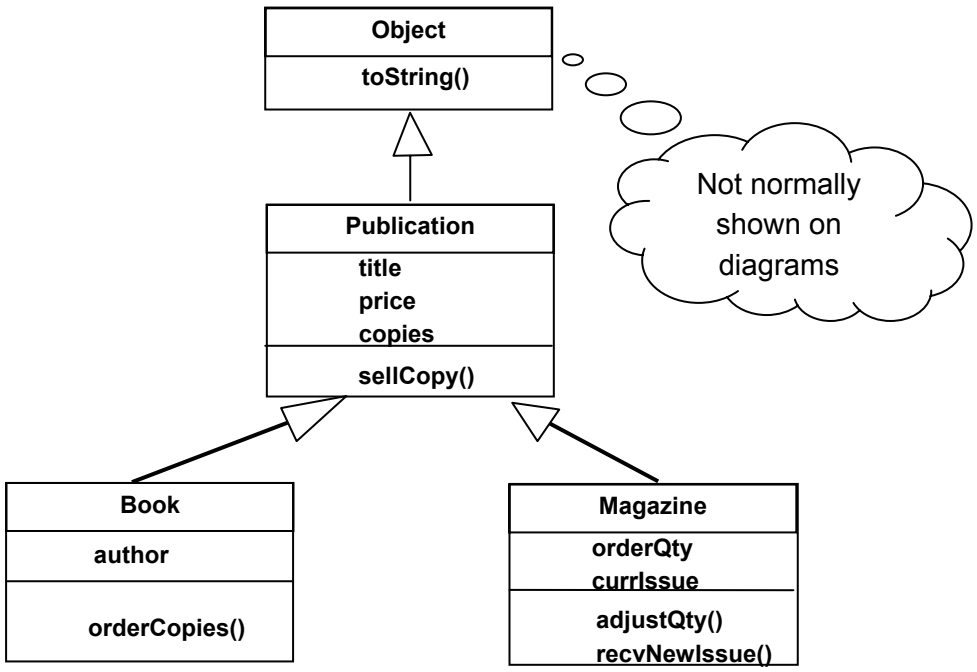
Object defines no instance variables but several methods. Generally these methods will be overridden by new classes to make them useful. An example is the **toString()** method.

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.

Download free eBooks at bookboon.com

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





Click on the ad to read more
Download free eBooks at bookboon.com

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

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

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

Download free eBooks at bookboon.com

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.

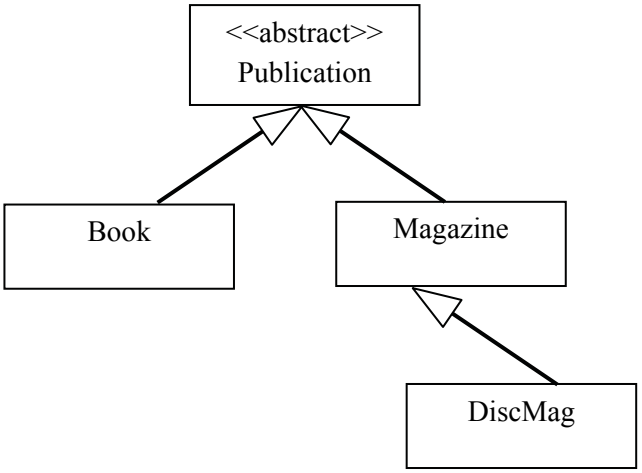
This is true in object oriented programs as well. Every time we define a class we create a new 'type'. Types determine compatibility between variables, parameters etc.

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.



Click on the ad to read more
Download free eBooks at bookboon.com



- 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 ie. 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.

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

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.



Click on the ad to read more

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

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

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 concatentation operator '+'. This implicitly invokes the **toString()** method for the subclass of this object, and remember that **toString()** is different for books and magazines.

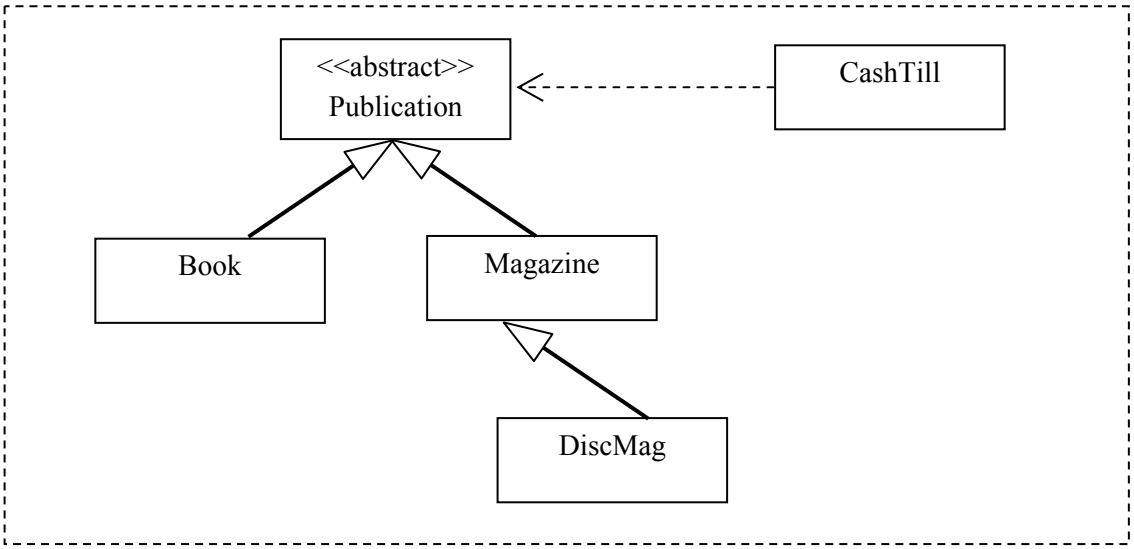
This is polymorphism in action – using the **toString()** operation to invoke the appropriate toString() method for the relevant class!



Click on the ad to read more

Download free eBooks at bookboon.com

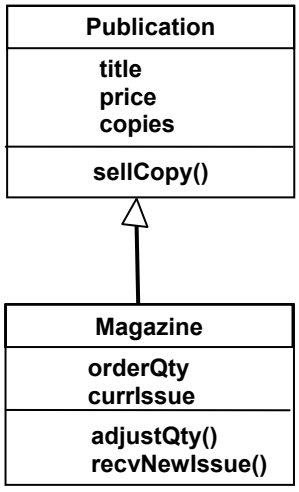
We can show CashTill on a class diagram as below:-



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();

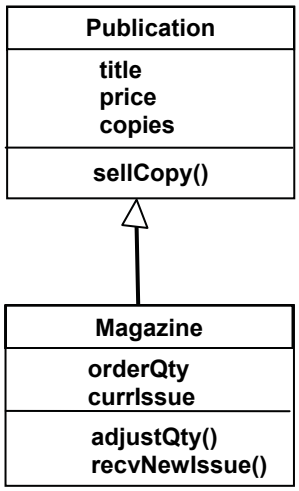
d) Publication p = new Magazine(…); p.recvNewIssue();

e) Magazine m = new Magazine(…); m.recvNewIssue();

Download free eBooks at bookboon.com

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();

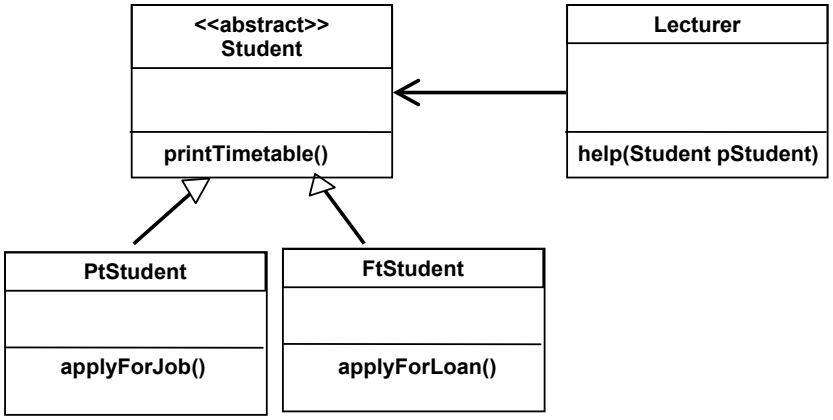
Feedback 2

a) Legal – you can invoke sellCopy() on a publication
b) Illegal – the recNewIssue() 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.

Download free eBooks at bookboon.com

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

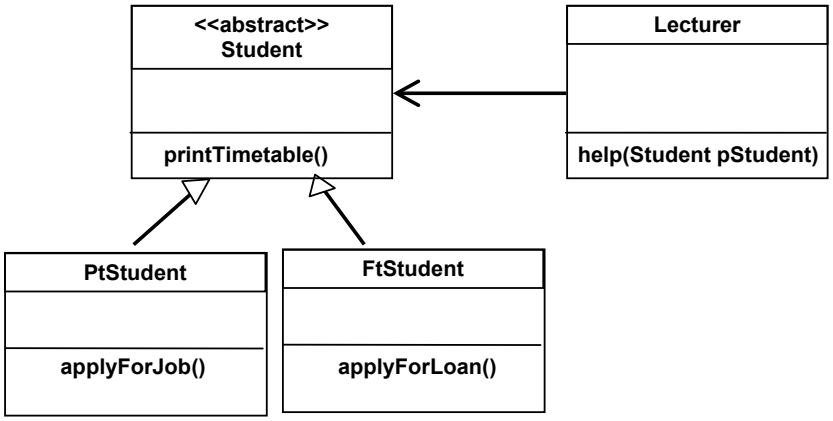
a) This is not valid as class Student is abstract and cannot be instantiated
b) This is valid. FtStudent is a type of Student and can be assigned to variable of type Student. This can then be passed as a parameter to l.help()

Download free eBooks at bookboon.com

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();

I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com



Click on the ad to read more

Download free eBooks at bookboon.com

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 sub**class** is a sub**type**. It's 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.

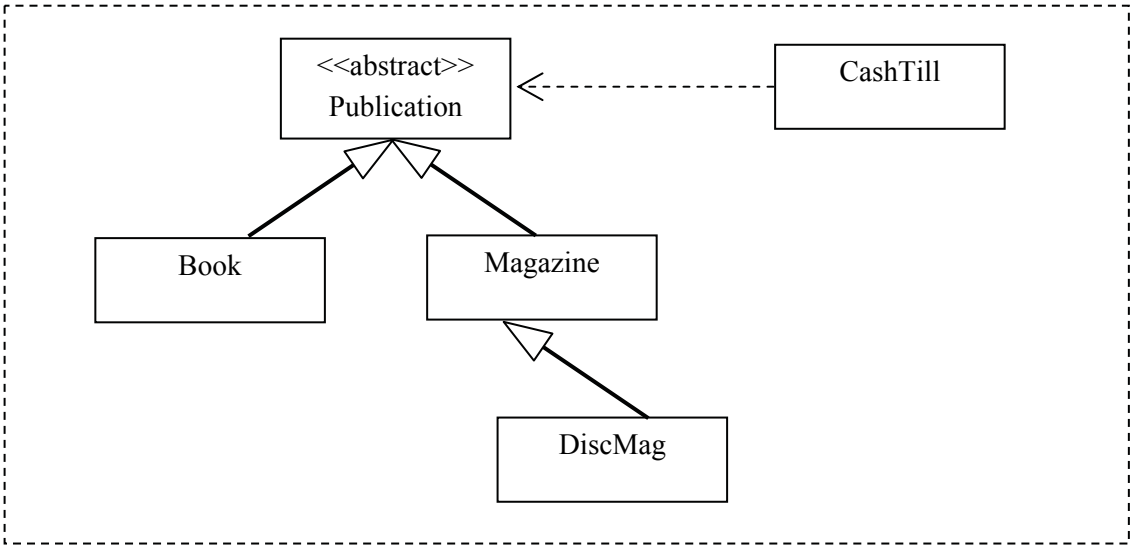
This is like making sure that two cars have controls that work in exactly the same way, but leaving it to different engineers to design engines which 'implement' the functionality of the car, possibly in quite different ways.

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

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

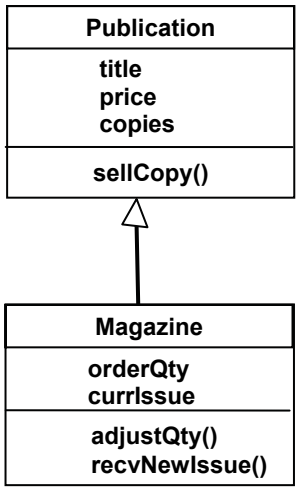
Tickets seem to have little in common with Publications – they share a small **interface** associated with being sold, but even for this the underlying **implementation** will be different because we will not be decrementing them from a current stock

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.

Download free eBooks at bookboon.com

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 form 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.



Click on the ad to read more

Download free eBooks at bookboon.com

```
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.

Download free eBooks at bookboon.com

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

Download free eBooks at bookboon.com

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:

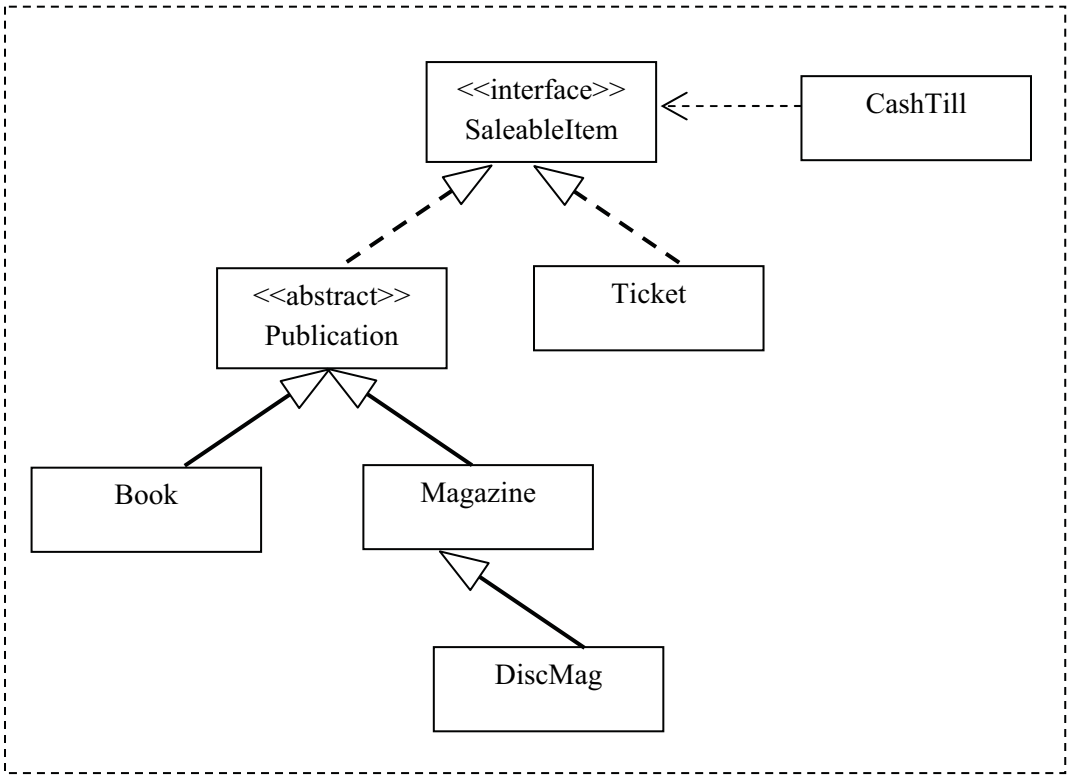
```
public void sellItem (SaleableItem pSelb)
```

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…



Click on the ad to read more
Download free eBooks at bookboon.com



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.

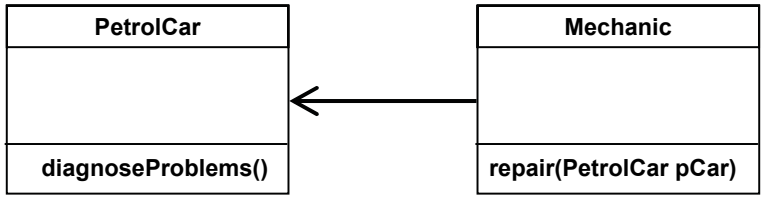
Download free eBooks at bookboon.com

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!

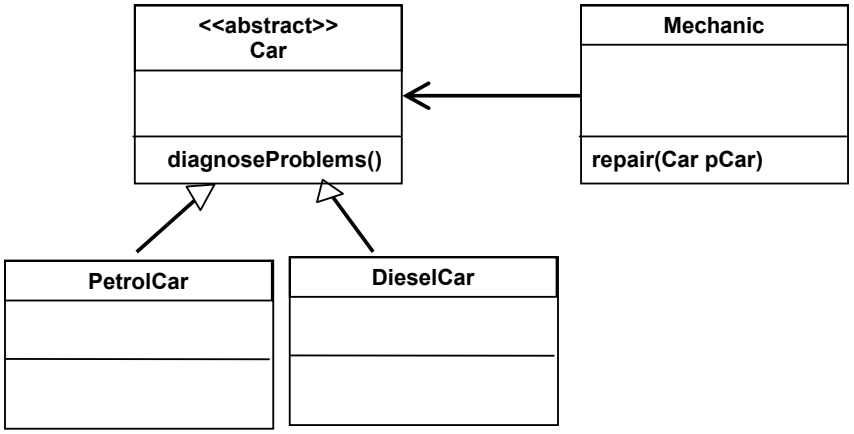
Activity 5

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.

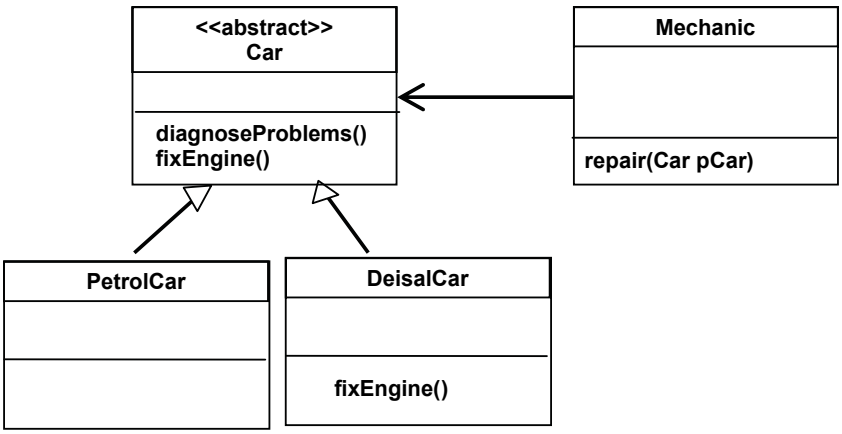
If an ElectricCar class was added Mechanic would still be able to work with them without making any changes to the Mechanic class.

Download free eBooks at bookboon.com

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 DiesalCar object was passed to the repair() method which actual method would be invoked by pCar.fixEngine(); ?

In the past 5 years we have drilled around

95,000 km

—that's more than **twice** around the world.

Who are we?

We are the world's leading provider of reservoir characterization, drilling, production, and processing technologies to the oil and gas industry.

Who are we looking for?

We offer countless opportunities in the following domains:

- Operations
- Research, Engineering, and Manufacturing
- Geoscience and Petrotechnical
- Commercial and Business

We're looking for high-energy, self-motivated graduates with vision and integrity to join our team.

careers.slb.com

What will you be?

Schlumberger

Click on the ad to read more

Download free eBooks at bookboon.com

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 DiesalCar 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 displas 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");
     }
}
```


Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

5 Overloading

Introduction

This chapter will introduce the reader to the concept of method overloading

Objectives

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

- Understand the concept of 'overloading'
- Appreciate the flexibility offered by overloading methods
- Identify overloaded methods in the online API documentation

This chapter consists of the following three sections:-

1) Overloading
2) Overloading To Aid Flexibility
3) Summary



Click on the ad to read more

Download free eBooks at bookboon.com

5.1 Overloading

Historically in computer programs method names were required to be unique. Thus the compiler could identify which method was being invoked just by looking at its name.

However several methods were often required to perform very similar functionality for example a method could add two integer numbers together and another method may be required to add two floating point numbers. If you have to give these two methods unique names which one would you call 'add()'?

In order to give each method a unique name the names would need to be longer and more specific. We could therefore call one method addInt() and the other addFloat() but this could lead to a proliferation of names each one describing different methods that are essentially performing the same operation ie. adding two numbers.

To overcome this problem in Java you are not required to give each method a unique name – thus both of the methods above could be called add(). However if method names are not unique the Java Runtime Environment (JRE) must have some other way of deciding which method to invoke at run time. ie. when a call is made to add(number1, number2) the machine must decide which of the two methods to use. It does this by looking at the parameter list.

While the two methods may have the same name they can still be distinguished by looking at the parameter list.:-

```
add(int number1, int number2)
add(float number1, float number2)
```

This is resolved at run time by the JRE. i.e. at run time the JRE looks at the method call and the actual parameters being passed. If two integers are being passed then the first method is invoked. However if two floating point numbers are passed then the second method is used.

Overloading refers to the fact that several methods may share the same name. As method names are no longer uniquely identify the method then the name is 'overloaded'.

5.2 Overloading To Aid Flexibility

Having several methods that essentially perform the same operation, but which take different parameter lists, can lead to enhanced flexibility and robustness in a system.

Download free eBooks at bookboon.com

Imagine a University student management system. A method would probably be required to enrol, or register, a new student. Such a method could have the following signature…

enrollStudent(String pName, String pAddress, String pCoursecode)

However if a student had just arrived in the city and had not yet sorted out where they were living would the University want to refuse to enrol the student? They could do so but would it not be better to allow such a student to enrol (and set the address to 'unkown')?

To allow this the method enrollStudent() could be overloaded and an alternative method provided as…

enrollStudent(String pName, String pCoursecode)

At run time the JRE could determine which method to invoke depending upon the parameter list provided. Thus given a call to

enrollStudent("Fred", "123 Abbey Gardens", "G700")

the JRE would use the first method.

Activity 1

Imagine a method withdrawCash() that could be used as part of a banking system. This method could take two parameters:- the account identity (a String) and the amount of cash required by the user (int). Thus the full method signature would be:-

withdrawCash(String pAccountID, int pAmount).

Identify another variation of the withdrawCash() method that takes a different parameter list that may be a useful variation of the method above.

Feedback 1

An alternative method also used to withdraw cash could be withdrawCash(String pAccountID) where no specified amount is provided but by default £100 is withdrawn.

These methods essentially perform the same operation but by overloading this method we have made the system more flexible – users now have a choice they can specify the amount of cash to be withdrawn or they can accept the default sum specified.

Overloading methods don't just provide more flexibility for the user they also provide more flexibility for programmers who may have the job of extending the system in the future and thus overloading methods can make the system more future proof and robust to changing requirements.

Constructors can be overloaded as well as ordinary methods.

Download free eBooks at bookboon.com

Activity 2

Go online and look at the Java Standard Edition API documentation by 1) going online to **java.sun.com/javase** 2) following the link to the API 3) selecting the link for Core API documents for the latest version.

At the time of writing this should take you to http://java.sun.com/javase/6/docs/api/

In the lower left panel you should see a long list of all the classes available to Java programmers. Scroll down this list until you find the String class. This will be quicker if you first select the Java.lang package in the upper left window (as the String class is in this package).

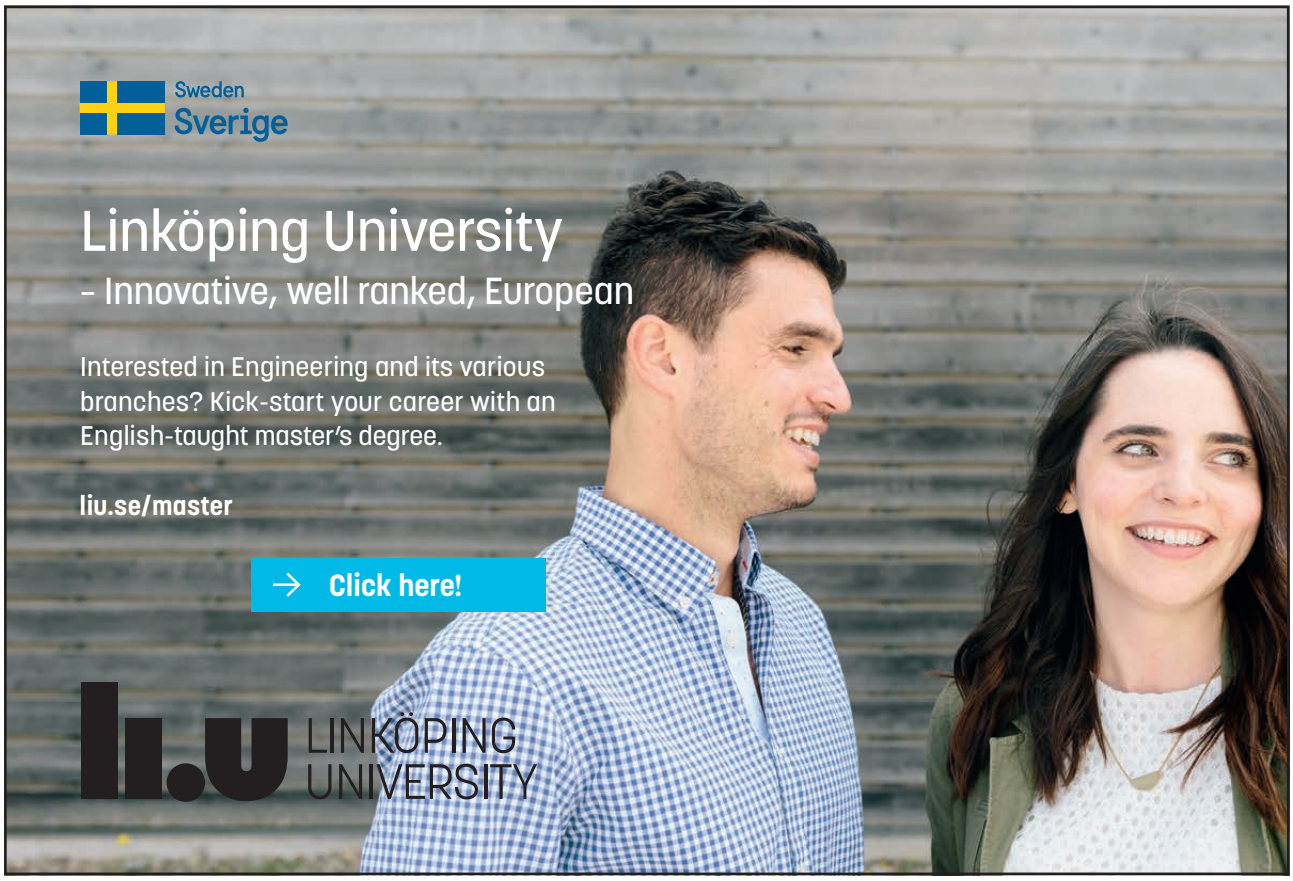
Look at the String class documentation in the main pane and find out how many constructors exist for this class.

Feedback 2

The String class specifies 15 different constructors. They all have the same method name 'String' of course but they can all be differentiated by the different parameters these methods require.

One of these constructors takes no parameters and creates an empty String object. Another requires a String as a parameter and creates a new String object that is a copy of the original.

By massively overloading the Sting constructor the creators of this class have provided flexibility for other programmers who may wish to use these different options in the future.



Click on the ad to read more

Download free eBooks at bookboon.com

We can make our programs more adaptable by overloading constructors and other methods. Even if we don't initially use all of the different constructors, or methods, by providing them we are making our programs more flexible and adaptable to meet changing requirements.

Activity 3

Still looking at the String class in the API documentation find other methods that are overloaded.

Feedback 3

There are many methods that are not overloaded but there are also many that are. These include:- format(), indexOf(), replace(), split(), subString() and valueOf().

Looking at the different subString methods we see that we can find a substring by either specifying the starting point alone or by specifying start and end points.

When we use the subString() method the JRE will select the correct implementation of this method, at run time, depending upon whether or not we have provided one or two parameters.

5.3 Summary

Method overloading is the name given to the concept that several methods may exist that essentially perform the same operation and thus have the same name. The JRE distinguishes these by looking at the parameter list. If two or more methods have the same name then their parameter list must be different.

At run time each method call, which may be ambiguous, is resolved by the JRE by looking at the parameters passed and matching the data types with the method signatures defined in the class.

By overloading constructors and ordinary methods we are providing extra flexibility to the programmers who may use our classes. Even if these are not all used initially, providing these can help make the program more flexible to meet changing user requirements.

Download free eBooks at bookboon.com

6 Object Oriented Software Analysis and Design

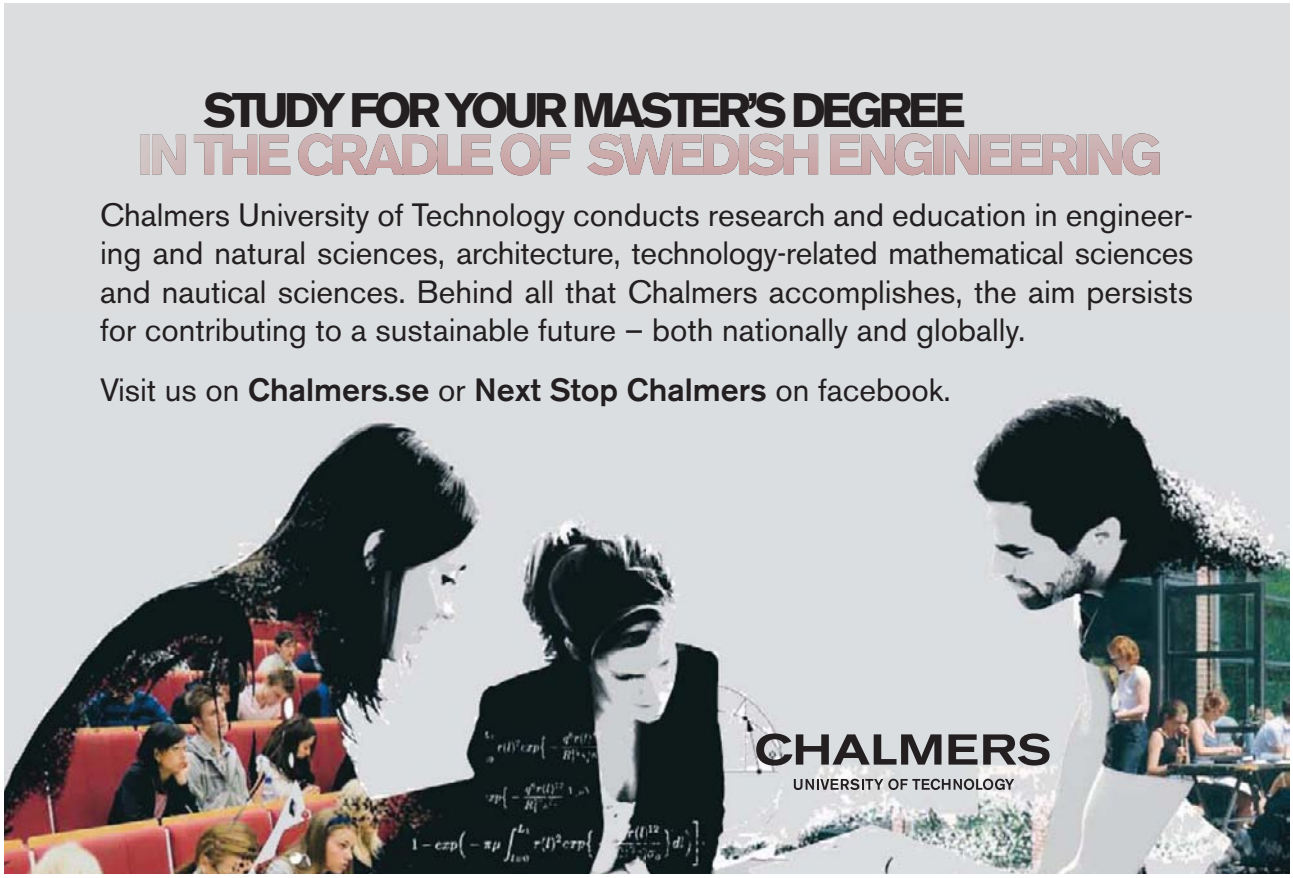
Introduction

This chapter will teach rudimentary analysis and modelling skills through practical examples. Leading the reader to an understanding of how to get from a preliminary specification to an Object Oriented Architecture.

Objectives

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

- Analyse a requirements description
- Identify items outside scope of system
- Identify candidate classes, attributes and methods
- Document the resulting Object Oriented Architecture



Click on the ad to read more

Download free eBooks at bookboon.com

This chapter consists of twelve sections:-

1) Requirements Analysis
2) The Problem
3) Listing Nouns and Verbs
4) Identifying Things Outside The Scope of The System
5) Identifying Synonyms
6) Identifying Potential Classes
7) Identifying Potential Attributes
8) Identifying Potential Methods
9) Identifying Common Characteristics
10) Refining Our Design using CRC Cards
11) Elaborating Classes
12) Summary

6.1 Requirements Analysis

The development of any computer program starts by identifying a need:-

- An engineer who specialises in designing bridges may need some software to create three dimensional models of the designs so people can visualise the finished bridge long before it is actually built.
- A manager may need a piece of software to keep track of personal, what projects they are assigned to, what skills they have and what skills needs to be developed.

But how do we get from a 'need' for some software to an object oriented software design that will meet this need?

Some software engineers specialise in the task of Requirement Analysis which is the task of clarifying exactly what is required of the software. Often this is done by iteratively performing the following tasks:-

1) interviewing clients and potential user of the system to find out what they say about the system needed
2) documenting the results of these conversations,
3) identifying the essential features of the required system
4) producing preliminary designs (and possibly prototypes of the system)
5) evaluating these initial plans with the client and potential users
6) repeating the steps above until a finished design has evolved.

Download free eBooks at bookboon.com

Performing requirements analysis is a specialised skill that is outside the scope of this text but here we will focus on steps three and four above ie. given a description of a system how do we convert this into a potential OO design.

While we can hope to develop preliminary design skills experience is a significant factor in this task. Producing simple and elegant designs is important if we want the software to work well and be easy to develop however identifying good designs from weaker designs is not simple and experience is a key factor.

A novice chess player may know all the rules but it takes experience to learn how to choose good moves from bad moves and experience is essential to becoming a skilled player. Similarly experience is essential to becoming skilled at performing user requirements analysis and in producing good designs.

Here we will attempt to develop rudimentary skills in the hope that you will have the opportunity to practise those skills and gain experience later.

Starting with a problem specification we will work through the following steps:-

- Listing Nouns and Verbs
- Identifying Things Outside The Scope of The System
- Identifying Synonyms
- Identifying Potential Classes
- Identifying Potential Attributes
- Identifying Potential Methods
- Identifying Common Characteristics
- Refining Our Design using CRC Cards
- Elaborating Classes

By doing this we will be able to take a general description of a problem and generate a feasible, and hopefully elegant, OO design for a system to meet these needs.

6.2 The Problem

The problem for which we will design a solution is 'To develop a small management system for an athletic club organising a marathon.'

For the purpose of this exercise we will assume preliminary requirements analysis has been performed and by interviewing the club managers, and the workers who would use the system, the following textual description has been generated.

Download free eBooks at bookboon.com

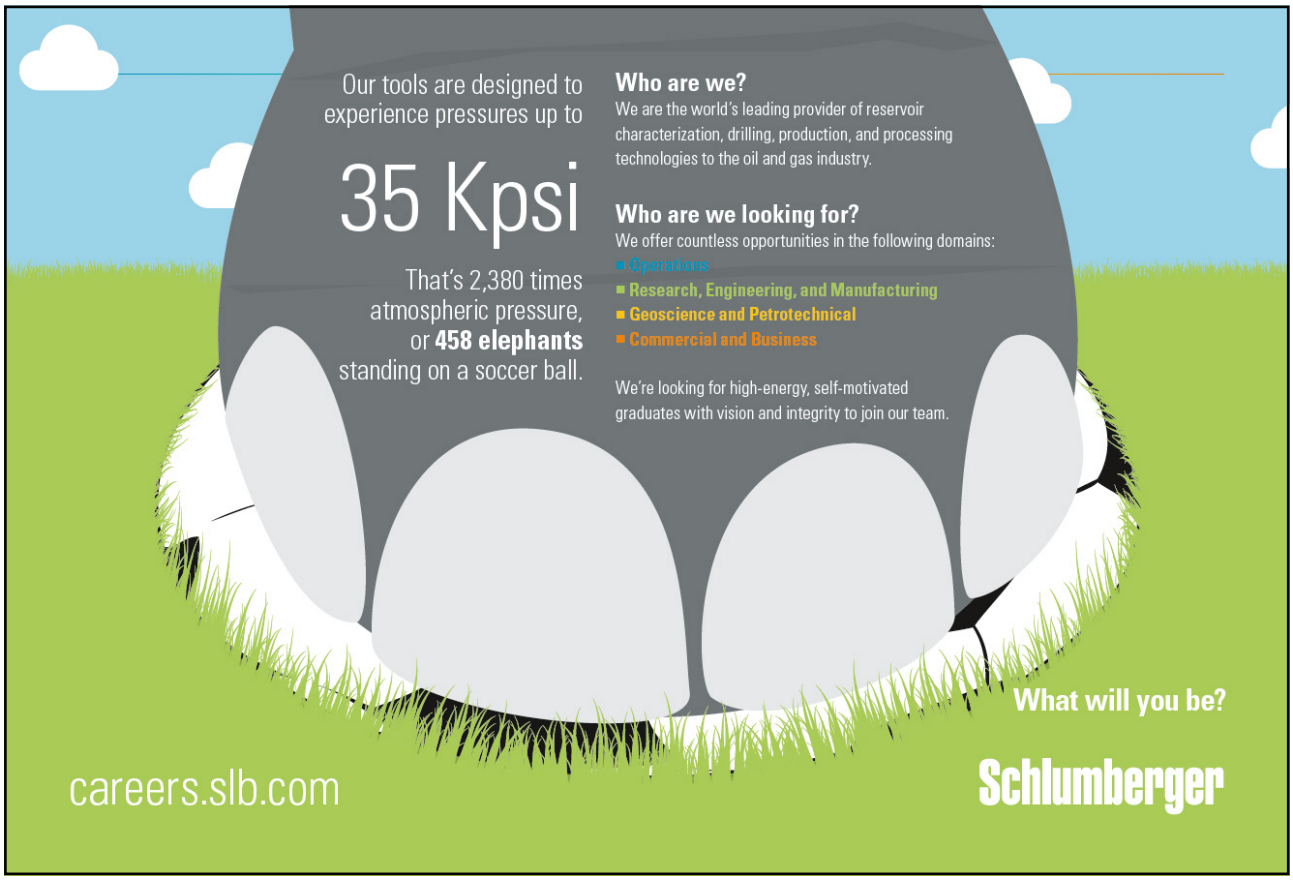
The 'GetFit' Athletic Club are organizing their first international marathon in the spring of next year. A field comprising both world-ranking professionals and charity fund-raising amateurs (some in fancy dress!) will compete on the 26.2 mile route around an attractive coastal location. As part of the software system which will track runners and announce the results and sponsorship donations, a model is required which represents the key characteristics of the runners (this will be just part of the finished system).

Each runner in the marathon has a number. A runner is described as e.g. "Runner 42" where 42 is their number. They finish the race at a specified time recorded in hours, minutes and seconds. Their result status can be checked and will be displayed as either "Not finished" or "Finished in hh:mm:ss".

Every competitor is either a professional runner or an amateur runner.

Further to the above, a professional additionally has a world ranking and is described as e.g. "Runner 174 (Ranking 17)".

All amateurs are fundraising for a charity so each additionally has a sponsorship form. When an amateur finishes the race they print a collection list from their sponsorship form.



Click on the ad to read more

Download free eBooks at bookboon.com

A sponsorship form has the number of sponsors, a list of the sponsors, and a list of amounts sponsored. A sponsor and amount can be added, and a list can be printed showing the sponsors and sponsorship amounts and the total raised.

A fancy dress runner is a kind of amateur (with sponsorship etc.) who also has a costume, and is described as e.g. "Runner 316 (Yellow Duck)".

6.3 Listing Nouns and Verbs

The first step in analysing the description above is to identify the nouns and verbs:-

- The nouns indicate entities, or objects, some of these will appear as classes in the final system and some will appear as attributes.
- The verbs indicate actions to be performed some of these will appear in the final system as methods.

Nouns and verbs that are plurals are listed in their singular form (e.g. 'books' becomes 'book') and noun and verb phrases are used where the nouns\verb alone are not descriptive enough e.g. the verb 'print' is not as clear as 'print receipt'.

Activity 1

Look at the description above list five nouns and five verbs (use noun and verb phrases where appropriate).

Feedback 1

The list below is a fairly comprehensive list of the nouns and verbs, not just the first five.

Nouns:- GetFit Althletic Club, field, world ranking professional, fund-raising amateur, fancy dress, 26.2 mile route, coastal location, software system, runner, result, sponsorship donation, model, key characteristic, a number, time, result status, competitor, professional runner, amateur runner, world ranking, charity, sponsorship form, collection list, sponsor, sponsorship amount, total raised, costume.

Verbs:- Organise, marathon, compete, track runners, announce results, describe (runner), finish race, specify time, check status, display status, describe (professional), print collection list, add (sponsor and amount), print list, describe (fancy dress runner)

Download free eBooks at bookboon.com

6.4 Identifying Things Outside The Scope of The System

An important part in designing a system is to identify those aspects of the problem that are not relevant or outside the scope of the system. Parts of the description may be purely contextual i.e. for general information purposes and thus not something that will directly describe aspects of the system we are designing. Furthermore while parts of the description may refer to tasks that are performed by users of the system as they are using the system, and thus describe functions that need to be implemented within the system, other parts may describe tasks performed by users while not using the system – and thus don't describe functions within the system.

By identifying things in the description that are not relevant to the system we are developing we keep the problem as simple as possible.

Activity 2

Look at the list of nouns and verbs above and identify one of each that is outside the scope of the system.

Feedback 2

Most of the first paragraph is contextual and does not describe functionality we need to implement within the system. We also need to look at other parts of the description to identify parts that are not relevant.

Things outside the scope of the system…

Nouns:-

- GetFit Althletic Club – this is the client for whom the system is being developed. It is not an entity we need to model within the system.
- Coastal location – the location of the run is not relevant to the functionality of the system as described. Again we do not need to model this as an object within the system.
- Software system – this is the system we are developing as a whole it does not describe an entity within the system.
- Verbs:-
- Organise – this is an activity done by members of the athletic club, these may be users of the system but this is not an activity that they are using the system for.
- Marathon – this is what the runners are doing. It is not something the system needs to do.

Note: 'Finish race' is something that a runner does however when this happens their finish time must be recorded in the system. Therefore this in NOT in fact outside the scope of the system.

Download free eBooks at bookboon.com

6.5 Identifying Synonyms

Synonyms are two words that have these same meaning. It is important to identify these in the description of the system. Failure to do so will mean that one entity will be modelled twice which will lead to duplication and confusion.

Activity 3

Look at the list of nouns and verbs and identify two synonyms, one from the list of nouns and one from the verbs.



Click on the ad to read more

Download free eBooks at bookboon.com

Feedback 3

Synonyms

Nouns:-

- world ranking professional=professional runner
- fund-raising amateur=amateur runner
- runner=competitor

Note runner is not a synonym of professional runner as some runners are amateurs.

Verbs:-

- marathon=compete
- check status=display status
- print collection list = print list
- finish race = record specified time

6.6 Identifying Potential Classes

Having simplified the problem by identifying aspects that are outside the scope of the system and by identifying different parts of the description that are in reality describing the same entities and operations we can now start to identify potentials classes in the system to be implemented.

Some nouns will indicate classes to be implemented and some will indicate attributes of classes.

Good OO design suggests that data and operations should be packaged together – thus classes represent complex conceptual entities for which we can identify associated data and operations (or methods).

Simple entities, such as an address, have associated data but no operations and thus these can be stored as simple attributes within a related class.

Activity 4

Look at the list of nouns above and identify five that could become classes.

Feedback 4

Nouns that could indicate classes:-

- Runner (or Competitor)
- Amateur (or Amateur Runner)
- Professional (or similar)
- FancyDresser (or FancyDressAmateur or similar)
- Sponsorshipform

Download free eBooks at bookboon.com

6.7 Identifying Potential Attributes

Having identified potential classes the other nouns could by used to identify attributes of those classes.

Activity 5

Look at the list of nouns and identify one that could become an attribute of the class 'Runner' and one for the class 'FancyDresser'.

Feedback 5

Nouns that could become attributes...

For Runner:-

number,
resultStatus ie. finished (boolean)
time (hours, minutes, seconds)

For FancyDresser:-

costume (String)

Of course we need to identify all of the attributes for all of the classes.

6.8 Identifying Potential Methods

Having identified potential classes we can now use the verbs to identify methods of those classes.

Activity 6

Look at the list of verbs and identify one that could become a method of the class 'Runner' and one for the class 'FancyDresser'.

Feedback 6

Verbs that could become methods....

For Runner:-

describe (this will actually become an overridden version of toString())
finishRace
displayStatus

For FancyDresser:-

describe (toString() will need to be overridden again to encompass the description of the costume)

Of course we need to identify all of the methods for all of the classes.

Download free eBooks at bookboon.com

6.9 Identifying Common Characteristics

Having identified the candidate classes with associated attributes and methods we can start structuring our classes into appropriate inheritance hierarchies by identifying those classes with common characteristics.

Activity 7

Look at the list of classes below and place four into an appropriate inheritance hierarchy. Identify the one class that would not fit into this hierarchy:-

Runner
Amateur
Professional
FancyDresser
Sponsorshipform



Click on the ad to read more

Download free eBooks at bookboon.com

Feedback 7

The most general class i.e. the one at the top of the inheritance tree is 'Runner'. Amateur and Professional are subclasses of 'Runner' and FancyDresser is a specific type of Amateur hence a subclass of Amateur.

We can fit these into an inheritance hierarchy because these classes are all related by an is-a relationship. A FancyDresser is-a Amateur which in turn is-a Runner. A Professional is-a Runner as well.

A SponsorshipForm is not a type of Runner and hence does not fit into this hierarchy. This class will be related to one of the other classes by some form of an association. Looking at the description we can see that not all runners have a sponsorship form only amateurs who are running for charity. There is therefore an association between Amateur and SponsoshipFrom. Of course FancyDressers inherit the attributes defined in Amateur and hence they automatically have a SponsorshipFrom.

6.10 Refining Our Design using CRC Cards

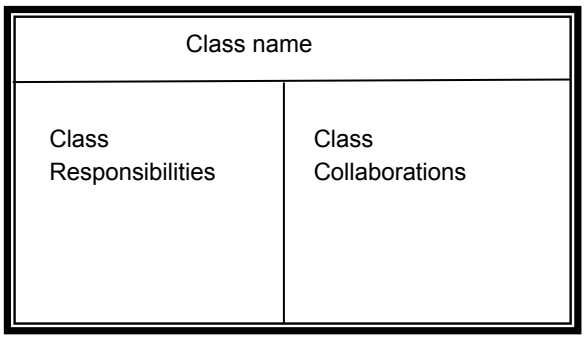
Having identified the main classes in our system, and the attributes and methods of these classes, we could now proceed to refine these designs by defining the data types and other small details, document this information on a UML diagram and program the system. However in a real world system the problem would be larger and less well-defined than the problem we are working on here and the analysis and refinement of design would therefore be a longer more complex process that we can realistically simulate.

As real world problems are more complex our initial designs are unlikely to be perfect therefore it makes sense to check our designs and to resolve any potential problems before turning these designs into a finished system.

One method of doing checking our designs is to document our designs using CRC cards and to check if these work by role-playing different scenarios.

CRC cards are not the only way of doing this and are not part of the UML specification.

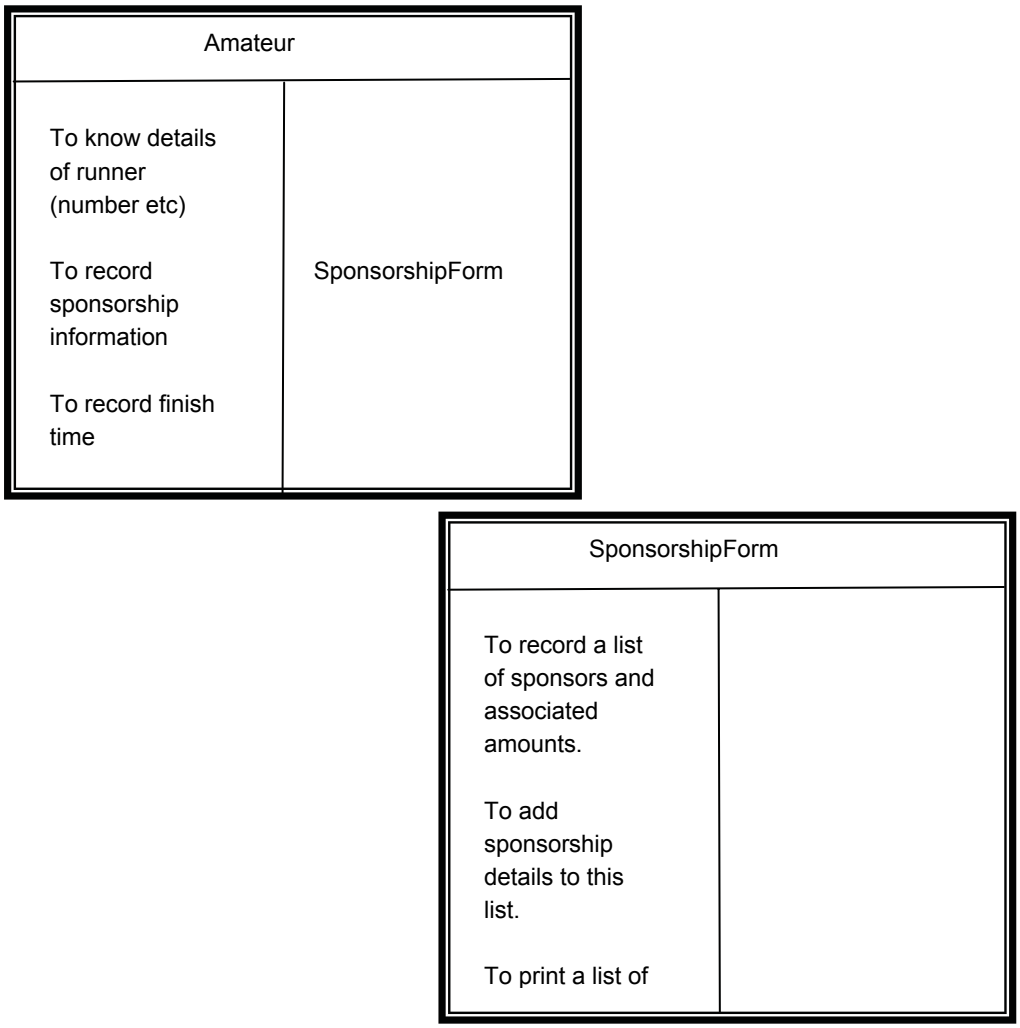
CRC stands for Class, Responsibilities and Collaborations. A CRC card is set out below and is made up of three panes with the class responsibilities shown on the left and the collaborations shown on the right.



Download free eBooks at bookboon.com

Responsibilities are the things the class needs to know about (ie the attributes) and the things it should do (ie. the methods) though on a CRC card these are not as precisely defined as on a UML diagram. The collaborations are other classes that this class must work with in order to fulfil its responsibilities.

The diagram below shows CRC cards developed for two classes in the system.



Having developed CRC cards we can roleplay a range of scenarios in order to check the system works 'on paper'. If not we can amend before getting into the time consuming process of programming a flawed plan.

One sample scenario would be when a runner gets an additional sponsor. In this case by looking at the CRC cards above a Runner is able to record sponsorship information in collaboration with the SponsorshipForm class. The SponsorshipForm class records a list of sponsors and can add additional sponsor to this list.

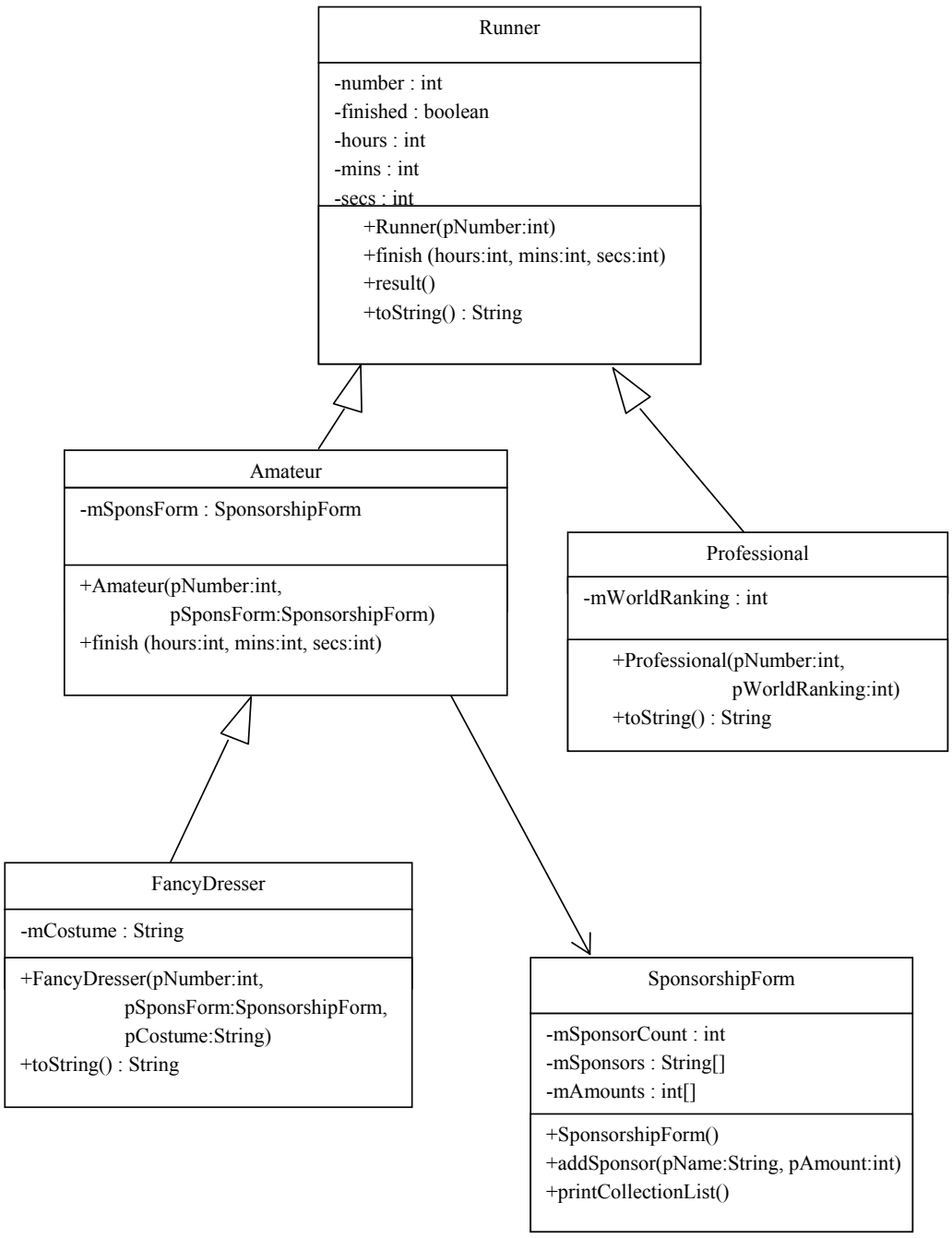
Testing out a range of scenarios may highlight flaws in our system designs that we can then fix – long before any time has been wasted by programming weak designs.

Download free eBooks at bookboon.com

6.11 Elaborating Classes

Having identified the classes in our system design, and documented and tested these using CRC cards, we can now elaborate our CRC cards and document our classes using a UML class diagram. To do this we need to take our general specification documented via CRC cards and our resolve any ambiguities e.g. exact data types.

Having elaborated our CRC cards we can now draw a class diagram for proposed design (see below):-



Download free eBooks at bookboon.com

6.12 Summary

Gathering User Requirements is an essential stage of the software engineering process (and outside the scope of this text).

Turning a complex requirements specification into an elegant simple object oriented architectural design is a skilled task that requires experience. However a good starting point is to follow a simple process set out in this chapter.

Through a sequence of tasks we have seen how to analyse a textual description of a problem. We have:-

- Looked for aspects of the description that are outside the scope of the system,
- Identified where the description refers synonymous items using different words
- Used the nouns and verbs to identify potential classes, attributes and methods
- Looked at the classes to identify potential inheritance hierarchies and to identify other relationships between classes (e.g. associations).
- Document the resulting classes using CRC cards and tested the validity of our design by role-playing a range of scenarios and amending our designs as appropriate
- Finally we can elaborate these details and document the results using a class diagram.

The design process set out in this chapter will be demonstrated again in detail using the case study described in chapter 11.

Download free eBooks at bookboon.com

7 The Collections Framework

Introduction

This chapter will introduce the reader to the collections framework – an important part of the standard Java library.

Objectives

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

- Understand the basic concepts of the Java Collections Framework
- Contrast the characteristics of various Collections interfaces
- Understand the related concept of an iterators



Click on the ad to read more

Download free eBooks at bookboon.com

This chapter consists of twelve sections:-

1) An Introduction to Collections
2) Collection Interfaces
3) Old and New Collections
4) Lists
5) Sets
6) Maps
7) Collection Implementations
8) Overview of the Collections Framework
9) An Example Using Un-typed Collections
10) An Example Using Typed Collections
11) A Note About Sets
12) Summary

7.1 An Introduction to Collections

Most software systems need to store various groups of entities rather than just individual items. Arrays provide one means of doing this, but the Java Collections support much more varied and flexible forms of grouping.

In Java, collections (or 'containers') are classes which serve to hold together groups of other objects. The Java platform provides a 'Collections Framework', a consistent set of interfaces and implementations

- interfaces define the available functionality
- implementations influence other issues including performance

7.2 Collection Interfaces

At the core of the collections framework is an interface called 'Collection'. This defines methods that are at the core of the framework. List and Set are interfaces that are extensions of Collection – they inherit all its operations and add some more.

Note that because these are interfaces, not classes, they only define operation signatures, not any aspect of their implementation as methods.

An important additional interface is 'Map'. This is not an extension of Collection – this is because maps do not entirely fit in to the Collection hierarchy for reasons we will see later, although they are still part of the 'Collections Framework'.

Download free eBooks at bookboon.com

Thus we have the following interfaces…

- **Collection**: the most general grouping
  - **List:** a collection of objects (which can contain duplicates) held in a specific order
  - **Set**: a collection of unique objects in no particular order
    - **SortedSet**: a set with objects arranged in ascending order

- **Map**: a collection of unique 'keys' and associated objects
  - **SortedMap**: a map with objects arranged in ascending order of keys

Just as List and Set extend the Collection interface, SortedSet is an extension of Set and SortedMap is an extension of Map.

We will look at List, Sets and Maps in this chapter.

7.3 Old and New Collections

Up to Java SDK 1.4, collections held items of type Object – i.e. objects of any class since all are subclasses of Object. This allowed a mixture of objects in a collection. However when objects are taken out of a collection the compiler does not know what the object is and this can cause some additional complications.

In practise we almost always want only objects of some specific type in a particular collection. Thus in Java JDK 5.0 'generics' were introduced. 'Generics' allow a class to be defined without specifying the types of data items it handles. Instead the types are specified when the class is instantiated, and so are fixed for particular objects.

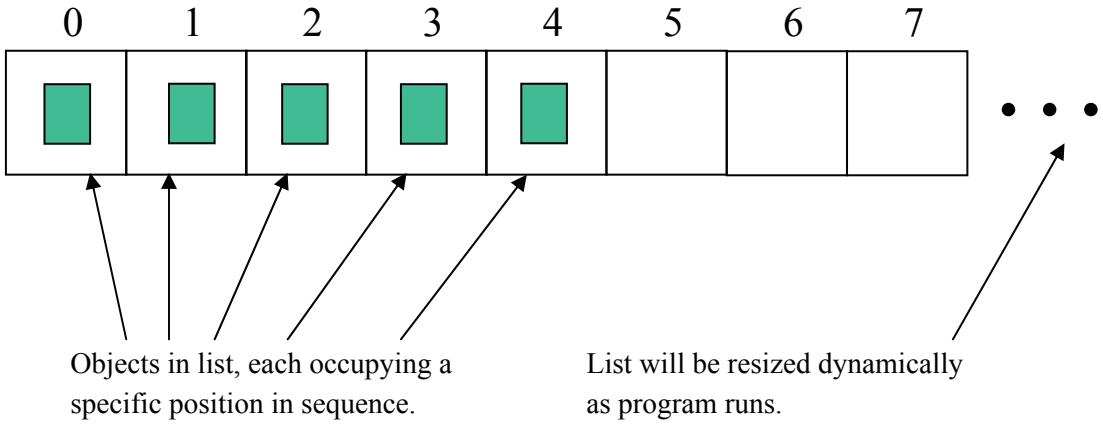
Thus using 'generics' we can create collections that only store objects of a specified type.

7.4 Lists

Lists are the most commonly used type of collection – where you might have used an array, a List will often provide a more convenient method of handling the data.

Lists are very general-purpose data structures, with each item occupying a specific position. They are in many ways like arrays, but are more flexible as they are automatically resized as data is added. They are also much easier to manipulate than arrays as many useful methods have been created that do the bulk of the work for you. Lists store items in a particular sequence (though not necessarily sorted into any meaningful order) and duplicate items are permitted.

Download free eBooks at bookboon.com



7.5 Sets

A set is like a 'bag' of objects rather than a list. They are based on the mathematical idea of a 'set' – a grouping of 'members' that can contain zero, one or many distinct items.

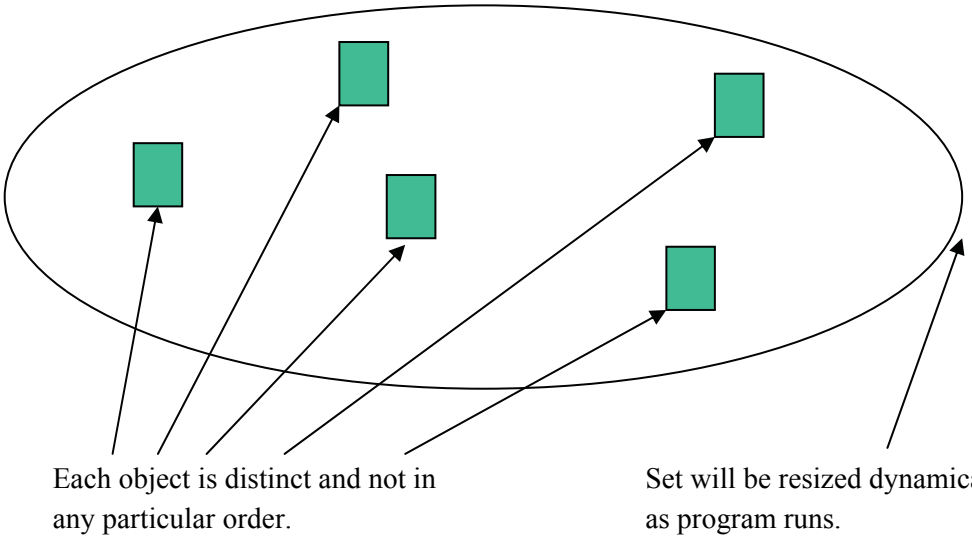
Unlike a List, duplicate items are not permitted in a set and a Set does not arrange items in order (but a SortedSet does).



Click on the ad to read more

Download free eBooks at bookboon.com

Like lists, sets are resized automatically as items are added



Many of the operations available for a List are also available for a Set, but

- we CANNOT add an object at a specific position since the elements are not in any order
- we CANNOT 'replace' an item for the same reason (though we can add one and delete another)
- retrieving all the items is possible but the sequence is indeterminate
- it is meaningless to find what position an element is in.

7.6 Maps

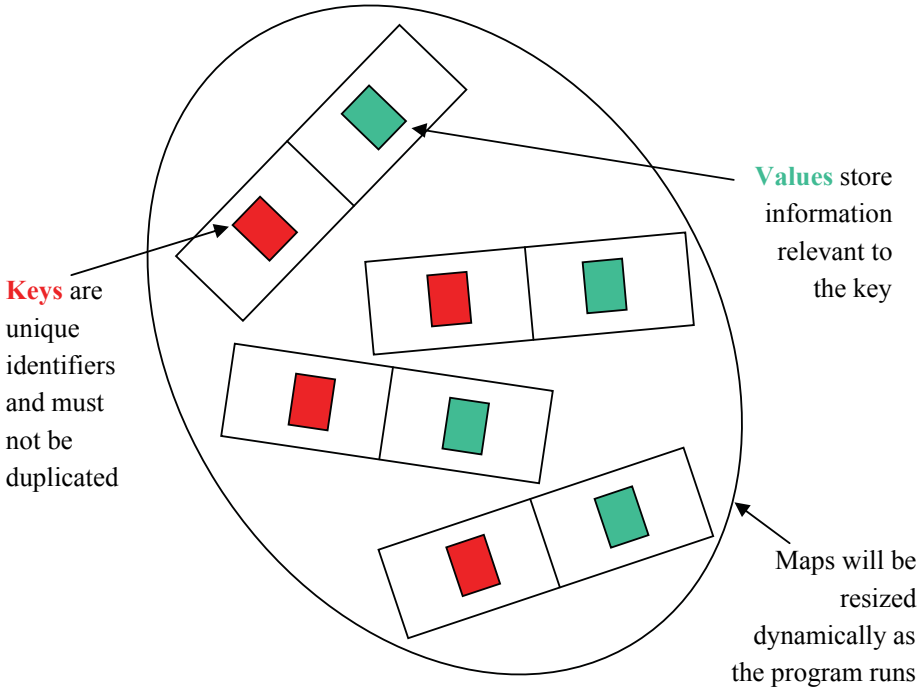
Maps are rather different from Lists and Sets because instead of storing individual objects they store pairs of objects. The pair is made up of a 'key' and a 'value'. The key is something which identifies the pair. The value is a piece of information or an object associated with the key.

Example: in an address book the keys would be people's names and the values their address, phone, email etc. There is only one value for each key, but since values are objects they can contain several pieces of data.

Duplicate keys are not permitted, though duplicate values are. So in the previous example if you looked up two people in the address book you may find them living at the same address – but one person would not have two homes.

Like a set, a Map does not arrange items in order (but a SortedMap does, in order of keys) and like lists and sets, maps are resized automatically as items are added or deleted.

Download free eBooks at bookboon.com



Activity 1

For each of the following problems write down which would be most appropriate:- a list, a set or a map.

1) We want to record the members of a club.
2) We want to keep a record of the money we spend (and what it was spent on).
3) We want to record bank account details – each identified by a bank account number

Feedback 1

1) For this we would use a Set. Members can be added and removed as required and the members are in no particular order.
2) For this we would use a List – this would record the items bought in the order in which they were purchased. Importantly as lists allows duplicate items we could buy two identical toys (perhaps for birthday presents for two different children),
3) We would not have two identical Bank accounts so a Set seems appropriate however as each is identified by a unique account number a Map would be the most appropriate choice.

7.7 Collection Implementations

Up to now we have been looking at the collections **interfaces** – specifications of functionality from which we can pick what we want to use.

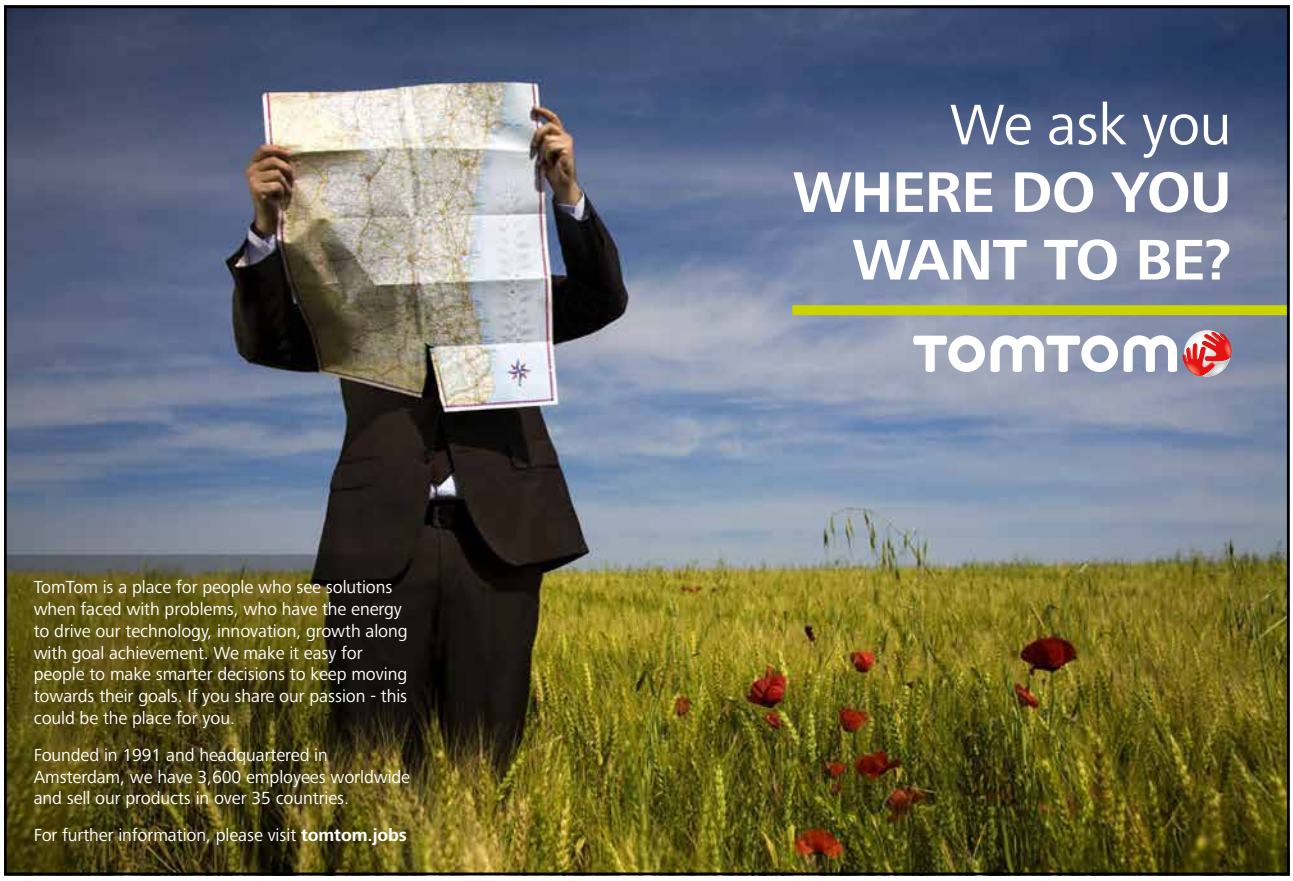
To actually use them we need **classes which implement** these interfaces.

Download free eBooks at bookboon.com

The implementations shown here are all part of the Java platform library packages, but programmers can also write implementations of their own for special purposes.

- **ArrayList** implements the List interface
  - generally fast access for an ordered list
- **LinkedList** this also implements the List interface
  - may be faster than ArrayList in some less usual circumstances
- **HashSet** implements the Set interface
  - It provide fast access but the items are unordered
- **TreeSet** this also implements the Set interface
  - slower than the HashSet but it maintain elements in order ie. it provides us with a SortedSet
- **HashMap** implements the Map interface
  - It provide fast access but the items are unordered
- and **TreeMap** which also implements the Map interface
  - slower than a HashMap it maintain elements in order of the Key ie. it provides us with a SortedMap

You can't create a 'List object' as List is an interface – but you can create an ArrayList object or a LinkedList object, and either of these will provide the functionality defined by the List interface.



Click on the ad to read more

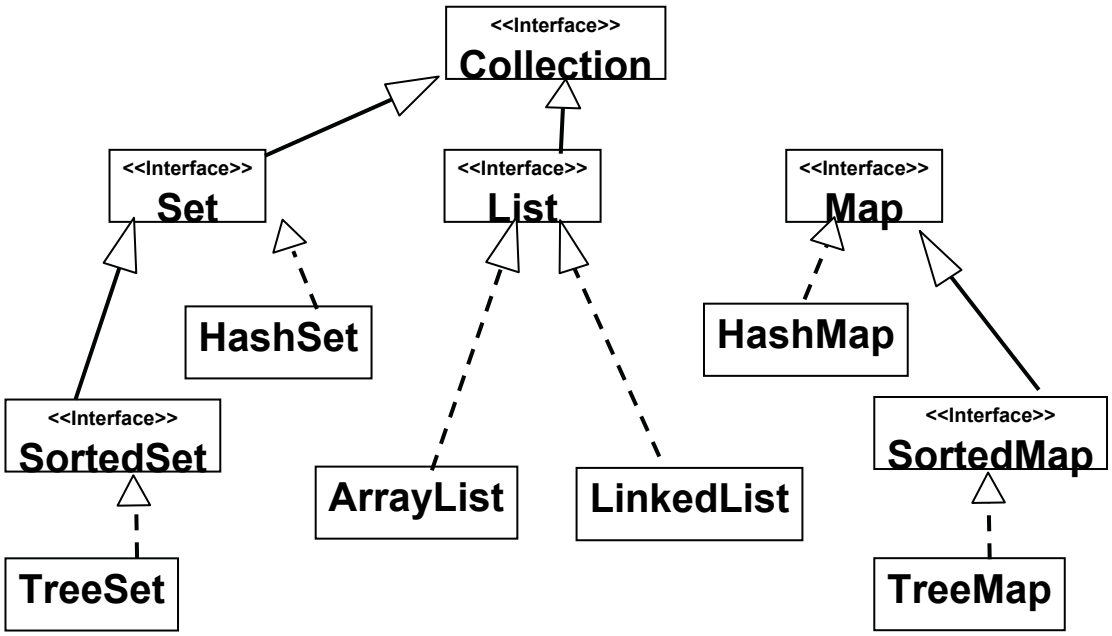
Download free eBooks at bookboon.com

7.8 Overview of the Collections Framework

The diagram below gives us a slightly simplified overview of the collections framework. It shows the interfaces and the classes that implement the interfaces. Remember, interfaces have no implementation within them, they just contain public method signatures with must be fulfilled by any class implementing that interface.

There are two types of arrows shown. Arrows with solid lines denote inheritance; e.g. the **Set** interface extends the **Collection** interface. Arrows with dotted lines denote implementation; e.g. the **HashSet** class implements the **Set** interface

Note that Map is not a subinterface of Collection. This is because, unlike Lists and Sets which store individual objects, Maps store pairings of key objects and value objects. We still consider Map to be part of the collections framework.



Activity 2

Documentation for all the collections interfaces and implementations is available to browse online or download at:

http://java.sun.com/javase/6/docs/api/

The interfaces and implementation classes of the Collections Framework can all be found in the **java.util** package.

Use the online API documentation to find the methods for an ArrayList.

List three of the methods you think would be most useful.

Download free eBooks at bookboon.com

Feedback 2

Some of the clearly useful methods include…

add() – which adds an element into a specified position in the list
clear() – which clears the list
contains() – which returns true if this list contains the specified object.
get() – which returns the object at a specified position and
remove() – which removes an object from a list

Note some of these methods are overloaded such as remove() which can either remove an object at a specified position or remove the first instance of a specified object.

INNOVATIVE
LIKE YOU.

If you're hoping for a truly modern education, one where you're encouraged to speak your mind and to think long-term, both when it comes to your own future and the future of the planet. Then the University of Gothenburg is the place for you.

Study a Master's programme in Gothenburg, Sweden | www.gu.se/education





Click on the ad to read more

Download free eBooks at bookboon.com

7.9 An Example Using Un-typed Collections

The code below shows an example of an Un-typed list of Strings.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
public class ListDemo
{
      List mAList;

      /**
      * Constructor
      */
      public ListDemo ()
      {
            mAList = new ArrayList();
      }

      /**
      * Append a string at the end of the list
      * @param pStr string to be added to list
      */
      public void appendString(String pStr)
      {
            mAList.add(pStr);
      }

      /**
      * Insert a string at a specified position in the list
      * @param pPos position at which string to be added
      *                                          (0 = before first)
      * @param pStr string to be added to list
      */
      public void insertString(int pPos, String pStr)
      {
            mAList.add(pPos, pStr);
      }

      /**
      * Delete the string at a specified position in the list
      * @param pPos position at which string to be added
      */
      public void deleteString(int pPos)
      {
            mAList.remove(pPos);
      }
```


Download free eBooks at bookboon.com

```
        /**
        * Display list of strings
        */
        public void display()
        {
                String nextItem;
                Iterator it = mAList.iterator();

                while (it.hasNext())
                {
                        nextItem = (String)it.next();
                        System.out.print(nextItem + " ");
                }
                System.out.println();
        }
}
```




Click on the ad to read more

Download free eBooks at bookboon.com

Mostly the code above is self explanatory however perhaps the display method needs some explanation. Firstly List, Set and Map classes implement the iterator() method this method returns an object of type Iterator and this object is capable of iterating around Lists, Sets or Maps. Note – this is not a simple process as each collection can be made up of different objects but, thankfully the hard work has been done for us.

Thus in the display() method above a variable 'it' is created of the general type Iterator. The method iterator() is then invoked on our specific collection thus returning an object capable of iterating around our List of Strings. The Iterator class has two very useful methods:- hasNext() which return true is another object exists in the collection and next() which returns the next object in the collection.

Thus the loop above iterates around the list rerunning and displaying each object.

There is one final thing to note: because we are using un-typed collection each object in the list could be a different type of object and hence when an object is retrieved from the list the JRE does not know what type of object it is – however as we are only using this list to store Strings we can cast each object retrieved onto a String. This will cause a compiler warning as this will fail if we add non String objects to the list.

While this program stores and manipulates a list of Strings the program could just as easily store a list of Publication – or any object constructed from a class we create.

7.10 An Example Using Typed Collections

Using typed collections this is simpler for two reasons:- 1) because we know what type of object will be returned from the list and hence we don't need to use the cast operator and 2) because Java 5.0 introduced an improved for loop that can loop through every item in a typed collection without using a loop counter (or iterator).

Download free eBooks at bookboon.com

Hence see example below….

```
import java.util.List;
import java.util.ArrayList;
public class ListDemo

{
	private List<String> mAList;

	/**
	* Constructor
	*/
	public ListDemo ()
	{
		mAList = new ArrayList<String>();
	}

	/**
	* Display list of strings
	*/
	public void display()
	{
		for (String nextItem: mAList)
		{
			System.out.print(nextItem + " ");
		}
		System.out.println();
	}
}
```

Creating a typed collection is very similar to creating an untyped collection except when we create the instance variable and create the actual collection we must specify what the collection contains (in this case it is simply a collection of Strings). You can think of < > syntax to mean 'of' ie in this case a list of strings

Appending, inserting and deleting elements is just the same as with untyped collections however displaying the collection is much simpler as we know what each element of the list contains. We don't need to cast the elements returned and in fact we don't even need an iterator (hence why there are only two import statements above.

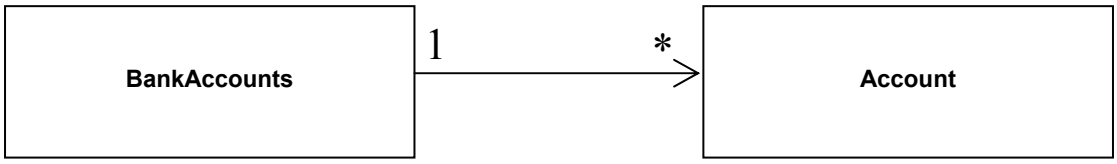
Download free eBooks at bookboon.com

nextItem is automatically assigned to the next item in the list. Note nextItem is defined as a String variable because here we are using a list of Strings. The compiler knows that and thus will ensure that the correct type of variable is used.

7.11 A Note About Sets

A thorough discussion of Sets is beyond the scope of this text however there is one complication with Sets that must be noted. Sets don't allow duplicate object to be stored. Thus we must ask what makes an object a duplicate. Are two object only the same if the name of the object is the same – or could they be the same if some of the data is identical?

Consider a set of bank accounts:-

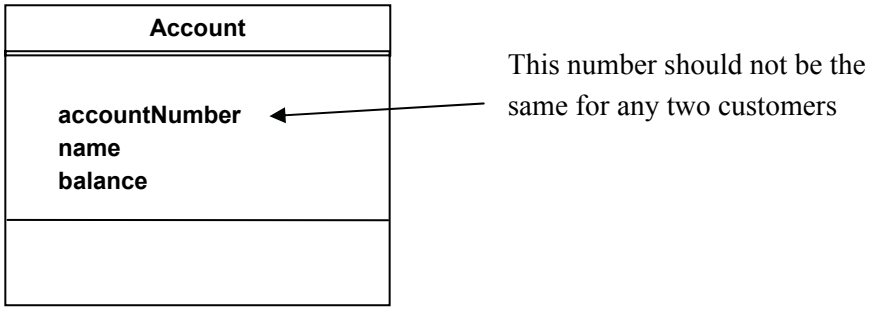




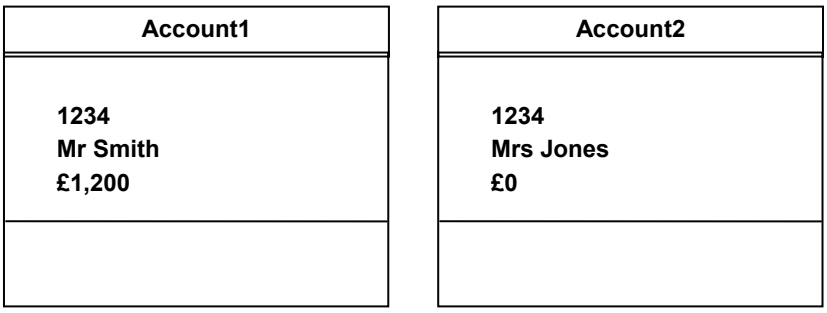
Click on the ad to read more

Download free eBooks at bookboon.com

Now consider two bank accounts one for Mr Smith and one for Mrs Jones



It should not be possible to create a second account with the same account number and add it to a set of bank accounts. However unless told otherwise Java will treat the two object below as different objects as both objects have different names (Account1 and Account2) and thus while Sets do not allow duplicates objects inside them both of these accounts could be created and added to a set.



To overcome this problem we need to override the equals() method defined in the Object class to say these objects are the same if the accountNumber is the same.

We can do this as below…

```
public boolean equals (Object pObj)
{
      Account account = (Account) pObj;
      return (accountNumber.equals(account.accountNumber);
}
```

This overrides Object.equals() for the Account class

Download free eBooks at bookboon.com

Even though it will always be an Account object passed as a parameter, we have to make the parameter an Object type (and then cast it to Account) in order to override the equals() method inherited from Object which has the signature

public boolean equals (Object obj)

(You can check this in the "API and Language" section of the JDK documentation.)

If we gave this method a signature with an Account type parameter it would not override Object.equals(). Therefore we need to cast the parameter to an Account before extracting its accountNumber to compare them with those of the current object.

One additional complication concerns how objects are stored in sets. To do this Java uses a hashcode based on the objects name. However two accounts with the same accountNumber should generate the same hashcode even if the object name is different. To make certain this happens we need to override the hashCode() method so that a hashcode is generated using the accountNumber rather than the object name (just as we needed to override the equals() method). We can ensure that the hashcode generated is based on the account number by overriding this method as shown below.…

```
/**
 * Override hashCode() inherited from Object
 * @return hashcode based on accountNumber
 */
public int hashCode()
{
      return accountNumber.hashCode();
}
```

The simplest way to redefine hashCode for an object is to join together the instance values which are to define equality as a single string and then take the hashcode of that string. In this case equality is defined purely by the accountNumber which must be unique.

It looks a little strange, but we can use the hashCode() method on this String even though we are overriding the hashCode() method for objects of type Account.

hashCode() is guaranteed to produce the same hash code for the same String. Occasionally the **same** hash code may be produced for **different** key values, but that is not a problem.

Download free eBooks at bookboon.com

By overriding equals() and hashCode() methods Java will prevent objects with duplicate data (in this case with duplicate account numbers) from being added to sets.

Activity 3

The code below will find a String in a set of Strings (called stringSet). Amend this so this will work find a Publication in a set of Publications (called publicationSet).

```
public void findString(String pStr)
{
        boolean found;

        found = stringSet.contains(pStr);

        if (found)
        {
                System.out.println("Element " + pStr + " found in set");
        }
        else
        {
                System.out.println("Element " + pStr + " NOT found in set");
        }
}
```



Click on the ad to read more

Download free eBooks at bookboon.com

Feedback 3

```
public void findPublication(Publication pPub)
{
        boolean found;

        found = publicationSet.contains(pPub);

        if (found)
        {
                System.out.println("Element " + pPub + " found in set");
        }
        else
        {
                System.out.println("Element " + pPub + " NOT found in set");
        }
}
```

Activity 4

Look at the code in the feedback to Activity 3 and answer the following questions.

1) Could the code above be used to store a collection of books?
2) Could it store a combination of books and magazines?
3) If a book was found in the set what would the following line of code do?

System.out.println("Element " + pPub + " found in set");

Feedback 4

1) Yes
2) Yes
3) pPub would invoke the toString() method on the publication. The JRE would determine at run time that this was in fact a book and assuming the toString() method had been overridden for book, to return the title and author, this would make up part of the message displayed.

Download free eBooks at bookboon.com

7.12 Summary

The Java 'Collections Framework' provides ready-made interfaces and implementations for storing collections of objects. This almost completely makes the use of arrays redundant.

There are 'untyped' collections, and as of JDK 5.0 'typed' collections also.

Collection interfaces include List, Set and Map, each defining appropriate operations.

Collection implementations include ArrayList, HashSet and HashMap which are implementations of the List, Set and Map interfaces respectively.

Special attention is required when defining objects to be stored in Sets (or as keys in Maps) to define the meaning of 'duplicate'. For these we need to override the methods equals() and hashCode() methods inherited from Object.

While we have not been able to provide a detailed discussion of collection in this chapter the case study, in Chapter 11, will demonstrate the use of Maps and Sets for storing our own objects (not just Strings). The code for this will be available to download and inspect if required.

SIMPLY CLEVER
ŠKODA


Do you like cars? Would you like to be a part of a successful brand?
As a constructer at ŠKODA AUTO you will put great things in motion. Things that will ease everyday lives of people all around Send us your CV. We will give it an entirely new new dimension.
Send us your CV on www.employerforlife.com
Click on the ad to read more

Download free eBooks at bookboon.com

8 Java Development Tools

Introduction

This chapter will introduce the reader to various development tools that support the development of large scale Java systems (e.g. Eclipse and NetBeans). The importance of the API documentation and JavaDoc tool will also be demonstrated.

Objectives

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

- Understand the roles of the JDK and JRE
- Investigate professional development environments (Eclipse and NetBeans)
- Understand the importance of the Javadoc tool and the value of embedding Javadoc comments within your code.
- Write Javadoc comments and generate automatic documentation for your programs.

This chapter consists of thirteen sections:-

1) Software Implementation
2) The JRE
3) Java Programs
4) The JDK
5) Eclipse
6) Eclipse architecture
7) Eclipse Features
8) NetBeans
9) Developing Graphical Interfaces Using NetBeans
10) Applying Layout Managers Using NetBeans
11) Adding Action Listeners
12) The Javadoc Tool
13) Summary

Download free eBooks at bookboon.com

8.1 Software Implementation

Before a computer can complete useful tasks for us (e.g. check the spelling in our documents) software needs to be written and implemented on the machine it will run on. Software implementation involves the writing of program source code and preparation for execution on a particular machine. Of course before the software is written it needs to be designed and at some point it needs to be tested. There are many iterative lifecycles to support the process of design, implementation and testing that involve multiple implementation phases. Of particular concern here are the three long established approaches to getting source code to execute on a particular machine…

- compilation into machine-language object code
- direct execution of source code by 'interpreter' program
- compilation into intermediate object code which is then interpreted by run-time system

Implementing Java programs involves compiling the source code (Java) into intermediate object code which is then interpreted by a run-time system called the JRE. This approach has some advantages and disadvantages and it is worth comparing these three options in order to appreciate the implications for the Java developer.

Compilation

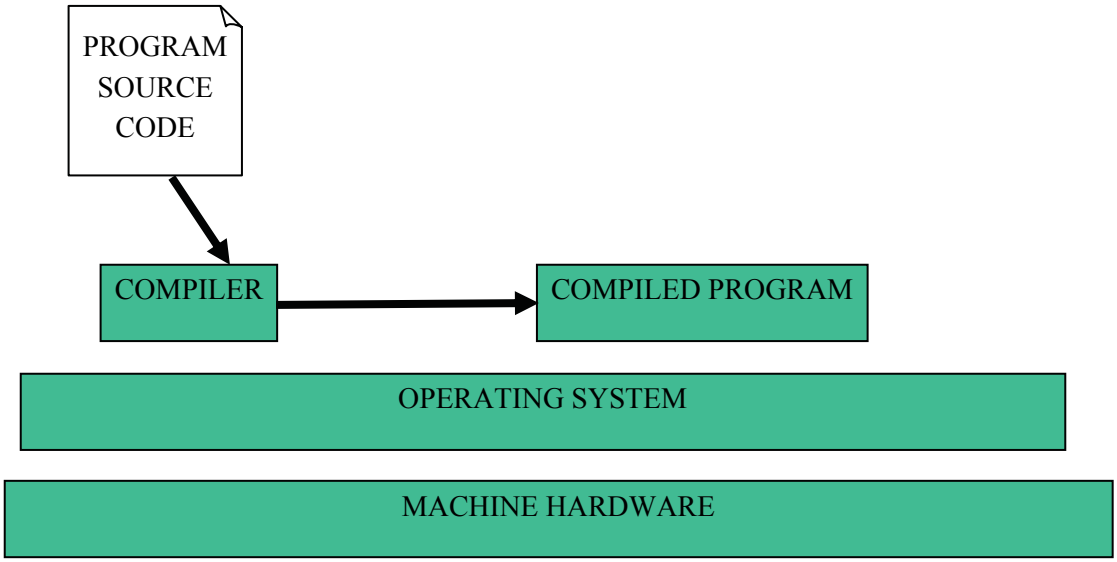
The compiler translates the source code into machine code for the relevant hardware/OS combination.

Strictly speaking there are two stages: compilation of program units (usually files), followed by 'linking' when the complete executable program is put together including the separate program units and relevant library code etc.

The compiled program then runs as a 'native' application for that platform.

This is the oldest model, used by early languages like Fortran and Cobol, and many modern ones like C++. It allows fast execution speeds but requires re-compilation of the program each time the code is changed.

Download free eBooks at bookboon.com

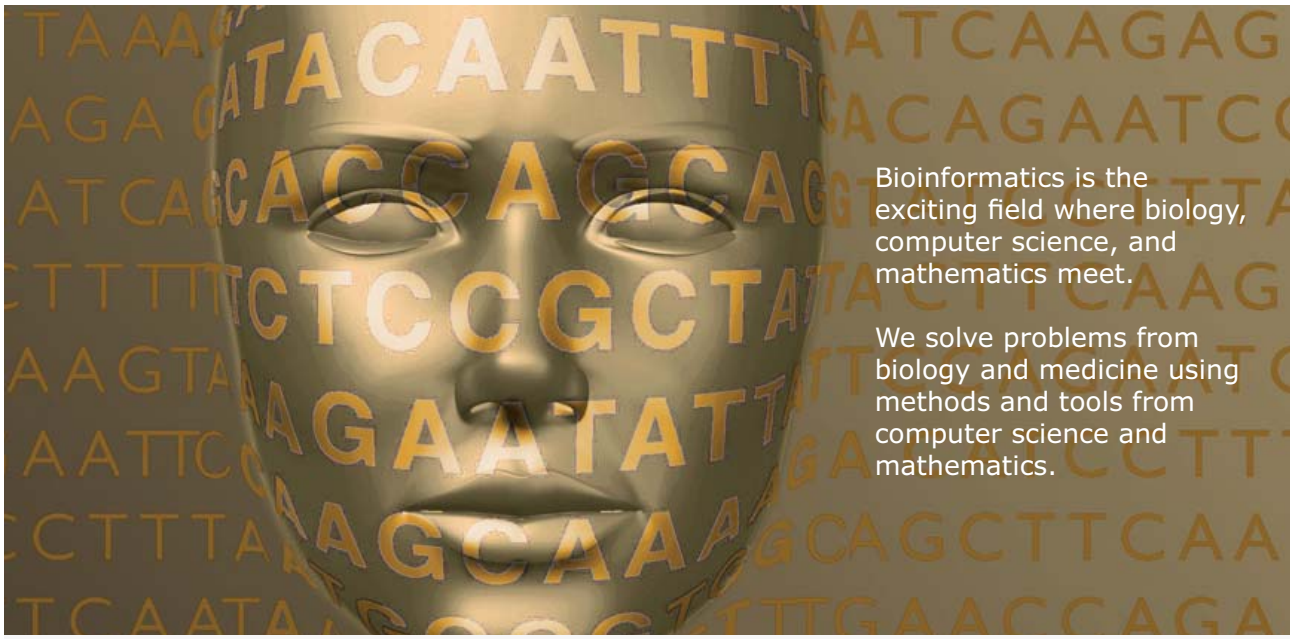


Interpretation

Here the source code is not translated into machine code. Instead an interpreter reads the source code and performs the actions it specifies.

UPPSALA UNIVERSITET

Develop the tools we need for Life Science
Masters Degree in Bioinformatics



Read more about this and our other international masters degree programmes at www.uu.se/master

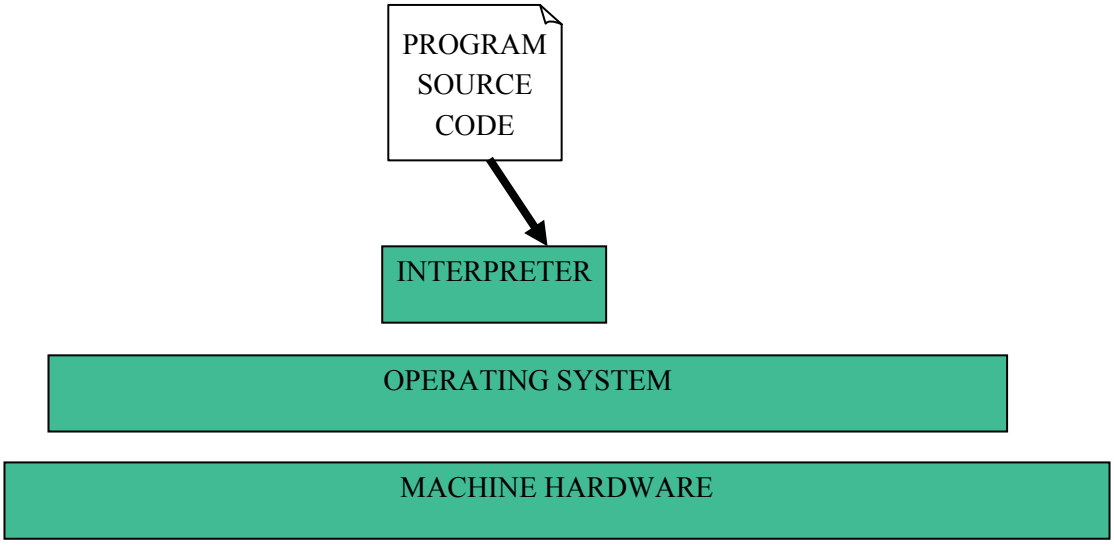
Click on the ad to read more

Download free eBooks at bookboon.com

We can say that the interpreter is like a 'virtual machine' whose machine language is the source code language.

No re-compilation is required after changing the code, but the interpretation process inflicts a significant impact on execution speed.

Scripting languages tend to work in this way.



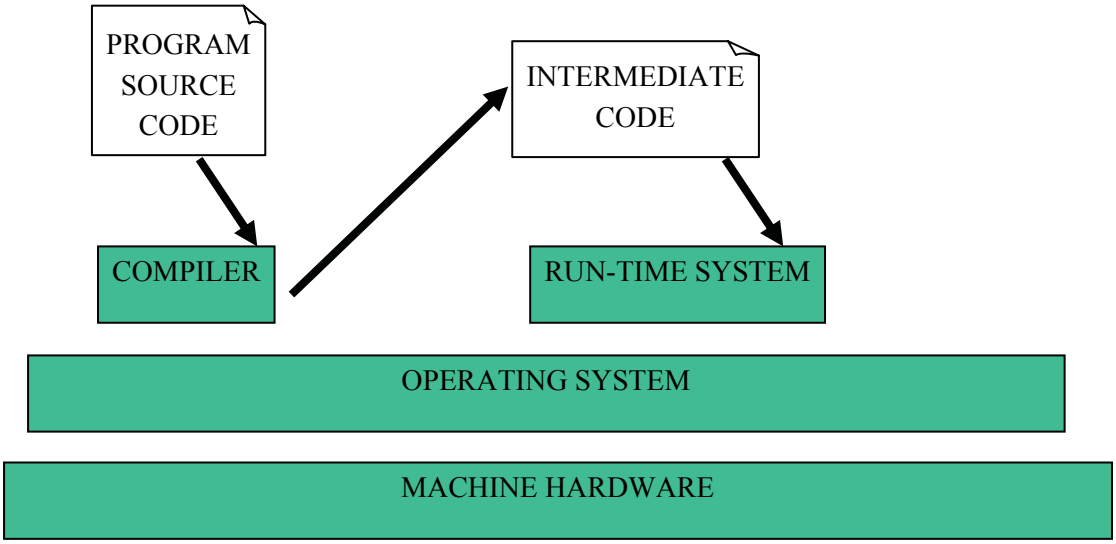
Intermediate Code

This model is a hybrid between the previous two.

Compilation takes place to convert the source code into a more efficient intermediate representation which can be executed by a 'run-time system' (again a sort of 'virtual machine') more quickly that direct interpretation of the source code. However, the use of an intermediate code which is then executed by run-time system software allows the compilation process to be independent of the OS/hardware platform, i.e. the same intermediate code should run on different platforms so long as an appropriate run-time system is available for each platform.

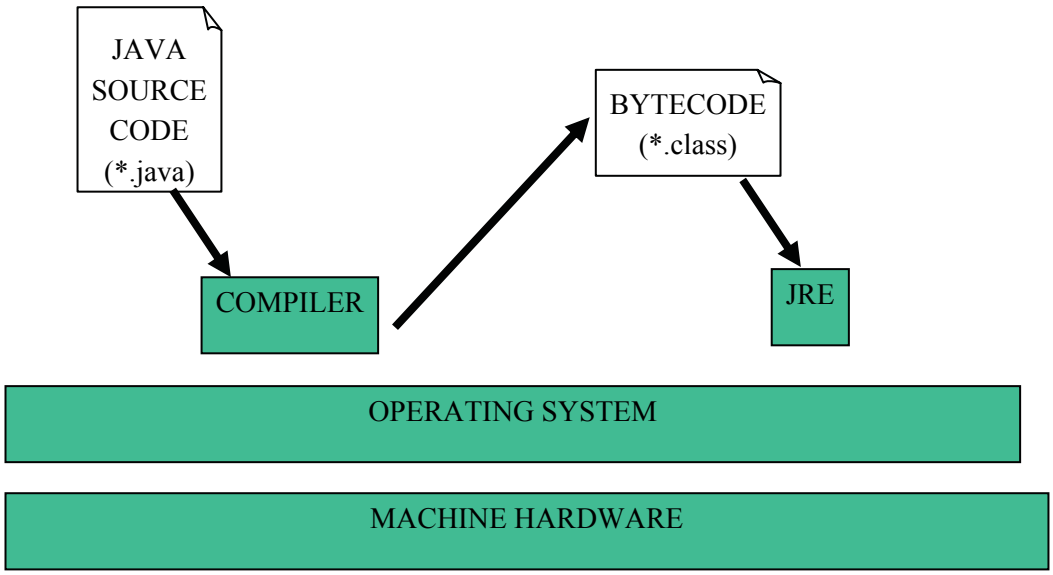
This approach is long-established (e.g. in Pascal from the early 1970s) and is how Java works.

Download free eBooks at bookboon.com



8.2 The JRE

To run Java programs we must first generate intermediate code (called bytecode) using a compiler available as part of the Java Development Kit (JDK) – see section 8.4 below.



A version of the Java Runtime Environment (JRE), which incorporates a Java Virtual machine (VM), is required to execute the bytecode and the Java library packages. Thus a JRE must be present on any machine which is to run Java programs.

The Java bytecode is standard and platform independent and as JRE's have been created for most computing devices (including PC's, laptops, mobile devices, mobile phones, internet devices etc) this makes Java programs highly portable.

Download free eBooks at bookboon.com

8.3 Java Programs

Whatever mode of execution is employed, programmers can work with a variety of tools to create source code. Options include the use of simple discrete tools (e.g. editor, compiler, interpreter) invoked manually as required or alternatively the use of an Integrated Development Environment (IDE) which incorporates these implementation tools behind a seamless interface. Still more sophisticated CASE (Computer Aided Software Engineering) tools which integrate the implementation process with other phases of the development cycle – such software could take UML class diagrams and generate classes and method stubs automatically saving some of the effort required to write the Java code.

When writing java programs each class (or interface) in a Java program has its own **name.java** file containing the source code.

These are processed by the compiler to produce **name.class** files containing the corresponding bytecode.

To actually run as an application, one of the classes must contain a main() method with the signature shown above.

UNIVERSITY OF COPENHAGEN



Copenhagen Master of Excellence



Copenhagen Master of Excellence are two-year master degrees taught in English at one of Europe's leading universities

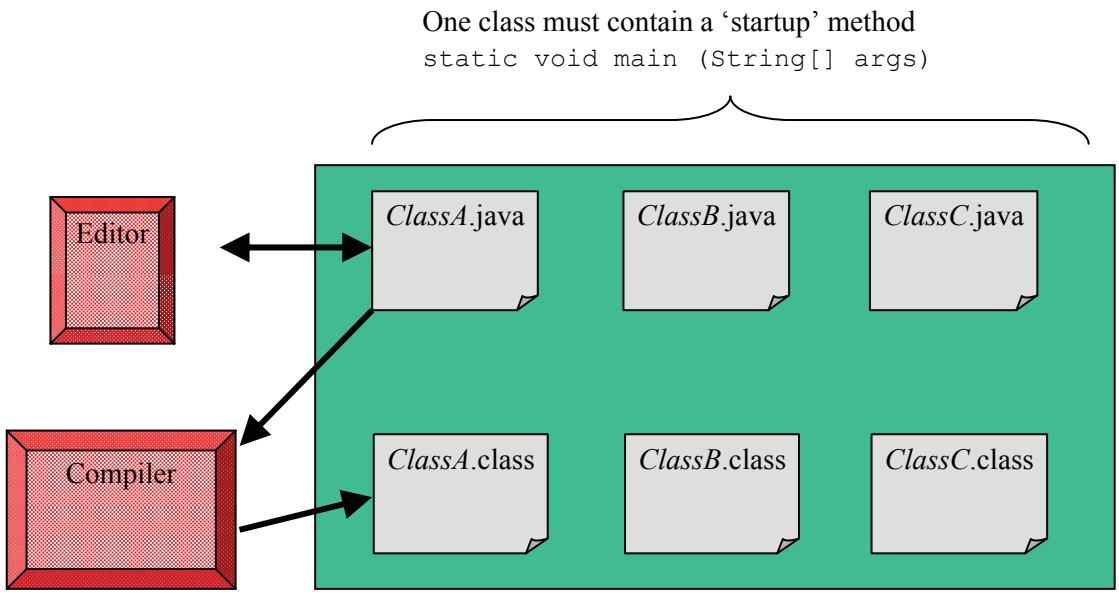
Come to Copenhagen - *and aspire!*

Apply now at
www.come.ku.dk



Click on the ad to read more

Download free eBooks at bookboon.com



8.4 The JDK

To develop Java programs you must first install the Java Development Kit (JDK). This was developed by Sun and is available freely from the internet via http://java.sun.com/. Prior to version 5.0 (or 1.5) this was known as the Java Software Development Kit (SDK).

A Java IDE's, e.g. Eclipse or NetBeans, sits on top of' the JDK to add the IDE features – these may include interface development tools, code debugging tools, testing tools and refactoring tools (more on these later). When using an IDE it is easy to forget that much of the functionality is in fact part of the JDK and when the IDE is asked to compile a program it is infact just passing on this request to the JDK sitting underneath.

We can use the JDK directly from the command line to compile and run Java programs though mostly it is easier to use the additional facilities offered by an IDE.

Somewhat confusingly the current version of Java is known both as 6.0, 1.6 and even 1.6.0. These supposedly have subtly different meanings – don't worry about it!

There are many tools in the JDK. A description of each of these is available from http://java.sun.com/javase/downloads/ and following the links for Documentation, APIs and JDK Programmer guides.

The two most important basic tools are:

javac – the java compiler
java – the java program launcher (that runs the VM)

Download free eBooks at bookboon.com

To compile MyProg.java we type

```
javac MyProg.java
```

If successful this will create MyProg.class

To run Myprog (assuming it has a main() method) we type

```
java MyProg
```

Another, extremely useful tool, is javadoc – this uses comments in Java source code to generate automatic documentation for programs.

8.5 Eclipse

Moving to an 'industrial strength' IDE is an important stepping stone in your progress as a software developer, like riding a bicycle without stabilisers for the first time. With some practice you will soon find it offers lots of helpful and time-saving facilities that you will not want to work without again…

Eclipse is a flexible and extensible IDE platform. It was first developed by IBM but is now open source and can be downloaded from the Eclipse Foundation at www.eclipse.org.

Eclipse was itself written in Java and like all Java programs it needs a JRE to work but using this it will therefore run on a large range of platforms. It comes with Java Development Tools (JDT) installed by default but also has extensions and additional tools to support the development of programs in other languages.

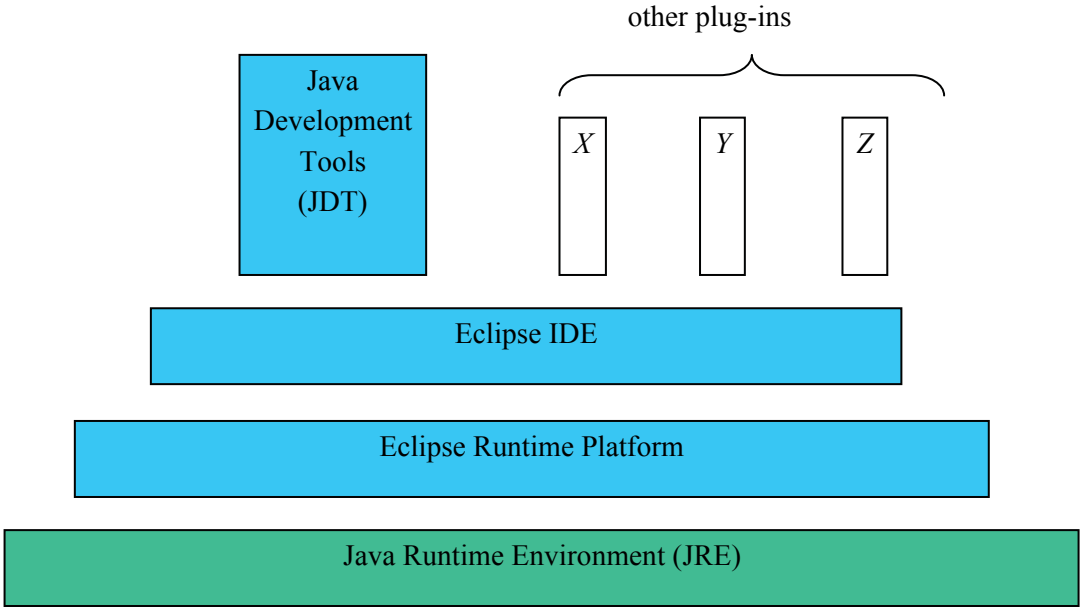
The Eclipse 'runtime platform' is sufficiently flexible that it can even be used as a framework for the development of applications unrelated to IDEs.

8.6 Eclipse Architecture

Because Eclipse relies on the JRE it is intrinsically multi-platform (Windows, Linux, Mac etc.)

The elements shown in blue below are part of the basic Eclipse download.

Download free eBooks at bookboon.com



Lots of other plug-ins are available to support additional tools (style-checking, testing, GUI design etc.) and languages (including C/C++).

Download .zip file from www.eclipse.org


Brain power
By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.
Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.
By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations.
Therefore we need the best employees who can meet this challenge!
The Power of Knowledge Engineering
Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge
SKF
Click on the ad to read more

Download free eBooks at bookboon.com

Like any IDE Eclipse requires an up-to-date JDK is installed as it will use this to compile programs and generate Java documentation.

8.7 Eclipse Features

Eclipse provides a range of features offering powerful support for a programmer wishing to develop large Java programs. These include:-

- code completion and help facilities,
- a code formatter which can automatically adjust a huge variety of code format aspects,
- powerful facilities to support program development including refactoring (see Chapter 10 section 4) and automatic testing facilities (see Chapter 10 section 5).

8.8 NetBeans

NetBeans is another very powerful IDE, originally developed and distributed by Sun Microsystems, NetBeans was made open source in 2000 and is extensively supported with plug-ins and video tutorials developed by the open source community.

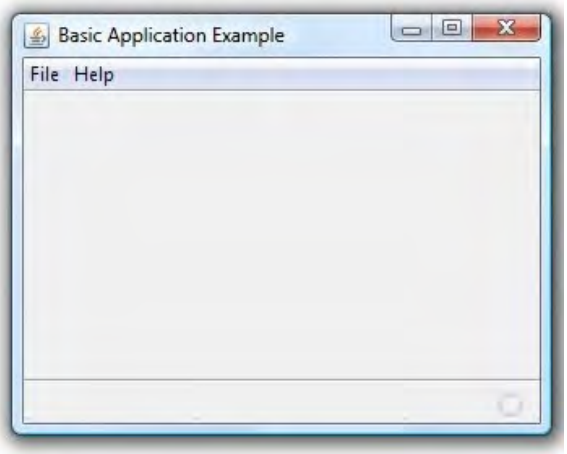
NetBbeans can be downloaded for free from www.netbeans.org

NetBeans has many features that have been developed to aid novice programmers and save time. These include…

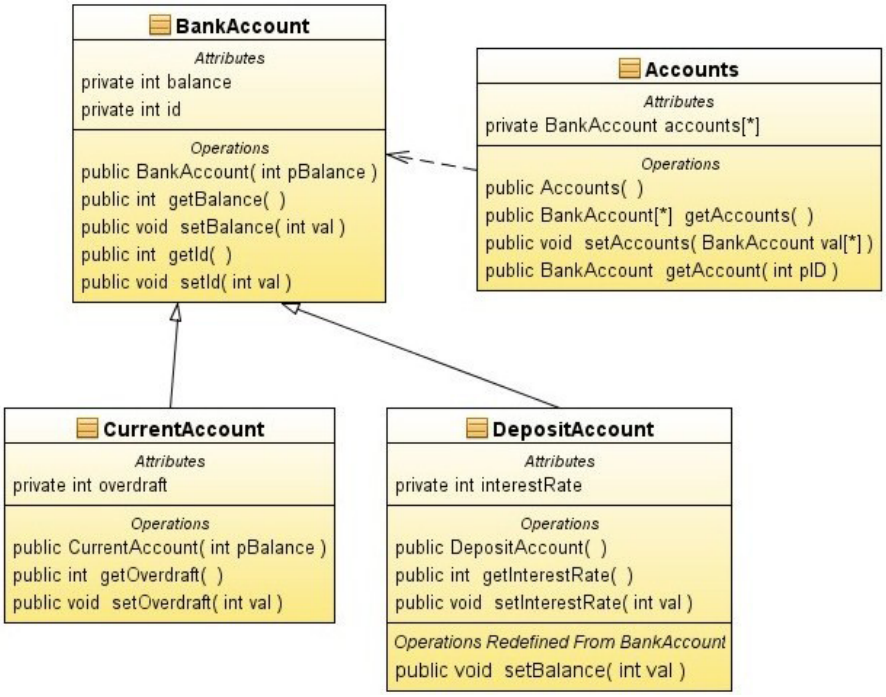
- A quick start guide, on line tutorials and online videos (covering among other topics an introduction to the IDE, the Javascript debugger, and PHP support).
- Help features with automatic pop up windows highlighting relevant parts of the Java API,
- formatting and debugging facilities
- code completion features that will auto insert for loops, try catch blocks and constructors (among others). These code completion facilities do far more than cut and paste sample blocks of text – for example when inserting a constructor for a subclass the code to call a super-constructor is automatically added and relevant parameters are defined for the sub-constructor and passed to the super-constructor.
- Project management features that monitor changes to a project that allow the project to be reverted to an earlier state.

Netbeans also includes templates for a range of Java applications including windows, enterprise, web and mobile applications. It even has templates for C++/PHP/ Ruby and database applications. Thus with just a few button clicks Netbeans will create a simple Java application with a main method that invokes a blank GUI as shown below…

Download free eBooks at bookboon.com



Additionally one of the many plug-ins available provides support for the creation of UML diagrams. While this tool does have its limitations it does not just support the creation of the diagrams but provides facilities that you may expect from a CASE tool such as automatic code generation (from a diagram) and reverse engineering (i.e. analysing current code to generate a UML diagram from the code).

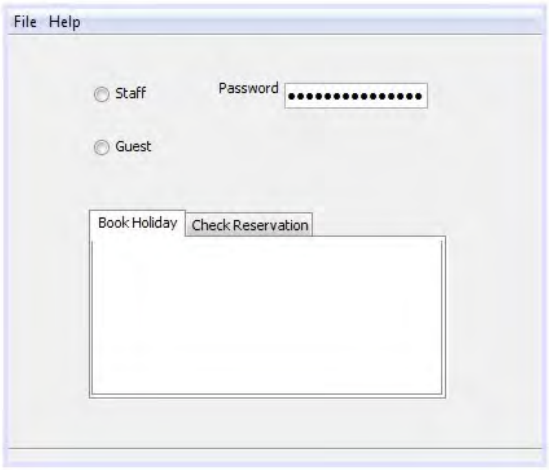


Thus given the diagram above NetBeans would create four classes, three of which would automatically be placed in an inheritance hierarchy. Method stubs would be created for each of the specified methods and while the code inside each method would still need to be written the constructor for CurrentAccount would automatically invoke the constructor for Bank Account passing the parameter pBalance upwards.

Download free eBooks at bookboon.com

8.9 Developing Graphical Interfaces Using NetBeans

NetBeans has a visual tool to help with the development of graphical user interfaces. This tools allows a window to be created and a range of objects to be dropped onto it including panels, tabbed panes, scrolled panes, buttons, radio buttons, check boxes, combo boxes, password fields, progress bars, and trees.



Trust and responsibility

NNE and Pharmaplan have joined forces to create NNE Pharmaplan, the world's leading engineering and consultancy company focused entirely on the pharma and biotech industries.

Inés Aréizaga Esteva (Spain), 25 years old
Education: Chemical Engineer

– You have to be proactive and open-minded as a newcomer and make it clear to your colleagues what you are able to cope. The pharmaceutical field is new to me. But busy as they are, most of my colleagues find the time to teach me, and they also trust me. Even though it was a bit hard at first, I can feel over time that I am beginning to be taken seriously and that my contribution is appreciated.



NNE Pharmaplan is the world's leading engineering and consultancy company focused entirely on the pharma and biotech industries. We employ more than 1500 people worldwide and offer global reach and local knowledge along with our all-encompassing list of services. **nnepharmaplan.com**

nne pharmaplan®

Click on the ad to read more

Download free eBooks at bookboon.com

8.10 Applying Layout Managers Using NetBeans

When designing a graphical user interface it is 'normal' for objects to be placed on a window in precise positions. Interfaces are thus displayed exactly how they are designed. For this to work well an interface designer usually has some idea of the size of the display they are designing an interface for. Most PC games are designed to run on window of 14"–21". Most mobile games are designed to work on much smaller screens.

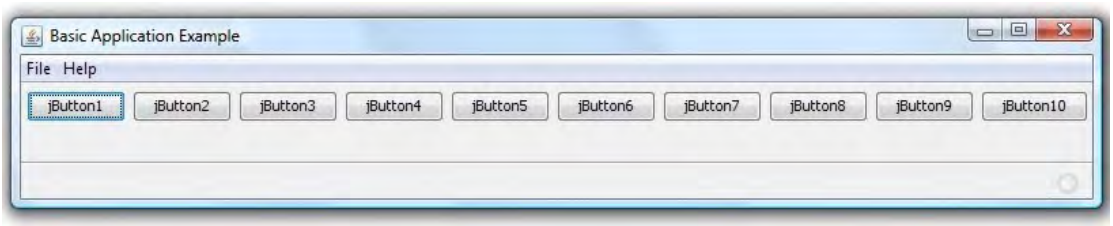
However Java programs are expected to be platform independent – one of the strengths of Java is that code written in Java will work on a PC, laptop or small mobile device irrespective of the hardware or operating systems these devices use (as long as they all have a JRE). Thus the same interface should work on a large visual display unit or small mobile screen. For this reason when developing interfaces in Java it is usual to give Java some control over deciding exactly how / where to display objects. Thus, at run time, Java can reconfigure a window to fit whatever device is being used to run the program. This has benefits as the display will be reconfigured automatically to fit whatever device is being used to run the program but this flexibility comes at a cost. As some control is given to Java the interface designer cannot be certain exactly how the interface will look though they can give Java some 'instructions' on how an interface should be displayed.

Layout Managers are 'objects' that define how an interface should be displayed and it is normal to create a layout manager and assign it to a 'container' object such as a window frame.

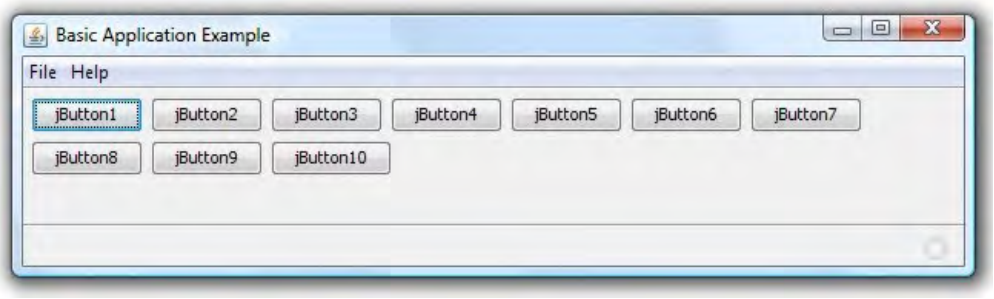
There are a range of common layout managers – two of the most common being flow layout and grid layout. Using flow layout Java will arrange objects in a row and objects will automatically flow onto another row if the window is not big enough to hold each object on one row.

Flow Layout In Action

Two windows are shown below with ten buttons on each – in the second figure Java places some of the buttons on a second row as the window has been resized and the object will not fit on one line.



Download free eBooks at bookboon.com

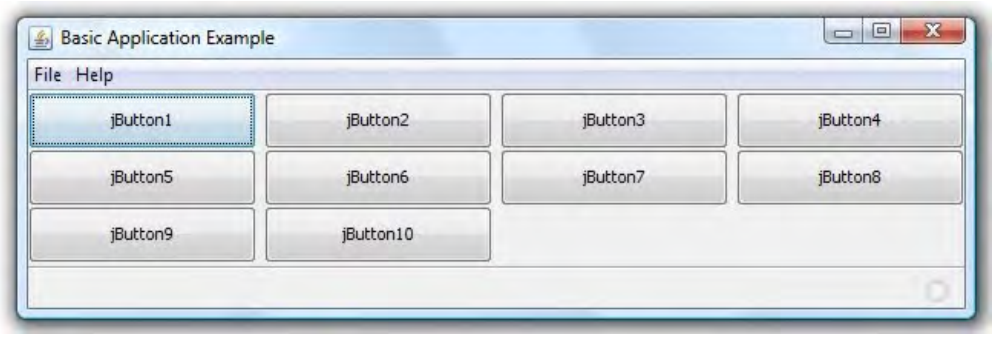


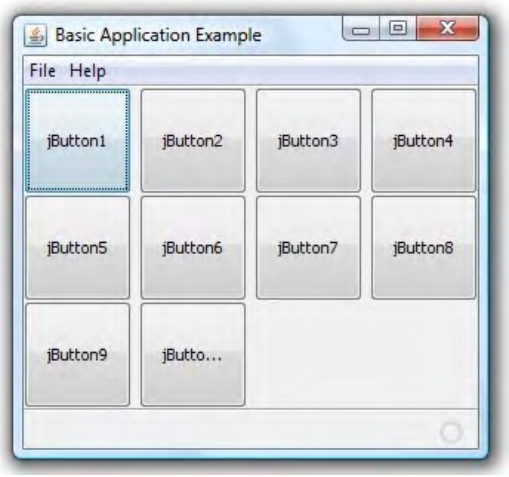
A flow manager can be created with various properties to define the justification of the object (centre, left or right) and to define the horizontal and vertical space between objects.

Grid Layout In Action

Another very common layout manager is gird layout. When applying objects in a grid a number of rows can be specified. Java will work out how many columns it needs to use in order to place the required number of objects in that number of rows. The objects will be resized to fit the window. Java will shrink or enlarge the objects to fit the window – even if this means that not all of the text can be shown.

The two figures below show the same 10 buttons being displayed on a grid with three rows.





Download free eBooks at bookboon.com

Note in the second of these the buttons have been resized to fit a different window – in doing so there is not enough space to display all of the text for button ten.

Layout managers can be used in layers and applied to any container object Thus we can create a grid where one element contains multiple objects arranged in a flow.

By using layout managers we can create displays that are flexible and can be displayed on a range of display equipment thus enabling our programs to be platform independent – however this does mean we lose some of the fine control in the design.

Creating a layout managers and setting their properties using NetBeans is very easy. Right clicking on a container object will bring up a context sensitive menu one option of which will be to apply a layout manager. Selecting this option will display the list of layout managers possible. Once a layout manager has been selected a window will appear to allow its properties to be specified.

This e-book
is made with
SetaPDF

SETASIGN



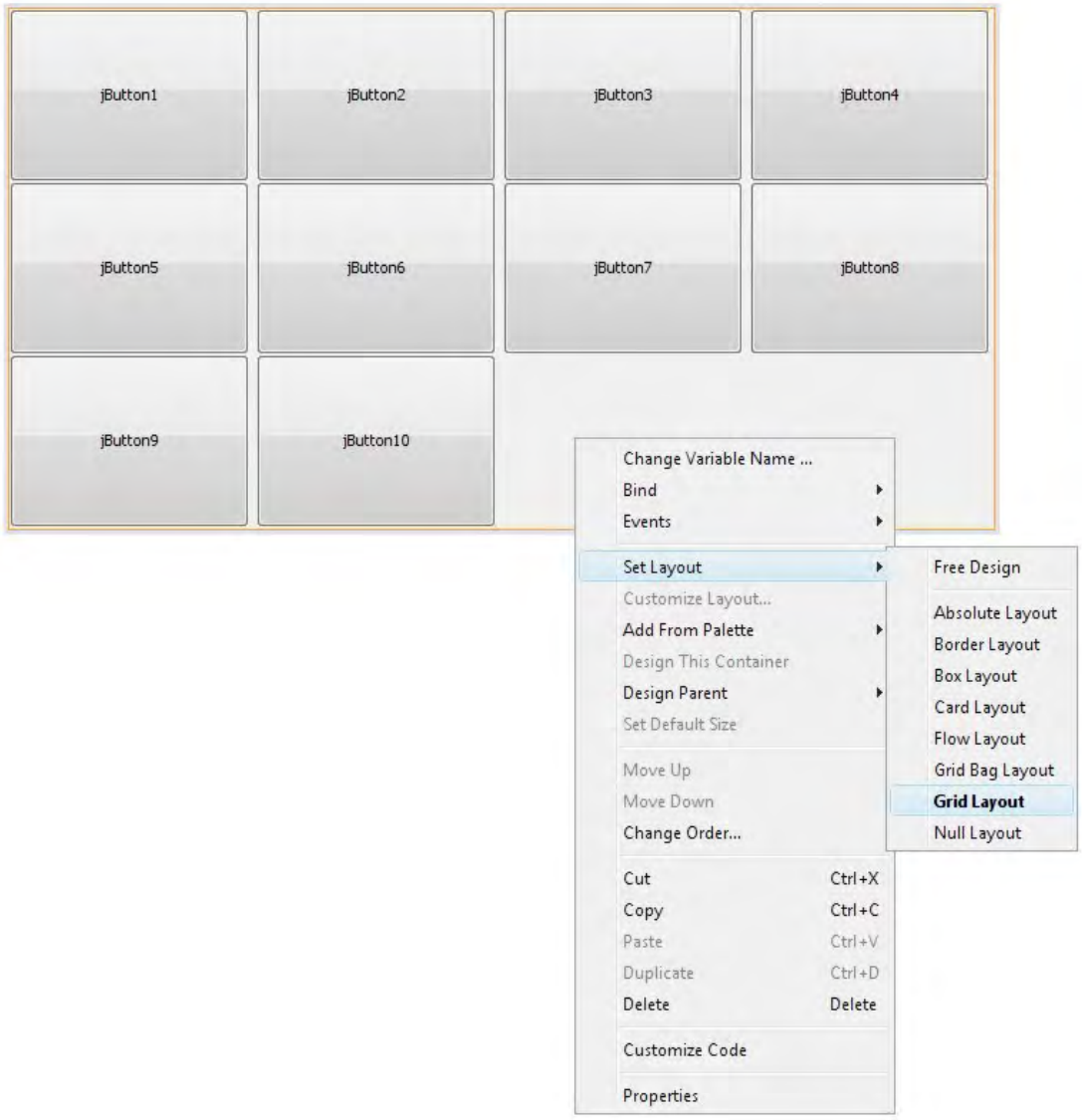
PDF components for **PHP** developers

www.setasign.com



Click on the ad to read more

Download free eBooks at bookboon.com



8.11 Adding Action Listeners

The final step in creating a working interface is to create Action Listeners and apply them to object on the interface. Action listeners are objects that listen for user actions and respond accordingly – thus when a 'calculate' button is pushed a program will calculate and display some results.

Action listeners can be created for all objects on an interface and for a range of actions for each object. Though many objects, such as labels, will be not be required to respond to a users interactions.

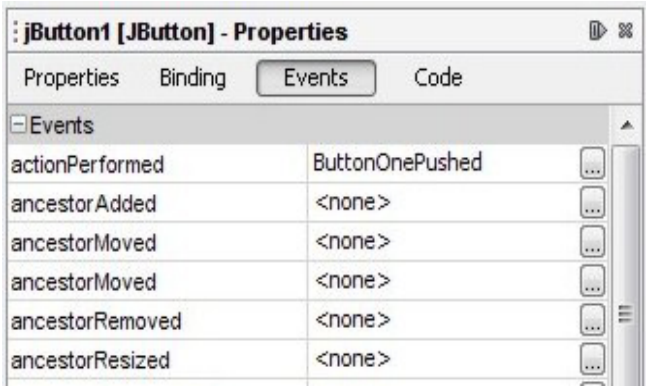
Creating an action listener in NetBeans is very easy.

Download free eBooks at bookboon.com

When any object is selected, in design mode, a properties window is displayed. Selecting the events tab will allow a range of actions to be created for this object. Once one has been selected an 'Action Listener' stub will be written. Code will automatically be written to create an object of this class and assign this to the object selected in the design.

All that remains is for the programmer to write the code to define the response to the users action.

In the example below an event has been created to listen for button1 being pushed.




```
private void ButtonOnePushed(java.awt.event.ActionEvent evt) {
    System.out.println("button 1 pushed");

}
```

8.12 The Javadoc Tool

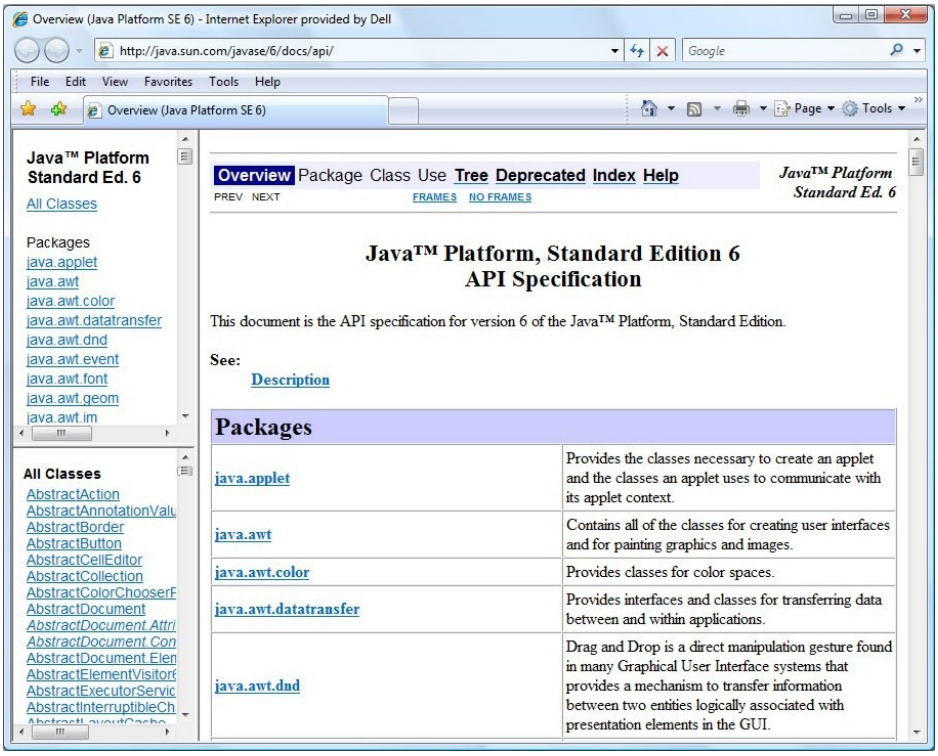
One particularly useful tool within the JDK is 'javadoc'.

Programmers for many years have been burdened with the tedious and time consuming task of writing documentation. Some of this documentation is intended for users and explain what a program does and how to use it. Other documentation is intended for future programmers who will need to amend and adapt the program so that its functionality will change as the needs of an organisation change. These programmers need to know what the program does and how it is structured. They need to know:-

- what packages it contains
- what classes exist in each of these packages and what these classes do,
- what methods exist in each class and for each of these methods
    - what does the method do
    - what parameters does it require
    - what, if any, value is returned.

Download free eBooks at bookboon.com

The javadoc tool won't produce a user guide but it will provide a technical description of the program. The tool analyses *.java source files and produces documentation as a set of web pages (HTML files) in the same format as API documentation on the Sun website.



This website provides, as a set of indexed web pages, essential details of the Java language specification including all packages, classes, methods and method parameters and return values. This extremely useful information was not generated by hand but generated using the javadoc tool.

The same tool can produce the same style of documentation doe any Java program. However to do this it relies on properly formatted (and informative!) javadoc-style comments in source files, including tags such as @author, @param etc.

Because this documentation is generated automatically, at the push of a button, it saves programmers from a tedious, time consuming and error prone task. Furthermore the documentation can be updated whenever a program is changed.

This tool does require a programmer to add meaningful comments to code as it is developed and poor attention to commenting of source code will result in poor javadoc output! On the other hand, if the commenting is done properly then the reference documentation is produced 'for free'!

Javadoc comments should be added at the start of every class using the standard tags @author and @ version to give an overview of that class in the following format.

Download free eBooks at bookboon.com

```
/**********************************
 * A description of the class
 *
 * @author name of author
 * @version details of version (and date).
**********************************/
```

A similar comment should be provided for every method using the @param and @return to describe each parameter and to describe the value returned. The details of each parameter, starting with the name of the parameter, should be provided on separate lines as shown below.

```
/**********************************
 * A description of the method
 *
 * @param nameOf1stParam description of first parameter
 * @param nameOf2ndParam description of second parameter
 * @return description of value returned
**********************************/
```

FOSS

Sharp Minds - Bright Ideas!

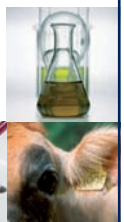
Employees at FOSS Analytical A/S are living proof of the company value - First - using new inventions to make dedicated solutions for our customers. With sharp minds and cross functional teamwork, we constantly strive to develop new unique products - Would you like to join our team?

FOSS works diligently with innovation and development as basis for its growth. It is reflected in the fact that more than 200 of the 1200 employees in FOSS work with Research & Development in Scandinavia and USA. Engineers at FOSS work in production, development and marketing, within a wide range of different fields, i.e. Chemistry, Electronics, Mechanics, Software, Optics, Microbiology, Chemometrics.

We offer
A challenging job in an international and innovative company that is leading in its field. You will get the opportunity to work with the most advanced technology together with highly skilled colleagues.

Read more about FOSS at www.foss.dk - or go directly to our student site www.foss.dk/sharpminds where you can learn more about your possibilities of working together with us on projects, your thesis etc.

The Family owned FOSS group is the world leader as supplier of dedicated, high-tech analytical solutions which measure and control the quality and production of agricultural, food, pharmaceutical and chemical products. Main activities are initiated from Denmark, Sweden and USA with headquarters domiciled in Hillerød, DK. The products are marketed globally by 23 sales companies and an extensive net of distributors. In line with the corevalue to be 'First', the company intends to expand its market position.





Dedicated Analytical Solutions
FOSS
Slangerupgade 69
3400 Hillerød
Tel. +45 70103370
www.foss.dk

Click on the ad to read more

Download free eBooks at bookboon.com

Activity 1

The method below takes two integer numbers and adds them together. This value is returned by the method. Write a javadoc comment, using appropriate tags, to describe this method.

```
public int add(int pNumber1, int pNumber2) {
        return (pNumber1 + pNumber2);
}
```

Feedback 1

```
/**********************************
 * This method adds two integer numbers.
 *
 * @param pNumber1 first number to be added
 * @param pNumber2 second number to be added
 * @return sum of the two numbers
**********************************/
```

The javadoc tool cannot analyse the description of a method to determine if this description provides an accurate summary of that method. However the tool does do far more than cut and paste a programmers comments onto a web document. One thing this tool does is analyse method signatures and compare this with the tags in the comments to check for logical errors. If a method requires three parameters but the comment only describes two then this error will be detected and reported to the programmer.

"warning – @param argument "pNumber3" is not a parameter name."

By analysing the code the tool also provides additional information about the class, the methods that have been inherited and the methods that have been overridden.

The reference documentation produced is for programmers using the class(es) concerned, it does not include comments within methods intended for programmers editing the class source code in maintenance work.

The generated documentation can be placed in a subfolder and can be inspected by opening the file index.html in a web browser.

'javadoc' will only document classes and members which are public or protected because only these are available to other programmers. Items which are private or for which no access qualifier is given are omitted.

Download free eBooks at bookboon.com

8.13 Summary

There are various ways in which source code in a programming language can be executed. Java uses an intermediate language called 'bytecode' and a main method is required.

We can go 'back to basics' creating Java programs using a text editor and the tools of the Java Development Kit but the use of a professional IDE, sitting on top of the JDK, can offer additional facilities to support programmers.

Specialist tools are available for aspects of development such as GUI design, diagramming and documentation but some IDEs go beyond the basics and provide some of these facilities in built into the IDE.

Eclipse and NetBeans are two open source powerful IDEs which support the development of Java programs as standard. These are both available as a free download.

These tools can initially seem daunting as they provides extensive support for professional development including code formatting and refactoring though online help does exist and both tools provide some level of automatic code generate, e.g. the main method, to make the job of the programmer easier.

The Javadoc tool is an extremely useful and timesaving device but it does require the programmer to inserting meaningful Javadoc style comments into the code (for all classes and all public methods).



Click on the ad to read more

Download free eBooks at bookboon.com

9 Creating And Using Exceptions

Introduction

If the reader has written Java programs that make use of the file handling facilities they will have written code to catch exceptions i.e. they will have used try\catch blocks. This chapter explains the importance of creating your own exceptions and shows how to do this by extending the Exception Class and using the 'Throw' mechanism.

Objectives

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

- Appreciate the importance of exceptions
- Understand how to create your own exceptions and
- Understand how to throw these exceptions.

This chapter consists of six sections:-

1) Understanding the Importance of Exceptions
2) Kinds of Exception
3) Extending the Exception Class
4) Throwing Exceptions
5) Catching Exceptions
6) Summary

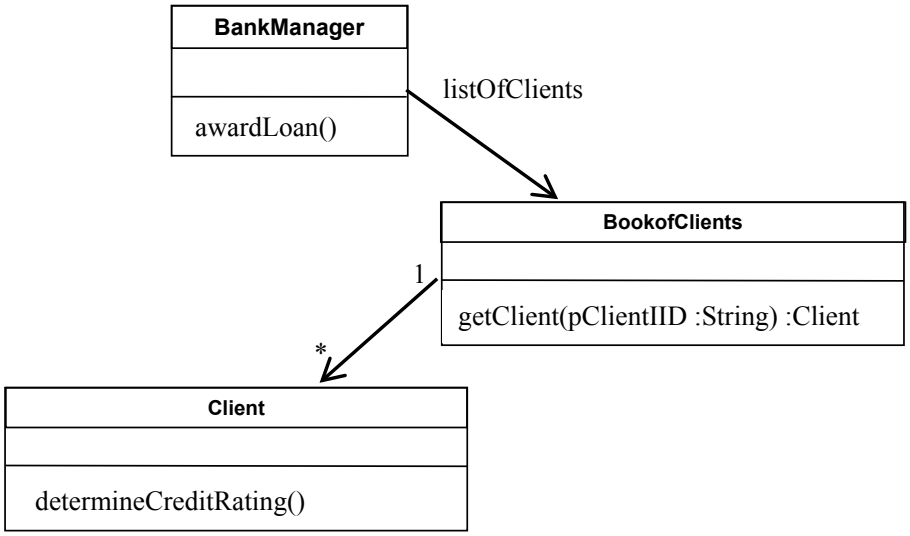
9.1 Understanding the Importance of Exceptions

Exception handling is a critical part of writing Java programs. The authors of the file handling classes within the Java language knew this and created routines that made use of Java exception handling facilities – but are these really important? and do these facilities matter to programmers who write their own applications using Java?

Download free eBooks at bookboon.com

Activity 1

Imagine part of a banking program made up of three classes, and three methods as shown below…



The system shown above is driven by the BankManager class. The awardLoan() method is invoked, either via the interface or from another method. This method is intended to accept or reject a loan application.

The BookofClients class maintains a set of account holders…people are added to this set if they open an account and of course they can be removed. However the only method of interest to us is the getClient() method. This method requires a string parameter (a client ID) and either returns a client object (if the client has an account at that bank) – or returns NULL (if the client does not exist).

The Client class has only one method of interest determineCreditRating(). This method is invoked to determine a clients credit rating – this is used by the BankManager class to decide if a loan should be approved or not.

Considering the scenario above look at the snippet of code below …

```
Client c = listOfClients.getClient(clientID) ;
c.determineCreditRating();
```

This fragment of code would exist in the awardLoan() method. Firstly it would invoke the getClient() method, passing a client ID as a parameter. This method would return the appropriate client object (assuming of course that a client with this ID exists) which is then stored in a local variable 'c'. Having obtained a client the determineCreditRating() method would be invoked on this client.

Look at these two lines of code. Can you identify any potential problems with them?

Download free eBooks at bookboon.com

Feedback 1

If a client with the specified ID exists this code above will work. However if a client does not exist with the specified ID the getClient() method will return NULL.

The second line of code would then cause a run time error (specifically a NullPointerException) as it tries to invoke the determineCreditRating() method on a non existent client and the program would crash at this point.

Activity 2

Consider the following amendment to this code and decide if this would fix the problem.

```
Client c = listOfClients.getClient(pClientID) ;
If (c !=NULL) {
c.determineCreditRating();
}
```

Feedback 2

If the code was amended to allow for the possible NULL value returned it would work – however this protection is insecure as it relies on the programmer to spot this potential critical error.

When writing the getClient() method the author was fully aware that a client may not be found and in this case decided to return a NULL value. However this relies on every programmer who ever uses this method to recognise and protect against this eventuality.



Click on the ad to read more
Download free eBooks at bookboon.com

If any programmer using this method failed to protect against a NULL return then their program could crash – potentially in this case losing the bank large sums of money. Of course in other applications, such as an aircraft control system, a program crash could have life threatening results.

A more secure programming method is required to ensure that that a potential crash situation is always dealt with!

Such a mechanism exists – it is a mechanism called 'exceptions'.

By using this mechanism we can ensure that other programmers who use our code will be alerted to potential crash situations and the compiler will ensure that these programmers deal with the 'issue'. Thus we can ensure that no such situation is 'forgotten'. How they are dealt with remains a choice with a programmer who uses our methods but the compiler will ensure that they at least recognise a potential crash situation.

In the situation above rather than return a NULL value the getClient() method should generate an exception. By generating an exception the Java compiler will ensure that this situation is dealt with.

9.2 Kinds of Exception

In order to generate meaningful exceptions we need to extend the Exception classes built into the Java language – there are two of these (normal exceptions and run time exceptions).

Subclasses of java.lang.Exception are used for anticipated problems which need to be managed. They must be declared in the originating method's **throws** clause and a call to method must be placed in **try/catch** block.

Subclasses of java.lang.RuntimeException are used for situations which lead to runtime failure and where it may not be possible to take any sensible remedial actions. They do not need to be declared in **throws** clause and a call need not be in **try/catch** block (but can be).

Thus we have the choice as to whether the Java compiler should force us to explicitly deal with a particular kind of exception.

Exception subclasses are appropriate for things which we know might go wrong and where we can take sensible recovery action – e.g. IO errors.

RuntimeException subclasses are appropriate for things which should not happen at all and where there is probably nothing we can do to recover the situation, e.g. an out of memory error or discovering that the system is in an inconsistent state which should never be able to arise.

Download free eBooks at bookboon.com

9.3 Extending the Exception Class

When writing our own methods we should look for potential failure situations (e.g. value that cannot be returned, errors that may occur in calculation etc). When a potential error occurs we should generate an 'Exception' object i.e. an object of the Exception class. However it is best to first define a subclass of the general Exception i.e. to create a specialised class and throw an object of this subtype.

A new exception is just like any new class in this case it is a subclass of **java.lang.Exception**

In the case above an error could occur if no client is found with a specified ID. Therefore we could create a new exception class called 'UnknownClientException'.

The parameter to the constructor for the Exception requires a Sting thus the constructor for UnknownClientException also requires a String. This string is used to give details of the problem that may generate an exception.

The code to create this new class is given below…

```
import java.lang.Exception;

/*******************************************************
 * Exception thrown when attempting to get an non-existent client ID
 *
 * @author Simon Kendal
 * @version 1.0 (11th July 2009)
 *******************************************************/
class UnknownClientException extends Exception
{
      /**
      * Constructor
      *
      * @param pMessage description of exception
      */
      UnknownClientException (String pMessage)
      {
            super(pMessage);
      }
}
```

In some respects this looks rather odd. Here we are creating a subclass of Exception but our subclass does not contain any new methods – nor does it override any existing methods. Thus its functionality is identical to the superclass – however it is a subtype with a meaningful and descriptive name.

If sublasses of Exception did not exist we would only be able to catch the most general type of exception i.e an Exception object. Thus we would only be able to write a catch block that would catch every single type of exception.

Download free eBooks at bookboon.com

Having defined a subclass we instead have a choice… a) we could define a catch block to catch objects of the general type 'Exception' i.e. it would catch ALL exceptions or b) we could define a catch block that would catch UnknownClientExceptions but would ignore other types of exception.

By looking at the online API we can see that many predefined subclass of exception already exist. There are many of these including:-

- IOException
  - CharConversionException
  - EOFException
  - FileNotFoundException
  - ObjectStreamException
- NullPointerException
- PrinterException
- SQLexception

Thus we could write a catch block that would react to any type of exception, or we could limited it to input \ output exceptions or we could be even more specific and limit it to FileNotFound exceptions.



Click on the ad to read more

Download free eBooks at bookboon.com

9.4 Throwing Exceptions

Having defined our own exception we must then instruct the getClient() method to throw this exception (assuming a client has not been found with the specified ID).

To do this we must first tell the compiler that this class may generate an exception – the complier will then ensure that any future programmer who makes use of this method catches this exception.

To tell the compiler this method throws an exception we add the following statement to the methods signature 'throws UnknownClientException'.

```
public Client getClient(String pClientID)
                                        throws UnknownClientException
```

We must create a new instance of the UnknownClientException class and apply the throw keyword to this newly created object.

We use the keyword 'throw' to throw an exception at the appropriate point within the body of the method.

```
if (foundClient != null)
{
 return foundClient;
}
else
{
 throw new UnknownClientException("BookOfClients.getClient():
                                        unknown client ID:" + pClientID);
}
```

In the example above if a client is found the method will return the client object. However it will no longer return a NULL value. Instead if a client has not been found the constructor for UnknownClientException is invoked, using 'new'. This constructor requires a String parameter – and the string we are passing here is an error message that is trying to be informative and helpful. The message is specifying:-

- the class which generated the exception (i.e. BookOfClients),
- the method within this class (i.e. getClient()),
- some text which explains what caused the exception and
- the value of the parameter for which a client could not be found.

By defining an UnkownClientException and using the throw clause in the header of the getClient() method we are providing a warning to all methods calling this one that an exception may be thrown. This enables the Java compiler to make sure a try/catch block is provided where required.

By doing this we are preventing potentially critical errors from going unnoticed!

Download free eBooks at bookboon.com

9.5 Catching Exceptions

Having specified to the compiler that this method may generate an exception we are forcing other programmers to protect against critical errors by placing calls to this method within a try / catch block. The code in the try block will be terminated if an exception is generated and the code in the catch block will be initiated instead.

Thus in the example above the awardLoan() method can decide what to do if no client with the specified ID is found…

```
try
{
      Client c = listOfClients.getClient(clientID) ;
      c.determineCreditRating();

      // add code to award or reject a loan application based on this
         credit rating

}

catch (UnknownClientException uce)
{
      System.out.println("INTERNAL ERROR IN BankManager.awardLoan()\n"
                                    + "Exception details: " + uce);

}
```

Now, instead of crashing with a NullPointerException if the client ID is not found, the UnknownClientException we have deliberately thrown will be handled by the Java Virtual Machine which will terminate the code in the **try** clause and invoke the code in the **catch** clause, which in this case will display a message warning the user about the problem.

9.6 Summary

Java exceptions provide a mechanism to deal with abnormal situations which occur during program execution.

By making use of the Java exception mechanism we are protecting against potentially life threatening program failure.

The exception mechanism will ensure other programmers who use our methods recognise and deal with error situations.

When exceptions are generated the code in a catch block will be initiated – this code could take remedial action or terminate the program generating an appropriate error message. In either case at least the program doesn't just 'stop'.

Download free eBooks at bookboon.com

10 Agile Programming

While a detailed discussion of Agile development methods is beyond the scope of this book, this chapter will explain the claims made by proponents of agile programming methods and show how modern IDE's (such as Eclipse) offers tools to support agile programming. In particular we will examine refactoring and JUnit testing tools.

Objectives

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

- Appreciate the importance of the claims made for Agile development
- Understand the need for refactoring and how a modern IDE supports this
- Understand the advantages of Unit testing and
- Understand how to create JUnit test cases.
- Understand the claims made for Test driven Development



Click on the ad to read more

Download free eBooks at bookboon.com

This chapter consists of fifteen sections:-

1) Agile Approaches
2) Refactoring
3) Examples of Refactoring
4) Support for Refactoring
5) Unit Testing
6) Automated Unit Testing
7) Regression Testing
8) JUnit
9) Examples of Assertions
10) Several Test Examples
11) Running Tests
12) Test Driven Development (TDD)
13) TDD Cycles
14) Claims for TDD
15) Summary

10.1 Agile Approaches

Traditional development approaches emphasized detailed advance planning and a linear progression through the software lifecycle *Code late, get it right first time* (Really??)

Recent 'agile' development approaches emphasize flexible cyclic development with the system evolving towards a solution *Code early, fix and improve it as you go along.*

This is a very hot topic in Software Engineering circles at the moment, and as with all such developments it has its share of zealots and ideologues!

Is the waterfall lifecycle model ***really*** successful in enabling large, complex projects to proceed from start to finish without ever looking back? Advocates of agile approaches contend that these better fit the reality of software development.

However agile programming requires tools that will enable software to change and evolve. Two specific tools provided by modern IDEs that support agile programming are refactoring and testing tools.

10.2 Refactoring

A key element of 'agile' approaches is 'refactoring'. This technique accepts that some early design and implementation decisions will turn out to be poor, or at least less than ideal.

Download free eBooks at bookboon.com

Refactoring means changing a system to improve its design and implementation quality without altering its functionality (in traditional development such work was termed 'preventive maintenance').

Although the idea of structurally improving existing software is not new, the difference is as follows. In traditional development it was seen as a remedial action taken when the software design quality had degraded, usually as a result of phases of functional modification and extension. In agile methodologies refactoring is regarded as a natural healthy part of the development process.

10.3 Examples of Refactoring

During the development process a programmer may realise that a variable within a program has been badly named. However changing this is not a trivial task.

Changing a local variable will only require changes in one particular method – if a variable with the same name exists in a different method this will not require changing.

Alternatively changing a public class variable could require changes throughout the system (one reason why the use of public variables are not encouraged).

Thus implementing a seemingly trivial change requires an understanding of the consequences of that change.

Other more complex changes may also be required. These include..

- Renaming an identifier everywhere it occurs
- Moving a method from one class to another
- Splitting out code from one method into a separate method
- Changing the parameter list of a method
- Rearranging the position of class members in an inheritance hierarchy

10.4 Support for Refactoring

Even the simplest refactoring operation, e.g. renaming a class, method, or variable, requires careful analysis to make sure all the necessary changes are made consistently.

Eclipse provides sophisticated automatic support for this activity.

Download free eBooks at bookboon.com

Don't confuse this with a simple text editor find/replace – Eclipse understands the Java syntax and works intelligently…e.g. if you have local variables with the same name in two different methods and rename one of them, Eclipse knows that the other is a different variable and does not change it. However if you rename a public instance variable this may require changes in other classes and even in other packages as methods from these classes may access and use this variable.

Rearranging existing classes within a package structure is also a refactoring activity

Eclipse makes this very easy to do. As a system design evolves, just as we may sometimes decide a class has become too large and complex and decide to split it into two or more separate classes, so we might decide to split a package. Less often we might merge packages or move classes between existing packages.

Eclipse will allow us to drag a class from one package to another. When this happens the package statement at the start of this class is changed and import statements within other classes are automatically adjusted to reflect the fact that the class concerned now resides in a new package.

The screen shot below shows the refactoring options provided by the Eclipse IDE.

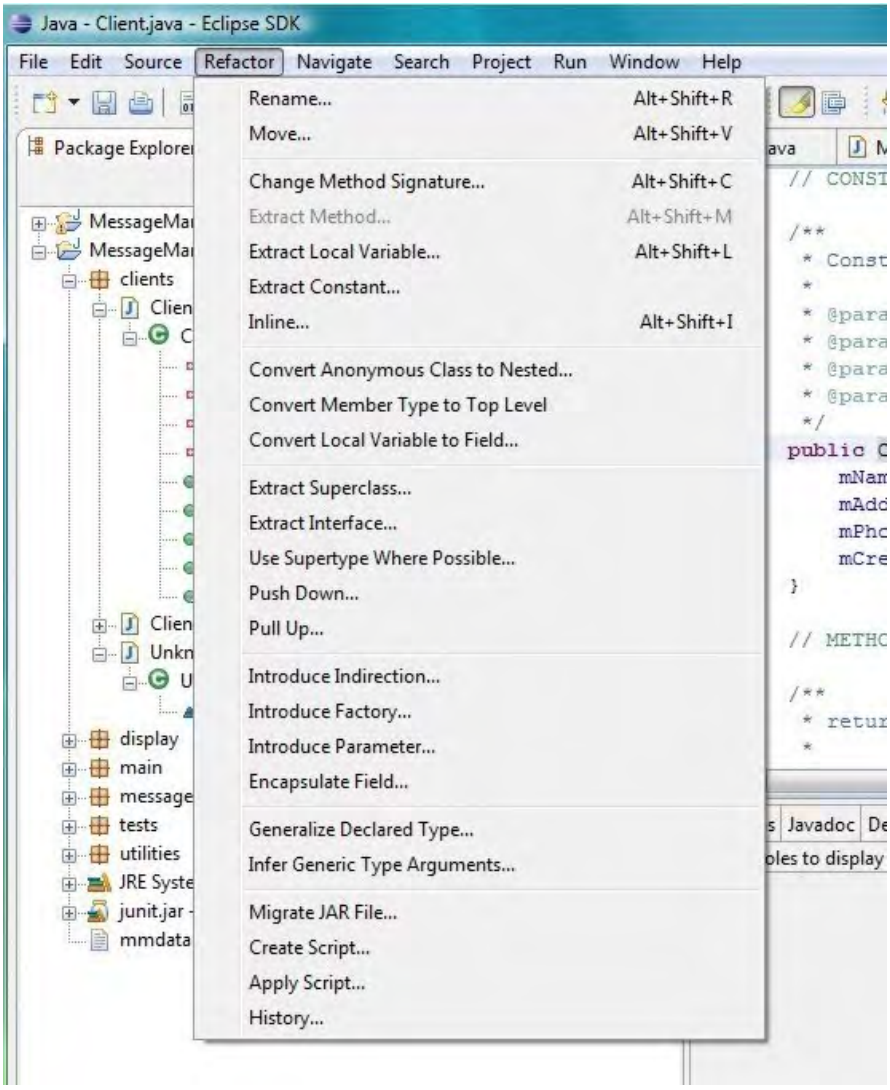
I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com



Click on the ad to read more

Download free eBooks at bookboon.com



Another essential tool provided by modern IDEs are automated testing tools.

10.5 Unit Testing

Testing the individual methods of a class in isolation from their eventual context within the system under construction is known as Unit Testing

This testing is generally 'wrapped into' the implementation process:

- Write class
- Write tests
- Run tests, debugging as necessary

Download free eBooks at bookboon.com

10.6 Automated Unit Testing

Tools and frameworks are available to automate the unit testing process. Using such a tool generally requires a little more effort than running tests once manually. However the main benefit arises from the ability to re-run the tests as often as desired just by pushing a button.

This is a great aid to the 'regression testing' which must be undertaken whenever previously tested code is modified.

Regression testing was developed long before agile methods were proposed and automated unit testing supports this. However automated unit testing also supports Test Driven Development processes and these play a major role in agile software development methods.

We will explore the Test Driven Development processes within this chapter but before doing so we will first explore conventional regression testing and the support offered for this via JUnit a testing framework within Java.

10.7 Regression Testing

Regression testing is required as a software is adapted to meet changing business needs. By running regression tests we want to ensure that changes to the code to not 'break' existing functionality. To do this as we write classes we must also write test cases that demonstrate these classes work. Thus we follow a process of…

- Write class
- Write tests
- Run tests, debugging as necessary
- Write more classes
- …

As we decide its necessary to change some of the earlier classes (or classes which they depend on) due to bugs, changing user requirements, or refactoring we need to re-run all previous tests to check they still pass. Without regression testing any modification of existing code is extremely hazardous!

As we regularly need to re-run sets of test cases it is helpful, and hugely timesaving, to have automated testing facilities such as those that exist within Java.

Download free eBooks at bookboon.com

10.8 JUnit

JUnit is a very widely used unit testing framework for Java. It has been developed as a suite of open-source classes, and is integrated into popular Java IDEs such as Eclipse and NetBeans.

While JUnit provides an automated testing framework we still need to set up the test cases – however once these have been set up a thousand test cases can be run at the push of a button – and the same set of test cases can be re-run every time a program is amended.

JUnit test classes are constructed as subclasses of junit.framework.TestCase

The correct behaviour of code being tested is checked using assert…() methods which must be true for the test to pass

Currently version 3.8 and version 4 of JUnit are widely used but there are significant differences between them. Here we are using the more widely established Junit 3.8

10.9 Examples of Assertions

When setting up test cases we make assertions. An assertion is a statement which should be true if the code has functioned correctly. Example of assertion include…

- assertTrue(…)
- assertFalse(…)
- assertEquals(…)
- assertNotEquals(…)
- assertNull(…)
- assertNotNull(…)
- fail()

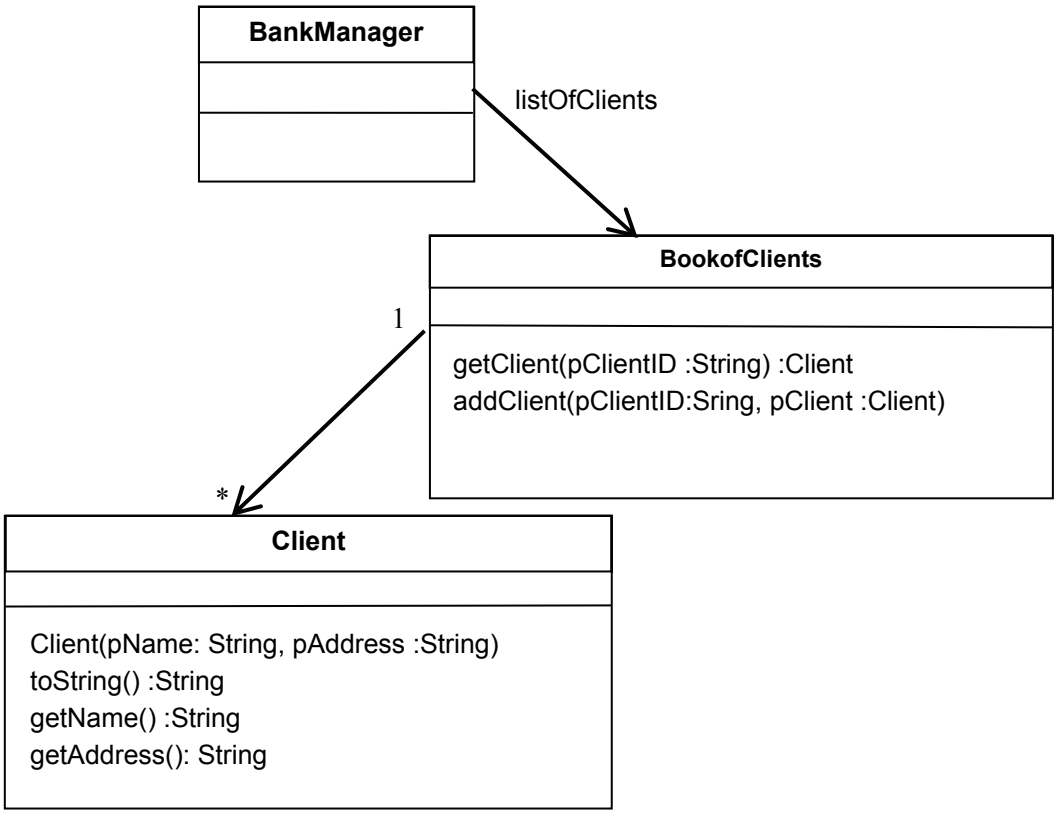
assertEquals() and fail() will be adequate for many testing purposes, as we will see here.

fail() indicates that having reached this line means the test has failed! (We will see an example).

10.10 Several Test Examples

To illustrate the JUnit testing framework we will create three test cases to test the functionality of a BankManagment system as represented below:-

Download free eBooks at bookboon.com





Click on the ad to read more

Download free eBooks at bookboon.com

In this system a BankManager class maintains a 'map' of clients. Client objects can be added by the addClient() method which requires an ID for that client and the client object to be added. Clients can be retrieved via the getClient() method which requires a ClientID as a parameter and returns a client object (if one exists) or generates an exception (if a client with the specified ID does not exist).

The Client class has a constructor that requires the name of the client and the clients address and provides associated accessor methods.

We will specifically test the ability to…

- Add a client to the BookofClients
- Trying to lookup a non-existent client
- The string representation yielded by the toString() method of the Client class.

This is of course only a small fraction of the test cases which would be needed to thoroughly demonstrate the correct operation of all classes and all methods within the system.

Testing adding a client

To test a client can be added we need to create a JUnit test case that

1) Creates a new empty BookofClients
2) Creates a new client and adds this to the BookofClients
3) Try's to retrieve a client with the same ID as the client just added (this of course should work) and finally
4) We need to test that the client retrieved has the same attributes as the client we just added – to ensure it was not corrupted.

The code for this is given below…

```
public void testAddClient() {
      BookofClients bofc = new BookofClients();
      Client c = new Client("Simon",
                            "No 5, Main St., Sunderland, SR1 0DD");
      Client c2 = null;
      bofc.addClient("SK001", c);
      try {
            c2 = bofc.getClient("SK001");
      } catch (UnknownClientException uce) {
            fail();
      }
      assertEquals("Simon", c2.getName());
      assertEquals("No 5, Main St., Sunderland, SR1 0DD",
                                                c2.getAddress());
}
```


Download free eBooks at bookboon.com

Note that a test method name always begins test…()

If an UnknownClientException is thrown the test fails because we should have found the "SK001" client which was added

If we successfully retrieve the client we assert that the value from getName() should equal "Simon" and the value from getAddress() should equal "No 5, Main St., Sunderland, SR1 0DD". If either of these is not the case then the test will fail.

Note that the two parameters of the assertEquals() method are first the **expected** value and second the **actual** value. Although the outcome of the assertion will be the same if these are reversed, it will result in JUnit giving misleading reports if the test fails.

If the test method ends without failing any assertions then the test is passed.

Note that for this test to work we need accessor methods getName() and getAddress() in the Client class. It is common practice in designing classes to create accessor methods which were intended purely to support unit testing even when these are not required in the actual system.

Testing for Unknown client

One test case we need to perform is to test that an exception is thrown if we try to retrieve a client that does not exist. To test this we create an new empty BookofClients and try to retrieve a client from this – any client!

The code for this is given below…

```
public void testGetUnknownClient() {
      BookofClients bofc = new BookofClients();
      try {
            bofc.getClient("SK001");
            fail();
      } catch (UnknownClientException uce) {
      }
}
```

In this case we expect an UnknownClientException to be thrown (as we haven't added the client!) and therefore if the line after the call to getClient() is reached the test fails.

If the exception is caught the fail() statement is skipped, the catch block has no action to take and the test then completes successfully.

Download free eBooks at bookboon.com

Testing the toString() method

One way of implicitly testing that ALL the attributes a client have been stored correctly is to test the toString() method returns the value expected. This is a little tricky because the format of the string must match **exactly** including every space, punctuation symbol, and newline.

The alternative however is to test the value returned by every accessor method.

Activity 1

Assuming the toString() method of the Client class is defined as below create a test method to test the value returned by the toString() method is as expected.

```
public String toString() {
       return ("Client name: " + mName + "\nAddress: " + mAddress);
}
```

Hint: Firstly create a new Client object with specified attributes then test the attributes of this object are as expected by using an assertion to test the string returned by the toString() method.

Feedback 1

One solution to this exercise is given below however clearly the attributes of the client created are arbitrary.

```
public void testClientToString() {
 Client c = new Client("Simon", "5 Main St., Sunderland");
 assertEquals("Client name: Simon\nAddress: 5 Main St.,
       Sunderland", c.toString() );
}
```

Here we assert that the string returned from c.toString() is equal to the string we are expecting. This is quite tricky because the format of the string must match **exactly** including every space, punctuation symbol, and newline.

This test is of course implicitly testing that ALL the attributes have been stored correctly which is useful.

10.11 Running Tests

Having designed a batch of test cases we need to set these up so that they can be run as often as required at the push of a button.

- To do this in Eclipse we
- create new package “tests”
- in “tests” create new JUnit test case “ClientTests”
- add in the three test methods
- Run As… JUnit test
- edit code to cause one of the tests to fail and observe result

Download free eBooks at bookboon.com

10.12 Test Driven Development (TDD)

Automated unit testing also supports Test Driven Development (TDD) which is a technique mainly associated with 'agile' development processes. This has become a hot topic in software engineering

The Test Driven Development approach is to

1) Write the tests (before writing the methods being tested).
2) Set up automated unit testing, which fails because the classes haven't yet been written!
3) Write the classes and methods so the tests pass

This reversal seems strange at first, but many eminent contributors to software engineering debates believe it is a powerful 'paradigm shift'

The task of teaching can be used as an analogy. Which of the following is simpler?

- Teach someone everything you know about a subject and then decide how to test their knowledge or
- Decide specifically what it is they need to learn (i.e. decide what to test) and then teach the person just what they need to know in order to pass that test.



Click on the ad to read more
Download free eBooks at bookboon.com

10.13 TDD Cycles

When undertaking test driven development the test will initially cause a **compilation** error as the method being tested doesn't exist!.

Creating a stub method enables the test to compile but it the test itself will fail because the actual functionality being tested has not been implemented in the method.

We then implement the correct functionality of the method so that the test succeeds.

For a complex method we might have several cycles of: write test, fail, implement functionality, pass, extend test, fail, extend functionality, pass… to build up the solution.

10.14 Claims for TDD

Among the advantages claimed for TDD are:

- testing becomes an intrinsic part of development rather than an often hurried afterthought.
- it encourages programmers to write simple code directly addressing the requirements
- a comprehensive suite of unit tests is compiled in parallel with the code development
- a rapid cycle of "write test, write code, run test", each for a small developmental increment, increases programmer confidence and productivity.

In conventional software lifecycles if a software project is running late financial pressures often result in the software being rushed to market not having been fully tested and debugged. With Test Driven Development this is not possible as the tests are written before the system has been implemented.

10.15 Summary

'Agile' development approaches emphasize flexible cyclic development with the system evolving towards a solution.

Unit testing is an important part of software engineering practice whatever style of development process is adopted.

An automated unit testing framework (e.g. JUnit) allows unit tests to be regularly repeated as system development progresses.

Test Driven Development reverses the normal sequence of code and test creation, and plays a major part in 'agile' approaches.

Download free eBooks at bookboon.com

11 Case Study

This chapter will bring together all of the previous chapters showing how these essential concepts work in practise through one example case study.

Objectives

By the end of this chapter you will see…

- how a problem description is turned into a UML model (as described in Chapter 6)
- an example of how typed collections can be effectively used, in particular you will see an example of the use of sets and maps (as described in Chapter 7)
- an example of polymorphism and see how this enables programs to be extended simply (as described in Chapter 4)
- a simple example of the use of inheritance and method overriding (as described in Chapter 3)
- several example of UML diagrams (as described in Chapter 2)
- finally you will see the use of the Javadoc tool (as described in Chapter 8).

The complete working application, developed as described throughout this chapter, is available to download for free as an exported Eclipse project.

Eclipse is freely available from www.eclipse.org.

This chapter consists of seventeen sections:-

1) The Problem
2) Preliminary Analysis
3) Further Analysis
4) Documenting the design using UML
5) Prototyping the Interface
6) Revising the Design to Accommodate Changing Requirements
7) Packaging the Classes
8) Programming the Message Classes
9) Programming the Client Classes
10) Creating and Handling UnknownClientException
11) Programming the Main classes
12) Programming the Interface
13) Using Test Driven Development and Extending the System

Download free eBooks at bookboon.com

14) Generating Javadoc
15) Running the System and Potential Compiler Warnings
16) The Finished System…
17) Summary

11.1 The Problem

User requirements analysis is a topic of significant importance to the software engineering community and totally outside the scope of this text. The purpose of this chapter is not to show how requirements are obtained but to show how a problem statement is modelled using OO principles and turned into a complete working system once requirements are gathered.

The problem for which we will design a solution is 'To develop a message management system for a scrolling display board belonging to a small seaside retailer.'

For the purpose of this exercise we will assume preliminary requirements analysis has been performed by interviewing the shop owner, and the workers who would use the system, and from this the following textual description has been generated:-



Click on the ad to read more

Download free eBooks at bookboon.com

Rory's Readables is a small shop on the seafront selling a range of convenience goods, especially books and magazines aimed at both the local and tourist trades. It has recently also become a ticket agency for various local entertainment and transport providers.

Rory plans to acquire an LCD message display board mounted above the shopfront which can show scrolling text messages. Rory intends to use this to advertise his own products and offers and also to provide a message display service for fee-paying clients (e.g. private sales, lost and found, staff required etc.)

Each client is given a unique ID string (e.g. "adams4") and has a name, an address, a phone number and an amount of credit in 'credit units'. A book of clients is maintained to which clients can be added and in which we can look up a client by their ID.

Each message is for a specific client and comprises the text to be displayed, the number of days for which it should be displayed and the cost of that message in units. No duplicate messages (i.e. the same text for the same client) are permitted.

A set of current messages is to be maintained: new messages can be added, the message set can be displayed on the display board, and at the end of each day a purge is performed – each message has its days remaining decremented and its client's credit reduced by the cost of the message, and any messages which have expired or whose client has no more credit are deleted from the message set.

The software is to be written before the display board is installed – therefore the connection to the board should be via a well-defined interface and a dummy display board implemented in software for testing purposes.

Given the description above this chapter describes how this problem may be analysed, modelled and a solution programmed – thus demonstrating the techniques discussed throughout this book.

11.2 Preliminary Analysis

To perform a preliminary analysis of these requirement as (described in Chapter 6) we must…

- List the nouns and verbs
- Identify things outside the scope of the system
- Identify the synonyms
- Identify the potential classes, attributes and methods
- Identify any common characteristics

From reading this description we can see that the first paragraph is purely contextual and does not describe anything specifically related to the software required. This has therefore been ignored.

Download free eBooks at bookboon.com

List of Nouns

From the remaining paragraphs we can list the following nouns or noun phrases:-

- LCD message display board
- shopfront
- scrolling text message
- client
- ID string
- name
- address
- phone number
- credit
- credit unit
- message
- day
- book of clients
- ID
- text
- number of days
- cost of message (units)
- set of current messages
- message set
- display board
- days remaining
- client's credit
- cost of message
- software
- connection
- interface
- dummy display board

List of Verbs

and the following verbs:-

- acquire
- mount
- show
- advertise
- give (a unique ID)
- display
- add (a client)
- look up
- permit (duplicates – NOT!)
- purge
- decrement
- reduce credit
- expire
- delete
- write (the software)
- install
- implement
- test

Outside Scope of System

By identifying things outside the scope of the system we simplify the problem…

- Nouns:
  - shopfront
  - software

- Verbs:
  - acquire, mount (the display board)
  - advertise
  - give (a unique ID)
  - write, install, implement, test (the software)

The shopfront is not part of the system and it is not a part of the system to acquire and mount the displayboard. The ID is assigned by the shop owner – not the system. And writing / installing the software is the job of the programmer – it is not part of the system itself.

Download free eBooks at bookboon.com

Synonyms

The following are synonyms:-

- Nouns:
  - LCD message display board = display board
  - scrolling text message = message
  - ID string = ID
  - credit units = client's credit = credit
  - set of current messages = message set
  - days = number of days = days remaining

- Verbs:
  - show = display

By identifying synonyms we avoid needless duplication and confusion in the system.



Click on the ad to read more

Download free eBooks at bookboon.com

Potential Classes, Attributes and Methods

Nouns that describe significant entities for which we can identify properties i.e. data and behaviour i.e. methods could become classes within the system. These include:-

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

Nouns that are would be better as attributes of a class rather than becoming a class themselves:-

- For a 'client':
  - ID
  - name
  - address
  - phone number
  - credit

- For a 'message':
  - text
  - days remaining
  - cost of message

Each of these *could* be modelled as a class (which Client or Message would have as an object attribute), but we decide that each of them is a sufficiently simple piece of information that there is no reason to do so – each one can be a simple attribute (instance variable) of a primitive type (e.g. int) or library class (e.g. String).

This is a **design judgement** – introducing classes for significant entities (Client, Message etc.) which have a set of information and behaviour belonging to them, but not overloading the design with trivially simple classes (e.g. credit which would just contain an 'int' instance variable together with a getter and setter!).

Download free eBooks at bookboon.com

Verbs describe candidate methods and we should be able to identify the classes these could belong to. For instance:-

- For a 'client':
  - decrease credit
- For a 'message':
  - decrement days

The other verbs describing potential methods should also be listed:-

- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

For each of these the associated class should be identified.

Common Characteristics

The final step in our preliminary analysis is to identify the common characteristics of classes and to identify if these classes should these be linked in an inheritance hierarchy or linked by an interface.

Looking at the list of candidate classes provided we can see that two classes that share common characteristics:-

- DisplayBoard
- DummyDisplayBoard

This either implies these classes should be linked within the system within an inheritance hierarchy or via 'an interface'(see section 4.5 Interfaces). In this case the clue is within the description "These will have a 'connection' to the rest of the system via a 'well-defined interface".

Ultimately our system should display messages in a real display board however it should first be tested on a dummy display board. For this to work the dummy board must implement the same methods as a real display board.

Download free eBooks at bookboon.com

Thus we should define a java 'interface'. No common code would exist between the two classes – hence why we are not putting these within an inheritance hierarchy. However the dummy board and the real display board should both implement the methods defined via a common interface. When our system is working we could replace the dummy board with the real board which implements the same methods. As the connection with the dummy board is via the interface changing the dummy board with the real display board should have no impact on our system.

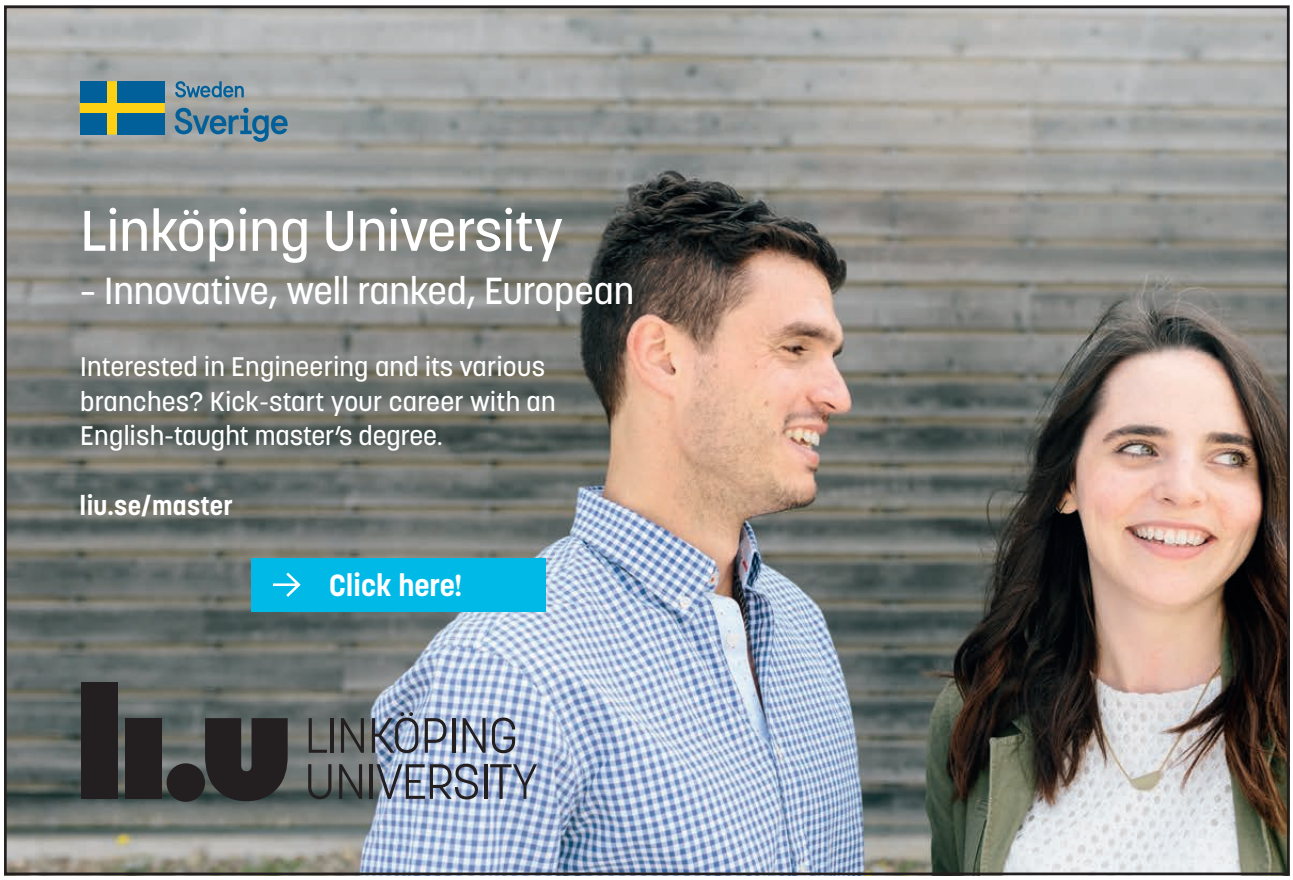
From our preliminary analysis of the description we have identified candidate classes, interfaces, methods and attributes. The methods and attributes can be associated with classes.

The classes are: -

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

The Interface is:-

- DisplayBoardControl (a name we have made up)



Click on the ad to read more

Download free eBooks at bookboon.com

And the methods include:-

- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

11.3 Further Analysis

We could now document this proposed design using UML diagrams and program a system accordingly. However before doing so it would be better to find any potential faults in our designs as fixing these faults now would be quicker than fixing the faults after time has been spent programming the system. Thus we should now refine our design using CRC cards and elaborate our classes.

CRC cards (see Chapter 6 section 6.10 and 6.11) allow us to role play various scenarios to check if our designs look feasible before refining these designs and documenting them using UML class diagrams.

The two CRC cards below have been developed to describe the Client and ClientBook classes. The panel on the left shows the class responsibilities and the panel on the right shows the classes they are expected to collaborate with.

Client	
Know name, address, phone	
Know current credit	
Decrease credit	

ClientBook	
Hold list of client IDs and associated clients	Client
Add a client	Client
Look up a client	Client

Download free eBooks at bookboon.com

We can now use these to roleplay, or test out, a scenario. In this case what happens when we get a new client in the shop? Can this client be created and added to the client book?

To do this the system must perform the following actions:-

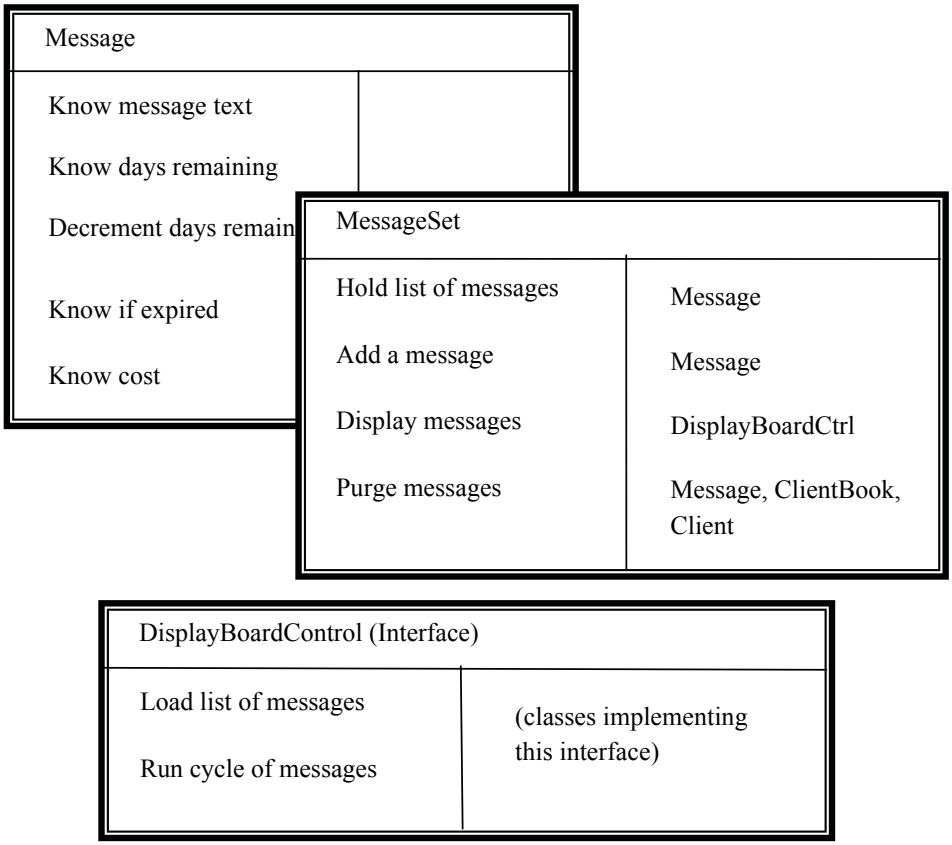
- create a new Client object
- pass it (along with the unique ID to associate with it) to the ClientBook object
- add the client to the client book.

By looking at the CRC cards we can see that:-

- the constructor for Client will be able to create a new client object
- The ClientBook has the capability to add a client and
- the ClientBook can hold the IDs associated with each client.

It would therefore appear that this part of the system will work at least in this respect – of course we need to create CRC cards to describe every class and to test the system with a range of scenarios.

Below are three CRC cards to describe the Message and MessageSet classes and the DisplayBoardControl interface.



Download free eBooks at bookboon.com

What we want to 'test' here is that messages can be created, added to the MessageSet and displayed on the display.

A point of requirements definition occurs here. There are two possibilities regarding the interface to the display board:-

a) we load one message at a time, display it, then load the next message, and so on.
b) we load a collection of messages in one go, then tell the board to display them in sequence which it does autonomously

The correct choice depends on finding out how the real display board actually works.

Note that (a) would mean a simple display board and more complexity for the "Display messages" responsibility of MessageSet, while (b) implies the converse.

For this exercise we will assume the answer to this is (b), hence the responsibilities of scrolling through a set of messages will be assigned to the DisplayBoardControl interface.

STUDY FOR YOUR MASTER'S DEGREE
IN THE CRADLE OF SWEDISH ENGINEERING
Chalmers University of Technology conducts research and education in engineering and natural sciences, architecture, technology-related mathematical sciences and nautical sciences. Behind all that Chalmers accomplishes, the aim persists for contributing to a sustainable future – both nationally and globally.
Visit us on **Chalmers.se** or **Next Stop Chalmers** on facebook.


Click on the ad to read more

Download free eBooks at bookboon.com

Looking at these CRC cards it would appear that we can

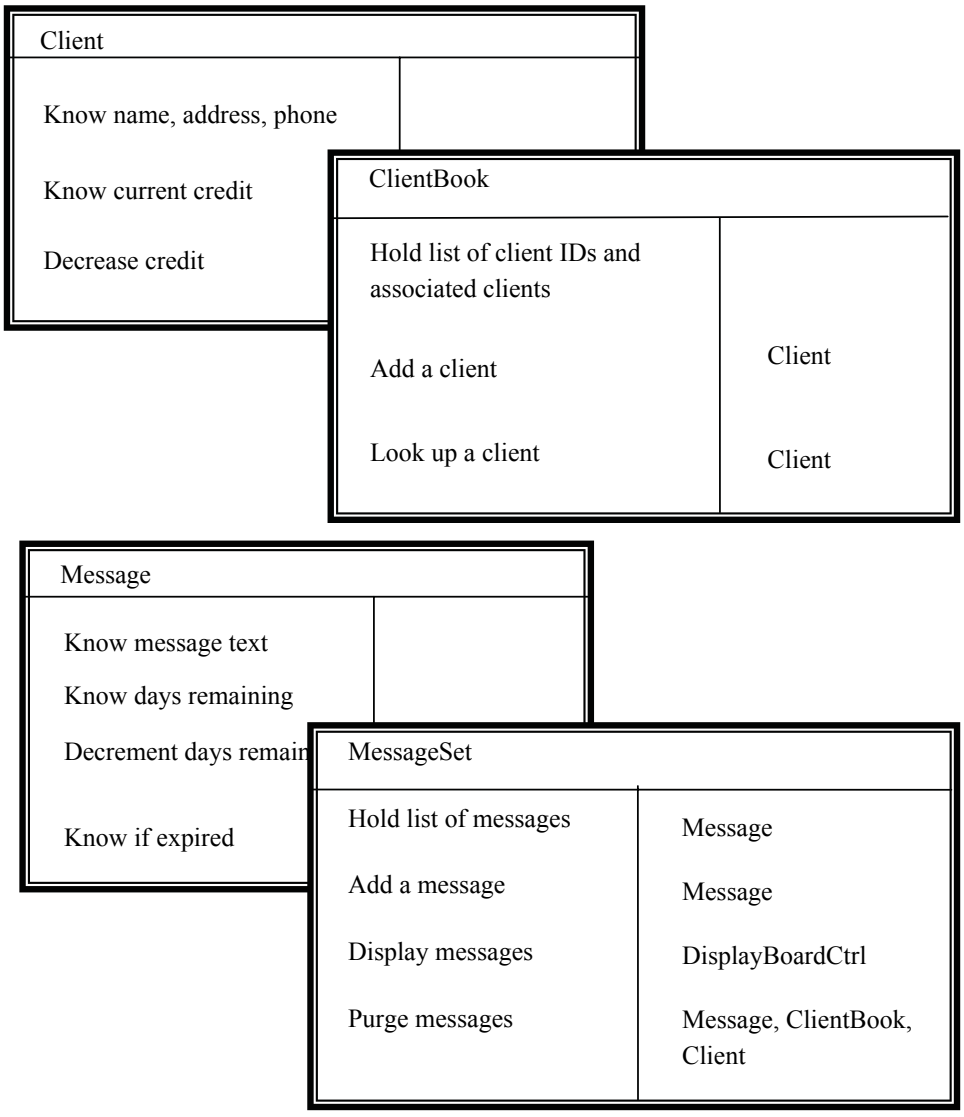
- Create a new message,
- Add this to the message set and
- Display these messages by invoking load 'list of messages' and 'run cycle of messages'

So this part of our design also seems to work.

The Message Purge Scenario

The final scenario that we want to run though here is the message purge scenario. At the end of the day when messages have been displayed the remaining days of the message need to be decremented and the message will need to be deleted if a) the message has run out of days or b) the client has run out of credit.

CRC cards for the classes involved in this have been drawn below…



Download free eBooks at bookboon.com

Activity 1

To purge the messages, the MessageBook cycles through its list of messages reducing the credit for the client who 'owns' this message, decrementing the days remaining for that message and deleting messages when appropriate.

Looking at the CRC cards above work through the following steps and identify any potential problems with these classes:-

For each message
- tell the Message to decrement its days remaining and
- tell the relevant Client to decrease its credit
  - ask the Message for its client ID
  - ask the Message for its cost
  - ask the ClientBook for the client with this ID
  - tell the Client to decrease its credit by the cost of the message
- if either the Client's credit is <= 0 or the Message is now expired
  - delete the message from the list

Feedback 1

A problem becomes evident when we try to find the client associated with a message as Message does not know the client ID.

We therefore need to add this responsibility to the Message class.

A revised design for the Message class is given below…

Message	
Know message text **Know clientID** Know days remaining Decrement days remaining Know if expired Know cost	

By drawing out CRC cards for each class and interface and by role playing a range of scenarios we have checked and revised our plans for the system – we can now refine these and document these using UML diagrams.

Download free eBooks at bookboon.com

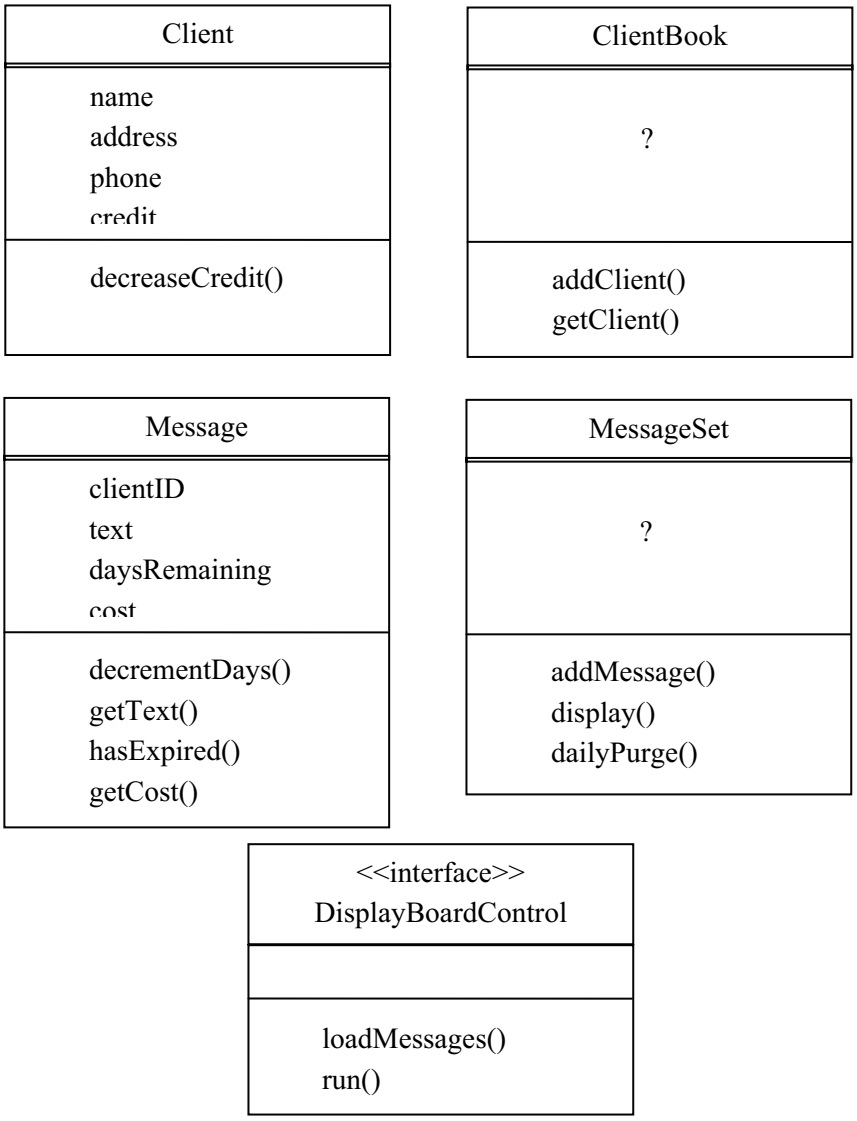
11.4 Documenting the design using UML

To fully document our designs we need to:-

- Determine in detail what attributes go in each class
- Determine how the classes are related and
- Put classes into appropriate packages.

Elaborating the Classes

Having worked through CRC scenarios we can make an initial assignment of instance variables and methods among our classes, including some accessors and mutators whose necessity has become evident (see diagram below).



Download free eBooks at bookboon.com

We don't know of any simple attributes which ClientBook and MessageSet will require, but they will need to be associated with other classes so we still have some work to do there – hence the ?s (which are not an official part of UML)!

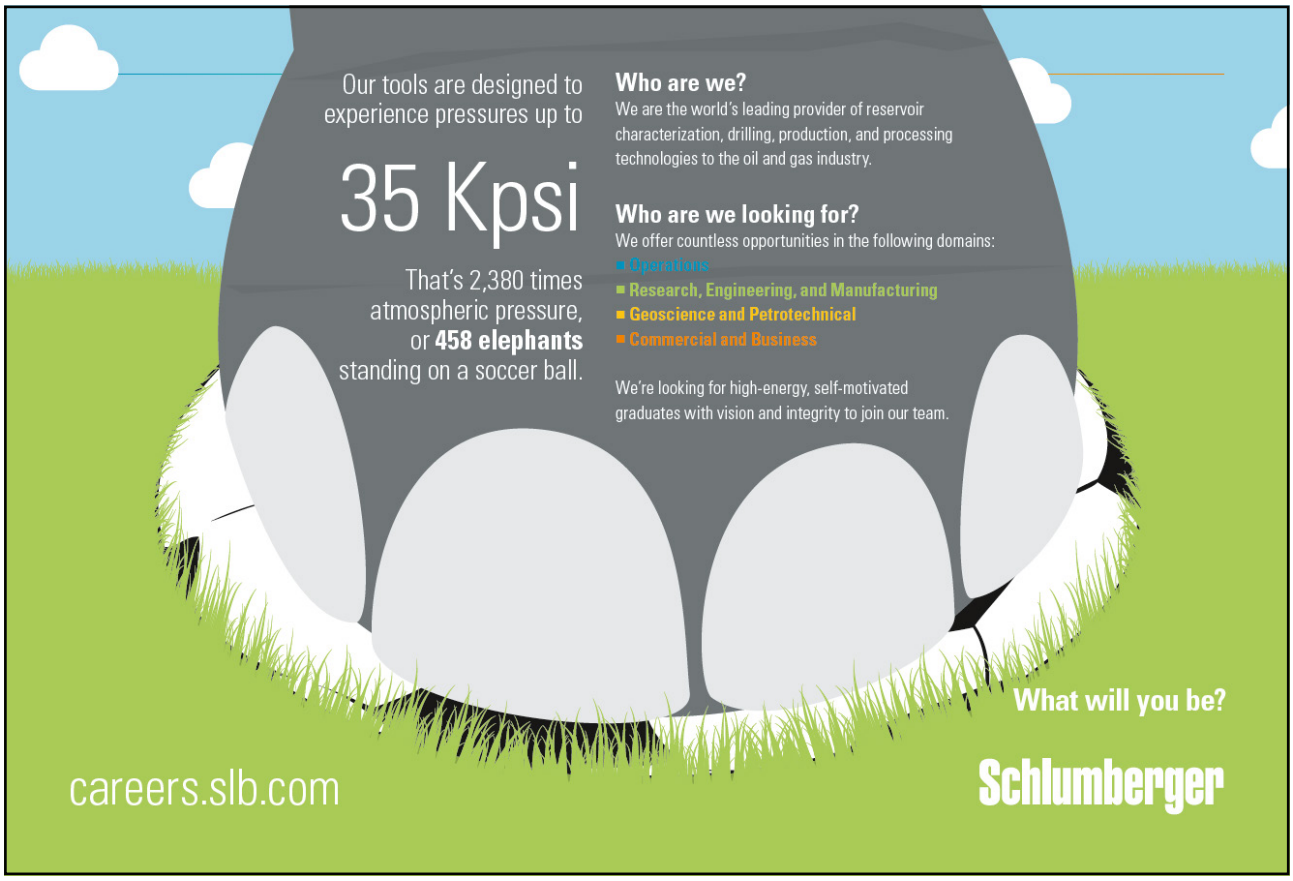
Relationships Between Classes

We can now start to work out how these classes are related.

Starting with ClientBook and Client:- a ClientBook will record details of zero or more clients.

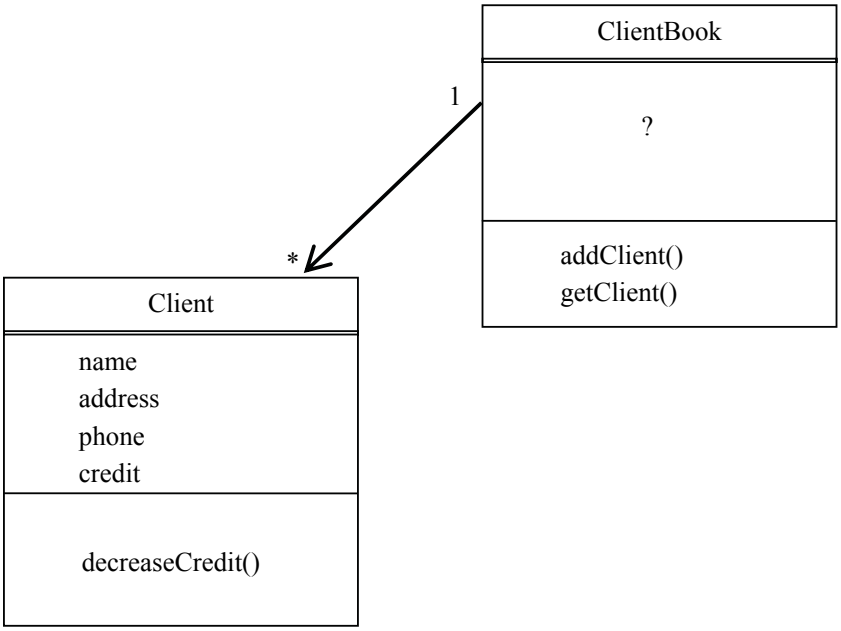
The navigability will be from ClientBook to client because the book "knows about" its Clients (in implementation terms it will have references to them) but the individual Clients will not have references back to the book.

The one-to-many relationship suggests that ClientBook will have a **Collection** (of some kind) of Clients. The specification states that each Client will have a unique ID thus the collection will in fact be a map where each entry is made up of a pair of values – in this case a clientID (a string) and a Client object.



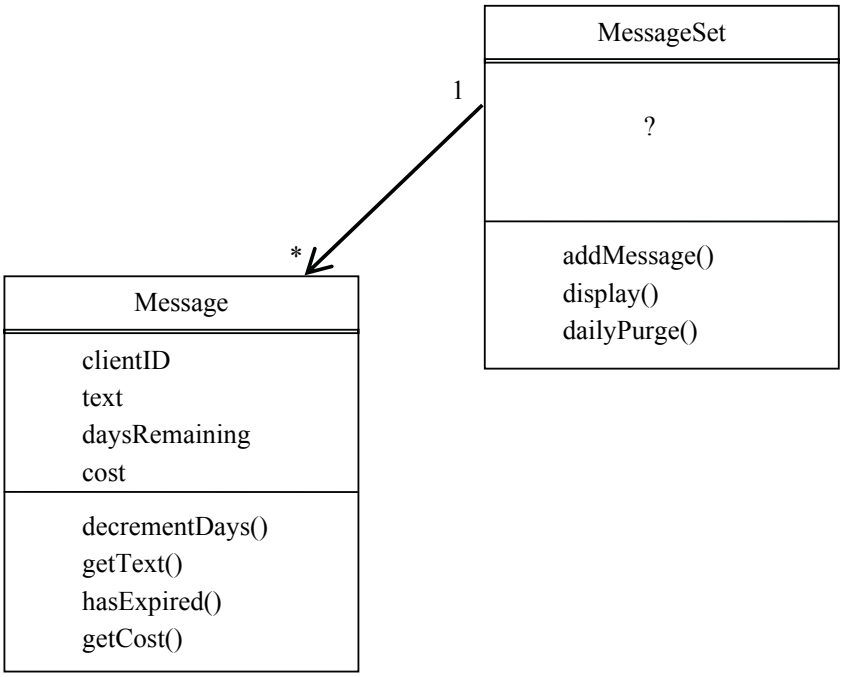
Click on the ad to read more

Download free eBooks at bookboon.com



The relationship between MessageSet and Message is very similar to the relationship between ClientBook and Clients.

Although MessageSet appears to have no attributes, its one-to-many association with Message again implies an attribute which is a Collection type. The specification states that messages must be unique but does not imply a key value is required thus a simple set will suffice.

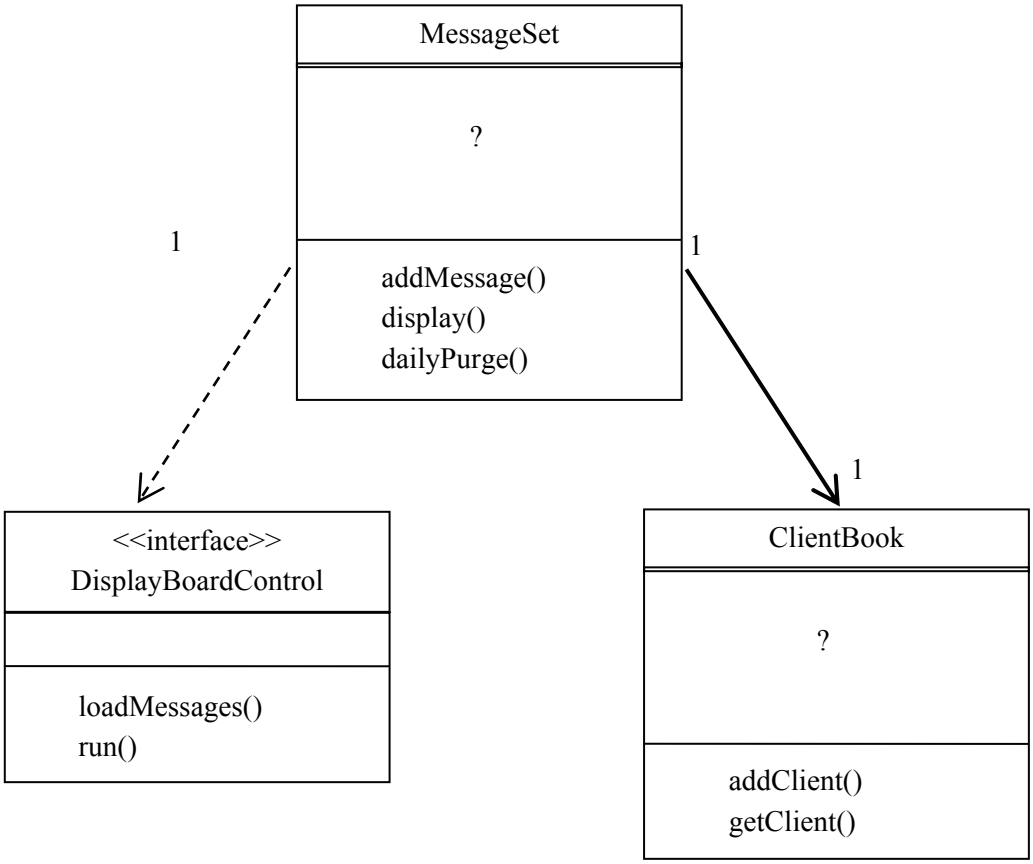


Download free eBooks at bookboon.com

Relating the Classes: MessageSet, ClientBook, and DisplayBoardControl

Because MessageSet is responsible for initiating the display of the messages on the display board it has a dependency on a class implementing the DisplayBoardControl interface.

MessageSet also has a relationship with ClientBook because it needs to access and update Client information when the daily purge is carried out. This is shown below.

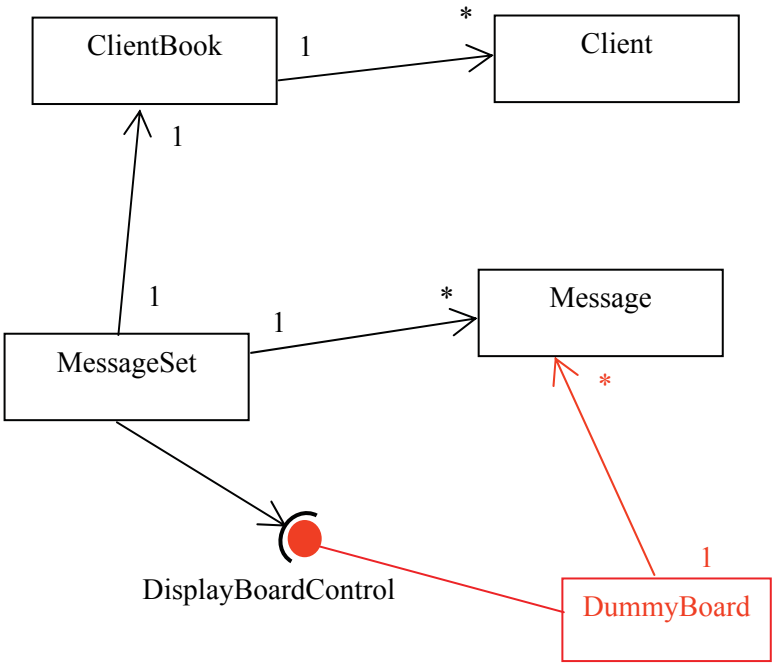


Relating the Classes Overall

The diagram below shows how all of these classes are related. An additional class, DummyBoard, has been included which will implement the DisplayBoardControl interface for testing purposes.

Since DummyBoard will have a collection of messages loaded it also has a one-to-many relationship with Message.

Download free eBooks at bookboon.com



Note the use of the concise "ball and socket" notation for the DisplayBoardControl interface.



Click on the ad to read more

Download free eBooks at bookboon.com

While the classes above will form the heart of the system two additional classes will be required to drive and manage the system as a whole.

One of these 'GUImain' will have the 'main' method required to run the system and this will invoke a graphical user interface through which the user will interact with the system adding clients, messages etc.

The other additional class 'Manager' will control additional functionality not specified by the shop owner but implicitly required – for instance at the end of the day the details of the ClientBook and MessageSet will need to be saved to file. This data will need to be restored next time the system is run as the shop owner will clearly not want to enter details of all the clients every time they run the program.

11.5 Prototyping the Interface

While methods for gathering user requirements is beyond the scope of this text – it is always a good idea to prototype an interface and get feedback on this before proceeding with the development.

The figure below shows the proposed interface for this system:-

MessageManager v7				
NEW CLIENTS		NEW MESSAGES		Find Client
				Increase Credit
Client ID				Delete Client
Name		Client ID		
Address				Display Message
Phone				Purge Messages
Credits				
	Add Client		Add Message	Save and Exit

This is made up of three areas. From left to right these are a) an area for adding new clients, b) and area for adding new messages and c) an area for buttons dedicated to other essential operations.

Each of these three areas will be implemented using a JPanel placed within one larger JFrame.

11.6 Revising the Design to Accommodate Changing Requirements

Changing software requirements are a fact of life and OO programming is intended to help software engineers make program adaptations easier, quicker, cheaper and with less risk of generating errors. The principles of inheritance, method overriding and polymorphism are essential OO features that help in this manner.

Download free eBooks at bookboon.com

In this project when gaining feedback from the shop owner on the prototype interface they comment that they generally like the interface but that they have an additional system requirement:-

> Some messages are 'urgent messages'. These should be highlighted on the display by placing three stars before and after the message and the cost of these messages will be twice the cost of ordinary messages. Other than that urgent messages are just like ordinary messages.

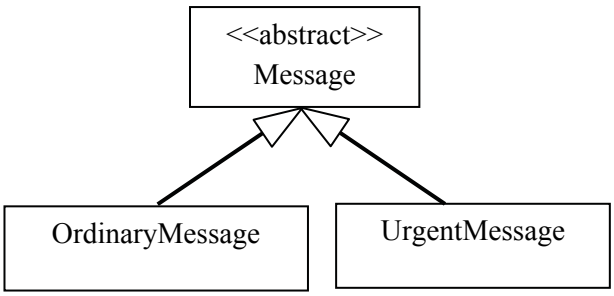
Modifying the interface design to accommodate this change is easy – we can either:-

- create an new panel to accommodate the creation of 'Urgent Messages' or
- since the data required is identical to normal messages we can just add an extra button to the middle panel.

But how will these extra requirements impact on the underlying classes within the system?

If OO principles work implementing this additional requirement should be relatively simple. Firstly there is clearly a strong relationship between a 'Message' and an 'Urgent Message'

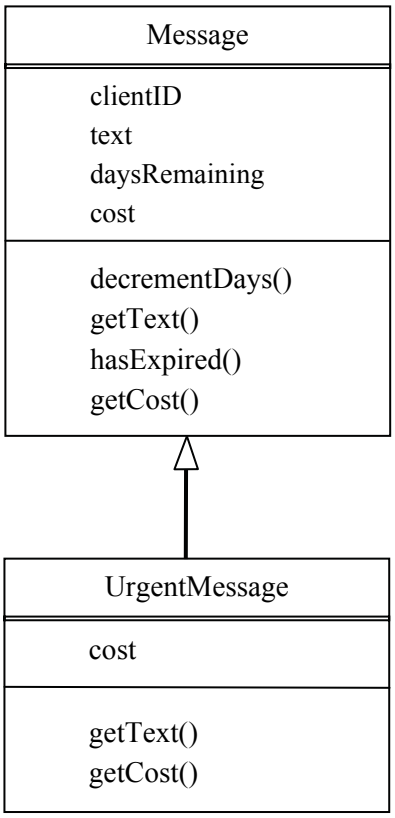
If both classes had some unique features but there was a significant overlap in functionality we could introduce an inheritance hierarchy to deal with this:-

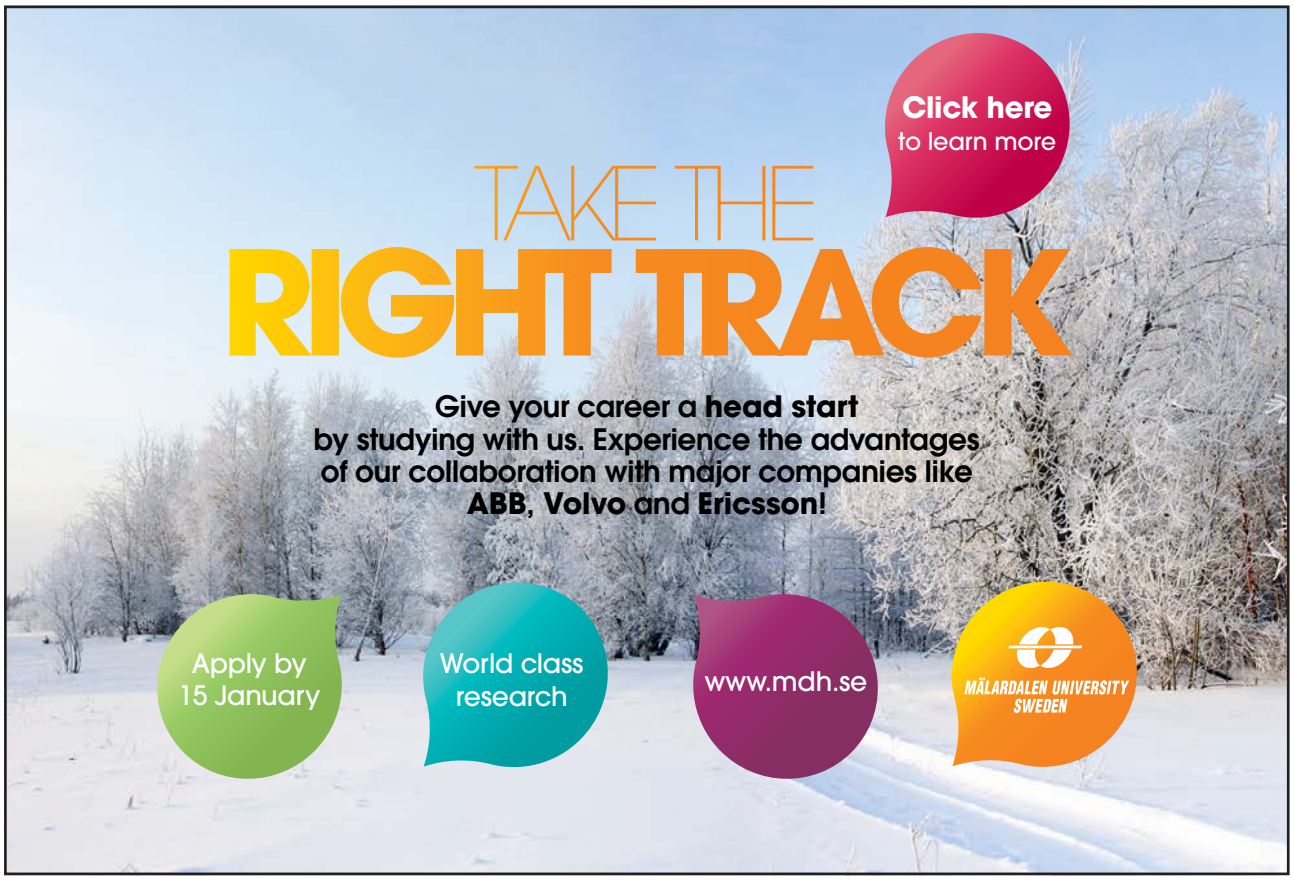


However in this case there are no unique features of an ordinary message – messages have an associated cost, the cost and text can be obtained and new messages can be created. All this is true for urgent messages. An urgent message is just the same as an ordinary message where the text and the cost has been changed slightly. Thus UrgentMessage is a type of Message and can inherit ALL of the features of Message with the cost and text methods being overridden.

Thus the Message and UrgentMessage classes are be related as shown below, with UrgentMessage inheriting all of the values and methods associated with Message but overriding getCost() and getText(0 methods to reflect the different cost and text associated with urgent messages.

Download free eBooks at bookboon.com

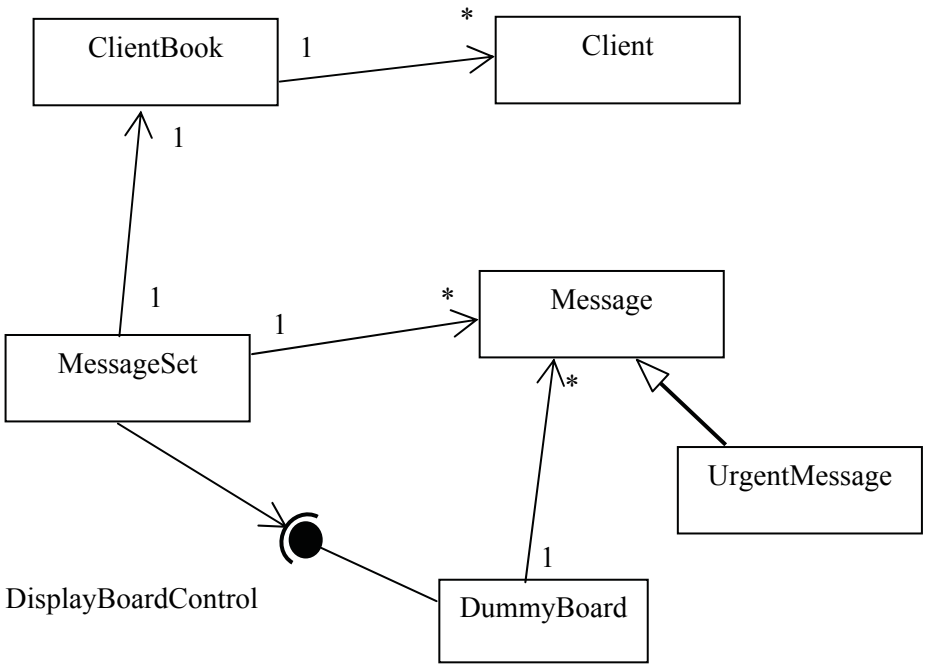




Click on the ad to read more

Download free eBooks at bookboon.com

A revised class diagram is below. But how will this change impact upon other parts of the system?



Thanks to the operation of polymorphism this change will have no impact at all on any other part of the system!

Looking at the class diagram above we can see that MessageSet keeps and manages a set of Messages (DummyBoard also keeps a set of messages – once they have been uploaded for display). But what about UrgentMessages?

Urgent messages are just a specific type of message. When the addMessage() method is invoked within MessageSet it requires an object of type Message i.e. a message to be added – but an object of the subtype UrgentMessage **is still a 'Message'** so the addMessage() method would accept an UrgentMessage object.

Therefore, without making any changes at all to MessageSet, MessageSet can maintain a set of all messages to be displayed (both urgent and ordinary)!

Furthermore when the dailyPurge() method is invoked it invokes the getCost() method on a Message object so that the client can be charged for that message. At run time the JVM will determine whether the object is of type Message or of type UrgentMessage and it will invoke the correct version of the getCost() method – remember this was overridden in UrgentMessage. This is polymorphism in action!

Download free eBooks at bookboon.com

MessageSet requires messages but, thanks to the application of polymorphism and method overriding, MessageSet will happily deal with any Message subtype as though it were a Message object. If later we decided to create new message types (such as a Christmas message) MessageSet would be able to deal with these as well without changing a single line of code!

Thus in this application we are able to extend the system to add the facility for urgent messages by adding only one class and making one small change to the interface.

Without the application of polymorphism we would need to have made additional changes to other parts of the system – namely MessageSet and DummyBoard.

Object Orientation has enabled to the system to be extended with minimal effort!

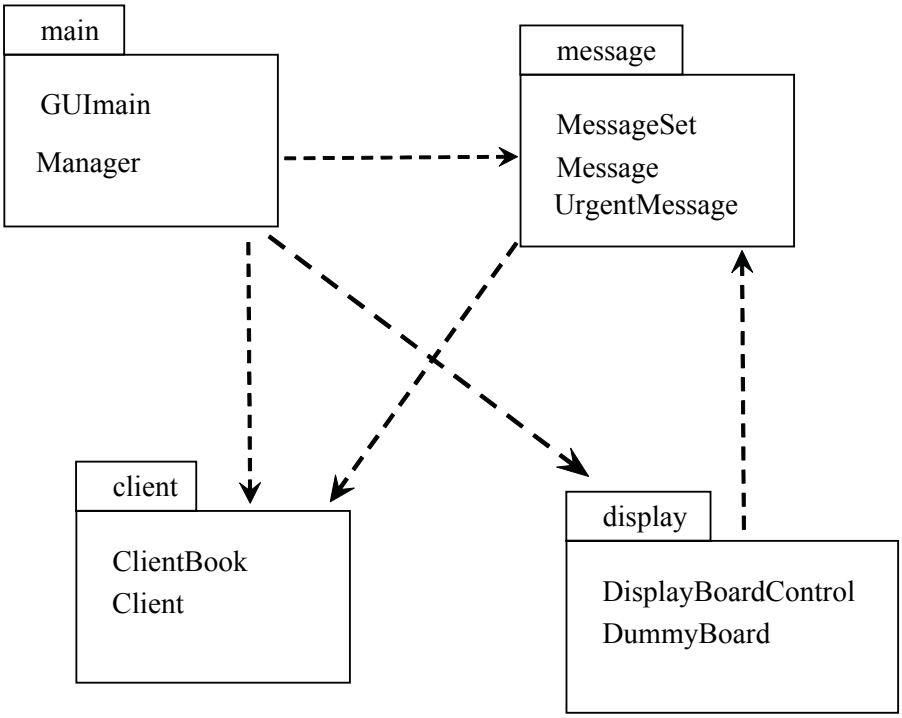
11.7 Packaging the Classes

Large programs should be segmented into packages as this provides an appropriate level of encapsulation and access control (as described in Chapter 2).

The system being used here to demonstrate the theory in this textbook hardly qualifies as large – nonetheless it has been decided to package related classes together as shown below.


GET A MASTER'S DEGREE IN UMEÅ!
• modern campus • world class research • 32 000 students
• top class teachers • ranked nr 1 in Sweden by international students
Master's programmes:
• Architecture
• Industrial Design
• Science
• Engineering
APPLY NOW!
UMEÅ UNIVERSITY
FACULTY OF SCIENCE & TECHNOLOGY
Click on the ad to read more

Download free eBooks at bookboon.com



This diagram shows the four packages used and the classes within each package. Also shown are associations between the packages. Not surprisingly the main package, which houses the system interface, is associated with all of the other packages – this is because the interface invokes functionality throughout the system.

Having completed the design, and accommodated changing requirements, we can start implementing the system. This will be done in two phases:-

In the first phase a basic system will be implemented which will allow messages and clients to be created, the details written to file and messages to be displayed.

In the second phase the system functionality will be extended to allow clients to be deleted and to allow their credit to be increased. This will be done in a way to allow the demonstration of Test Driven Development (as described in Chapter 10).

Download free eBooks at bookboon.com

11.8 Programming the Message Classes

Message, UrgentMessage and MessageSet are relatively straight forward to program.

Message has various instance variables (Strings: clientID, text and int: daysRemaining). It has a constructor to initialize the instance variables and it has the following methods:-

void decrementDays ()
boolean hasExpired()
String getClientID()
int getCost()
String getText ()

The Manager class will need to store the ClientBook and MessageSet objects to a file. To do this all Client objects and Message objects will also need to be stored hence these classes (including the Message class) will need to implement the Serializable interface.

Finally the requirements state that "No duplicate messages (i.e. the same text for the same client) are permitted."

Therefore Message must override the equals() and hashCode() methods to ensure that duplicates will not be permitted when the messages are stored in a Set.



Click on the ad to read more

Download free eBooks at bookboon.com

The complete code for this class is given below – though comments have been excluded for the sake of brevity.

```
package messages;
import java.io.Serializable;

public class Message implements Serializable
{
      final int COST=1;

      private String clientID;
      private String messageText;
      private int daysRemaining;

      public Message (String pClientID, String pText,
                                          int pDaysRemaining) {
            clientID = pClientID;
            messageText = pText;
            daysRemaining = pDaysRemaining;
      }

      public void decrementDays() {
            daysRemaining--;
      }

      public boolean hasExpired() {
            return (daysRemaining == 0);
      }

      public String getClientID() {
            return clientID;
      }

      public String getText () {
            return messageText;
      }

      public int getCost() {
      return COST;
      }

      public int hashCode () {
            return (clientID + messageText).hashCode();
      }

      public boolean equals (Object pOther) {
            Message otherMsg = (Message)pOther;
            return (clientID.equals(otherMsg.clientID) &&
            messageText.equals(otherMsg.messageText));
      }

      public String toString(){
            return ("Message text: " + messageText +
                  "\nClient: " + clientID +
                  "\nDays left: " + daysRemaining);
      }
}
```


Download free eBooks at bookboon.com

The UrgentMessage class is extremely short and sweet as it inherits almost all of its functionality from Message:-

```
package messages;

public class UrgentMessage extends Message
{

      final int COST=2;

      public UrgentMessage (String pClientID, String pText,
                                          int pDaysRemaining){
            super(pClientID, pText, pDaysRemaining);
      }

      public String getText () {
            return "*** "+super.getText()+ " ***";
      }

      public int getCost() {
            return COST;
      }
}
```

The MessageSet class has a one-to-many relationship with Message. This implies a collection type and the fact that duplicate massages are not allowed (at least for the same client) implies the collection should be a Set.

The MessageSet class requires an instance variable to hold the set of messages and also one to reference a ClientBook – as it needs access to the clients when performing a daily purge.

A constructor is required to assign a new HashSet() to messageSet and to initialize clientBook to a parameter. The following methods are also required:-

void addMessage(Message pMsgToAdd)
void display(DisplayBoardControl db)
void dailyPurge()

Download free eBooks at bookboon.com

Some of the code from this class is shown below:-

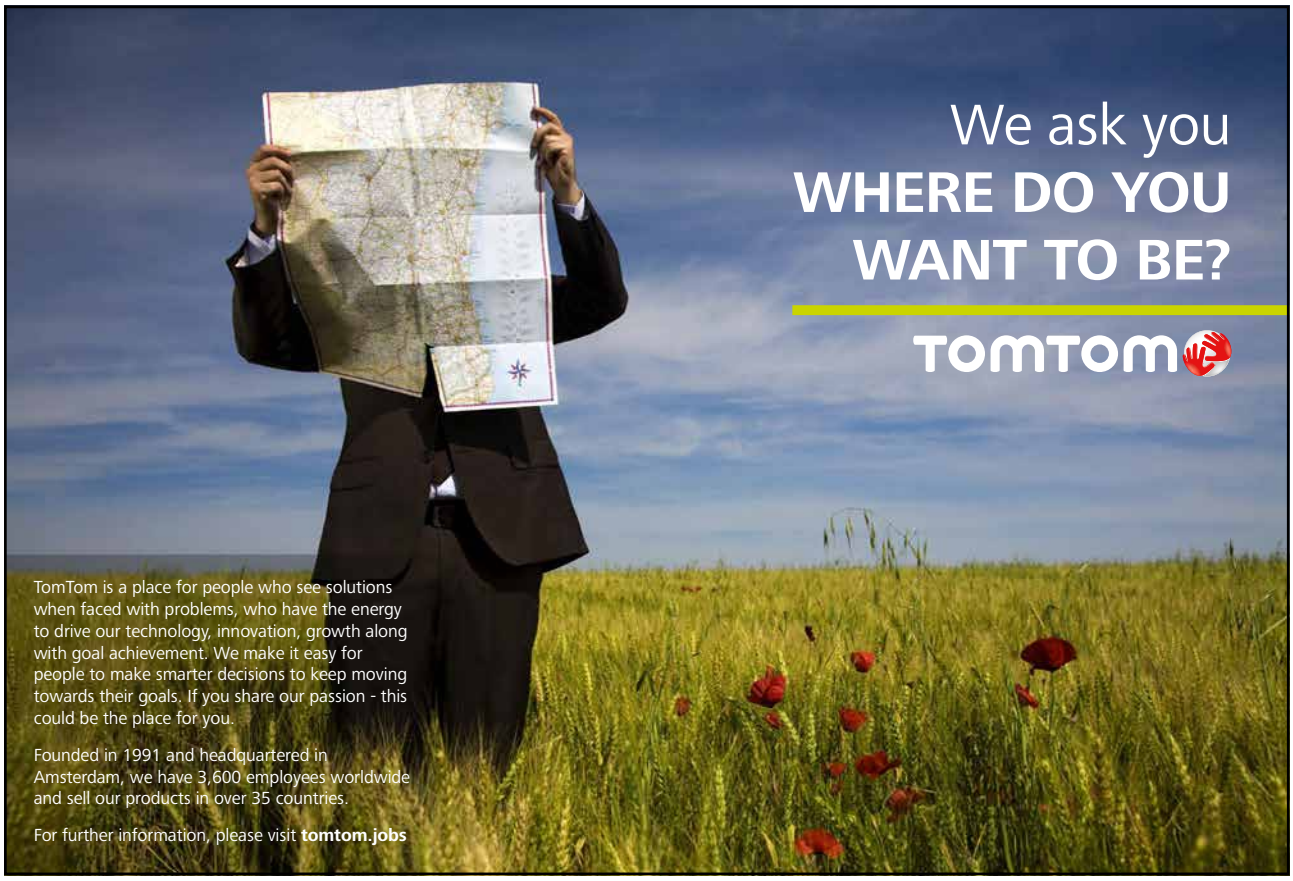
```
package messages;

import ...

public class MessageSet implements Serializable
{

	private Set<Message> messageSet;
	private ClientBook clients;

	public MessageSet (ClientBook pClients){
		clients = pClients;
		messageSet = new HashSet<Message>();
	}

	public void addMessage(Message pMsgToAdd) {
		messageSet.add(pMsgToAdd);
	}

	public void display(DisplayBoardControl db)
	{
		db.loadMessages(messageSet);
		db.run();
	}

	public void dailyPurge() {

		// code omitted here

	}

	public void saveToFile(ObjectOutputStream oos) {
		try {
			oos.writeObject(this);

		} catch (IOException ioe) {
			JOptionPane.showMessageDialog(null, ""+ioe);
		}
	}

	static public MessageSet readFromFile(ObjectInputStream ois) {
		MessageSet cb = null;

		try {
			cb = (MessageSet) ois.readObject();

		} catch (IOException ioe) {
			JOptionPane.showMessageDialog(null, ""+ioe);
			System.exit(1);
		} catch (ClassNotFoundException cnfe) {
			JOptionPane.showMessageDialog(null, ""+cnfe);
			System.exit(1);
		}
		return cb;
	}
}
```


Download free eBooks at bookboon.com

The code above shows the creation of a typed collection of 'Message' and methods to add and display messages.

The method to display messages requires and object of type DisplayBoardControl to be passed as a parameter. Initially a DummyBoard object will be provided however when a real display board is purchased then this object will replace the DummyBoard object. This will have no impact on the code within the display() method as both objects are of the more general type DisplayBoardControl. This is another example of the application of polymorphism.

Two additional methods have been created saveToFile() and readFromFile() which read and write the entire set of messages to file using the technique of object serialisation.



Click on the ad to read more

Download free eBooks at bookboon.com

The dailyPurge() method was excluded from the code above so we could concentrate on this method below:-

```
public void dailyPurge() {

      Client client;

      // loop through all current messages

      for (Message msg: messageSet) {

            // deduct 1 from days remaining for message
            msg.decrementDays();

            try
            {
                  // decrease client credit for this message
                  client = clients.getClient(msg.getClientID());
                  client.decreaseCredit(msg.getCost());

                  // if message expired or client out of credit remove it
                  if (msg.hasExpired() || client.getCredit() <= 0) {
                        messageSet.remove(msg);
                  }
            }
            catch (UnknownClientException uce) {
                  JOptionPane.showMessageDialog(null,
                  "INTERNAL ERROR IN MessageSet.Purge()\n" +
                  "Exception details: " + uce +
                  "\nMesssage details:\n" + msg);

                  if (msg.hasExpired()) {
                        messageSet.remove(msg);
                  }

            }
      }
}
```

The dailyPurge() method performs the following actions:-

For each message

Decrement the days remaining for that message
Find the client who paid for that message
Find the cost of the message and deduct this from that clients credit
If the message has expired or if the client has run out of credit then
Remove the message

Note it is possible that a client could not be found – hence the try catch block. This will be discussed in the next section.

Download free eBooks at bookboon.com

11.9 Programming the Client Classes

Programming the Client class is very similar to programming the Message class and is not shown here.

Programming the ClientBook class is also very similar to programming MessageSet, and most of this class is omitted here however there are two significant differences:-

- All clients have a clientID so ClientBook uses a Map instead of a Set.
- The method getClient() could fail if no client exists with the specified clientID. We need to build in protection in case a client cannot be found.

```
package clients;

import ...

public class ClientBook implements Serializable {

	private Map<String,Client> clientMap;

	public ClientBook() {
		clientMap = new HashMap<String,Client>();
	}

	public void addClient(String pClientID, Client pNewClient) {
		clientMap.put(pClientID, pNewClient);
	}

	public Client getClient(String pClientID)
		// details omitted

	}

	public void saveToFile(ObjectOutputStream oos) {
		// details omitted
	}

	static public ClientBook readFromFile(ObjectInputStream ois) {
		// details omitted
	}
}
```

The code above shows the creation of the Map and the other methods required by the ClientBook class.

Download free eBooks at bookboon.com

11.10 Creating and Handling UnknownClientException

The getClient() method in the ClientBook class will return a null value if no client exists with the specified ID. In such a case during the daily purge our program will crash as we try to invoke the decreaseCredit() method without having a client object to invoke this method on (it will crash with a NullPointerException).

We therefore need to build in protection against this eventuality. To protect against this we need to:-

- Create a new kind of exception (as described in Chapter 9) called UnknownClientException
- tell the ClientBook class to throw this exception if a client is not found
- catch and deal with this exception in the dailyPurge() method.

The first step is simple and not shown here.

INNOVATIVE
LIKE YOU.

If you're hoping for a truly modern education, one where you're encouraged to speak your mind and to think long-term, both when it comes to your own future and the future of the planet. Then the University of Gothenburg is the place for you.

Study a Master's programme in Gothenburg, Sweden | www.gu.se/education

UNIVERSITY OF GOTHENBURG

Click on the ad to read more

Download free eBooks at bookboon.com

Telling the getClient() method and deleteClient() method to generate this exception is relatively straight forward (as shown below):-

```
public Client getClient(String pClientID)
                                        throws UnknownClientException {
            Client foundClient;

            foundClient = clientMap.get(pClientID);

            if (foundClient != null) {
                  return foundClient;
            } else {
                  throw new UnknownClientException(
                         "ClientBook.getClient():
                        unknown client ID:" + pClientID);
            }
      }
```

Firstly we must tell the compiler that this method can generate an exception. Then, under the appropriate condition, we invoke the constructor of the exception using the keyword 'new' and pass a string message required by the constructor. The object returned by the constructor is then 'thrown'.

To be helpful the string specifies the method where this exception was generated from and the clientID for which a client was not found.

The compiler will then ensure that the programmer writing the dailyPurge() method catches this exception – hopefully they will then deal with it to prevent a crash situation.

The final step is to catch and deal with UnkownClientException within the dailyPurge() method – as shown in section 11.8 (Programming the Message Classes).

If a client does not exist we may could remove the message. However in this case we have chosen to be more cautious since we simply don't know how we have come to have an 'unowned' message.

We have therefore decided that if the message has not expired we will not to take any action other than to report the error. The message will continue to be displayed (even without having a client to charge!).

If an unowned message has expired we of course still need to remove it from the display set.

Download free eBooks at bookboon.com

11.11 Programming the Main classes

There are two classes, not shown on the class diagrams given previously, that 'drive' the system.

'Manager' is the main class that manages the system. It performs the following functions:-

- it has a permanent reference to the client book and message set.
- it sets up the data file
- it defines what happens when the system starts and
- it defines what happens when the system shuts down
- finally it has getClientBook() and getMessageSet() methods so the other parts of the system can access the clientbook and message set.

The startup() method is shown below:-

```
public void startUp() {
      try {
            FileInputStream fis = new FileInputStream(MMS_DATA_FILE);
            ObjectInputStream ois = new ObjectInputStream(fis);

            cb = ClientBook.readFromFile(ois);
            ms = MessageSet.readFromFile(ois);
            fis.close();

      } catch (FileNotFoundException fnfe) {
            JOptionPane.showMessageDialog(null, "No existing
                                    client/message data found");
            cb = new ClientBook();
            ms = new MessageSet(cb);
      } catch (IOException ioe) {
            JOptionPane.showMessageDialog(null, ""+ioe);
            System.exit(1);
      }
}
```

The startup() method tries to setup an Object Input stream from which it then tries to reconstruct ClientBook and MessageSet objects.

It has two catch blocks. The first will catch a FileNotFoundException – this will occur if the file of data cannot be found e.g. the first time this program is run. In such a case the system will create a new and empty ClientBook object and a new MessageSet object.

The second catch block will catch any other IO error and in this case will then exit the program.

Download free eBooks at bookboon.com

11.12 Programming the Interface

The other driving class is GUImain. This has a main() method which invokes the createGUI() method. This creates three JPanels and adds buttons, text boxes and labels to these according to the preliminary design. A Grid layout manager was applied to these JPanels (as described in section 10 of chapter 8).

The GUImain class also defines action listeners for each of the buttons on the GUI.

Shown below is the action listener associated with the FindClient button:

```
private class FindClientListener implements ActionListener {
      public void actionPerformed(ActionEvent arg0) {
            String id = JOptionPane.showInputDialog(null,
                                                "Enter client ID");
            if (id == null) id = ""; //in case Cancel pressed

            try {
                  JOptionPane.showMessageDialog(null,
                  manager.getClientBook().getClient(id).toString());
            }
            catch (UnknownClientException uke){
                  JOptionPane.showMessageDialog(null,
                                                "No such client");
            }
      }
}
```



Click on the ad to read more

Download free eBooks at bookboon.com

This action listener does the following:-

- It opens a dialog box to ask the user for a clients ID,
- If cancel is pressed it replaces the null returned with an ID of "".
- It then asks the manager object to return the client book.
- On the client book object it invokes the getClient() method passing the ID as a parameter
- Assuming a client is returned the toString() method is then invoked to get a string representation of the client and this is passed as a parameter to the showMessageDialog() method (which displays the details of the client with that ID).
- If getClient() fails to find a client this method will throw an UnknownclientException – this will be caught here and an appropriate message will be displayed.

11.13 Using Test Driven Development and Extending the System

Now we have a working system – though two important methods have yet to be created. We need a method to increase a clients credit – this should be placed within the Client class. We also need a method to delete a client – as this means removing them from the client book this should be placed in the ClientBook class.

It has been decided to use Test Driven Development to extend the system by providing this functionality (as discussed in Chapter 10 Agile Programming).

In TDD we must:-

1) Write tests
2) Set up automated unit testing, which fails because the classes haven't yet been written!
3) Write the classes so the tests pass

After creating these methods we must then adapt the interface so that this will invoke these methods.

Download free eBooks at bookboon.com

Two test cases are given below:-

```
public void testIncreaseCredit() {
      Client c = new Client("Simon", "Room 217", "x2756", 10);
      c.increaseCredit(10);
      assertEquals(20, c.getCredit());
}

public void testDeleteClient() {
      ClientBook cb = new ClientBook();
      Client c = new Client("Simon", "Room 217", "x2756", 10);
      cb.addClient("sk", c);
      try {
            cb.deleteClient("sk");
      } catch (UnknownClientException uce) {
            fail();
      }
      try {
            c = cb.getClient("sk");
            fail();
      } catch (UnknownClientException uce) {
}
}
```

The first of these creates a client with 10 units of credit, adds an additional 10 units of credit and then checks that this client has 20 units of credit.

One test does alone not sufficiently prove that the increaseCredit() method will always work so we made need to define additional tests.

The second test creates and empty client book, creates a client, adds the client to the client book and tries to delete this client – if at this point an unknownClientexception is generated then we know there is a problem. Once deleted we try to delete the object for a second time and this should fail as an unkownClientException should now be generated. If an exception is not generated at this point then the class is flawed and the test should 'fail'.

Having created test cases these will generate complier errors as the methods increaseCredit() and deleteClient() do not exist.

We must now create the methods and revise them until these tests pass.

The increaseCredit() method is given below…

```
public void increaseCredit(int pExtraCredit) {
      credit = credit + pExtraCredit;
}
```


Download free eBooks at bookboon.com

Theory suggest that TDD leads to simple code.

In this case by focusing our minds on what the increaseCredit() method needs to achieve we reduce the risk of over complicating the code. Of course we may need a range of test case to make sure the method has all of the essential functionality it needs.

11.14 Generating Javadoc

Documentation is essential and can be generated automatically (as described in Chapter 8 – Java Development Tools) assuming appropriate comments have been placed in the code.

Javadoc comments have been placed in the code to describe all classes, all constructors and all methods. All parameters and return values have been described.

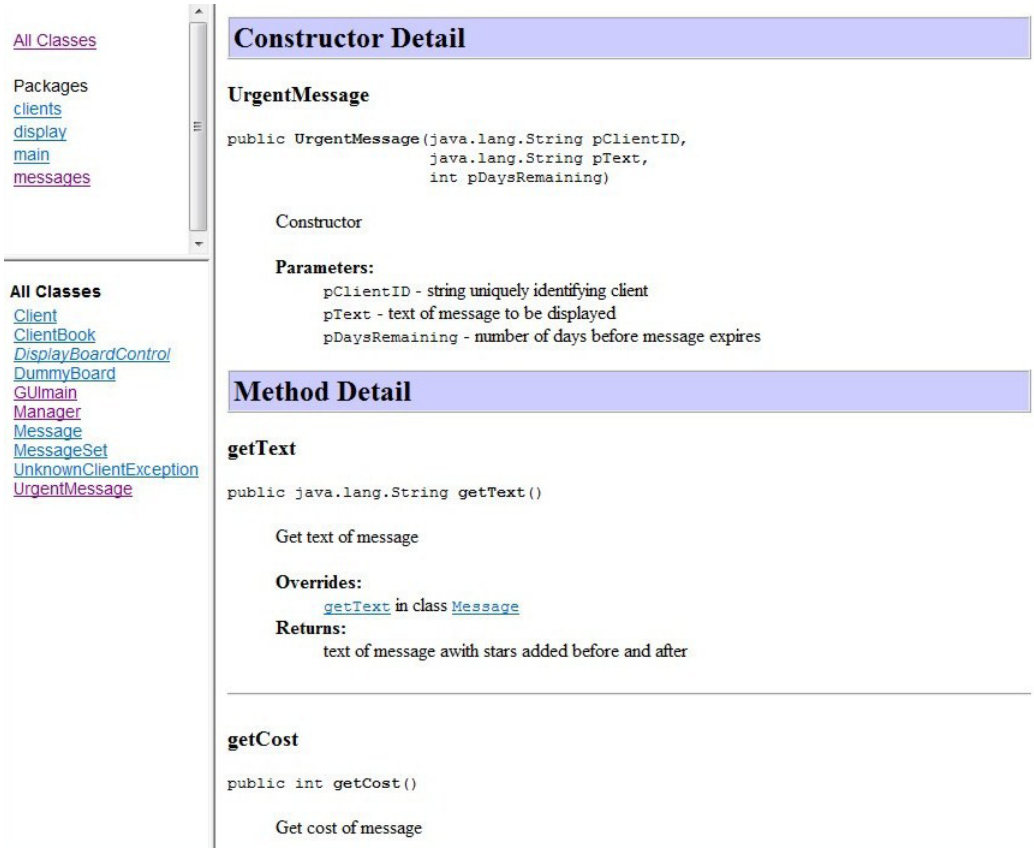
Three of the comments taken from the Client class are shown below:-

```
/**********************************************************
 * A customer of the message display service
 *
 * @author Simon Kendal
 * @version 1.0 (26 June 2009)
 **********************************************************/
public class Client implements Serializable {

      ... lines missing ...

      /**
       * Constructor
       *
       * @param pName name of client
       * @param pAddress client's address
       * @param pPhone client's phone number
       * @param pCredit initial credit for client
       */
      public Client(String pName, String pAddress, String pPhone,
                                                      int pCredit) {

      ... lines missing ...

      /**
       * return the client's current credit
       *
       * @return credit units remaining
       */
      public int getCredit() {

      ... lines missing ...

      }
```


Download free eBooks at bookboon.com

Once Javadoc style comments have been placed throughout the code initiating the javadoc tool will generate a set of web pages to describe the system (in Eclipse this is done by selecting Project | Generate Javadoc menu items).

The following picture shows part of the Java documentation describing the UrgentMessage class:-



11.15 Running the System and Potential Compiler Warnings

The complete program, as described in this chapter, is available with this textbook as an exported Eclipse project. To run this program:-

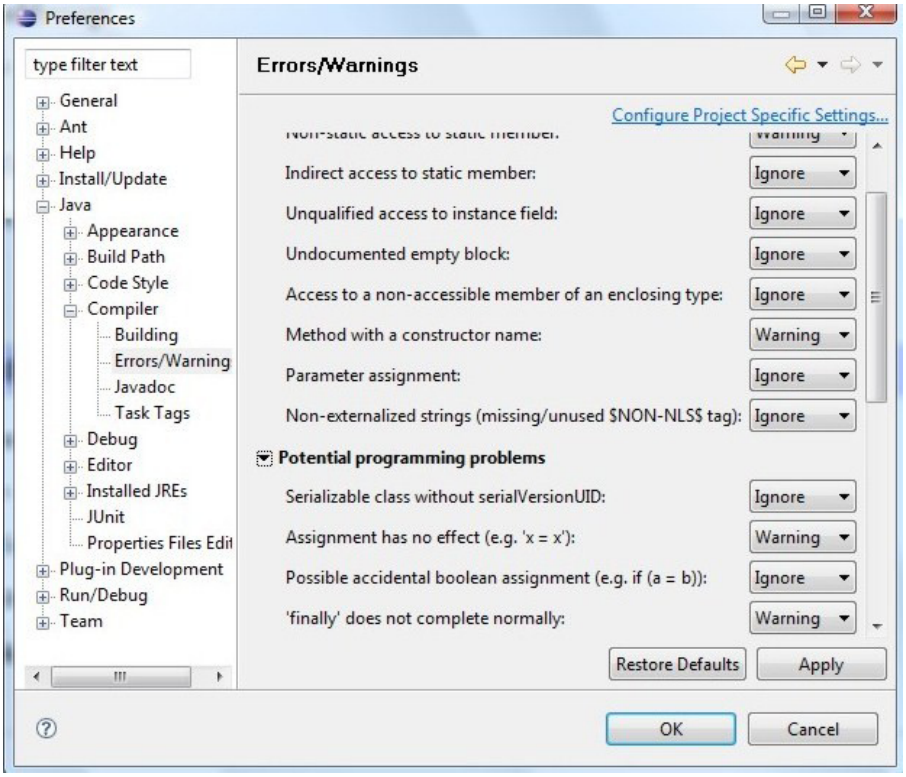
- Download the file 'OOP Using Java'
- Import it into Eclipse by selecting File | Import | Existing Project Into Workspace | Select Archive File 'OOP Using Java' and then select the Message Management System project that is inside the archive file.
- Using the Eclipse package explorer window, right click on the GUImain class which is inside the package 'main' and select 'Run as Java application'.

In this exported file are all classes, methods and test cases discussed in this chapter along with the Javadoc generated by the Javadoc tool. To view the Javadoc go to the .doc folder and double click on the index.html page.

Download free eBooks at bookboon.com

When examining this program, depending upon your compiler settings, you may notice some warning messages. One in particular that you are likely to see refers to the serializable classes (MessageSet, Message etc) as I have not given them version numbers.

These warnings should have no impact on running the system and can be switched of if required by going to Window | Preferences |Compiler | Errors & Warnings and selecting ignore for those warnings you wish to suppress (as shown below).



11.16 The Finished System…

The following screen shots show the finished system.

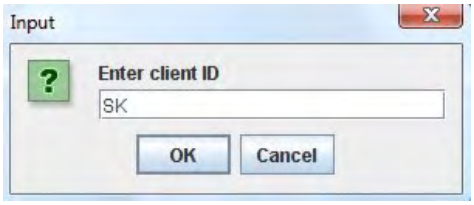
Firstly the main interface window – this is very similar to the design. The only change was one extra button that was added to allow a message to be designated as an urgent message.



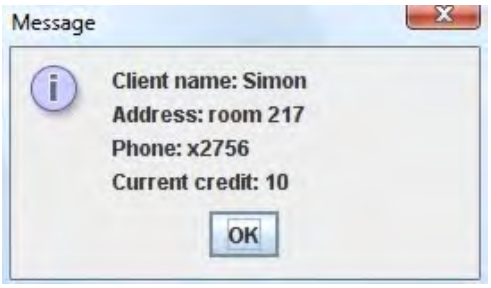
Download free eBooks at bookboon.com

The next two images show the pop up dialogues that appear when the 'Find Client' button is pressed.

Firstly asking for a client ID…



Secondly displaying the client details – assuming a client with this ID has been added.





Click on the ad to read more

Download free eBooks at bookboon.com

The 'Display Messages' button shows each of the messages on the screen that should be display using the DummyBoard class. This is only crudely simulating a real display board and makes no effort to scroll the messages or display them in any graphically interesting way.

'Purge Messages' invokes the purgeMessages() method. It does nothing visible but decrements the days remaining for each message, decreases the clients credits and deletes the messages if appropriate. This can be tested by running Find Client before and after doing a daily purge.

11.17 Summary

The fundamental principles of the Object Orientated development paradigm are

- abstraction
- encapsulation
- generalization/specialization (inheritance)
- polymorphism

These principles are ubiquitous throughout the Java language and library package APIs as well as providing a framework for our own software development projects.

A well-established range of tools and reference support is available for OO development in Java, some of it allied to modern 'agile' development approaches.

Throughout this chapter you will hopefully have seen how Object Orientation supports the programmer by:-

- using abstraction and encapsulation to enables us to focus on and program different parts of a complex system without worrying about 'the whole'.
- using inheritance to 'factor out' common code
- using polymorphism to make programs easier to change
- using tools help document and manage large software projects.

This has been exemplified using Java but the same principles and benefits apply to all OO programming languages and the tools demonstrated here are available in many modern IDE's.

Through reading this book, and doing the small exercises, you will hopefully have gained some understanding of these principles.

I hope you have found this book helpful and I wish you all the best for the future.

Download free eBooks at bookboon.com