Note to lecturer and student
The Lecturer’s Guide for this course includes a free CD, which provides valuable extra resources:

- background information on SQL for Level 3
- Visual Basic source code and Access databases for examples and case studies in Topics 1 and 2
- web code for the examples and case studies of web applications in Topics 3, 4 and 5, HTML for website designs, including multimedia applications, so that students do not need to input code to try out the examples
- in-depth case studies for lecturers to print and hand out to students as well as sample solutions: these consist of system development briefings, such as a programmer would work from in the IT industry; detailed design processes and sample documentation and coding – illustrated with screenshots of the working applications.
Contents

Topic 1 Object-oriented programming .................................................. 1

Module 1 Basic object-oriented terminology ..................................... 3
  Unit 1.1 Basic principles of a class ................................................. 4
  Unit 1.2 Basic principles of an object ........................................... 7
  Unit 1.3 Basic principles of information hiding and encapsulation ..... 8
  Unit 1.4 Basic principles of inheritance ......................................... 10
  Unit 1.5 Basic principles of polymorphism .................................... 12

Module 2 Differences between object-oriented and procedural programming ... 14
  Unit 2.1 Implementing classes in object-oriented programming ........ 15
  Unit 2.2 Encapsulation of data and functions .................................. 24
  Unit 2.3 Global data sharing, weak coupling and functional cohesion .. 35
  Unit 2.4 Explain and identify possible classes for simple examples and problems .................................................. 35

Module 3 Implementing object-oriented techniques ............................. 41
  Unit 3.1 Identify objects and classes .............................................. 41
  Unit 3.2 Implement objects and classes in a solution ....................... 46

Topic 2 Database application design ............................................... 71

Module 4 Database access requirements for a computer programming solution . 73
  Unit 4.1 Review of the feasibility of the requirements ....................... 74
  Unit 4.2 Database access objectives and critical performance factors .... 76
  Unit 4.3 Estimation of costs for the development effort required .......... 80
  Unit 4.4 Review procedure to ensure that the outcomes meet the database access requirements .................................................. 84

Module 5 Database access design for a computer application ................... 86
  Unit 5.1 Implementation of the design per user requirements .............. 86
  Unit 5.2 Design of the database structure ...................................... 88
  Unit 5.3 The third normal form .................................................... 90
  Unit 5.4 Methods of accessing data .............................................. 103
  Unit 5.5 The key relationships between the tables in the database ....... 106
  Unit 5.6 Data types for primary and foreign .................................. 107

Module 6 Program code for SQL database access ................................ 109
  Unit 6.1 External data connection and access ................................ 109
  Unit 6.2 Programming language constructs ................................... 121
  Unit 6.3 Joining tables in a query ............................................... 143
  Unit 6.4 Data integrity in a multi-use, multi-process environment ...... 158

Module 7 Testing database applications .......................................... 179
  Unit 7.1 Testing program logic .................................................... 180
  Unit 7.2 Identifying and correcting errors ..................................... 182
  Unit 7.3 Verifying database access ............................................. 183

Module 8 Documentation of computer application programs .................... 187
  Unit 8.1 Program code documentation .......................................... 187
  Unit 8.2 Documentation and the self-documenting attributes of the program code .................................................. 188
  Unit 8.3 Documentation to support the design, program code and the solution .................................................. 189

Topic 3 Principles of developing software for the Internet ....................... 195

Module 9 Network issues related to Internet development ...................... 197
  Unit 9.1 The Internet and sessionless network protocols .................... 198
  Unit 9.2 Sessionless application development ................................ 200
  Unit 9.3 The impact of bandwidth on Internet usage and data transfer ... 202
  Unit 9.4 The implications of bandwidth for application design ........... 205
Topic 1
Object-oriented programming
Object-oriented programming

Introduction

Object-oriented (OO) programming is a style of programming that attempts to represent real-world problems in terms of objects. Objects ‘talk to’ or interact with other objects and tell each other to do things. The objects communicate with each other by sending messages, and carry out their instructions using methods (Subs and Functions in Visual Basic). Objects also have properties that describe its characteristics.

For example, a lecturer, Ms Sifuba, could be an object of type Teacher, and this object would have a name (MsSifuba), an ID number, an address, and so on. The Teacher object would have a set of methods, such as doRollCall, markAssignments, setExamPapers, and so on. Remember, a method describes something that the object does.

Object-oriented programming is very popular, because it allows code to be re-used easily and safely, and make programs more modular and understandable. These benefits also help to reduce programming development costs and development times. There is an ongoing debate about whether object-oriented programming is better than procedural programming. Both styles have the benefits and weaknesses, and so each has its place in the programming world. Similarly, you will also read heated arguments about which OO language is the best. Is it Java, or Visual Basic, or Ruby on Rails, or Delphi? Each language has its followers and they will never agree with each other.

We have chosen Visual Basic .NET 2008 in this book, not because it is necessarily the best OO language, but because:

- it is widely used
- Visual Basic programmers are always in demand
- Microsoft will continue investing money in Visual Basic and keeping up with its competitors
- Visual Basic is easy and fun to use.

So don’t worry about the choice of language. Once you have learnt one OO language, you will find it easy to pick up another, if you need to.
Module 1

Basic object-oriented terminology

Overview
At the end of this module, you will be able to:
• explain the basic principles of a class
• explain the basic principles of an object
• explain the basic principles of information hiding and encapsulation
• explain the basic principles of inheritance
• explain the basic principles of polymorphism.

Introduction
Object-oriented programming is not new. It was already being used in 1967, in a language called Simula67. Today, most program development uses object-oriented techniques using languages such as Java, C++, and Visual Basic .NET (VB .NET). The reason why it is still being used is because it is so effective. OO techniques allow you to produce better programs that are more robust, easier to maintain and quicker and cheaper to produce. The many ready-made objects built into the languages or available on the Internet increase these benefits even further.

In order to understand OO programming and to use OO techniques, you first need to become familiar with the terminology. Some of the terms you will learn are familiar words that have slightly different meanings in the OO world. For example, a class defines a set of objects and the properties that the objects possess. An object is an instance of a class. Objects can inherit properties from their class or from one another, saving you the trouble of repeatedly adding properties to new objects. Objects have methods associated with them. These methods define how objects carry out actions. Objects also store data, which is hidden inside the object. We call this encapsulation. The data in the object can only be accessed by using the methods of the object. So encapsulation means you don’t have to worry protecting the data from misuse. The methods take care of that for you. Finally, when an object is asked to do something, such as print some data, it’s smart enough to understand what to do and how to do it. For example, there might be lots of different data types and structures inside the object, but you don’t need to worry about that. All you need to do is ask the object to print the necessary data, and the object knows what to do. We call this feature of objects polymorphism.
So, as you can see, there is a lot to learn, but don’t worry, you will get it!
Unit 1.1: Basic principles of a class

At the end of this unit, you will be able to explain the basic principles of a class.

Classification and abstraction

Everyone knows how to classify things and the OO use of the word classification is the natural one that you are used to. In OO terminology, we refer to the category we classify things into as a **class**. For example, if you were asked to classify a list of things, such as a sausage, a potato and a bread roll, you could say they are all types of **Food**. In this example, **Food** is a class and it contains the **objects** Sausage, Potato and Bread roll.

The class called **Food** has a number of properties that apply to all the specific examples in the category. Therefore, we say that the class is an **abstraction** of the specific foods Sausage, Potato and Bread roll. In other words, **Food** is a more general description of Sausage, Potato and Bread roll.

You could add many more food objects to the class, such as Rice, Lamb chops and Brown bread. Can you see that, even though they are different types of food, they all share common characteristics, and so fall under the class called **Food**?

You might also want to have more detailed abstraction levels than simply **Food**. For example, you might have classes such as Meat, Fruit or Vegetables. All of these fall under the general class of **Food**. The result is that you have a **hierarchy** of classes, each becoming more specific as you move down the hierarchy. We use a tool called a **class hierarchy diagram** to show how all these classes are related to each other. Figure 1.1 shows an example of a class hierarchy diagram. The diagram in Figure 1.1 shows the classes becoming more specific as you read it from left to right. But it’s also perfectly valid to show it vertically, with the more general classes at the top. In this case, the diagram would resemble a family tree.

**Class**: A class defines the nature of the objects that belong to it. For example, Student is a class, and you, a unique student, are an object of that class.

**Object**: An object is an example of a class. For example, Thandi is an object of the class Student.

**Abstraction**: The process of identifying and extracting the common parts of classes and summarising them in a higher level class. For example, Food is an abstraction of Bread, Meat, Fruit, and so on.

**Hierarchy**: A ranking system that groups classes as being above (more general), below (more specific) or on the same level (independent of each other).

**Class hierarchy diagram**: Connected classes can be represented in a class hierarchy diagram, which shows how classes are related to each other.

**Unified Modelling Language (UML)**: A diagrammatic way of describing OO systems and structures.

The notation used for these diagrams is the **Unified Modelling Language (UML)**. You can find out more about UML on the Internet. Using a search engine, such as Google, type in ‘uml primer’ or ‘beginner’s guide’, and you will find many websites that describe UML. One good example of a website that describes UML is http://cswww.essex.ac.uk/staff/sml/UMLPrimer.html.
You will come across different terminologies to describe the relationship between classes in a class hierarchy. For example, the Food class is the parent of the Meat, Vegetable and Fruit classes. The Meat class is a child of the Food class and the Lamb chop class is a child of the Meat class. Another way to describe the relationships is to say that Food is the super-class of the Meat, Vegetable and Fruit classes. In our example, Apple is a sub-class of Fruit.

Information stored in a class

We can group the information stored in a class into three parts, as shown in the UML diagram in Figure 1.2. The three parts are:

- the class name
- the class attributes or properties
- the class methods.

The attributes of a class define the data that you will store in the class. In the example shown in Figure 1.2, the Fruit class has two attributes. When we create an object based on this class, we’ll assign a value to each of these attributes. So if we create an object for an apple, we will assign the value ‘Apple’ to the Name attribute, and the value ‘Deciduous’ to the Type attribute. Similarly, we can create an object for an orange, and assign the values ‘Orange’ and ‘Citrus’ to the two attributes. Both objects are based on the same class, and so we say that the objects are instances of the class.

The methods of the class describe what the class is able to do and the functions that it can perform. In our example, the Fruit class has three methods, namely getName, getType and PrintDetails. The first two methods enable your program to ask an object what the values are that are stored in each of its attributes. The PrintDetails method lets you send a command to the object telling it to print out all the stored information.

The methods are programmed as functions or subroutines and can be called or invoked from other objects.

Generally, the programmer would also add some descriptive comments to the class coding. The comments would describe the purpose of the class, its constraints, and how to use and maintain it. This forms part of the program documentation.
How does a class hierarchy help you as a programmer? The answer is that it stops you having to add the same properties to each class on the same level in the diagram. For example, the class hierarchy in Figure 1.1 shows three sub-classes of Food, namely Meat, Vegetables and Fruit. Each of the sub-classes will have some properties and methods in common, such as a Name and Type property. Using a class hierarchy, you simply add the common properties to the parent class, and all the children can inherit these properties automatically. This saves time and prevents mistakes.

Case Study 1.0 Part 1 (Employee records)
We can develop another example from a company’s information systems. A company has employees and some of the employees are managers and some are workers. Your lecturer will give you a copy of the full Case Study 1.0 and will work through it with you as you progress through this topic. Have a look at it now. So we could have a class for Employee with sub-classes Worker and Manager. The two sub-classes could inherit many attributes from the parent class Employee; such as salary, annualBonus. They could also inherit some methods, for instance; showAddress, showCV. These methods would print or display the address and CV of a particular person (object). Objects created or instantiated from these classes would be particular employees and managers. So the Phindiwe object could be the Human Resources manager and Jacob and Viwe could be two of her employees. You will see later in this module how to code classes, objects and methods in Visual Basic.

Class access specifiers
Class access specifiers or modifiers are used to control access to your data attributes and methods. For instance, you might want to protect the data of objects belonging to a class so that only other objects in that class can access the data, to do this you would use the class access specifier Private. Some other access specifiers are: Public, Protected, Friend, Protected Friend, Static, Shared and ReadOnly, These are described in Unit 2.1.

Case Study 1.1 (PhoneNet) Part 1
We will be using a case study in this topic to give you some practice in applying the theory to the development of a practical system. Your lecturer will give you the required information as you go along. Before doing the assessment activity below, ask your lecturer to go through the Case Study 1.1 Part 1 notes with you up to the class definition point.

Assessment activity 1.1
(Group work, discussion activity, lecturer assessed) Total marks [21]
Discuss these questions in your class.

Question 1 (21 marks, three marks per question)
(a) What is a class? Give three examples of a class.
(b) What is an object? (You will need to use what you have gathered so far).
At the end of this unit, you will be able to explain the basic principles of an object.

**Objects**

An object is a specific instance of a class. So, if we consider pets as an example, then we have:

- A *Cat* is a class, a *Tabby* is a sub-class of the *Cat* class and *Fluffy* (which might be the name of your pet tabby cat) is an object. *Tabby* can be abstracted to the class *Cat*.
- The *Fluffy* object is an instance of the class *Tabby*.
- The *object* is the real thing, whereas the *class* that the object belongs to defines what the object is and what its characteristics are.

Objects can represent anything you wish to store information about.

An object is fully described by its *attributes*. In our previous example:

- the object has a *Name* attribute, which will have a value, such as *Fluffy*
- there is a *class* that the object is based on, for example, *Tabby*
- the *state* of the object, such as what it is doing right now, such as sleeping or playing or eating
- the behaviour of the object, which defines how it interacts with other objects.

In your programs, objects are designed to be modules that can exist on their own, having their own data and process logic (information and operations or behaviour).

Think of objects as living things or intelligent robots, with memory (data) and rules of behaviour (methods or logic). If you design and code an object well, you can then interact with that object in your program without having to worry about the details of how the object was designed. You can even use someone else’s objects, provided you trust them to have done a good job! This saves time and effort and your programs and systems will be robust and have few bugs. An object definition diagram is a graphical tool we use to define an object, and is similar to a class diagram. Figure 1.3 shows an example of an object definition diagram.

c) What is meant by parent and child classes? Give three examples.
d) Give other terms for parent and child classes.
e) Give three examples of abstraction using the classes in your answers above.
f) List six examples of information that are stored in a class and explain those using examples from your answers above.
g) Using UML, draw a class hierarchy diagram to represent your answer in e).
(Pair work, written activity, peer assessed) Write down the answers to these questions, working in pairs.

**Question 1** (25 marks, five marks per question)

a) Define an object and give five examples of objects.

b) What are the differences between an object and a class?

c) Classify the following items into classes and objects:
   - Boy, Girl, Boat, Emily, Person, John, Rudder, Engine, Free Life (name of a boat)

d) Draw a class hierarchy diagram of the classes in c).

e) Draw an object definition diagram for one of the objects in c).

---

**Unit 1.3: Basic principles of information hiding and encapsulation**

At the end of this unit, you will be able to explain the basic principles of information hiding and encapsulation.

In OO terms, the words **information hiding** and **encapsulation** mean the same thing, and are used interchangeably. In this book, we will use the term encapsulation. As O’Docherty says, ‘encapsulation is about an object hiding its attributes behind its methods by sealing its attributes in a capsule’. He also says ‘we can think of objects being courteous to one another, in the same way that if you wanted to borrow money from someone, you wouldn’t take their wallet and pull out the money you needed, you would ask them instead and they would look in the wallet, take out the cash and give it to you’ (2005, 20-21).

The main benefit of encapsulation is that when you need to change the internal working or representation of an object, you make the change in one place only, and you can be sure that the rest of the program and other objects will still be able to access it correctly. However, if your program accessed the object’s attributes directly, instead of through the methods of the object, then you would have to check through the entire system for references to any of the attributes, and make sure they were still being used correctly after the change.

Encapsulation also protects the internal data of a class, by only allowing the data to be accessed or changed through the methods or interfaces of the class. For example, by using a method to store a value in an attribute, the object can make sure that the value is a valid one. Your program can simply ask to store a value, and if it’s not valid, the object will reject it, hopefully with a useful message that explains why so that you can correct the error.
Case Study 1.0 (Employee records) Part 2

Refer to the Case Study 1.0 notes that your lecturer will give you.

A practical example of encapsulation or data hiding can be found in the Employee, Worker, Manager application described previously. Sensitive information such as a person’s salary would have to be carefully managed, both in terms of visibility (generally you wouldn’t allow everyone to see someone’s salary) and in terms of changing it (usually, only authorised people in the Human Resources department would be allowed to change salaries and then only as part of a formal process). The salary information would be declared as Private and the only way to change it would be by using a special salary update method, which could be called updateSalary. The access to the updateSalary method would also be restricted. In this way the salary information would be hidden and protected from everyone except a few specified people.

Assessment activity 1.3

(Group work, practical activity, peer assessed) Total marks [60]

Form groups of three. Work out your answers together, write them down, and then discuss your answers in the class with the other groups.

Question 1 (30 marks, 10 marks per question)
Consider an object called searchSong that tells you the title of a song when you give it a section of the lyrics. Internally it would take the text string of the lyrics you are trying to match, and it would search a database to find a match. Then it would send you the title of the song in another text string.

a) Construct an object diagram for the searchSong object, showing the attributes that it would need for accepting the lyric text string, and then for returning the title text string. Also show the search method.

b) Describe what has been encapsulated, and how.

c) Assume that you decide to improve the object’s functionality by adding more song databases. How would this affect users of the object, and what changes, if any, would they have to make in their messages to your searchSong object? How has encapsulation helped here?

Question 2 (30 marks)

a) Now suppose you want your object to use Google to search the Internet for lyric matches. How would this affect the users of your object? How has encapsulation helped here? (20 marks)

b) In a) do you think you would have to pay Google Inc. to use Google in your application? (10 marks)

In the workplace

In practice, instead of writing the searchSong object yourself, you could first search the Internet for suitable, ready-made objects and use one of those. Note that you might have to get the object owner’s permission to use the object. The owner might require you to acknowledge your source publicly, or you might have to pay for the right to use the object. However, don’t just steal objects; that is piracy!
Unit 1.4: Basic principles of inheritance

At the end of this unit, you will be able to explain the basic principles of inheritance.

Inheritance in OO can be complex, and a full treatment is beyond the scope of this book. If you want to dig deeper, there are many good references, such as Mike O’Docherty’s book (2005, 54-760). However, we will cover the basics here.

Inheritance allows you to derive new classes from a base class. The new classes, called derived classes, can share some or all of the capabilities of the base class. This allows the programmer to re-use common program code, thus saving time and money and reducing program bugs. Inheritance also allows you to use a base class coded by someone else and add additional functionality to it, then, if the base class is improved later on, your inherited classes will automatically get the benefit of the improvements.

Inheritance is the ability of a sub-class to automatically have access to the attributes, methods and relationships of its super-class. This helps in the reuse of code, in that you only need to code the details in the super-class, and then any sub-classes can use them. Consider the class diagram in Figure 1.4.

In our example, the super-class is Aircraft and it has an attribute of maxPeople (the maximum number of people that the aircraft can carry). There are also two methods. The first, getMaxPeople, lets your program ask an object based on the class for the value stored in maxPeople. The other method, setMaxPeople, lets your program give the object a value to store in the attribute. So you can use these methods to read and set the value.

The sub-classes of JetLiner, Helicopter and ParaGlider have each inherited the attributes and methods from their super-class.

In addition, because of the special requirements of the JetLiner, Helicopter and ParaGlider classes, different, new attributes and methods have been added to each of them. For example, for the JetLiner the runway length might be important, while for the ParaGlider the height above the ground that it can fly (the altitude) could be important.

So, for example, the Helicopter object has two attributes, namely maxPeople and maxRange. The first attribute is inherited from the parent class (super-class). Similarly, the Helicopter object also has four methods, two that are defined in the class itself, and two that it has inherited from the parent class.

An important principle in class design is to push common attributes, messages and methods as high up the hierarchy as possible so as to get the maximum advantage of code reuse from inheritance. However, if you find that this means that some sub-classes have elements that they don’t need or can’t use, then you have probably pushed those elements too far up the hierarchy.
As another example of inheritance, consider a system describing bicycles. There are many different kinds of bicycles and they share some common behaviours and useful information, but they each might have some unique features that are not found in all type of bicycles. Let’s call the base class Bicycle and some of its possible sub-classes, RoadBike, RacingBike, MountainBike and TandemBike.

You would then define all the common things in the Bicycle class, such as Price, Size and GearType. These would be automatically inherited by the sub-classes of Bicycle, so you would not need to declare the Price, Size and GearType variables in the RoadBike, RacingBike, MountainBike and TandemBike classes. They would be there automatically through inheritance.

In each sub-class you would put the things that were unique to that class. The TandemBike might have a maximum weight restriction on its two riders, while the other sub-classes might not.

Case Study 1.0 (Employee records) Part 3
Your lecturer will have given you the notes on this case study.

Our business example with the Employee, Worker and Manager classes can also make use of inheritance. There are many data attributes and methods that are common to workers and managers and these can all be defined in the parent class, Employee, as shown below.

Common data attributes:
- Name
- Office address
- Office phone number
- Salary

Common methods and functions:
- Update name and address details
- Show name and address details
- and display salary (this will have restricted access as discussed in Unit 1.3 – encapsulation)

Note that not everything could be inherited by the Worker and Manager classes because of the differences between these classes. For instance, managers might be given company cars and workers not, so data and methods to do with company cars would only be in the Manager class.

(Pair work, written practical activity, self assessed) Total marks [100]
Choose a partner and answer these questions together. After the exercise, your lecturer will give you sample answers so you can assess your work.

Question 1 (15 marks, 5 marks per question)
(a) What is meant by inheritance in OO?
(b) Give three advantages of inheritance.
(c) How do you decide where in a class hierarchy you should place particular attributes and methods? Give an example to show your reasoning.
Question 2 (45 marks)
You are required to produce a class hierarchy diagram, showing sensible inheritance for a system that calculates the surface areas of the following shapes:
- a square
- a rectangle
- a triangle
- a circle
- a sphere.
Your diagram should show:
- class names
- methods (such as `getArea` and `calcArea`)

Question 3 (40 marks)
You are required to produce a class hierarchy diagram for the *Employee, Worker, Manager* example discussed above (Case Study 1.0).
The data attributes should include:
- Name and address details
- Salary
- Company car details
The methods should include:
- Update name and address details
- Display name and address details
- Update salary parameters
- Show salary
- Update company car details
- Show company car details
a) Show the data attributes and methods in the *Employee* class that can be inherited by the *Worker* and *Manager* classes.
b) Also show data attributes and methods in the sub-classes that shouldn’t be inherited from the *Employee* class.

Unit 1.5: Basic principles of polymorphism

At the end of this unit, you will be able to explain the basic principles of polymorphism.

The word polymorphism is derived from ‘poly’, meaning many and ‘morph’, meaning change. In OO terms, polymorphism is the ability of an object to understand a message sent to it by another object according to rules defined in the receiving object. Polymorphism also enables the calling object to send a message or instruction to another object to do something without having to know anything about the receiving object (other than that the message concerned is defined in the receiving object).

Another way to explain polymorphism is that it allows you to have different classes which have different methods or functionality, but these methods can all have the same name. Other objects could then execute these methods using the same name. This might sound a bit odd, but it can be useful. Consider the example of a financial system, which produces invoices, receipts, bank statements, price lists and general account reports. The system would need to be able to print any of these and, in most cases, some processing would have to be carried out before the printing can be done. In this case, it simplifies the programming if you can just say PRINT to whichever object you want printed and it will do all the necessary
processing and print the reports. The actual Visual Basic command would be something like `CurrentInvoice.Print`, which would invoke the `Print` method of the `CurrentInvoice` object.

Consider our `Aircraft` class hierarchy example in Figure 1.4. We could add a new method to `Aircraft` called `takeOff`. Then we would implement the `takeOff` method in the sub-classes in ways that are appropriate for each sub-class. The process of taking off is very different for a `JetLiner`, `Helicopter` and `ParaGlider`. With a jet aircraft, the pilot would go through the test procedures, get clearance from the control tower and then release the brakes, get onto the correct runway, open the throttles and when the plane has reached the correct speed, it would be able to take off. With a helicopter, the pilot would go through the test procedures, get control tower clearance, open the throttle and pull back on the collective control to change the rotor pitch, and the aircraft would lift into the air. With a paraglider (which is like a big kite that the pilot is strapped to), the pilot would check the glider set up and the pilot harness, check the wind speed and direction, and then run into the wind and jump off the cliff edge! So now, each sub-class has a method called `takeOff`, but each implements it differently, according to its needs. But from the point of view of a program using these objects, all they need to know is that there is a method called `takeOff`.

The advantages of polymorphism are that the calling object doesn’t need to know anything about the sub-classes and associated objects. It can just issue the `takeOff` message and the receiving object will do the right thing. Also, if you add a new object at a later stage (for example, a seaplane), the calling objects don’t need to know how a seaplane takes off either, because that knowledge will be built into the `SeaPlane` object. This makes it easier to maintain and change programs and systems, and reduces costs and program bugs.

Note that the name of the message must be the same in all the classes and objects that use the polymorphic message.

---

**Assessment activity 1.5**

**Total marks [60]**

Write down the answers to these questions.

**Question 1 (40 marks)**

a) Define polymorphism, and give an example based on objects you are familiar with. (15 marks)

b) Draw a class hierarchy for your example, showing the messages and short descriptions of how the associated methods could be implemented. (15 marks)

c) What are the advantages of polymorphism? Give two examples. (10 marks)

**Question 2 (20 marks)**

Using the details (Case Study 1.0 Part 4) given in Assessment Activity 1.4 Question 3, about the Employee, Worker, Manager application; demonstrate the use of polymorphism to cater for salary calculations for the workers and managers. You will need to add methods to calculate salary (the calculation method is different for managers and workers). Draw the revised class hierarchy diagram showing the polymorphic methods.
Module 2

Differences between object-oriented and procedural programming

Overview
At the end of this module, you will be able to:
- explain the implementation of classes in object-oriented programming, using examples
- explain encapsulation of data and functions/methods (in classes)
- describe how global data sharing is minimised to enable weak coupling, and modules exhibit functional cohesion
- explain and identify possible classes for simple examples and problems.

Introduction
In the previous course, Principles of Computer Programming NQF Level 3, you learnt the Visual Basic language. Visual Basic is an OO language. For example, the Visual Basic controls you used, such as textboxes, are objects. In this course, we will again be using Visual Basic, but this time we will be using the .NET version, Visual Basic 2008 Express Edition. This is a fully OO language with a rich set of classes and objects for you to use in developing systems.

In this module, we will explore classes and encapsulation of data/functions/methods in more detail, and explain the differences between OO and procedural programming.

A procedural language, such as Pascal, keeps the data separate from the program. In procedural programming, the programmer breaks down the overall program or task into a set of procedures using variables and data structures. Usually, a top-level calling program calls the procedures (which are like subroutines or functions) in the desired order. The logic of the program is contained in the calling program and within each procedure. You can still get the benefits of code reuse by reusing your procedures wherever you can. However, it is difficult to use other people’s procedures because of their dependency on variable names and data structures.

On the other hand, OO programming focuses on classes and objects. Good design of the classes and objects allow efficient reuse of objects without having to know too much about how they work. This means you can safely and easily use objects written by other programmers.
OO programs are also easier to maintain and change, because you can change or add functionality within the methods of an object without having to make many alterations in the rest of the system to accommodate your changes. Objects are more self-contained and self-sufficient than procedures or subroutines.

**Unit 2.1: Implementing classes in object-oriented programming**

At the end of this unit, you will be able to explain how to implement classes in object-oriented programming, using examples.

You learnt about classes and objects in Module 1. We will now do an actual example of a class in Visual Basic to bring the definitions to life. The example is taken from Microsoft’s MSDN library, which is an excellent source of technical information, coding examples and general help. Access to MSDN Express is also free, as is Visual Basic Express 2008. Your lecturer will ensure that you have access to these. Google is also an excellent resource, and help you to you solve problems and point you to good sources of information. For example, if you search for ‘creating classes in Visual Basic .Net’, you will find the relevant pages in the MSDN library that we will go through here.

Note that in terms of copyright and licensing (you will learn about this later in this course), Microsoft makes the Express Edition material available under the Creative Commons licensing arrangements, which means we are free to use it in this book.

The example we have chosen is a class and an associated object that allows you to break up a sentence of text in a string into the separate words. It is a simple example aimed at showing you how the code works.

Are you ready? Switch on your computer, open Visual Basic and let’s get started.

Open your browser as well (Internet Explorer or Firefox or whatever you are used to using) and navigate to the address http://msdn.microsoft.com/en-us/library/ms973814.aspx. Ignore the bits about the differences between Visual Basic .NET and Visual Basic 6, as they are not relevant now. Just read the overview and do the examples.

**Example 2.1 Create a Class**

2. Under ‘Recent Projects’ on the left-hand of the screen, select ‘Create’ and the ‘New Project’ window will open up.
3. Select ‘Windows Forms Application’ as the project template. Set the name of the application to ClassCreation at the foot of the screen and click on OK. The Form Design screen will open.
4. In the Solution Explorer window on the right-hand of the screen, select the `Form1.vb` form and rename it to `frmLineTest.vb` in the Solution Explorer window. You can do this by right-clicking the form and choosing ‘Rename’ on the shortcut menu.

5. Create the form shown in Figure 1.5 by adding the appropriate controls and setting the properties of those controls, as outlined in Table 2 in MSDN. You might have to open the Toolbox on the extreme left-hand side of the screen. Once you have access to the Toolbox, select ‘Common Controls’ to get the controls you need (Label, TextBox, CommandButton).

**Saving your work**

At this stage, we need to briefly discuss how Visual Basic stores its many files, and the easiest way to manage your projects and Assessment Activities.

When you want to start working in Visual Basic, it forces you to name the various components and will provide default names for you. It is good practice to override the default names and put in your own names so you can identify your projects and work. It is also good to adopt a simple set of **naming conventions** for projects, classes, forms and variables. Please refer to Appendix 1 for a suggested set of naming conventions. However, your lecturer might already be using different naming conventions, which is fine.

**Visual Basic files and folders**

For the discussion below, assume that the project name is `ClassCreation` and the form name is `frmLineTest`. Visual Basic uses a parent folder for the project, and a set of subfolders for the different parts of the project. The parent folder is created and named by you. Use the project name for the folder (in this case, `ClassCreation`). In the parent folder, `ClassCreation`, Visual Basic creates three items:

- a subfolder, also called `ClassCreation`
- `ClassCreation`, a Microsoft Visual studio solution file
- `ClassCreation`, a Visual Studio solution user options file.

In the subfolder, `ClassCreation`, there are several files and folders:

- the `bin` folder
- the `My Project` folder
- the `obj` folder
- `ClassCreation`, a Visual Basic project file
- `ClassCreation.vbproj`, a user file
- `frmLineTest`, a .NET managed resource file
- `frmLineTest.Designer.vb`, a Visual Basic source file (contains your form details)
- `frmLineTest.vb`, a Visual Basic source file (contains the event coding for your form)
- Line.vb, a Visual Basic source file (contains the method coding for the Line method).

The important thing to note is that you must never fiddle with these folders and files outside of Visual Basic, otherwise you will get many confusing error messages and your programs won’t work! If you do have to copy these folders (for example, to make a backup), copy the highest level folder (the parent folder you created initially) and all its subfolders.

**Controlling where Visual Basic puts its files and folders**

Your lecturer will tell you where you should save your work. Perhaps you will be given a special folder on a shared network drive, or perhaps you will save them in a special folder on your computer.

To save your project, the easiest and safest way is to click on the ‘File’ menu and then select ‘Save All’ (or use the ‘Save All’ icon at the top of the Visual Basic screen). The first time you save a project with ‘Save All’, Visual Basic doesn’t know where you want to save the project, so the program will allow you to select a project name and a path to a folder on the computer. When you click on ‘Save All’, Visual Basic will present you with a dialogue box similar to Figure 1.6 below.

![Figure 1.6 Saving a project for the first time](image)

The project name will already be filled in (and so will the solution name), but you can change them if required.

The location will usually have to be changed to the path that your lecturer gives you. To change it, click on the ‘Browse’ button and navigate your way to the correct folder. Then click ‘Save’.

The next time you want to save your work (and you should do this often!), simply click on the ‘Save All’ icon and your work will be saved in the correct place.

Before continuing with this example, please save your work as described above and exit from Visual Basic. Then reopen Visual Basic and make sure you can find and load your previously saved project (‘File’, ‘Open Project’ and then find your ClassCreation project and load it by clicking on the ClassCreation solution file). Another method is to click on the ‘Most recent project’, if that is the correct one.
Creating the *Line* class

1. Open the ‘Add New Item’ dialog box by clicking ‘Project’ and then clicking ‘Add Class’.
2. Set the ‘Name’ property to *Line*, and click OK.
3. This creates a new file called *Line.vb*. It is displayed in the code window of Visual Basic.

```
Public Class Line
End Class
```

You must enter all the properties and methods that you need for this class between these lines of code.

Note that in the *Public Class* statement, *Public* is called an access modifier or specifier in Visual Basic. There are many different access modifiers that you can use.

**Access modifiers or specifiers**

These control how variables, methods and classes can be used and from where they can be used.

*Public* means that there is no restriction on access. This should be used with caution; it is good practice to limit access as much as possible while still allowing your application to work. This reduces the risk of accidental misuse of your data and methods in your classes.

*Private* is usually applied to variables and means that only other objects within the class containing the variable can access it. You would use this when you want the variables within a class to be protected against access from external objects.

*Protected* is used to restrict access to only those objects within the same class or from a derived class (so that inherited features can be accessible only from within the family of the base class and its derived sub-classes).

*Friend* is used when you want all the classes in a project to have access and to prevent access from classes outside your project. You can’t do this with *Private* or *Protected* and that is why the *Friend* concept was introduced. The *Friend* modifier can be applied to classes and variables.

*Protected Friend* is a combination of *Protected* and *Friend* access specifiers and allows access to class members in the same project and all the inherited classes.

*Static* variables will keep their value even when the procedure (e.g. *Sub*) in which they were defined stops running. Normally, local variables in a procedure disappear once the procedure terminates – this is to save memory resources.
Shared variables of a class are shared across all objects created from that class. Normally variables in an object are part of that particular instance only. This is useful when you want a variable and its current value to be accessible across the whole application. If one instance of the class changes the value of the variable, then all other instances see the same, changed value.

ReadOnly variables are similar to Const variables in that you assign a value once and then nothing can change it. ReadOnly would be used when you can’t use Const, for instance Const sets the value of a variable at compile time, but you might only know the value at run time – you would then use ReadOnly.

Creating properties

Data can be stored in an object using variables as you would in non-object oriented programming, but data can also be stored in the properties of an object. Using properties is a good way of achieving encapsulation, because the actual variables can then be completely hidden within the object and made inaccessible by using the Private access specifier.

The best way to create properties within a class is use the Property statement and Private variables.

Doing it this way allows you to:
- control whether you want the property to be read-only, write-only or read-write
- implement error-handling within the Property statement
- store calculated values in properties (as we will do for the Length property).

For our example, we will need two properties for the Line class, namely LineText and LengthText. First, you need to create a private variable called mstrLine in which you will store the line of text to be analysed and split up. Then you can add the Property statements.

```vbnet
Public Class Line
    Private mstrLine As String

    Property LineText() As String
        Get
            Return mstrLine
        End Get
        Set (ByVal Value As String)
            mstrLine = Value
        End Set
    End Property

    ReadOnly Property LengthText() As Integer
        Get
            Return mstrLine.Length
        End Get
    End Property
End Class
```
Note that, as you start typing in code in the code window, the Visual Basic Integrated Development Environment (IDE), which is Visual Studio, assists you by suggesting what should come next, and adding lines of code automatically when it knows you must have them. As you start typing, Visual Basic displays a drop-down list, and you can select the item you want by pressing the Tab key to accept your selection. Visual Studio also automatically indents the code for you. This is the Intellisense feature of Visual Studio. You will find it extremely helpful, because it means that you don’t need to remember or continually look up the methods and properties associated with a particular object.

Creating an event handler

You have now created a class and a form. To actually use the class and test it, you need to enter data into the lineText variable, and the length of the data into the lengthText property.

This is how you do it. You will probably remember most of this from your studies last year.

1. Go to the ‘Solution Explorer’ window and double-click on frmLine.vb. This will put you into ‘Design Mode’ again.
2. Double-click on the DisplayLength button. Visual Basic creates an event handler template for you, and puts you back into the code window, ready for you to add your code to the event handler.
3. Now, add the code as below.

```vbnet
Public Class frmLineTest

Private Sub btnGetWord_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnGetWord.Click

End Sub

Private Sub btnDisplay_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnDisplay.Click

Dim oLine As Line 'Creates the oLine variable in the Line class
oLine = New Line() 'Instantiates an object in the class Line
'Put the text in the textbox into LineText
oLine.LineText = txtLine.Text
'Convert the length value into a string and put it into the txtLength textbox
txtLength.Text = oLine.lengthText.ToString

End Sub
End Class
```

Have a look at the comments in the code to see what the code does (unbolded text).
Testing

Having saved your ClassCreation project, you now test it by pressing F5. The LineClass Test form will display and wait for you to do something. Try pressing the Display Length button. What happens? The integer 21 should appear in the text box to the right of the Display Length button. Count the number of letters (and spaces!) in the text string ‘How to create a class’. It is 21!
Now try typing some other text string into the top text box, and press the Display Length button again. Check that the new number displayed is correct.

Adding a new method to your class

Methods can be public or private. Private methods can only be called by other methods within the same class. In our example, we want to be able to call a method from a form, so we must use the Public keyword.

A method can be a procedure that carries out some operations on the data in the class, or it can be a function that does something and returns data to the calling object. A procedure does not return a value to the calling object. We will use a function here. This is how to do it:

1. Click on the Line.vb tab at the top of your centre window on your Visual Basic screen. This puts you into code view in the Line class.
2. Now add the code as below just after the end of the last of the Property blocks of code.

```vbnet
Public Function GetWord() As String
    Dim astrWords() As String

    ' astrWords is the array to hold the split up text
    ' Split splits the string into an array of words
    astrWords = Split(mstrLine, " ")

    ' Return the contents of the FIRST word in the array (using the zeroith element)
    Return astrWords(0)
End Function
```

Words & Terms

Procedure: In Visual Basic terms, a procedure is a Sub or subroutine that doesn’t have a value attached to its name, although you can pass values to it as parameters.

Function: In Visual Basic terms, a function is a similar to a Sub, but it can have a value attached to its name, and you can also pass values to it as parameters. The value assigned to the name of function is the value returned to the object that called it.
Calling the new method

As usual, get into design view of your form and double-click on the Get word button. Visual Basic will create an event handler and put you into code view.

Add the code to call the GetWord method as below:

```vbnet
Private Sub btnGetWord_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnGetWord.Click
    'Create the variable and an instance of the Line class
    Dim oLine As New Line()

    'Put the contents of the txtLine textbox into the LineText property
    oLine.LineText = txtLine.Text
    txtWord.Text = oLine.GetWord() 'Calls the GetWord method
End Sub
```

Now save your work and press F5 to run your application. Try entering different strings of text and pressing the two buttons.

Remember, the benefits of classes are:

- they make safe and efficient reuse of code easier to implement
- encapsulating calculations and operations within a class hides complexity and lets you concentrate on the real design issues in your program
- you can make changes within a class without necessarily affecting the users of the class
- you can use inheritance to create new classes from existing classes and add new functionality without having to duplicate all the existing functionality and change the original class.

Congratulations, you have created and tested your first class!
(Pair work, practical activity, lecturer assessed)  

Choose a partner, open Visual Basic and go through the practical exercise below. Take turns to do the actual coding. Remember to put comments in the code to explain what the code does. And save your work often!

**Question 1** (20 marks)  
You are going to extend the functionality of Example 2.1.  
The requirement is to split the line of text and show the separate words on the form instead of just the first word. There are many ways to do this, but to keep it simple at this stage, you should allow for the program to display the first six words of the line of text.

Before you start working on this, you need to get a new folder ready to take a copy of the ClassCreation folder. This will be your starting point for the exercise. Unfortunately Visual Basic 2008 Express doesn’t offer a ‘Save Project As …’ facility, which is what we need.

So instead, we have to resort to copying projects into new folders, like this:
1. Create a new folder under at the same level as the Example 2.1 ClassCreation folder. Call this new folder Assessment Activity 2.1.
2. Copy the entire contents of the ClassCreation folder to the Assessment Activity 2.1 folder.
3. Open Visual Basic and navigate your way to the new Assessment Activity 2.1 folder, click on the solution file, and the original example will load from its new home.
4. To avoid confusion, you should now change something in the new version. For example, you could change the Text property of the form to Assessment Activity 2.1. Then click on ‘Save All’.
5. If you are feeling really worried about losing your work, you could then exit from Visual Basic, reload it, and see if you can still get your Assessment Activity 2.1 project to load!

For the activity itself, here is a hint: add a textbox for each of the five new array elements you will get from GetWord, and change btnGetWord and GetWord to allow an array index to be passed to specify which array element you want returned.

Write the code, save the project, print a full listing of the code for your Portfolio of Evidence (PoE), and demonstrate the project to your lecturer.

**Question 2** (20 marks)  
You might have noticed that you get an error if your line of text has less than 6 words in it.  

Refer back to the Level 3 Course section on debugging and see if you can figure out what is going wrong and how to fix it (see Principles of Computer Programming NQF Level 3, Module 19).

See what happens when you enter different strings with different numbers of parts, separated by blanks. Try one with 10 parts and a blank string. If you get errors, then debug them and find a way to fix the problems.

Write the code, save the project, print a full listing of the code for your PoE, and demonstrate the project to your lecturer.

**Array index:** The index of an array specifies which element of the array is being accessed. In Visual Basic, the first element of the array usually has an index of 0, the second 1, and so on.
Unit 2.2: Encapsulation of data and functions

At the end of this unit, you will be able to explain the encapsulation of data and functions or methods in classes.

Encapsulation means hiding data and functions or methods within a class. The benefit is that programmers using that class don’t have to worry with the details of what is stored where, under what name, and how the calculations are done. All the programmer needs to know is how to interface (communicate) with the object using its name and its arguments. For example, in Example 2.1, the Line class has two methods, LengthText() and GetWord(). In order to use these methods, you don’t need to know that GetWord is using the Split function, and you don’t need to know how to use the Split function and how the Split parameters work.

It is not easy to implement encapsulation using procedural programming, because data is kept separate from the program logic, this results in the data being exposed throughout the system. If you need to make any changes to the data structures, this can affect any module in the entire system. However, in object-oriented programming, as long as you keep the interfaces unchanged, you can change anything within a class without affecting the rest of the system.

Example 2.2 Encapsulation and inheritance

You learnt about encapsulation and inheritance in Module 1; we will now do some simple, worked examples to demonstrate these concepts. Firstly, we will use the Employee example from Module 1 (Case Study 1.0). We have a base class (Employee) with two sub-classes (Worker and Manager). The main variables and methods are defined in the Employee class. This allows them to be used in the Manager and Worker classes with the minimum of additional coding through the use of inheritance. The example also demonstrates the use of encapsulation, implemented by the use of properties and private variables, allowing the salary and salary increase variables to be protected against misuse.

Create the form

2. ‘Recent Projects’ on the left-hand of the screen, select ‘Create’ and the ‘New Project’ window will open up.
3. Select ‘Windows Forms Application’ as the project template. Set the name of the application to EmployeeRecords at the foot of the screen and click on OK. The Form Design screen will open.
4. In the Solution Explorer window on the right-hand of the screen, select the Form1.vb form and rename it to frmEmployeeRecords.vb in the Solution Explorer window. You can do this by right clicking the form and choosing ‘Rename’ on the shortcut menu.
5. Create the form shown in Figure 1.6a below by adding the appropriate controls and setting the properties of those controls. Refer to the code listing below to get the field names.

6. Save the project as EmployeeRecords or use your own choice of name.

Create the base class, Employee

1. Open the ‘Add New Item’ dialog box by clicking ‘Project’ and then clicking ‘Add Class’.
2. Set the ‘Name’ property to Employee, and click OK.
3. This creates a new class called Employee. It is displayed in the code window of Visual Basic.
4. Now add the code for the variables that your application needs. These are:
   - empName for the employee name
   - empAddress for the employee address
   - empType for the employee type (manager or worker)
   - empSalary for the salary, this is a Private variable (for use in the property code)
   - empSalIncrease for the salary increase percentage, this is a Private variable (for use in the propert code).
5. Next we need to define the properties needed for the salary information, so that it can be fully encapsulated. We have chosen to create two properties:
   - empSalValue with a private variable empSalary with holds the original salary.
   - empSalIncrValue with a private variable empSalIncrease which holds the salary increase percentage. See the code listed below.

Create the sub-classes, Manager and Worker

The next step is to add the Manager and Worker classes, making use of inheritance.

1. Create new classes for Manager and Worker and add the statement Inherits Employee to both classes.
2. In the form, add a field for the company car price limit, call it txtmgrCarLimit. This will only be valid in the case of managers in our example. The poor workers have to use their own cars, but don’t worry, the managers pay full tax on their cars, so it is just part of their salary.
3. In the Manager class, add the code for validating the company car value.
4. In the Worker class you would add any worker-specific code (none at present).
Create the event handlers

Add the code given below for:
1. The update name and address details (btnupdNameAddress_Click).
2. The salary calculation (btncalcSalary_Click).
3. The selection of manager or worker mode, lstempType_SelectedIndexChanged. This is triggered when the list selection is changed.
4. The validation of the manager car price limit, txtmgrCarLimit_Leave. This is triggered when the cursor leaves the field.
5. The validation of the old salary entry, txttempSalary_Leave. This is triggered when the cursor leaves the field.
6. Note the New sub. This is automatically generated for you, but you can add additional code in it if you need to, such as initialisation of variables.

We have put the properties methods themselves into the Employee class so that they can be inherited by the Manager and Worker classes. See the updNameAddress() and calcSalary() subs and the empSalValue and empSalIncrValue properties in Employee.

The full listing is shown below.

```vbnet
Public empType As String 'Worker or Manager
Private empSalary As Double 'Old salary
Private empSalIncrease As Double 'Salary percentage increase

Property empSalValue() As Double
'Protects the empSalary variable within a property
'Returns the current value of the property
Get
    Return empSalary
End Get
'Sets the property value
Set (ByVal Value As Double)
    empSalary = Value
    'Salary validation code
    If empType = "Manager" And empSalary > 30000 Then _
    MsgBox("Manager salary must be less than R30 000") _
    : empSalary = 30000
    If empType = "Worker" And empSalary > 10000 Then _
    MsgBox("Worker salary must be less than RIO 000") _
    : empSalary = 10000
End Set
End Property
```
Property empSalIncrValue () As Double
    'Protects the empSalary variable within a property
    'Returns the current value of the property
    Get
        Return empSalIncrease
    End Get
    'Sets the property value
    Set (ByVal Value As Double)
        empSalIncrease = Value
        ' Insert salary increase validation code here
    End Set
End Property

Public Sub updNameAddress()
    'Updates the name and address fields
    empName = frmEmployeeRecords.txttempName.Text()
    empAddress = frmEmployeeRecords.txttempAddress.Text
    empType = frmEmployeeRecords.lstempType.Text
    frmEmployeeRecords.txttempNewSalary.Text = "Press Calculate button"
    ' Insert database update code here
End Sub

Public Sub calcSalary()
    'Calculates the new salary using the old and the increase %
    empSalIncrValue = CDb1(frmEmployeeRecords.txttempSalIncr.Text)
    empSalValue = frmEmployeeRecords.txttempSalary.Text
    frmEmployeeRecords.txttempNewSalary.Text = _
        empSalValue*(1 + empSalIncrValue / 100)
End Sub

End Class

Public Class Manager
    Inherits Employee
    Public mgrCarLimit As Double
    Public Sub updCarLimit ()
        ' Checks that the max car price limit is not exceeded
        If mgrCarLimit >= 300000 Then
            MsgBox ("Car value must be less than R 300 000")
            mgrCarLimit = 300000
        End If
    End Sub
End Class

Public Class Worker
    Inherits Employee
    ' Insert worker specific variables and methods here
End Class
Public Class frmEmployeeRecords
    Dim empfirst As New Employee
    Dim emp As New Manager
    Private Sub btnupdNameAddress_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles btnupdNameAddress.Click
        emp.updNameAddress() 'Calls the inherited method in Manager or Worker
    End Sub

    Private Sub btnCalcSalary_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles btnCalcSalary.Click
        emp.calcSalary() 'Calls the inherited method in Manager or Worker
    End Sub

    Public Sub stempType_SelectedIndexChanged(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles lstempType.SelectedIndexChanged
        'Checks employee type and instantiates a Manager or Worker object
        empfirst.empType = lstempType.Text
        If empfirst.empType = "Manager" Then _
        txtmgrCarLimit.Enabled = True: txtTypeSelected.Text = "Manager" _
        Else txtmgrCarLimit.Enabled = False _
        : txtmgrCarLimit.Text = "" : txtTypeSelected.Text = "Worker"
        End If
        btnCalcSalary.Enabled = True
        btnupdNameAddress.Enabled = True
        If empfirst.empType = "Worker" Then
            Dim emp As New Worker
        End If
        emp.updNameAddress() 'Initialises fields
    End Sub

    Private Sub txtmgrCarLimit_Leave(ByVal sender As Object,
        ByVal e As System.EventArgs) Handles txtmgrCarLimit.Leave
        'Calls the inherited method mgrCarLimit to check car limit
        emp.mgrCarLimit = CDbl(txtmgrCarLimit.Text)
        txtmgrCarLimit.Text = emp.mgrCarLimit 'Re-displays in case of error
    End Sub

    Private Sub txttempSalary_Leave(ByVal sender As Object,
        ByVal e As System.EventArgs) Handles txttempSalary.Leave
        'Sets empSal Value property which also validates salary
        emp.empSalValue = CDbl(txttempSalary.Text)
        txttempSalary.Text = Str(emp.empSalValue) 'Re-display
    End Sub

    Public Sub New ()
        ' This call is required by the Windows Form Designer.
        InitializeComponent()
    End Sub
Add any initialization after the InitializeComponent() call.

End Sub

Protected Overrides Sub Finalize ()
    MyBase. Finalize ()
End Sub
End class

Explanation of the code

- An employee object, empfirst is instantiated in the Dim empfirst As New Employee statement. This is a temporary object which will be used just until the employee type (manager or worker) has been selected.
- An emp object is created from the Manager class, later, if the user selects the worker employee type instead, a new emp object will be created from the Worker class.
- The empSalValue and empSalIncrValue variables are set as Private so as to make them inaccessible outside of the property construct. They can only be changed by changing the relevant property variables, so it is easier to manage validation and access. This provides the encapsulation functionality.
- frmEmployeeRecords is the main module containing the update, calculate and some validation subs. These subs call the actual methods from the Employee class.
- Note the empSalValue = frmEmployeeRecords.txtempSalary.Text statement which sets the empSalValue property has some validation code.
- Similarly, the empSalIncrValue = CDbl(frmEmployeeRecords.txtempSalIncr.Text) statement sets the empSalIncrValue to the entered value.
- To calculate the new salary, the old salary and the salary increase are retrieved from the relevant properties and the salary increase calculation is performed in the calcSalary sub.
  There are additional comments in the code to help your understanding.
  You should enter this code, save it and then test it until it is working correctly and then save it again.
  Try to access the empSalary and empSalIncrease variables directly from within the form. You will find that they are not even visible from within the form, you can only update them using the property get and set functions and these have built-in validation.
To further explore encapsulation, inheritance and polymorphism, we will use and expand on the area calculation exercise in Assessment Activity 1.2 Question 2.

In this exercise, we will also introduce the idea of Constructor classes. These are special classes associated with every class in Visual Basic. This class is always called New and is used when a class is instantiated. We usually use Constructors to initialise variables in classes. Constructors and object insubstantiation are explained in more detail in Unit 3.2.

### Assessment activity 2.2

(Pair work, practical activity, lecturer assessed)  

**Question 1** (40 marks)

Design and develop the program described below, following the detailed instructions. Take turns in doing the actual coding.

**Area Calculator—inheritance and encapsulation**

**Program requirements definition**

Create a program to calculate the area of different regular plane shapes and regular solids. Regular plane shapes are 2D shapes (square, triangle, and so on), and regular solids are 3D objects (cube, cylinder, and so on). Start with a rectangular shape area calculator, but design the program so that it is extendable to calculate areas of other regular shapes and 3D solids. The program should:

- allow the user to select the type of shape, such as:
  - rectangular plane shapes
  - triangular plane shapes
  - others to be added later (circle, spherical solid, rectangular solid)

- allow user to enter the relevant dimensions of the shape, for example:
  - xSize (for example, length of a rectangle)
  - ySize (for example, breadth of a rectangle)
  - zSize (not relevant to a plane rectangular shape, but needed for some solids to indicate the height of the object)

- calculate the area of the shape and display the calculated result.

Work through the example on the following pages with your partner, taking turns to do the coding. Please keep a copy of your written design notes and diagrams for your PoE.

**Design** (5 marks)

The design in this example was chosen to demonstrate some OO concepts.

Remember that there are many different ways to design and code a program for a particular requirements definition and there is no one right way.
The Base Class, `frmAreaCalculator`, has the attributes of `shapeType` (with values Rectangle, Triangle or Circle, entered by the user into a TextBox) and `shapeArea` (the calculated area of the selected shape type). Later we will add an inheritable method to `AreaCalculator`.

The Derived Classes: Rectangle, Triangle and Circle, inherit the `shapeType` and `shapeArea` attributes. They each have their own `calcArea` methods.

Form design
A suggested design is shown in Figure 1.8.

Writing the code (5 marks)
2. Under ‘Recent Projects’ on the left-hand side of the screen, select ‘Create’ and the ‘New Project’ window will open up.
3. Select ‘Windows Forms Application’ as the project template. Set the name of the application to `AssessmentActivity2.2` at the foot of the screen, and click on OK. The form design screen will open.
4. In the ‘Solution Explorer’ window on the right-hand side of the screen, select the `Form1.vb` form and rename it to `frmAreaCalculator.vb`.
5. Select the form in design view, go to the Text property, and change it to `Area Calculator`.

Base class

<table>
<thead>
<tr>
<th>Derived classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rectangle</strong></td>
</tr>
<tr>
<td><code>shapeType</code></td>
</tr>
<tr>
<td><code>shapeArea</code></td>
</tr>
<tr>
<td><code>calcArea</code></td>
</tr>
</tbody>
</table>

| **Triangle**    |
| `shapeType`     |
| `shapeArea`     |
| `calcArea`      |

| **Circle**      |
| `shapeType`     |
| `shapeArea`     |
| `calcArea`      |

Figure 1.7 Class diagram for Assessment Activity 2.2

Figure 1.8 Area Calculator form

**Base class**: A base class is a class that doesn’t have a superclass. In other words, the class is at the top of the tree in a class hierarchy diagram.

**Derived class**: This is another word for a sub-class or child class.
6. Before you go any further, save the new project using the ‘Save All’ icon on the Visual Basic toolbar. Leave the project details ‘Name’ as AssessmentActivity2.2 and ‘Solution Name’ also as AssessmentActivity2.2. You should change the ‘Location’ to whatever path your lecturer gave you with a new folder called AssessmentActivity2.2 on the end. Make sure the ‘Create directory for solution’ box is ticked.

7. Create the form shown in Figure 1.8 by adding the appropriate controls as detailed in the table below. Note the naming conventions used.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name property</th>
<th>Text property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>frmAreaCalculator</td>
<td>Area Calculator</td>
</tr>
<tr>
<td>Label</td>
<td>lblChooseShape</td>
<td>Choose Shape</td>
</tr>
<tr>
<td>ListBox</td>
<td>lstChooseShape</td>
<td>Rectangle Triangle Circle</td>
</tr>
<tr>
<td>Label</td>
<td>lblDimensions</td>
<td>Enter dimensions</td>
</tr>
<tr>
<td>Label</td>
<td>lblxSize</td>
<td>xSize</td>
</tr>
<tr>
<td>TextBox</td>
<td>txtXSize</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td>lblYSize</td>
<td>ySize</td>
</tr>
<tr>
<td>TextBox</td>
<td>txtYSize</td>
<td></td>
</tr>
<tr>
<td>CommandButton</td>
<td>btnCalcArea</td>
<td>Calculate Area</td>
</tr>
<tr>
<td>TextBox</td>
<td>txtCalcArea</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1 Form control details for Assessment Activity 2.2

**Coding the base class frmAreaCalculator:** (5 marks)

1. We will use the automatically generated class from your form (frmAreaCalculator).

2. Now we need to add some code in frmAreaCalculator to declare the following variables:
   - *selectedShape* identifies which shape was selected (for example, a rectangle)
   - *calculatedArea* stores the result of the calculated area
   - *xSize* the x-dimension that the user enters (for example, the length of a rectangle)
   - *ySize* the y-dimension that the user enters (for example, the width of a rectangle)
   - *Pi* the universal constant, Pi, used in area calculations for circles and spheres.

**Adding a method to the base class calcArea:** (5 marks)

This method will be inherited by the derived classes (sub-classes), and we will allow the method to be overridden in the sub-classes (because different shapes have different area calculations).
1. Just above the *End Class* statement of `frmAreaCalculator`, add the sub procedure, as shown below. Note the *Overridable* modifier in the *Sub* statement. This allows derived classes to override methods inherited from the base class.

```vbnet
Public Class frmAreaCalculator
    Private selectedShape As String
        'Only accessible in the base class (AreaCalculator)
    Protected calculatedArea As Double
        'Accessible in base and derived classes
    Protected xSize As Double        'Accessible in base and derived classes
    Protected ySize As Double        'Accessible in base and derived classes
    Protected Const Pi As Double = 3.1416
        'Pi used for calculating areas of circles and spheres

    Public Overridable Sub calcArea()
        calculatedArea = xSize * ySize    'Area = length x breadth
    End Sub
End Class
```

**Coding the derived class** `planeRectangle` *(5 marks)*

1. Create a new class called `planeRectangle`, and add the line *Inherits frmAreaCalculator*.  
2. Now let’s override the base class method, `calcArea`. We do this by adding a method in `planeRectangle` with the modifier *Overrides* in the *Sub* statement.  
3. The variables `xSize`, `ySize` and `calculatedArea` are all inherited from the base class, so you must not declare them here as well.  
4. Note that, because you are in a different class to the form class, you need to specify where the form fields come from by giving the form name, `frmAreaCalculator`.  
5. The actual calculation is done in the line `MyBase.calculatedArea = xSize * ySize`.  
   This calculation will change in the methods for the other shapes, which you will be adding later.  

   The listing for the `planeRectangle` derived class is shown below.

```vbnet
Public Class planeRectangle     'Class for calculating areas of rectangles
    Inherits frmAreaCalculator
        'Inherits the variables and methods from frmAreaCalculator

    Public Overrides Sub calcArea()
        'This method overrides the base class method

        xSize = Val(frmAreaCalculator.txtXSize.Text)        'Refreshes the dimensions from the form
            'Note that you have to say in which form txtXSize and txtYSize are defined.

        ySize = Val(frmAreaCalculator.txtYSize.Text)        'Refreshes the dimensions from the form

        MyBase.calculatedArea = xSize * ySize              'calculates the area and puts it into a variable

        frmAreaCalculator.txtCalcArea().Text = calculatedArea    'converts to string and displays
    End Sub
End Class
```
Coding the event handlers for the form (5 marks)

You learnt how to do this in the NQF Level 3 course, *Principles of Computer Programming NQF Level 3 Module 8.*

1. Open the Code view of the *frmAreaCalculator* form, and double-click on the *lstChooseShape* listbox. This will create an event handler template for you (called *lstChooseShape_SelectedIndexChanged*).
2. Add a line of code in this event handler to store the selected shape item into the variable `selectedShape`.
3. Now double-click on the *btnCalcArea* button (labeled *Calculate Area*).
4. This event handler calls the relevant area calculation method, depending on the shape that was selected. Note that the correct overriding method is specified by prefixing the method name with the derived class name. For example, `recac.calcArea()` will call the version of the method for the rectangular shape.

The coding for these two event handlers is shown below:

```vbnet
Private Sub lstChooseShape_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles lstChooseShape.SelectedIndexChanged
    selectedShape = lstChooseShape.Text
End Sub

Private Sub btnCalcArea_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCalcArea.Click
    Dim recac As planeRectangle = New planeRectangle
    ‘Instantiates object recac
    If selectedShape = "Rectangle" Then recac.calcArea()
End Sub
```

Testing and debugging

Please refer back to *Principles of Computer Programming NQF Level 3, Module 19,* to refresh your testing and debugging skills.

As you write the code, you should be testing what you can as you go along. You do this by pressing F5, which will run your program in debugging mode. The program will stop whenever you get an **unhandled exception** or similar **run-time error**. Generally, it will be obvious what has gone wrong, but if it isn’t obvious, you will have to look more carefully at the programming logic and variable values to see what the problem is. Remember, there are various ways to do this:

- **Set temporary breakpoints:** do this by going into code view and clicking in the margin on the line where you want the program to stop. To continue, you can use the F5 key to continue to run the program normally, or to the next breakpoint, or use the F8 key to run the program one line at a time. To stop the program running altogether, you press Ctl-Alt-Break at the same time.
Unit 2.3: Global data sharing, weak coupling and functional cohesion

At the end of this unit, you will be able to describe how global data sharing is minimised to enable weak coupling, and how modules exhibit functional cohesion.

Global data sharing

Global data sharing means declaring data variables so that they are accessible from any part of a program. This practice is dangerous because it can lead to problems, such as when different modules use the same variable for different purposes. For example, UserID might mean the login name in one module and a person’s identity number in another. A situation like this can quickly become very confusing. In
Interdependency:
When several things depend on each other. If you change one thing in an interdependent module, it could prevent other modules from working properly.

this case, you might accidentally store a person’s ID number in the same place as their login name, and they would then not be able to log in to the system. These sorts of logic errors are very difficult and time-consuming to find.

**Minimising global data sharing**
Variables should be kept as local as possible. This means declaring variables in the classes and methods where you use them. When you do need to communicate between classes, then you can always pass parameters between them. Sometimes you do have to use global variables, but try to keep this to a minimum, and always name them very carefully.

**Weak coupling**
In OO terms, coupling refers to the interdependency of modules and methods. In other words, how dependent are different classes on one another. Global variables are an example of links between classes. In this case, there is a single variable that both classes can access and change. If you change something about the variable, such as the name, you would have to change all references to it in both classes. Here we say that the two classes are interdependent. Weak coupling means that there are not strong links between different classes. This implies low interdependency between parts of a system. This is good, because it means you can change one class or method without having to change many others. This makes program maintenance simpler and safer.

**Enabling weak coupling**
Try to make each method independent of others and able to stand on its own. Sometimes this will mean repeating some common calculations in different methods. You should also avoid using global variables, as these lead to stronger coupling.

**Functional cohesion**
Cohesion means sticking together. In OO programming, cohesion refers to how closely related the methods in a single class are. The more closely-related the methods, the higher the cohesion. For example, you might have a class that only contains accounting functions, or another that contains methods that only relate to stock control.

High cohesion is good because classes and methods are then focused on their main purpose, instead of trying to do everything. Imagine creating a single class that tried to do everything. It would be big and complex, and difficult to maintain. It would also require a large amount of computer memory to run, because you would have to load the entire class each time you wanted to use one method. It would be far better to break the class up into similar functional areas.
Enabling high functional cohesion

When you design your classes, methods and modules, try to design them so that each has a clear task and only one job to do. This will result in more classes, methods and modules, but the advantage is that the program is easier to understand and maintain. Good documentation is essential to help you keep track of all the classes, and diagrams like the class hierarchy diagram is very useful here.

Best practice guidelines:

- Avoid using global data sharing.
- Keep the coupling weak and the modules and methods as independent of each other as possible.
- Keep the functional cohesion high, and rather have more classes and methods each doing less.

Note that both OO programming and procedural programming can achieve these goals if you have a good design and are disciplined about using the best practice guidelines. However, the principles we have described here tend to be easier to implement in an OO programming language.

Assessment activity 2.3

(Group work, discussion activity, lecturer assessed) Total marks [45]

Discuss these questions in your class.

**Question 1** (15 marks, 5 marks per question)

a) Define global data sharing. Why is this bad?

b) Define weak coupling. Why is this good?

c) Define functional cohesion. Should functional cohesion be high or low? Give reasons for your answer.

**Question 2** (30 marks, 10 marks per question)

Using the Visual Basic project in Assessment activity 2.2, discuss the following:

a) Does the module make use of global data sharing? If so, how could this be reduced or avoided? Be specific, and list the variables, classes and methods you are referring to in your answer.

b) Do you think that the program elements are weakly coupled or strongly coupled? Discuss your conclusions and give examples to support your ideas.

c) Which modules have a high degree of functional cohesion? Discuss your answers and give examples to support your ideas.
Unit 2.4: Explain and identify possible classes for simple examples and problems

At the end of this unit, you will be able to explain and identify possible classes for simple examples and problems.

Imagine that you are a robotics programmer, and you have been asked to design a system to enable a robot to play different games. Remember, there are always many ways to design systems, and there is no one correct way. The design process will include the following steps:

- prepare a system/program requirements definition (see Principles of Computer Programming NQF Level 3, Module 10)
- identify classes
- group the classes so that similar behaviours are grouped together (this will help you get high functional cohesion)
- abstract each group into higher-level groups (this will help you to exploit inheritance and save on coding time)
- draw a class hierarchy diagram
- add the specified methods to the lowest-level classes
- identify the data attributes required for the lowest-level classes.
- analyse how you can store the methods higher up in the class hierarchy, so that you can use inheritance and reduce the amount of coding that you have to do.
- update the class diagram showing messages and methods
- remember to follow the OO best practices you have learnt so far.

The program should include the following classes:

- soccer, chess, baseball, hockey, checkers, tennis, cricket, table tennis and rugby

The program should also include the following methods, messages and data:

- getTeamSize
- setTeamSize
- getGameName
- setGameName
- getTeamName
- setTeamName
- getGameType
- getScore
- setScore
- getBall
- strikeBall
- movePiece
- killPiece
- getRules.