NATIONAL OPEN UNIVERSITY OF NIGERIA

COURSE CODE: CIT383

COURSE TITLE: INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

COURSE GUIDE

CIT383

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Course Team O. R. Vincent (Writer) - UNAAB
A. A. Afolorunso (Editor) - NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Office
No. 5 Dar es Salaam Street
Off Aminu Kano Crescent
Wuse II, Abuja
Nigeria

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published By:
National Open University of Nigeria

First Printed 2012

Reviewed and Reprinted 2020

ISBN: 978-058-694-6

CONTENTS **PAGE**

Introduction

CIT383 – Introduction to object-oriented programming is a two- credit unit course consisting of 15 units. The course presents background concepts of objects and describes a widely used programming methodology in object-oriented design (OOD). In OOD, the first step in the problem solving process is to identify the components called objects, which form the basis of the solutions and determine how these objects interact with one another.

This course is divided into three modules. Module 1 introduces the basic concept of object-oriented programming; discusses objects and classes as the basis for OOD. The module also describes encapsulation, abstraction, message passing and introduces composition, inheritance and polymorphism.

Using Java programming language- one of the object-oriented programming languages, module 2 deals with creating a simple class and addressing its properties. Constructors and destructors are discussed in details with concrete examples. Another concept discussed in the module is the static behaviour of classes. Inheritance and polymorphism are addressed in details with examples and possible output.

In module 3, units 1 and 2 are discussed using Java while with C sharp is used in units 3-5. This is done because Java does not support operator overloading. Overloading is discussed in details. Some types of overloading discussed are: method overloading, constructor overloading, basic operators overloading. Others include overloading true and false, logical operator overloading indexers.

This course is aimed at enhancing your former knowledge in programming by introducing objects to improve your knowledge in writing program through possible use of objects and classes. By the end of this course, you should be able to solve any programming problem by dividing some into *modules*. You should also be able to create constructor and destructors and be able to analyse a problem by writing its main class using any object-oriented language.

This Course Guide gives you a brief overview of the course content, course duration, and course materials.

Some of the concepts treated in the course require you to have some basic background on some topics in computer science, especially programming course. There is the need to have a fore-knowledge of programming syntax. Therefore, you are advised to read through further

reading to enhance the knowledge you will acquire from this course material.

What You Will Learn in This Course

The main purpose of this course is to introduce you to the concepts of object-oriented programming by using any object-oriented programming language. This will be achieved through the following:

Course Aims

- Introduce the basic concepts of object-oriented programming (OOP).
- Discuss objects and classes in details and expose their functions in OOP.
- Create classes and set its properties in OOP.
- Discuss the relationship that exists between constructors and destructors.
- Expose the relationships of composition, inheritance and polymorphism in OOP.
- Introduce modular programming and discuss how to divide a program into modules.
- Discuss in details overloading focussing on the different type of overloading.
- Analyse static methods and fields associated with an entire class rather than specific instances of the class.
- Learn how to create a class that inherits fields and methods from another class.
- Learn how the object of a class can reference members of another class (composition).

Course Objectives

In order to ensure that this course achieves its aims, some general objectives have been set though, every unit of this course has some set objectives. By the end of every unit, you may need to check the objectives to ensure that you have properly understood the topic treated. After studying through the course, you should be able to:

- define a class and objects
- create a simple class
- explain the meaning of constructors and destructors
- discuss when a class could become static
- relate inheritance to polymorphism
- create false and true overloading

- describe the function of modular programming
- have a good knowledge about composition
- describe abstraction
- differentiate between instance variables of a class and local variables of a method
- explain the concept of object initialisation
- create your own constructor
- differentiate between constructors and methods
- create multiple constructors in a class and use them as applicable
- explain how static methods and fields are associated with an entire class rather than specific instances of the class.

Working through This Course

In order to have a proper understanding of the course units, you will need to study and understand the content, practise after reading through examples given. You should also learn to write your own program using the basic concepts treated in this course which can be implemented using any programming language of your choice.

This course is designed to cover approximately 16 weeks, and it will require a thorough understanding of the concepts. You should do the exercises in the Tutor-Marked Assignments and submit to your tutors.

Course Materials

These include:

1. Course Guide
2. Study Units
3. Recommended Texts
4. A file for your assignments and records to monitor your progress.

Study Units

There are 15 study units in this course:

Module 1 Basic Concepts of Object-Oriented Programming

Unit 1 Introduction to Classes and Objects
Unit 2 Passing Message and Encapsulation
Unit 3 Inheritance
Unit 4 Polymorphism and Modulation
Unit 5 Composition and Abstraction

Textbooks and References

Bertrand Meyer Object-Oriented Software Construction (Book/CD-ROM) (2nd ed.).

Bill Venners (2001). “Objects and Java: Building Object-Oriented, Multi-Threaded Applications with Java”.

Bill Venners (2001). “Objects and Java: Building Object-Oriented, Multi-Threaded Applications with Java”.

Bjarne Stroustrup (1987). What is "Object-Oriented Programming"? Proceedings ECOOP '87, LNCS, Vol. 276, pp. 51-70, Springer-Verlag.

Brad A. Myers, Dario A. Giuse & Brad Vander Zanden (1992). Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods, Proceedings OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, pp. 184-200.

Brad J. Cox (1984). “Message/Object Programming: An Evolutionary Change in Programming Technology, IEEE Software, 1(1).

Breu. R. (1991). Algebraic Specification Techniques in Object-Oriented Programming Environments, LNCS, Vol. 562, Springer-Verlag.

Cannon, H.I. (1982). Flavors: A Non-Hierarchical Approach to Object-Oriented Programming.

Conrad Bock & James Odell (1994). A Foundation for Composition, Journal of Object-Oriented Programming, 7(6).

Cormen, T. H.; Leiserson, C. F.; Rivest, R. L. & Stein, C. (2001). *Introduction to Algorithms* (2nd ed.). Cambridge: M/T Press.

David Walker (1990). π-calculus Semantics of Object-Oriented Programming Languages, ECS-LFCS-90-122.

Denis Caromel (1990). "Programming Abstractions for Concurrent Programming, Pacific '90, pp. 245-253.

Dijkstra, E.W. (1972). Notes on Structured Programming-Structured Programming, pp. 1-82, Academic Press, Inc.

Giuseppe Castagna (1997). Object-Oriented Programming: A Unified Foundation, Birkhaeuser.

Gregory R. Andrews (1991). Concurrent Programming — Principles and Practice. The Benjamin Cummings Publishing Co. Inc.

Günther Blaschek (1991). Type-Safe Object-Oriented Programming with Prototypes- The Concepts of Omega Structured Programming, Vol. 12, pp. 217-225, Springer-Verlag.

Henry Lieberman (1987). Concurrent Object-Oriented Programming in Act 1-Object-Oriented Concurrent Programming, pp. 9-36, MIT Press.

Huet, G. (ed.) (1990). Logical Foundations of Functional Programming, Addison Wesley.

Ian Craig (2000). The Interpretation of Object-Oriented Programming Languages. Springer-Verlag.

Jaffar, J. & Maher, M. (1994). "Constraint Logic Programming: A Survey. The *Journal of Logic Programming*, Number 19, 20, pp. 503-581.

James O. Coplien (1995). Advanced C++ Programming Styles: Using C++ as a Higher-Level Language.

James W. Cooper , Principles of Object-Oriented Programming in Java 1.1: The Practical Guide to Effective, Efficient Program Design.

John R. Koza (1992). Genetic programming: On the Programming of Computers by Natural Selection, MIT Press.

Ken Arnold & James Gosling (1996). The Java Programming Language. Addison Wesley.

Malik, D.S. (2006). Java Programming: From Problem Analysis to Design, (2nd ed.).

Murray, H. J. R. (1902)."The Knight's Tour, Ancient and Oriental." *British Chess Magazine*, pp. 1-7.

Per Brinch, Hansen (1972). Structured Multi-Programming, CACM, 15(7), pp. 574-578.

Philip Wadler (1995). Monads for Functional programming: Advanced Functional Programming, LNCS, Vol. 925, Springer-Verlag.

Philippe Mougin & Stéphane Ducasse (2003). OOPAL: Integrating Array Programming in Object-Oriented Programming, Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 65-77.

Reiss, S.P. (1987). An Object-Oriented Framework for Conceptual Programming Research Directions in Object-Oriented Programming, pp. 189-218, MIT Press.

Satoshi Matsuoka & Akinori Yonezawa (1993). Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, Research Directions in Concurrent Object-Oriented Programming, pp. 107-150, MIT Press.

Sergey, Dimitriev (2004). “Language Oriented Programming: The Next Programming Paradigm, Online Magazine, 1(1).

Soren Brandt & Ole Lehrmann Madsen (1994). Object-Oriented Distributed Programming in BETA, Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, LNCS, Vol. 791, pp. 185-212, Springer-Verlag.

Timothy A. Budd (1991). An Introduction to Object-Oriented Programming, Addison Wesley.

(Timothy Budd (1998). Understanding Object-Oriented Programming with Java, Addison Wesley.

Timothy Budd (2000). Understanding Object-Oriented Programming with Java Updated Edition, Addison Wesley.

Tom Cargill (1992). C++ Programming Style, Addison Wesley.

Assignment File

These are of two types: the self-assessment exercises and the Tutor-Marked Assignments. The self-assessment exercises will enable you monitor your performance by yourself, while the Tutor-Marked Assignment is a supervised assignment. The assignments take a certain percentage of your total score in this course. The Tutor-Marked Assignments will be assessed by your tutor within a specified period. At the end of the course, the examination will test your understanding as regards the concept and principle of the subject matter. This course includes 15 Tutor-Marked Assignments and each must be done and submitted accordingly. Your best scores however, will be recorded for you. Be sure to send these assignments to your tutor before the deadline to avoid loss of marks.

Presentation Schedule

The *Presentation Schedule* included in your course materials gives you the important dates for the completion of Tutor- Marked Assignments and attending tutorials. Remember, you are required to submit all your assignments by the due date. You should guard against lagging behind in your work.

Assessment

There are two aspects to the assessment of the course. First are the Tutor- Marked Assignments; second, is a written examination.

In order to solve the problems in the Tutor-Marked Assignment and also pass the examination, you are expected to apply knowledge acquired during this course. The assignments must be submitted to your tutor for formal assessment in accordance with the deadlines stated in the Assignment File. The work you submit to your tutor for assessment will count for 30% of your total course mark.

At the end of the course, you will need to sit for a final three-hour examination. This will also count for 70% of your total course mark.

Tutor -Marked Assignment (TMA)

There are 15 Tutor-Marked Assignments in this course. You need to submit all the assignments. The total marks for the best four (4) assignments will be 30% of your total course mark.

Assignment questions for the units in this course are contained in the Assignment File. You should be able to complete your assignments from the information and materials contained in your set textbooks, reading and study units. However, you may wish to use other references to broaden your viewpoint and provide a deeper understanding of the subject.

After completing the assignment, you should send it together with the form to your tutor. Make sure that each assignment reaches your tutor on or before the deadline given. If, however, you cannot complete your work on time, contact your tutor before the assignment is done to discuss the possibility of an extension.

Final Examination and Grading

The final examination for the course will carry 70% percentage of the total marks available for this course. The examination will cover every aspect of the course, so you are advised to revise all your corrected assignments before the examination.

This course endows you with the status of a teacher and that of a learner. This means that you teach yourself and that you learn, as your learning capabilities would allow. It also means that you are in a better position to determine and to ascertain the what, the how, and the when of your learning. No teacher imposes any method of learning on you.

The course units are similarly designed with the introduction following the table of contents, then a set of objectives and then the dialogue and so on.

The objectives guide you as you go through the units to ascertain your knowledge of the required terms and expressions.

Course Marking Scheme

This table shows how the actual course marking is broken down.

Table 1: Course Marking Scheme

Assessment	Marks
Assignment	Four assignments, best three marks of the four count at 30% of course marks
Final Examination	70% of overall course marks
Total	100% of course marks

Course Overview

Unit	Title of Work	Weeks Activity	Assessment (End of Unit)
	Course Guide	Week 1	
	Module 1 Basic Concepts of Object-Oriented Programming		
1	Introduction to Object and Classes	Week 1	Assignment 1
2	Message Passing and Encapsulation	Week 2	Assignment 2
3	Inheritance	Week 3	Assignment 3
4	Polymorphism and Modulation	Week 4	Assignment 4
5	Composition and Abstraction	Week 5	Assignment 5
	Module 2 Object-Oriented Programming Properties Using any Programming Language		
1	Classes and Objects	Week 6	Assignment 6
2	Constructors and Destructors	Week 7	Assignment 7
3	Static Behaviours	Week 8	Assignment 8
4	Inheritance and Composition	Week 9	Assignment 9
5	Polymorphism	Week 10	Assignment 10
	Module 3 Overloading		
1	Methods and Method Overloading	Week 11	Assignment 11
2	Basic Operators Overloading	Week 12	Assignment 12
3	Logical Operator Overloading	Week 13	Assignment 13
4	Overloading True and False	Week 14	Assignment 14
5	Conversion Operator Overloading and Indexers	Week 15	Assignment 15
	Revision	Week 16	
	Examination	Week 17	
	Total	**17 weeks**	

How to Get the Most from this Course

In distance learning the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

Remember that your tutor's job is to assist you. When you need help, don't hesitate to call and ask your tutor to provide it.

1. Read this *Course Guide* thoroughly.
2. Organise a study schedule. Refer to the 'Course Overview' for more details. Note the time you are expected to spend on each unit and how the assignments relate to the units. Whatever method you choose to use, you should decide on it and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they lag behind in their course work.
4. Turn to *Unit 1* and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will always need both the study unit you are working on and one of your set of books on your desk at the same time.
6. Work through the unit. The content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit you will be instructed to read sections from your set books or other articles. Use the unit to guide your reading.

7. Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.
8. When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
9. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have any questions or problems.
10. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this *Course Guide*).

Facilitators/Tutors and Tutorials

There are 15 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, together with the name and phone number of your tutor, as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail or submit your Tutor-Marked Assignments to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, or e-mail if you need help. The following might be circumstances in which you would find help necessary. Contact your tutor if:

- you do not understand any part of the study units or the assigned readings,
- you have difficulty with the self-tests or exercises,
- you have a question or problem with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have face to face contact with your tutor and to ask questions which

are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussions actively.

Summary

Introduction to object-oriented programming will introduce you to the concepts of objects-data structures consisting of data fields and methods and their interactions to design applications and computer programs. The methodology focuses on data rather than processes, with programs composed of self-sufficient modules, each containing all information needed to manipulate its own data structure. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects and can be viewed as an independent 'machine' with a distinct role or responsibility. Concrete examples with illustrations to help you get the best from the course are given.

Therefore, I wish you all the best you can get with the hope that you will understand and find the course interesting.

Course Code CIT383
Course Title Introduction to Object-Oriented Programming

Course Team O. R. Vincent (Writer) - UNAAB
B. A. Afolorunso (Editor) - NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Office
No. 5 Dar es Salaam Street
Off Aminu Kano Crescent
Wuse II, Abuja
Nigeria

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published By:
National Open University of Nigeria

First Printed 2012

ISBN: 978-058-694-6

CONTENTS PAGE

MODULE 1 BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

UNIT 1 INTRODUCTION TO CLASSES AND OBJECTS

CONTENTS

1.0 INTRODUCTION

Object-oriented programming (OOP) is a programming paradigm that uses “objects” – data structures consisting of data fields and methods and their interactions to design applications and computer programmes. Programming techniques may include features such as information hiding, data abstraction, encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s. Many modern programming languages now support OOP.

Object-oriented programming has roots that can be traced to the 1960s. As hardware and software became increasingly complex, quality was often compromised. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasising discrete, reusable units of programming logic. The methodology focuses on data rather than processes, with programmes composed of self-sufficient modules (objects) each containing all the information needed to manipulate its own data structure. This is in contrast to the existing modular programming which had been dominant for many years that focused on the function of a module, rather than specifically the data, but equally provided for code reuse, and self-sufficient reusable units of programming logic, enabling collaboration through the use of linked modules (subroutines). This more conventional approach, which still persists, tends to consider data and behaviour separately.

An object-oriented program may thus be viewed as a collection of cooperating objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects and can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "operators") on these objects are closely associated with the object. For example, the data structures tend to carry their own operators around with them (or at least "inherit" them from a similar object or class).

The category of those programming languages that support the object-oriented programming paradigm are the main object-oriented programming languages which are: Ada programming language, C Sharp programming language family, C++, Fortran, Java programming language, Smalltalk programming language family.

2.0 OBJECTIVES

At the end this unit, you should be able to:

- define the term classes
- list object-oriented programming concepts
- describe objects
- enumerate how to access a class member.

3.0 MAIN CONTENT

3.1 Objects and Classes

3.1.1 Objects

Objects are keys to understanding object-oriented technology. Real-world objects share two characteristics: They all have state and behaviour. Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour. An object stores its state in fields (variables in some programming languages) and exposes its behaviour through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

By attributing state (current speed, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

- **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- **Information-hiding**: By interacting only with an object’s methods, the details of its internal implementation remain hidden from the outside world.
- **Code re-use**: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- **Pluggability and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

3.1.1.1 Object Creation and Destruction

To create an object of a particular class, use the new operator. For example, now that there is a constructor for the Box class you can make specific instances or discrete copies of a box by using the assignment operator and the new memory allocation operator as in:

- *Box box_1=new Box (3, 4, 5) ;*

Once a class has been specified, a data type exists with the same name. You do not need to destroy or remove an object when it is no longer needed. Java automatically flags unused objects and applies garbage collection when appropriate. However you may occasionally need to use finalise() method to insure that a non-Java resource such as a file handle or a window font character is released first. The general form is:

```
void finalise ( )

{
   / / cleanup code goes here
   Super.finalise ( )  / / parent  too!
}
```

3.1.1.2 Accessing Objects

Object variables and methods are accessed using dot notation. Use instance_name.variable or instance_name.method_name(args) to reference instance objects declared with new. Use class_name.variable or class_name.method_name(args) to reference static variables or methods.

```
Employee e=new Employee();
e.name="Al Bundy"; e.setSalary(1000.00); // refs instance
Employee.getCount(); // references static class variable
```

3.1.2 Classes

As discuss earlier, the first step in problem-solving with object-oriented design (OOD) is to identify the components called objects. An object combines data and the operations on that data in a single unit; the mechanism in Java that allows one to combine data and the operations on that data in a single unit is called a class.

A class is a collection of a fixed number of components. The components of a class are called the *members* of the class. The general syntax for defining a class is:

```
modifier (s) class ClassIdentifier modifier (s)
{
    classMembers
}
```

Where modifier(s) are used to alter the behaviour of the class and, usually, class members consist of named constants, variable declarations, and/ or methods. That is, a member of a class can be either a variable (to store data) or a method. Some of the modifiers that are used are public, private, and static.

Note:

- If a member of a class is a named constant, you declare it just like any other named constant.
- If a member of a class is a variable, you declare it just like any other variable.
- If a member of a class is a method, you define it just like any other method
- If a member of a class is a method, it can (directly) access any member of the class- data members and methods. Therefore, when you write a definition of a method, you can directly access any data member of the class (without passing it as a parameter).

In Java, class is a reserved word, and it defines only a data type; no memory is allocated. It announces the declaration of a class. The data members of a class are also called fields. The members of a class are usually classified into three categories: private, public and protected. These will be discussed later.

In the real world, one often finds out that many individual objects are all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created. The following class is one possible implementation of a bicycle:

```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:"+cadence+"       speed:"+speed+"
gear:"+gear);
    }
}
```

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The field's cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world. You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application. The next example is a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

```
class BicycleDemo {
    public static void main(String[] args) {

        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();
```

```
        // Invoke methods on those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

- cadence:50 speed:10 gear:2
- cadence:40 speed:20 gear:3

3.1.2.1 Accessing Class Members

Once an object of a class is created, the object can access the members of the class. The general syntax for an object to access a data member or a method is:

referenceVariableName.memberName

The class members that the class object can access depend on where the object is created.

- If the object is created in the definition of a method of the class, then the object can access both the public and private members.
- If the object is created elsewhere (for example, in a user's program) then the object can access only the public members of the class.

3.1.2.2 Class Specification

A class specifies the properties (data) and methods (actions) that objects can work with. It is a template or prototype for each of the many objects made to the class design. The syntax for a class is:

["public"] ["abstract"| "final"] "class" Class_name

```
  ["extends" object_name] ["implements" interface_name]
"{"
// properties declarations
// bahaviour declarations
"}"
```

The first optional group indicates the visibility (public) or scope of accessibility from other objects. The default is package or visible within the current package only. The second optional group indicates the capability of a class to be inherited or extended by other classes. Abstract classes must be extended and final classes can never be extended by inheritance. The default indicates that the class may or may not be extended at the programmers' discretion.

Class_name has initial letter capitalised by Java convention.
The third option of extends is described in the tutorial on inheritance. The fourth option of implements is described in the tutorial on interfaces. A simple example of a class specification is a box. The box has length, width and height properties as well as a method for displaying its volume.

```
public class box
{
    // what are the properties or fields
    private int length, width, height;
    // what are the actions or methods
    public void setLenght (int p) {length=p;}
    public void setWidth (int p) {width=p;}
public void setHeight (int p) {height=p;}
public int displayVolume( )
{System.out.println(length*width*height);}
}
```

Note: There is no main method in a class defining template! Class names begin with a capital. Use lowercase for all other names. It is a good programming practice to write separate files for the class templates and the driver or main user program. This allows separate compilation as well as class reuse by other driver programmes. A class file can contain more than one associated class but normally its filename is that of the first defined file. A driver program is named the same as the class that contains the main(). Its file may contain other classes as well.

3.1.2.3 Properties

Properties are sometimes called field variables or states. To declare a property, use the following syntax:

```
[  "public" | "private" | "protected"] [ "final"]
[  "static" | "transient" | "volatile" ]
   data_type var_name  [=var_initialiser  ]   ";"
```

The items in the first optional group indicate the visibility or accessibility from other objects; public means visible everywhere (global); private indicates accessible only to this class and nested classes; protected means visible to this class or inherited (extended) classes only; final indicates continuous retention and unchangeable after initial assignment (it is read only or constant). The default is friendly or visible within the current package only. The third optional group indicates how long a value is retained in the variable. Static indicates that the value is shared by all members of the class and exists for all runtime. Static properties can be referenced without creating an instance of the class. Transient prevents the variable from being transferred during a serial operation such as file i/o. Volatile is used in multi-threading to prevent overwrite issues. Many programmers make all properties private and force access through public accessors and mutators which can include validation steps.

3.1.2.4 Class Methods

Class behaviour is represented in Java by methods. To declare a method, use the following syntax:

```
[  "public" | "private" | "protected"] [ "final"]
[  "static" | "transient" | "volatile" ]
   return_data_type method_name  "(" parameter_list ")"
   "{"
   / / some defining actions
   "}"
```

Accessibility keywords are the same as for properties. The default (omitted) is package (friendly) or visible within the current package only. Static methods are shared by all members and exist for all runtime. Static methods can be referenced without creating an instance of the class. Abstract methods must be redefined on inheritance. Native methods are written in C but accessible from Java. The return data type defines the type of value that the calling routine receives from the object (the reply message in object terminology). It can be any of the primitive types or the reserved word void (default value) if no message is to be returned. The statement returns varName; is used to declare the value to be returned to the calling routine.

The parameter list can contain from zero to many entries of datatype varName pairs. Entries are separated by commas. Parameters are passed by value, thus upholding the encapsulation principle by not allowing unexpected changes or side effects. Object references (such as arrays) can also be passed. The projects page has some simple problems based on array passing. Some examples of method header parameter lists are:

- *public static void example1 ( ) {}*
- *public static int add2 (int x) {x+=2; return x;}*
- *public static double example3 (int x, double d) {return x*d;}*
- *public static double example4 (int x, int y, Boolean flagger) {}*
- *public static void example5 (int arr[]) {} / / note: this is an object*

3.1.2.4.1Types of Methods

Constructor methods allow class objects to be created with fields initialised to values as determined by the methods' parameters. This allows objects to start with values appropriate to use (salary set to a base level or employeeNumber set to an incrementing value to guarantee uniqueness). For our simple box class:

```
public Box ( )  //  default box is point
   { lengtht=0;  width=0;  height; }

public Box{int  1,int  w, int  h) //  allows giving initial  size
   { length=1; width=w;  height=h; }
```

Note: That there is no class keyword or return data type keyword. Also the method name is the same as the class name. This is what marks the fragment as a constructor method. If no constructor method is defined for a class, a default constructor is automatically used to initialise all fields to 0, false or unicode(0) as appropriate to the data type. The best programming device is to declare the constructor with no parameters as private and use it to initialise all properties. Then other constructors can first call it using this() and then do their own specific property validations/initialisation.

Accessor (or observer) methods read property (i.e. field variable) values and are conventionally named *getFoobar()* or whatever the property is called while Mutator (or transformer) methods set property values and are often named *setFoobar()* etc. Mutators can be used to ensure that the property's value is valid in both range and type.

It is a good programming practice to make each property in a class private and include accessor and mutator methods for them. This is a

good example of object encapsulation. The exceptions to writing accessor/mutator methods for each property is for those that are used only within the class itself or for properties that are set in more complex ways. Helper methods are those routines that are useful within the class methods but not outside the class. They can help in code modularisation. Normally they are assigned private access to restrict use. Recursive methods are methods that are defined in terms of it. A classic recursion is factorials where n factorial is the product of positive integer n and all the products before it down to one. In Java this could be programmed as:

```
Class Factorial
{
  Int factorial  ( int  n)
{
  If  ( n= =1 )  { return  1};
  Return  ( n *  factorial (n-1 ) ) ;
  }
}
```

Note: This short method is not very well written as negative and floating calling parameters are illegal in factorials and will cause problems in terminating the loop. Bad input should always be trapped.

3.1.2.5 Inner Classes

Inner classes are classes nested inside another class. They have access to the outer class fields and methods even if marked as private. Inner classes are used primarily for data structures, helper classes, and event handlers. A brief example of how an inner class can be used as a data structure is:

```
public class Main2
{
 class Person
 {
   // inner class defines the required structure
   String first;
   String last;
 }
 // outer class creates array of person objects with specific properties
 // the objects can be referenced by personArray[1].last for example
 Person personArray[]={new Person(), new Person(), new Person()};
}
```

4.0 CONCLUSION

A class normally consists of one or more variables that represent the attributes of a particular object of a class. These variables are called instance variables. Methods within the class manipulate these variables. Most instance variable declarations are preceded with the private access modifier. Variables or methods declared with access modifier *private* are accessible only to methods of the class in which they are declared. Classes often provide public methods that allow clients of the class to *set* or *get* private instance variables. The names of these methods need not begin with *set* or *get*, but this naming convention is highly recommended in Java.

5.0 SUMMARY

In this unit, the following have been discussed:

- Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication
- A class is a collection of a fixed number of components. The components of a class are called the *members* of the class
- In Java, class is a reserved word, and it defines only a data type; no memory is allocated. It announces the declaration of a class. The data members of a class are also called fields. The members of a class are usually classified into three categories: private, public and protected
- Recursive methods are methods that are defined in terms of it. A classic recursion is factorials where n factorial is the product of positive integer n and all the products before it down to one.

6.0 TUTOR-MARKED ASSIGNMENT

Circle is a class with property radius and methods *setRadius*(), *getRadius*(), *calcDiameter*(), *calcArea*(). Use double precision for everything. Since this is such a short program, you may want to use only one file say MakeCircle.java that contains both the main (or driver) class and the Circle class. The driver creates an instance of the Circle class and then displays the diameter and area. Test with radius=3.0. The diameter should read as 6.0 and the area as 28.25999. Once it is working, see if you can factor it into two files (Circle and MakeCircle) and compile separately, (the Circle class file first). As an enhancement, you may want to add a command line interpreter to get the radius for the driver to use.

7.0 REFERENCES/FURTHER READING

Bacchetta, M. *et al* (1998). *Electronic Commerce and the Role of the* Bill Venners (2001). "Objects and Java: Building Object-Oriented, Multi-Threaded Applications with Java".

Bertrand Meyer (2002). *Object-Oriented Software Construction* (2nd ed.).

James W. Cooper (nd). *Principles of Object-Oriented Programming in Java 1.1: The Practical Guide to Effective, Efficient Program Design.*

Malik, D.S. (2006). *Java Programming: From Problem Analysis to Design* (2nd ed.).

UNIT 2 PASSING MESSAGE AND ENCAPSULATION

CONTENTS

1.0 INTRODUCTION

Message passing in computer science, is a form of communication used in parallel computing, object-oriented programming, and inter-process communication. Message passing systems have been called “shared nothing” systems because the message passing abstraction hides underlying state changes that may be used in the implementation of sending messages.

2.0 OBJECTIVES

At the end this unit, you should be able to:

- outline what message passing entails
- define encapsulation
- describe the concept of encapsulation
- list the uses of message passing.

3.0 MAIN CONTENT

3.1 The Model

Message passing model based programming languages typically define messaging as the (usually asynchronous) sending (usually by copy) of a data item to a communication endpoint (Actor, process, thread, socket, etc.). Such messaging is used in Web Services by SOAP. This concept is the higher-level version of a datagram, except that messages can be larger than a packet and can optionally be made reliable, durable, secure, and/or transacted.

Messages are also commonly used in the same sense as a means of inter-process communication; the other common technique being streams or pipes, in which data are sent as a sequence of elementary data items instead (the higher-level version of a virtual circuit). Examples of Message passing styles are:

a. Actor model implementation
b. Amorphous computing
c. Antiobjects
d. Flow-based programming
e. SOAP (protocol)

3.2 Influences on other Programming Models

In the terminology of some object-oriented programming languages, a message is the single means to pass control to an object. If the object 'responds' to the message, it has a method for that message. In pure object-oriented programming, message passing is performed exclusively through a dynamic dispatch strategy. Sending the same message to an object twice will usually result in the object applying the method twice. Two messages are considered to be the same message type, if the name and the arguments of the message are identical.

Objects can send messages to other objects from within their method bodies. Message passing enables extreme late binding in systems. Alan Kay has argued that message passing is a concept more important than objects in his view of object-oriented programming, however people often miss the point and place too much emphasis on objects themselves and not enough on the messages being sent between them. The live distributed objects programming model builds upon this observation; it uses the concept of a distributed data flow to characterise the behaviour of a complex distributed system in terms of message patterns, using high-level and functional-style specifications. Some languages support the forwarding or delegation of method invocations from one object to another if the former has no method to handle the message, but 'knows' another object that may have one.

3.3 Encapsulation

In OOD, objects combines data and operations on that data in a single unit, the feature called encapsulation. The object is first identified, then the relevant data and then the operations which are needed to manipulate the objects.

Encapsulation is the ability of an object to be a container (or capsule) for related properties (data variables) and methods (functions). Older languages did not enforce any property/method relationships. This often

resulted in side effects where variables had their contents changed or reused in unexpected ways and spaghetti code that was difficult to unravel, understand and maintain. Encapsulation is one of the three fundamental principles in object- oriented programming.

Data hiding is the ability of objects to shield variables from external access. It is a useful consequence of the encapsulation principle. Those variables marked as private can only be seen or modified through the use of public accessor and mutator methods. This permits validity checking at run time. Access to other variables can be allowed but with tight control on how it is done. Methods can also be completely hidden from external use. Those that are made visible externally can only be called by using the object's front door (i.e. there is no 'goto' branching concept).

Encapsulation is the concept of hiding the implementation details of a class and allowing access to the class through a public interface. For this, there is need to declare the instance variables of the class as private or protected. The client code should access only the public methods rather than accessing the data directly. Also, the methods should follow the Java Bean's naming convention of set and get. Encapsulation makes it easy to maintain and modify code. The client code is not affected when the internal implementation of the code changes as long as the public method signatures are unchanged. For instance:

```
public class Employee
{
private float salary;
public float getSalary()
{
return salary;
}
public void setSalary(float salary)
{
this.salary = salary;
}
```

4.0 CONCLUSION

Message passing enables extreme late binding in systems. Message passing is a concept more important than objects in Alan Kay's view of object-oriented programming, however people often miss the point and place too much emphasis on objects themselves and not enough on the messages being sent between them. Encapsulation is the ability of an object to be a container related properties and methods.

5.0 SUMMARY

In this unit, we have learnt that:

- Messages are also commonly used in the same sense as a means of inter-process communication. Examples of Message passing styles are: Actor model implementation ; Amorphous computing; Antiobjects; Flow-based programming; SOAP (protocol).
- Message passing is performed exclusively through a dynamic dispatch strategy. Sending the same message to an object twice will usually result in the object applying the method twice. Two messages are considered to be the same message type, if the name and the arguments of the message are identical.
- Those variables marked as private can only be seen or modified through the use of public accessor and mutator methods. This permits validity checking at run time
- Encapsulation makes it easy to maintain and modify code. The client code is not affected when the internal implementation of the code changes as long as the public method signatures are unchanged.

6.0 TUTOR-MARKED ASSIGNMENT

1. List some examples of passing message style.
2. Write a program to explain encapsulation.
3. How does message passing affect other programming model.

7.0 REFERENCES/FURTHER READING

Bertrand Meyer, *Object-Oriented Software Construction* (Book/CD-ROM) (2nd ed.).

Bill Venners (2001). "Objects and Java: Building Object-Oriented, Multi-Threaded Applications with Java".

Günther Blaschek (1991). Type-Safe Object-Oriented Programming with Prototypes- *The Concepts of Omega Structured Programming*, Vol. 12, pp. 217-225, Springer-Verlag, 1991.

James W. Cooper (nd). *Principles of Object-Oriented Programming in Java 1.1: The Practical Guide to Effective, Efficient Program Design.*

Malik, D. S. (2006). *Java Programming: From Problem Analysis to Design*. (2nd ed.).

UNIT 3 INHERITANCE

CONTENTS

1.0 INTRODUCTION

In Java, the mechanism that allows us to extend the definition of a class without making any physical changes to the class is the principle of inheritance. Inheritance has to do with relationship. It helps in the creation of new classes from existing classes.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define inheritance
- differentiate between a subclass and a super class
- outline the uses of a super class
- differentiate between inheritance and overridden.

3.0 MAIN CONTENT

3.1 What is Inheritance?

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. An example of where this could be useful is with an employee records system. One could create a generic employee class with states and actions that are common to all employees. Then more specific classes could be defined for salaried, commissioned and hourly employees. The generic class is known as the super class or base class and the specific classes as subclasses or derived classes. The concept of inheritance greatly

enhances the ability to reuse code as well as making design a much simpler and cleaner process.

For instance, once the problem domain is partitioned into types, one will likely want to model relationships in which one type is a more specific or specialised version of another. For example you may have identified in your problem domain two types, Cup and CoffeeCup, and you want to be able to express in your solution that a CoffeeCup is a more specific kind of Cup. In an object-oriented design, you model this kind of relationship between types with inheritance.

3.2 Building Inheritance Hierarchies

The relationship modeled by inheritance is often referred to as the "is-a" relationship. In the case of Cup and CoffeeCup, a "CoffeeCup is-a Cup." Inheritance allows you to build hierarchies of classes, such as the one shown in Figure 3.1. The upside-down tree structure shown in Figure 5-1 is an example of an inheritance hierarchy displayed in UML form. Note that the classes become increasingly more specific as you traverse down the tree. A CoffeeCup is a more specific kind of Cup. A CoffeeMug is a more specific kind of CoffeeCup. Note also that the is-a relationship holds even for classes that are connected in the tree through other classes. For instance, a CoffeeMug is not only more specific version of a CoffeeCup, it is also a more specific version of a Cup. Therefore, the is-a relationship exists between CoffeeMug and Cup: a CoffeeMug is-a Cup.

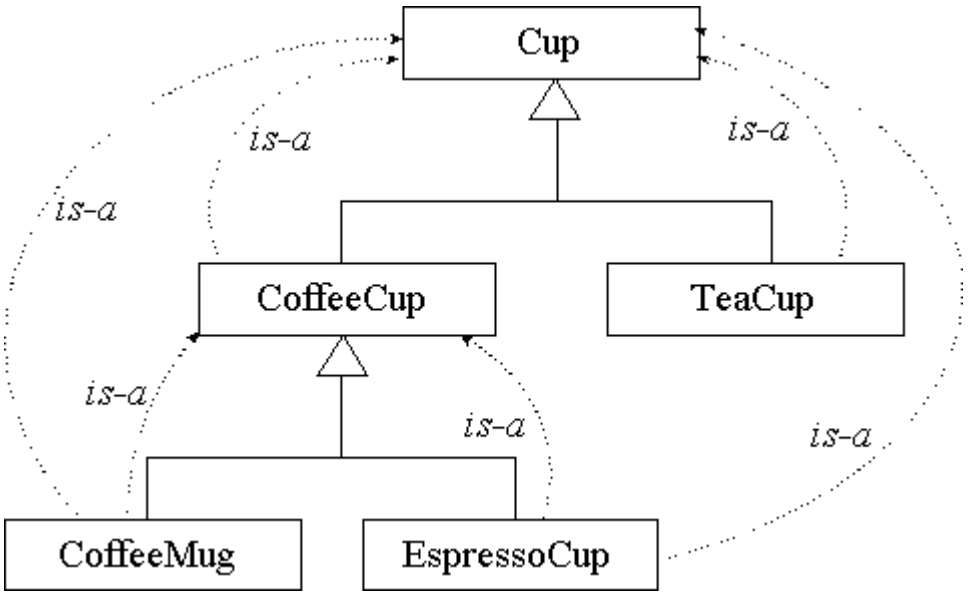

Fig. 3.1: The is-a Relationship of Inheritance

When programming in Java, you express the inheritance relationship with the extends keyword:

```
class Cup {
}
class CoffeeCup extends Cup {
}
class CoffeeMug extends CoffeeCup {
}
```

In Java terminology, a more general class in an inheritance hierarchy is called a superclass. A more specific class is a subclass. In Figure 3.1, Cup is a superclass of both CoffeeCup and CoffeeMug. Going in the opposite direction, both CoffeeMug and CoffeeCup are subclasses of Cup. When two classes are right next to each other in the inheritance hierarchy, their relationship is said to be direct. For example Cup is a direct superclass of CoffeeCup, and CoffeeMug is a direct subclass of CoffeeCup.

The act of declaring a direct subclass is referred to in Java circles as class extension. For example, a Java guru might be overheard saying, “Class CoffeeCup extends class Cup.” Owing to the flexibility of the English language, Java in-the-knows may also employ the term “subclass” as a verb, as in “Class CoffeeCup subclasses class Cup.” One other way to say the same thing is, “Class CoffeeCup descends from class Cup.”

An inheritance hierarchy, such as the one shown in Figure 3.1, defines a family of types. The most general class in a family of types, the one at the root of the inheritance hierarchy is called the base class. In Figure 3.1, the base class is Cup. Because every class defines a new type, you can use the word “type” in many places you can use “class.” For example, a base class is a base type, a subclass is a subtype, and a direct superclass is a direct supertype. In Java, every class descends from one common base class: Object. The declaration of class Cup above could have been written:

```
class Cup extends Object { // "extends Object" is optional
}
```

This declaration of Cup has the same effect as the earlier one that excluded the "extends Object" clause. If a class is declared with no extends clause, it by default extends the Object class. (The only exception to this rule is class Object itself, which has no superclass.) The inheritance hierarchy of Figure 3.1 could also have shown the Object class hovering above the Cup class, in its rightful place as the

most super of all superclasses. In this case, class Object remained invisible, because the purpose of the figure was to focus on one particular family of types, the Cup family.

In Java, a class can have only one direct superclass. In object-oriented parlance, this is referred to as single inheritance. It contrasts with a multiple inheritance, in which a class can have multiple direct superclasses. Although Java only supports single inheritance of classes through class extension, it supports a special variant of a multiple inheritance through "interface implementation."

3.3 Inheriting Interface and Implementation

Modeling is-*a relationship* is called inheritance because the subclass inherits the interface and, by default, the implementation of the superclass. Inheritance of interface guarantees that a subclass can accept all the same messages as its superclass. A subclass object can, in fact, be used anywhere a superclass object is called for. For example, a CoffeeCup as defined in Figure 3.1 can be used anywhere a Cup is needed. This substitutability of a subclass (a more specific type) for a superclass (a more general type) works because the subclass accepts all the same messages as the superclass. In a Java program, this means you can invoke on a subclass object any method you can invoke on the superclass object.

This is only half of the inheritance story, however, because by default, a subclass also inherits the entire implementation of the superclass. This means that not only does a subclass accept the same messages as its direct superclass, but by default it behaves identically to its direct superclass when it receives one of those messages. Yet unlike inheritance of interface, which is certain, inheritance of implementation is optional. For each method inherited from a superclass, a subclass may choose to adopt the inherited implementation, or to override it. To override a method, the subclass merely implements its own version of the method.

Overriding methods is a primary way a subclass specialises its behaviour with respect to its superclass. A subclass has one other way to specialise besides overriding the implementation of methods that exist in its direct superclass. It can also extend the superclass's interface by adding new methods. This possibility will be discussed in detail later. Suppose there is a method in class Cup with the following signature:

```
public void addLiquid(Liquid liq) {
}
```

The addLiquid() method could be invoked on any Cup object. Because CoffeeCup descends from Cup, the addLiquid() method could also be invoked on any CoffeeCup object.

If you do not explicitly define in class CoffeeCup a method with an identical signature and return type as the addLiquid() method shown above, your CoffeeCup class will inherit the same implementation (the same body of code) used by superclass Cup. If, however, you do define in CoffeeCup an addLiquid() method with the same signature and return type, that implementation overrides the implementation that would otherwise have been inherited by default from Cup.

When you override a method, you can make the access permission more public, but you cannot make it less public. So far, you have only been introduced to two access levels, public and private. There are, however, two other access levels that sit in-between public and private, which form the two ends of the access-level spectrum. In the case of the addLiquid() method, because class Cup declares it with public access, class CoffeeCup must declare it public also. If CoffeeCup attempted to override addLiquid() with any other access level, class CoffeeCup would not compile.

For an illustration of the difference between inheriting and overriding the implementation of a method, see Figure 3.2. The left side of this figure shows an example of inheriting an implementation, whereas the right side shows an example of overriding the implementation.

The method in question is the familiar addLiquid() method. In the superclass, Cup, a comment indicates that the code of the method, which is not shown in the figure, will cause the liquid to swirl clockwise as it is added to the cup. Liquid added to an instance of the CoffeeCup class defined on the left will also swirl clockwise, because that CoffeeCup inherits Cup's implementation of addLiquid(), which swirls clockwise. By contrast, liquid added to an instance of the CoffeeCup class defined on the right will swirl counterclockwise, because this CoffeeCup class overrides Cup's implementation with one of its own. A more advanced CoffeeCup could override addLiquid() with an implementation that first checks to see whether the coffee cup is in the northern or southern hemisphere of the planet, and based on that information, decide which way to swirl.

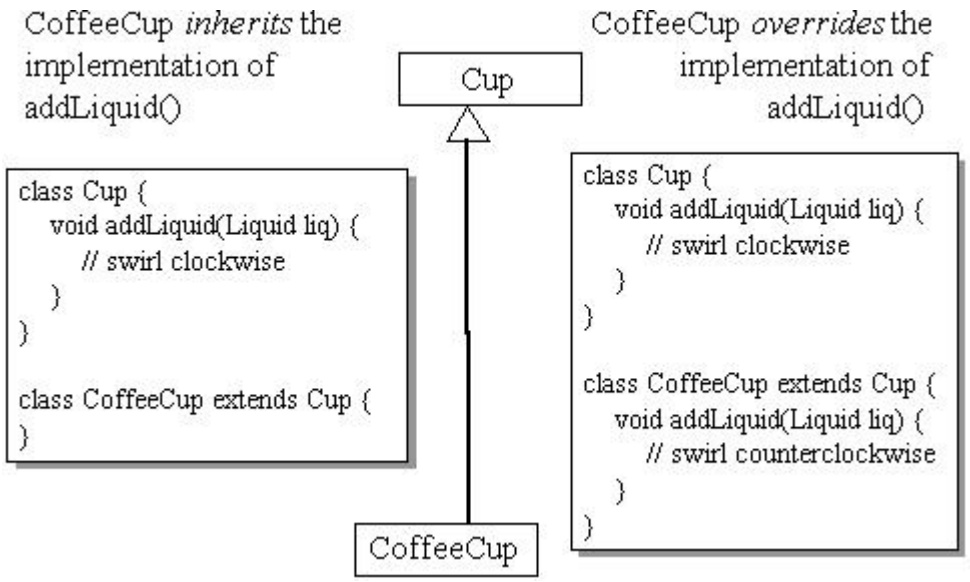

Fig.3.2: Inheriting vs. Overriding the Implementation of a Method

In addition to the bodies of public methods, the implementation of a class includes any private methods and any fields defined in the class. Using the official Java meaning of the term "inherit," a subclass does not inherit private members of its superclass. It only inherits accessible members. Well- designed classes most often refuse other classes direct access to their non-constant fields and this policy generally extends to subclasses as well. If a superclass has private fields, those fields will be part of the object data in its subclasses, but they will not be "inherited" by the subclass. Methods defined in the subclasses will not be able to directly access them. Subclasses, just like any other class, will have to access the superclass's private fields indirectly, through the superclass's methods.

3.4 Hiding Fields

If you define a field in a subclass that has the same name as an accessible field in its superclass, the subclass's field hides the superclass's version. (The type of the variables need not match, just the names.) For example, if a superclass declares a public field, subclasses will either inherit or hide it. (You can't override a field.) If a subclass hides a field, the superclass's version is still part of the subclass's object data; however, the subclass doesn't "inherit" the superclass's version of the field, because methods in the subclass can't access the superclass's version of the field by its simple name. They can only access the subclass's version of the field by its simple name. You can access the superclass's version by qualifying the simple name with the super keyword, as in super.fieldName.

Java permits you to declare a field in a subclass with the same name as a field in a superclass so you can add fields to a class without worrying about breaking compatibility with already existing subclasses. For example, you may publish a library of classes that your customers can use in their programmes. If your customers subclass the classes in your library, you will likely have no idea what new fields they have declared in their subclasses. In making enhancements to your library, you may inadvertently add a field that has the same name as a field in one of your customer's subclasses. If Java didn't permit field hiding, the next time you released your library, your customer's program might not run properly, because the like- named field in the subclass would clash with the new field in the superclass from your library. Java's willingness to tolerate hidden fields makes subclasses more accepting of changes in their superclasses.

3.4.1 Abstract Classes and Methods

As you perform an object-oriented design, you may come across classes of objects that you would never want to instantiate. Those classes will nevertheless occupy a place in your hierarchies. An example of such a class might be the Liquid class from the previous discussions. Class Liquid served as a base class for the family of types that included subclasses Coffee, Milk, and Tea. While you can picture a customer walking into a cafe and ordering a coffee, milk, or a tea, you might find it unlikely that a customer would come in and order a "liquid." You might also find it difficult to imagine how you would serve a "liquid." What would it look like? How would it taste? How would it swirl or gurgle?

Java provides a way to declare a class as conceptual only, not one that represents actual objects, but one that represents a category of types. Such classes are called abstract classes. To mark a class as abstract in Java, you merely declare it with the abstract keyword. The abstract keyword indicates the class should not be instantiated. Neither the Java compiler nor the Java Virtual Machine will allow an abstract class to be instantiated. The syntax is straightforward:

```
// In Source Packet in file inherit/ex6/Liquid.java
abstract class Liquid {

   void swirl(boolean clockwise) {
      System.out.println("One Liquid object is swirling.");
   }
   static void gurgle() {
      System.out.println("All Liquid objects are gurgling.");
   }
}
```

The above code makes Liquid a place holder in the family tree, unable to be an object in its own right.

Note that the Liquid class shown above still intends to implement a default behaviour for swirling and gurgling. This is perfectly fine; however, classes are often made abstract when it doesn't make sense to implement all of the methods of the class's interface. The abstract keyword can be used on methods as well as classes, to indicate that the method is part of the interface of the class, but does not have any implementation in that class. Any class with one or more abstract methods is itself abstract and must be declared as such. In the Liquid class, you may decide that there is no such thing as a default swirling behaviour that all liquids share. If so, you can declare the swirl() method abstract and forgo an implementation, as shown below:

```
// In Source Packet in file inherit/ex7/Liquid.java
abstract class Liquid {

   abstract void swirl(boolean clockwise);

   static void gurgle() {
      System.out.println("All Liquid objects are gurgling.");
   }
}
```

In the above declaration of Liquid, the swirl() method is part of Liquid's interface, but doesn't have an implementation. Any subclasses that descend from the Liquid class shown above will have to either implement swirl() or declare themselves abstract. For example, if you decided there were so many varieties of coffee that there is no sensible default implementation for Coffee, you could neglect to implement swirl() in Coffee. In that case, however, you would need to declare Coffee abstract. If you didn't, you would get a compiler error when you attempted to compile the Coffee class. You would have to subclass Coffee (for example: Latte, Espresso, CafeAuLait) and implement swirl() in the subclasses, if you wanted the Coffee type to ever see any action.

Most often you will place abstract classes at the upper regions of your inheritance hierarchy, and non- abstract classes at the bottom. Nevertheless, Java does allow you to declare an abstract subclass of a non- abstract superclass. For example, you can declare a method inherited from a non-abstract superclass as abstract in the subclass, thereby rendering the method abstract at that point in the inheritance hierarchy. This design implies that the default implementation of the method is not applicable to that section of the hierarchy. As long as you

implement the method again further down the hierarchy, this design would yield an abstract class sandwiched in the inheritance hierarchy between a non-abstract superclass and non- abstract subclasses.

4.0 CONCLUSION

Inheritance can be viewed as a hierarchical structure wherein a superclass is declared with its subclass. Any new class that you create from an existing class is called a subclass or derived class; existing classes are called superclasses or base classes. The subclass inherits the properties of the superclass. Rather than creating completely new classes from scratch, inheritance can be used to reduce software complexity. Modeling the relationships between types is a fundamental part of the process of object-oriented design. This unit shows you how to model relationships using composition and inheritance. It describes many facets of inheritance in Java, analyse the flexibility and performance implications of inheritance and composition with guidelines on the appropriate use of each.

5.0 SUMMARY

In this unit, we have discussed the following:

- The generic class is known as the superclass or base class and the specific classes as subclasses or derived classes. The concept of inheritance greatly enhances the ability to reuse code as well as making design a much simpler and cleaner process.
- In Java terminology, a more general class in an inheritance hierarchy is called a superclass. A more specific class is a subclass.
- When a class has only one direct super class, it is referred to as single inheritance. In contrasts with a multiple inheritance, in which a class can have multiple direct superclasses.
- Overriding methods is a primary way a subclass specialises its behaviour with respect to its superclass.
- Well-designed classes most often refuse other classes direct access to their non-constant fields, and this policy generally extends to subclasses as well. If a super class has private fields, those fields will be part of the object data in its subclasses, but they will not be "inherited" by the subclass.
- Java permits one to declare a field in a subclass with the same name as a field in a superclass so that one can add fields to a class without worrying about breaking compatibility with already existing subclasses.

6.0 TUTOR-MARKED ASSIGNMENT

1. Draw a class hierarchy in which several classes are subclasses of a single superclass.
2. Suppose that a class Employee is derived from the class Person. Give examples of data and method members that can be added to the class Employee.

7.0 REFERENCES/FURTHER READING

Bertrand Meyer (nd).*Object-Oriented Software Construction* (Book/CD-ROM) (2nd ed.).

Bill Venners (2001). "Objects and Java: Building Object-Oriented, Multi-Threaded Applications with Java".

James W. Cooper (nd). *Principles of Object-Oriented Programming in Java 1.1: The Practical Guide to Effective, Efficient Program Design.*

Malik, D. S. (2006). *Java Programming: From Problem Analysis to Design* (2nd ed.).

UNIT 4 POLYMORPHISM AND MODULATION

CONTENTS

1.0 INTRODUCTION

Polymorphism is the capability of an action or method to do different things based on the object that it is acting upon. This is the third basic principle of object-oriented programming. Overloading and overriding are two types of polymorphism.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- describe polymorphism
- state types of polymorphism
- define modular programming
- list the functions of modular programming.

3.0 MAIN CONTENT

3.1 What is Polymorphism?

Java allows one to treat an object of a subclass as an object of its superclass. In other words, a reference variable of a superclass can point to an object of its subclass. For instance, one could declare classes Person and PartTimeEmployee as:

```
Person name, nameRef;                          //Line 1

PartTimeEmployee employee, employeeRef;        //Line 2

name = new Person ("Abraham", " Lincoln");     //Line 3

employee = new PartTimeEmployee (" Susan", "Johnson", 6.40, 35);
//Line 4
```

The statement in line 1 declares name and nameRef to be reference variables of the type Person. The statement in line 2 declares employee and employeeRef to be reference variables of the type PartTimeEmployee. The statement in line 3 instantiates the object name while the statement in line 4 instantiates the object employees. Consider the following:

nameRef = employee; *//Line 5*

System.out.println ("nameRef: " + nameRef); *//Line 6*

The statement in line 5 makes nameRef point to the object employee. After executing the statement in line 5, the object nameRef is treated as an object of the class PartTimeEmployee. The statement in line 6 outputs the value of the object nameRef. The output of the statement in line 6 is:

nameRef: Susan Johnson's wages are : $ 224.0 *//Line 7*

Notice that even though nameRef is declared as a reference variable of the type Person, when the program executes, the statement in line 7 outputs the first name, the last name, and the wages. This is because when the statement in line 6 executes to output nameRef, the method toString of the class PartTimeEmployee executes, not the methodtoString of the class Person. This is called late binding, dynamic binding, or run-time binding; that is, the method that gets executed is determined at execution time, not at compile time.

In a class of hierarchy, several methods may have the same name and the same formal parameter list. Moreover, a reference variable of a class can refer to either an object of its own class or an object of its subclass. Therefore, a reference variable can invoke, that is, execute, a method of its own class or of its subclass(es). Binding means associating a method definition with its invocation, that is, determining which method definition gets executed. In early binding, a method's definition is associated with the method's invocation at execution time, that is, when the method is executed. Except for a few cases, java uses late binding for all methods. Furthermore, the term polymorphism means assigning multiple meanings to the same method name.

The reference variable name or nameRef can point to any object of the class Person or the class PartTimeEmployee. Loosely speaking, one could say that these reference variables have many forms, that is, they are polymorphism reference variables. They can refer to either objects of their own class or objects of the subclasses inherited from their class. The following example further explains polymorphism.

Example 4.1

Consider the following definitions of the classes RectangleFigure and BoxFigure

```
public class RectangleFigure
{    private double length ;
     private double width;
     public rectangleFigure ( )
     {
          length  = 0;
          width = 0;
     }
     public RectangleFigure ( double 1, double w );
    {
         setDimension ( 1, w );
     }
     public  void setDimension ( double 1, double  w)
    {
          if ( 1 >=  0 )
                length  = 1 ;
          else
                length = 0 ;
          if ( w  <= 0 )
               width  = w;
          else
}
public double getwidth ( )
{
   return length ;
}
public double getwidth ( )
{
   return;
}
public double area ( )
{
     return length  *  width;
}
     public double perimeter( )
     {
          return  2  * ( length  +  width );
      }
      public void print( )
      {
 System.out.print(" Length  =  "  +  length  +  " ; Width =  "  +  width
```

```
  +" \n" + " Area = " + area ( ));
      }
}
```

3.2 Types of Polymorphism

The two main types of polymorphism are overloaded method and overridden methods. Overloaded methods are methods with the same name signature but either a different number of parameters or different types in the parameter list. For example 'spinning' a number may mean increase it, 'spinning' an image may mean rotate it by 90 degrees. By defining a method for handling each type of parameter you achieve the effect that you want. Overridden methods are methods that are redefined within an inherited or subclass. They have the same signature and the subclass definition is used. The third type is dynamic method binding. These will be discussed in details in module two.

3.3 Modularity

Methods in Java are like automobile parts', they are building blocks. Methods are used to divide complicated program into manageable pieces. They are both predefined methods, methods that are already written and provided by java, and user-defined methods, methods that you create. Using methods has several advantages:

- While working on one method, you can focus on just that part of the program and construct it, debug it and perfect it.
- Different people can work on different methods simultaneously.
- If a method is needed in more than one place in a program, or in different program, you can write it once and use it many times.
- Using methods greatly enhance the programmes readability because it reduces the complexity of the main method.
- Methods are often called modules. They are like miniature programmes; you can put them together to form a larger program. The ability is less apparent with the predefined methods because their programming code is not available to us. However, because predefined methods are already written for us, this will be learnt first, so that it could be used when needed. To use a predefined method in your program (s), you need to know only how to use it.

Figure 4.1 gives a picture of what modular programming is all about. The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters.

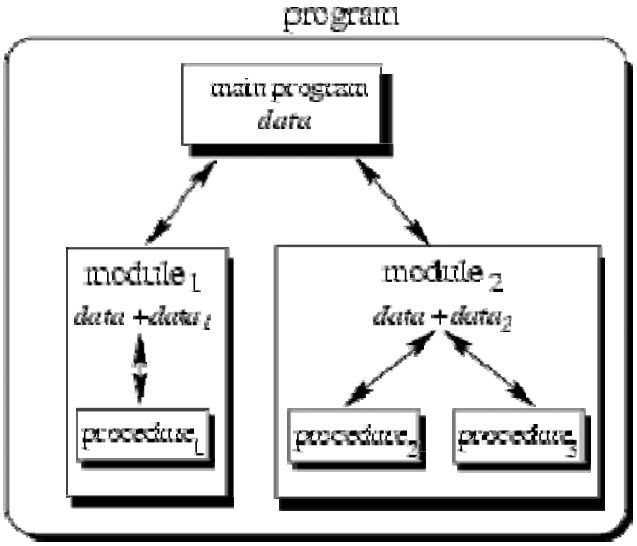

Fig.4.1: Modular Programming

With modular programming procedures, common functionality is grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program. Each module can have its own data. This allows each module to manage an internal *state* which is modified by calls to procedures of this module. However, there is only one state per module and each module exists at most once in the whole program.

3.3.1 Methods

A car can be seen as an object of a class vehicle. When you drive a car, pressing its accelerator sends a message to the car to perform a task i.e. make the car move faster. Similarly, messages are sent to an object of a class. Each message is known as a *method call* and tells a method of the object to perform its task. Methods are used to facilitate the design, implementation, operation and maintenance of large programmes. A method is invoked by a method call, and when the called method completes its task, it either returns a result or simply the control to the caller.

Although, most methods execute in response to method calls on specific objects, this is not always the case. Sometimes a method performs a task that does not depend on the contents of any object. Such a method is called a *static method.* An example of a static method is the main method. Every Java application has a main method. The main method is where Program execution begins. The main method is declared with the header shown below:

public static void main (String[]args)

Note: Static methods will be discussed in details in unit 3.

The syntax for creating a method is:

accessmodifier retuntype methodname([parameters])
{
Method body
[return value];
}

Note: Anything contained inside square bracket [] is optional.

In the above syntax, *accessmodifier* describes how other methods in the class will be able to access this method. The *accessmodifier* can either be *public or private.* A *private accessmodifier* makes the method only accessible to the members of the class; it cannot be used outside the class. A *public accessmodifier* makes the method accessible by other members of the class and it can also be used outside the class. The *returntype* describes the data type the method will return to its caller. Return type can be any valid data type. A method that has a return type of *int* returns an integer to its caller, a method that has a return type of *String* value returns a String value to its caller. Sometimes a method may not return any value to its caller. Such methods have a return type of *void.*

The *methodname* is any valid variable name. The method may or may not have parameters depending on the method action. If a method has parameters, when such method is to be called, the signature of the arguments must be equal to the specified signature in the called method. Note that signature refers to the number of arguments and the data type of the arguments. Just like classes, methods have a method body that begins and ends respectively with a left brace "{"and a right brace "}". A method may or may not return a value. A method that has a return type of *void* does not return any value to its caller, thus, the second to the last line is absent i.e. return value. It is important to Note that for methods that do not have a return type of void, the last line before the closing brace in the method should be the return value.

4.0 CONCLUSION

Polymorphism in java is implemented using late binding. The two main types of polymorphism are overloaded method and overridden methods. Overloaded methods are methods with the same name signature but either a different number of parameters or different types in the parameter list.

Methods are often called modules. They are like miniature programmes; you can put them together to form a larger program. Modular programming involves the procedures of a common functionality which are grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program.

5.0 SUMMARY

In this unit, the following have been discussed:

- Several methods may have the same name and the same formal parameter list in a class of hierarchy.
- A reference variable can invoke, that is, execute, a method of its own class or of its subclass(es). Binding means associating a method definition with its invocation, that is, determining which method definition gets executed.
- Overloaded methods are methods with the same name signature but either a different number of parameters or different types in the parameter list.
- Methods are often called modules. They are like miniature programmes; you can put them together to form a larger program. With modular programming procedures, common functionality is grouped together into separate *modules*.
- Although, most methods execute in response to method calls on specific objects, this is not always the case. Sometimes a method performs a task that does not depend on the contents of any object. Such a method is called a *static method*.

6.0 TUTOR-MARKED ASSIGNMENT

1. List the two static members of a class that we can have?
2. Explain why the main method is always declared as static?
3. Write a program that computes the volume of a sphere. The static field of Math class should be used to provide the constant required for this computation.

7.0 REFERENCES/FURTHER READING

Giuseppe Castagna (1997). *Object-Oriented Programming: A Unified Foundation*. Birkhaeuser.

Per Brinch Hansen (1972). Structured Multi-Programming, *CACM*, 15(7), pp. 574-578.

Philip Wadler (1995). Monads for functional programming: Advanced Functional Programming, *LNCS*, Vol. 925, Springer-Verlag, 1995.

UNIT 5 COMPOSITION AND ABSTRACTION

CONTENTS

1.0 INTRODUCTION

Volumes have been written regarding the application layers, web sites. One of the fundamental activities of any software system design is establishing relationships between classes. Two fundamental ways to relate classes are inheritance and composition. Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition. This unit discusses and compares these two approaches to relating classes and will also provide guidelines on their use.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define composition
- describe abstraction
- define an abstract class
- list abstract data types.

3.0 MAIN CONTENT

3.1 Composition

Composition is a relationship between two classes that is based on the aggregation relationship. Composition takes the relationship one step further by ensuring that the containing object is responsible for the lifetime of the object it holds. If Object B is contained within Object A, then Object A is responsible for the creation and destruction of Object B. Unlike aggregation, Object B cannot exist without Object A.

Examples 1.5.1: Imagine you create a Student class that holds information about individual students at a school. One piece of information stored is the student's date of birth. It's held in a Gregorian Calendar object:

```
import java.util.GregorianCalendar;

public class Student {

  private String name;
  private GregorianCalendar dateOfBirth;

  public Student(String name, int day, int month, int year)
  {
    this.name = name;
    this.dateOfBirth = new GregorianCalendar(year, month, day);
  }

//rest of Student class..

}
```

As the student class is responsible for the creation of the GregorianCalendar object it will also be responsible for its destruction (i.e., once the Student object no longer exists neither will the GregorianCalendar object). Therefore the relationship between the two classes is composition because Student has-a GregorianCalendar and it also controls its lifetime. The GreogrianCalender object cannot exist without the Student object.

It might be useful if the coffee cup object of a program could contain coffee. Coffee itself could be a distinct class, which your program could instantiate. One would recognise coffee with a type if it exhibits behaviour that is important to your solution. Perhaps it will swirl one way or another when stirred, keep track of a temperature that changes

over time, or keep track of the proportions of coffee and any additives such as cream and sugar.

To use composition in Java, you use instance variables of one object to hold references to other objects. For the CoffeeCup example, you could create a field for coffee within the definition of class CoffeeCup, as shown below by implementing the methods.

```
// In Source Packet in file inherit/ex1/CoffeeCup.java
class CoffeeCup {

   private Coffee innerCoffee;

   public void addCoffee(Coffee newCoffee) {
      // no implementation yet
   }

   public Coffee releaseOneSip(int sipSize) {
      // no implementation yet
      // (need a return so it will compile)
      return null;
   }

   public Coffee spillEntireContents() {
      // no implementation yet
      // (need a return so it will compile)
      return null;
   }
}

// In Source Packet in file inherit/ex1/Coffee.java
public class Coffee {

   private int mlCoffee;

   public void add(int amount) {
      // No implementation yet
   }

   public int remove(int amount) {
      // No implementation yet
      // (return 0 so it will compile)
      return 0;
   }

   public int removeAll() {
```

```
        // No implementation yet
        // (return 0 so it will compile)
        return 0;
    }
}
```

In the example above, the CoffeeCup class contains a reference to one other object, an object of type Coffee. Class Coffee is defined as a separate source file. The relationship modeled by composition is often referred to as the "has-a" relationship. In this case a CoffeeCup has Coffee. As you can see from this example, the has-a relationship doesn't mean that the containing object must have a constituent object at all times, but that the containing object may have a constituent object at some time. Therefore the CoffeeCup may at some time contain Coffee, but it need not contain Coffee all the time. (When a CoffeeCup object does not contain Coffee, its innerCoffee field is null.) In addition, note that the object contained can change throughout the course of the containing object's life.

3.2 Abstraction

Abstraction is the process or result of generalisation by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose. For example, abstracting a leather soccer ball to a ball retains only the information on the general ball attributes and behaviour. Similarly, abstracting happiness to an emotional state reduces the amount of information conveyed about the emotional state. Computer scientists use abstraction to understand and solve problems and communicate their solutions with the computer in some particular computer language.

Abstraction is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics. In object-oriented programming, abstraction is one of the three central principles (along with encapsulation and inheritance). Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. In the same way that abstraction sometimes works in art, the object that remains is a representation of the original, with unwanted detail omitted. The resulting object itself can be referred to as an abstraction, meaning a named entity made up of selected attributes and behaviour specific to a particular usage of the originating entity. Abstraction is related to both encapsulation and data hiding.

In the process of abstraction, the programmer tries to ensure that the entity is named in a manner that will make sense and that it will have all the relevant aspects included and none of the extraneous ones. A real-world analogy of abstraction might work like this: You (the object) are arranging to meet a blind date and are deciding what to tell them so that they can recognise you in the restaurant. You decide to include the information about where you will be located, your height, hair colour, and the colour of your jacket. This is all the data that will help the procedure work smoothly. You should include all that information. On the other hand, there are a lot of bits of information about you that aren't relevant to this situation: your social security number, your admiration for obscure films, and what you took to "show and tell" in fifth grade are all irrelevant to this particular situation because they won't help your date find you. However, since entities may have any number of abstractions, you may get to use them in another procedure in the future.

3.2.1 Abstract Classes

From the previous example, the superclass is more general than its subclass(es). The superclass contains elements and properties common to all of the subclasses. The previous example was of a concrete superclass that instance objects can be created from. Often, the superclass will be set up as an abstract class which does not allow objects of its prototype to be created. In this case, only objects of the subclass are used. To do this, the reserved word abstract is included in the class definition.

Abstract methods are methods with no body specification. Subclasses must provide the method statements for their particular meaning. If the method was one provided by the superclass, it would require overriding in each subclass. And if one forgot to override, the applied method statements may be inappropriate.

```
Public abstract class Animal  // class is abstract
{
    Private String name;
     Public Animal(String nm)  //constructor method
     { name=nm;}
      Public String getName()    //regular method
       { return (name); }
       Public abstract void speak();  // abstract method
}
```

3.2.2 Interfaces

Interfaces are similar to abstract classes but all methods are abstract and all properties are static final. Interfaces can be inherited (i.e. you can have a sub-interface). As with classes, the extend keyword is used for inheritance. Java does not allow multiple inheritances for classes (i.e. a subclass being the extension of more than one superclass). An interface is used to tie elements of several classes together. Interfaces are also used to separate design from coding as class method headers are specified but not their bodies. This allows compilation and parameter consistency testing prior to the coding phase. Interfaces are also used to set up unit testing frameworks.

As an example, we will build a Working interface for the subclasses of Animal. Since this interface has the method called work(), that method must be defined in any class using the Working interface.

```
public interface Working
{
    public void work();
}
```

When you create a class that uses an interface, you reference the interface with the phrase implements interface list. Interface list consists one or more interfaces, as multiple interfaces are allowed. Any class that implements an interface must include code for all methods in the interface. This ensures commonality between interfaced objects.

```
public class WorkingDog extends Dog implements Working
{
    public WorkingDog(String nm)
    {
        super(nm);  //builds parent
     }
     public void work()  // this method is specific to WorkingDog
{
     speak();
     System.out.printly(" I can herb sheep and cow");
     }
}
```

3.2.3 Abstract Data Types (ADT)

If you want to keep a collection of various elements such as address: students, employees, department and projects. This is a structure and is called a list; a list is an example of ADT (abstract data type). An ADT is an abstraction of a commonly appearing data structure, along with defined operations on the data structure. ADT is a data type that specifies the logical properties without the implementation details.

Historically, the concept of ADT in computer programming developed as a way of abstracting the common data structure and the associated operations. Along the way ADT provides information hiding. That is, ADT hides the implementation details of the operations and the data from the users of the ADT. Users can use the operation of an ADT without knowing the operation is implemented. An abstract data type (ADT) is characterised by the following properties:

1. It exports a type.
2. It exports a set of operations. This set is called interface.
3. Operations of the interface are the one and only access mechanism to the type's data structure.
4. Axioms and preconditions define the application domain of the type.

With the first property, it is possible to create more than one instance of an ADT as exemplified with the employee example. You might also remember the list example of chapter 2. In the first version, we have implemented a list as a module and were only able to use one list at a time. The second version introduces the “handle” as a reference to a “list object”. From what we have learned now, the handle in conjunction with the operations defined in the list module defines an ADT List:

1. When we use the handle we define the corresponding variable to be of type List.
2. The interface to instances of type List is defined by the interface definition file.
3. Since the interface definition file does not include the actual representation of the handle, it cannot be modified directly.
4. The application domain is defined by the semantic meaning of the operations. Axioms and preconditions include statements such as:

"An empty list is a list."

"Let l=(d1, d2, d3, ..., dN) be a list. Then l.append(dM) results in l=(d1, d2, d3, ..., dN, dM)."

"The first element of a list can only be deleted if the list is not empty."

However, all of these properties are only valid due to our understanding of and our discipline in using the list module. It is in our responsibility to use instances of List according to these rules.

3.2.4 Importance of Data Structure Encapsulation

The principle of hiding the used data structure and to only provide a well-defined interface is known as encapsulation. Why is it so important to encapsulate the data structure?

To answer this question, consider the following mathematical example where we want to define an ADT for complex numbers. For the following it is enough to know that complex numbers consists of two parts: real part and imaginary part. Both parts are represented by real numbers. Complex numbers define several operations: addition, subtraction, multiplication or division to name a few. Axioms and preconditions are valid as defined by the mathematical definition of complex numbers. For example, there exists a neutral element for addition.

If you think of more complex operations, the impact of decoupling data structures from operations becomes even clearer. For example the addition of two complex numbers requires you to perform an addition for each part. Consequently, you must access the value of each part which is different for each version. By providing an operation ``add" you can encapsulate these details from its actual use. In an application context, you simply ``add two complex numbers" regardless of how this functionality is actually achieved. Once you have created an ADT for complex numbers, for instance Complex, you can use it in the same way like other well-known data types such as integers.

3.2.5 Generic Abstract Data Types

ADTs are used to define a new type from which instances can be created. As shown in the list example, sometimes these instances should operate on other data types as well. For instance, one can think of lists of apples, cars or even lists. The semantical definition of a list is always the same. Only the type of the data elements change according to what type the list should operate on. This additional information could be specified

by a generic parameter which is specified at instance creation time. Thus, an instance of a generic ADT is actually an instance of a particular variant of the ADT. A list of apples can therefore be declared as follows:

List<Apple> listOfApples;

The angle brackets now enclose the data type for which a variant of the generic ADT List should be created. listOfApples offers the same interface as any other list, but operates on instances of type Apple.

4.0 CONCLUSION

Abstract classes and methods force prototype standards to be followed, that is, they provide templates. As ADTs provide an abstract view to describe properties of sets of entities, their use is independent from a particular programming language. Each ADT description consists of two parts:

- **Data**: This part describes the structure of the data used in the ADT in an informal way.
- **Operations**: This part describes valid operations for this ADT, hence, it describes its interface. We use the special operation constructor to describe the actions which are to be performed once an entity of this ADT is created and destructor to describe the actions which are to be performed once an entity is destroyed. For each operation, the provided arguments as well as preconditions and post-conditions are given.

The separation of data structures and operations and the constraint to only access the data structure via a well-defined interface allows you to choose data structures appropriate for the application environment.

5.0 SUMMARY

In this unit, the following were discussed:

- Composition takes the relationship one step further by ensuring that the containing object is responsible for the lifetime of the object it holds.
- to use composition in Java, instance variables of one object to hold references to other objects are employed.
- Abstraction is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.

- Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency.
- An interface is used to tie elements of several classes together. Interfaces are also used to separate design from coding as class method headers are specified but not their bodies. This allows compilation and parameter consistency testing prior to the coding phase.

6.0 TUTOR-MARKED ASSIGNMENT

Create a class called Employee that has fields that stores employee's first name, last name, identification number, department name and salary. Provide the necessary methods that provide meaningful output when the class is used. Ensure data hiding. Test your class with an executable class.

7.0 REFERENCES/FURTHER READING

Bill Venners (1998). Composition versus Inheritance: A Comparative Look at Two Fundamental Ways to Relate Classes, Published in JavaWorld.

Brad J. Cox (1984). "Message/Object Programming: An Evolutionary Change in Programming Technology, *IEEE Software*, 1(1).

Denis Caromel (1990). "Programming Abstractions for Concurrent Programming, Pacific '90, pp. 245-253, November 1990.

Jaffar, J. & Maher, M. (1994). "Constraint to Logic Programming: a Survey, *The Journal of Logic Programming*, no. 19, 20, pp. 503-581.

Sergey Dimitriev (2004). "Language-Oriented Programming: The Next Programming Paradigm, *Online Magazine*, 1(1), November 2004. BibTeX.

MODULE 2 OBJECT-ORIENTED PROGRAMMING PROPERTIES USING ANY PROGRAMMING LANGUAGE

UNIT 1 CLASSES AND OBJECTS PROPERTIES

CONTENTS

1.0 INTRODUCTION

This unit discusses how classes are created in Java (one of the most prominent and widely used Object-Oriented Programming Language). The properties of classes and how to control access to members of a class are also discussed. Object- oriented design (OOD) models software in terms similar to those that people use to describe real world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles have the same characteristics, for example, cars, lorries and trailers have many characteristics in common.

A class is the fundamental building block of code when creating object-oriented software. A class describes in abstract all of the characteristics and behaviour of a type of object. Once instantiated, an object is generated that has all of the methods, properties and other behaviours defined within the class. A class should not be confused with an object.

The class is the abstract concept for an object that is created at design-time by the programmer. The objects based upon the class are the concrete instances of the class that occur at run-time.

2.0 OBJECTIVES

At the end this unit, you should be able to:

- describe classes, objects, methods and instance variables
- state a class and use it to create an object
- declare methods in a class to implement the class's behaviours
- state instance variables in a class to implement the class's attributes
- list an object's methods to make those methods perform their tasks
- differentiate between instance variables of a class and local variables of a method.

3.0 MAIN CONTENT

3.1 Classes and Objects

Object-Oriented Programming (OOP) allows computer programmers to implement an object-oriented design as a working system and in such instances; the unit of programming is the class from which objects are created. Creating an instance (preferably object) of a class is called *Object Instantiation*. Classes exist in the general with abstract attributes but objects exist in the specific and automatically possess all the attributes of the general class.

Classes are to objects as blueprints are to houses, just as a house can be built from a blueprint. You cannot sleep in the room of a blueprint but you can sleep in the room of a house. Many objects can be instantiated from a class just like many houses can be built from a blueprint. OOP programmers concentrate on creating classes and specify how those classes will behave; they later instantiate objects of such classes to act as it has been specified in the general class.

3.2 Characteristics of a Class

Classes contain methods (class behaviours) which implement operations and fields which then implement attributes. The methods of a class help in manipulating the fields of the class and thus provide services to the clients of the class. The clients of a class are other classes that use the class.

Classes can relate with other classes e.g. in an object- oriented design of a hospital, the *"Patient"* class need to relate with the *"Nurse"* class which then relates with the *"Doctor"* class. All these classes cannot exist in isolation. These relationships are called *association.* The association property of a class makes software development an easy task because a large project can be broken down into smaller classes which then interact with one another to make the whole project.

Another property that makes a class a vital tool in Object- Oriented Programming is that it allows *software reusability*. Packaging software as classes makes it possible for future software systems to reuse the class. Groups of related classes are often packaged as reusable components. The most important factor that affects the future of software development is *software reuse*. The advantages of software reusability are as follows:

- Programmers build more dependable and efficient systems because existing classes and components often have gone through extensive testing, debugging and performance tuning.
- It saves programmers the time and effort they would have used in re-inventing the wheels.

With software reuse, each new class you create will have the ability to become an asset that you and other programmers can use to speed and enhance the quality of future software development efforts. Two major aspects of Object- Oriented Programming that promotes software reuse are Inheritance and Polymorphism. These two concepts will be discussed later in this module.

Note: Since we will be using Java as a case study of Object- Oriented Programming, everything that will be discussed from this moment on will be in relation to the Java Programming Language.

3.3 Predefined Classes and User Defined Classes

A great strength of Java is its rich set of predefined classes that can be reused rather than creating new classes. This is an aspect of software reuse discussed earlier. Related classes are grouped into packages and collectively referred to as the *Java Class Library* or the Java Application Programming Interface (API). These predefined classes are the building blocks for constructing new classes. Programmers use the *import* declaration to identify the predefined classes used in a Java Program. The syntax of the import statement is:

import packagename.classname;

If there are many classes in a package that programmers want to use, each class will be imported individually as follows:

Import packagename.class$_1$name;
Import packagename.class$_2$name;
……..
import packagename.class$_n$name;

This will be time consuming if there are many classes to be imported, in such instance; Programmers prefer to import all the classes in the package at once. When this is done they will be able to access whichever class they want to use. The syntax for importing all the classes in a package is:

import packagename.;*
Note: In Computer terminology, "" means all.*

Some packages are implicitly imported into every Java Program. Thus when the classes contained in such packages are to be used, there's no need to import such packages. An example of this is the package Java.lang which contains the classes System and String. *User defined classes* are the classes newly created by a programmer to assist in achieving his desired task. Every Java program consist of at least one user defined class declaration that is defined by the programmer. User defined classes are also known as *Programmer defined classes.* The *class* keyword introduces a user defined class declaration in Java and it is followed immediately by the class name. A Java class name is a valid identifier i.e. a series of characters consisting of letters digits, underscores and dollar sign that does not begin with a digit neither does it contain spaces.

The syntax for declaring a class is:

public class classname
{
class body
}

The *class body* contains methods that specify the actions that the objects of the class can exhibit. The class body begins and ends respectively with a left brace "{"and a right brace "}".

3.4 Creating a Simple Class

Here is an example of a Java class.

Example 2.1.1

Write a Java Program to print the message "I love Java Programming"

Program Input:

```
public class FirstProgram
{
 public static void main(String[] args)
 {
 System.out.println("I love Java Programming");
 }
}
// A program to display a line of text.
```

Program Output:

run:

I love Java Programming
BUILD SUCCESSFUL (total time: 2 seconds)

Program Analysis:

The name of the class is *FirstProgram.* This class is an executable class. An executable class is a class that has a main method. A non executable class is a class that does not have a main method. Such classes cannot execute on their own unless they are used inside an executable class. The *public static void main (String[]args)* is the header of the main method where program execution begins. Inside this main method we have the line of action to print out the text "*I love Java Programming*".

System.out is known as the standard output object. Thus, *System.out.printl()* displays or prints a line of text specified in the parenthesis. This is why the line of action in the main method was able to display *I love Java Programming* in the output. The message in parenthesis must be double quoted because it is regarded as a string literal. Next example creates a non executable class i.e. a class that has no main method. This class will then be used inside an executable class to carry out an action.

Example 2.1.2

Create a non executable class that has a method. This method prints a line of text that reads “Object- Oriented Programming is fun”.

Program Input:

```
public class MessageClass
{
public void displayMessage()
{
   System.out.println("Object Oriented Programming is fun");
}
}
```

Program Analysis:

The program has no output because it is a non executable class. An attempt to run this program raises an error but it can be compiled to check for errors. The name of the class is *MessageClass*, It has a public method called *displayMessage.* It is public because it will be still be used outside the *MessageClass.* Thus, public makes it accessible. This method returns no value to its caller thus it has a return type of void and it has no parameter. This is because it is only meant to print out a line of text and that is its main action. It is not acting on any data and it is not returning any value.

3.5 Object Instantiation

The methods of a non executable class will be useless if specific instances of such classes are not created. Just like the room in the blueprint of a house will be useless if a house is not built from the blueprint, if a house is built from the blue print, the rooms in the house will become useful. Objects are instances of a class that makes the actions defined in the class to be carried out. The act of creating an object of a class is called *object instantiation.* The syntax for creating an object of a class is:

Classname objectname=new Classname ();

In the syntax, *Classname* is the name of the class whose object is to be created and *objectname* is the name of the object, new represents a call to initialise the object. This is done by a constructor. Constructor will be discussed in details in unit 2. The new keyword is again followed by the class name. The parenthesis following the class name may or may not be empty depending on the constructor. Our next example creates an

executable class to instantiate the object of the class *MessageClass* created earlier. The method displayMessage() will then be called on the object of the class and the effect will be seen.

Example 2.1.3

Create an executable class. The object of MessageClass should be instantiated in this class and the displayMessage method of the MessageClass should be called on this object.

Program Input:

```
public class MessageClassTest
{
   public static void main(String[] args)
   {
       MessageClass msgclass=new MessageClass();
     msgclass.displayMessage();
   }
}
// An executable class that creates an object of MessageClass and
invokes the displayMethod method on the object.
```

Program Output:

```
run:
Object- Oriented Programming is fun
BUILD SUCCESSFUL (total time: 3 seconds)
```

Program Analysis:
MessageClassTest is an executable class, the object of *MessageClass* was created in this executable class and it is given the name *msgclass*. The method *displayMessage()* in *MessageClass* is called *msgclass*, this makes a line of text *Object-Oriented Programming is fun* to be displayed in the output.

3.6 Local Variables and Instance Variables

Local variables are variables that are declared in the body of a particular method, such variables can be used only in that method. When that method terminates, the values of its local variables are lost. Instance variable must be initialised to their default value before they are used. The syntax for declaring an instance variable within a method is:

Datatype variablename;

Datatype is any valid data type and *variablename* is any valid variable name. Instance variables are the attributes of a class. They are declared inside the body of a class but outside the bodies of the class's method declarations. They are also called Fields. Each object of a class has its own copy of attribute. Unlike local variables, instance variables are initialised by default by a constructor. Constructors will be discussed in details in Unit 2. The syntax for declaring an instance variable within a class is:

accessmodifier datatype variablename;
accesmodifier can either be private, public or protected

private makes the variable accessible only within the class, *public* makes it accessible within and outside the class, *protected* makes it accessible within the class and other subclasses of the class. You will understand how to use *protected* access modifier in unit 4 (inheritance).

Example 2.1.4

Create a non executable class that has a method that computes factorial.

Program Input:

```
public class FactorialClass {
   public void getFactorial (int num)
   {
      int result=1;
      for(int i=1; i<=num;i++)
            result=result*i;
      System.out.println("The factorial of "+num+" is "+result);
   }
}
// A non executable class that has a method that computes the factorial
of a number.
```

Program Analysis:

The method for the computation of factorial is an iterative process and it is expected that before taking OOP, you have an idea of control structures. The *getFactorial* method has a return type of void since it will only print out a line of text that shows the result of the factorial. The method has an argument since factorial has to be computed for a number. The datatype of the argument is integer and the name of the argument is *num.* The Factorial method also has a local variable *result* of integer data type. It is this variable that accumulates the value generated by each iterative process.

3.7 Set and Get Methods

Classes often provide public methods to allow clients (users) of a class to set (assign values to) or get (obtain values of) private instance variables. These methods are used to manipulate the instance variables of a class to respectively assign values to them (set) and retrieve values from them (get). When this is done, the instance variables of a class can be made private because these methods are used to internally manipulate them within the class. Other clients of the class do not have direct access to these private instance variables but they can be accessed through the get and set method. This is an important OOP concept and it is called *data hiding*. Another name for the get and set methods are *mutator* and *accessor* methods.

The names of these methods need not begin with set or get but this naming convention is highly recommended in Java. The next two examples explain how the *mutator* and the *accessor* methods can be used to manipulate the private instance variables of a class.

Example 2.1.5

Create a non executable class that has a private instance variable that can hold the first name of a student. Your class should contain methods to assign values to and retrieve values from the instance variable.

Program Input:

```
public class SetGetClass {
private String FirstName;
public void setFirstName(String name)
{
   FirstName=name;
}
public String getFirstName()
{
   return FirstName;
}
}
// A program that has a private instance variable and methods to manipulate the instance variable
```

Program Analysis

The instance variable *FirstName* is a String data type because it accepts the first name of students and it is set to be private because it will not be used directly outside the class, it will be used by the public methods that

are within the class. Method *setFirstName* stores a student's first name. It does not return any data when it completes its task, so its return type is void. The method accepts one parameter name which is the first name that will be passed to the method as an argument. In the body of this method, the value name in the parameter is assigned to the instance variable *FirstName*. i.e. this method sets the instance variable.

Method *getFirstName* retrieves the value of the class instance variable *FirstName*, this is why the method has a return type of string i.e it returns value of the instance variable which is a string data type. The method has an empty parameter list, so it does not require additional information to perform its task. Next example uses this class to see how the new methods work and the default value of the integer instance variable will be seen.

Example 2.1.6

Create an executable class that shows how the instance variables of the *SetGetClass2* can be manipulated by the methods of the class.

```
import java.util.Scanner;
  class SetGetClass2Test
{
   public static void main(String[] args)
   {
      Scanner myinput=new Scanner(System.in);
     SetGetClass2 setgetclass2=new SetGetClass2();
     String myname;
     int myscore;
     System.out.println("Unassigned first name is
"+setgetclass2.getFirstName());
      System.out.println("Unassigned score is "+setgetclass2.getScore());
      System.out.println("Enter your First name");
      myname=myinput.nextLine();
     System.out.println("Enter your Score");
     myscore=myinput.nextInt();
     setgetclass2.setFirstName(myname);
     setgetclass2.setScore(myscore);
     System.out.println(setgetclass2.getFirstName()+" scored
"+setgetclass2.getScore());
}
  }
// A Program that accepts user's input and uses the methods of
SetGetClass to manipulate the Input.
```

Program Output:

run:
Unassigned first name is null
Unassigned score is 0
Enter your First name
Rebecca
Enter your Score
90
Rebecca scored 90
BUILD SUCCESSFUL (total time: 12 seconds)

Program Analysis:

The new thing you will discover in the output is that the default value of an integer instance variable is 0. A careful examination of the lines of codes will make you understand it better.

4.0 CONCLUSION

The instance of a class is called an *object*. This is one of the reasons Java is known as an Object- Oriented Programming Language. A class may contain one or more methods that that are designed to carry out the class's tasks. A method can perform a task and return a result. Inside the method, you put the mechanisms that make the method do its tasks. Each message sent to an object is known as a *method call* and tells a method of the object's class to perform its task. A method declaration that begins with keyword *public* indicates that the method is "available to the public"- that is, it can be called by other classes declared outside the class declaration. Keyword *void* indicates that a method will perform a task but will not return any information when it completes its task.

When you attempt to execute a class, Java looks for the class's main method to begin execution. Any class that contains the main method can be used to execute an application. A method that belongs to another class may not be called until an object of the class is created. Variables declared in the body of a particular method are known as *local variables* and can be used only in that method. Variables declared in the body of a class are known as *instance variables*.

5.0 SUMMARY

In this unit, we have learnt the following:

- A class describes in abstract all of the characteristics and behaviour of a type of object. Once instantiated, an object is

generated that has all of the methods, properties and other behaviours defined within the class

- Creating an instance (preferably object) of a class is called *Object Instantiation*. Classes exist in the general with abstract attributes but objects exist in the specific and automatically possess all the attributes of the general class.
- Classes contain methods (class behaviours) which implement operations and fields which implement attributes. The methods of a class help in manipulating the fields of the class and thus provide services to the clients of the class.
- Related classes are grouped into packages and collectively referred to as the *Java Class Library* or the Java Application Programming Interface (API).
- The methods of a non executable class will be useless if specific instances of such classes are not created.

6.0 TUTOR-MARKED ASSIGNMENT

Review Exercise

i. What is the difference between a Class and an Object?
ii. List some of the properties of a class
iii. What do you understand by software reuse and list two advantages of software reusability.
iv. Differentiate between Predefined classes and user defined classes
v. Differentiate between local variables and instance variables
vi. Blueprints are to houses as Classes are to _____
vii. Man is to walk as a Class is to a _____

Programming Exercise

i. Write a Program to create an executable class to display the line of text.
"I am doing well in my Java lectures"
ii. Create a non executable class that has two methods that respectively calculate the perimeter and the area of a circle. The object of this class should be created in an executable class and the two methods should be invoked on it. Users should be able to enter the radius of the circle for processing. Your output should display a meaningful message that reflects the computation process.

7.0 REFERENCES/FURTHER READING

Deitel (nd). *Java*: *How to Program* (7th ed.).

Wrox (nd).*Beginning Java* ™ 2, JDK ™(5th ed.).

Zbingniew M. S. (nd).*Java Practical Guide for Programmers.*

UNIT 2 CONSTRUCTORS AND DESTRUCTORS

CONTENTS

1.0 INTRODUCTION

In this unit, you will learn how the object of a class is been initialised. Any time the object of a class is created, the object is implicitly initialised, and this makes the object have a default value. Constructors are used for such initialisations. Java provides a special kind of method, called a *constructor* that executes each time an object of a class is created. The constructor is used to initialise the state of an object. Apart from the constructor provided by Java, Programmers can also create their constructors, In such an instance, the mode of constructor creation looks like that of a method but care must be taken not to mistake methods for constructors. In this unit, you will learn how to create your own constructor and you will be able to differentiate between a method and a constructor.

There can be more than one constructor in a class, in such an instance we have overloaded constructions. When objects of that class are created, the default values of the objects depend on the kind of the constructor invoked on the object.

2.0 OBJECTIVES

At the end this unit, you should be able to:

- define a constructor
- explain the concept of object initialisation
- create a constructor
- differentiate between constructors and methods
- list multiple constructors in a class and use them as applicable.

3.0 MAIN CONTENT

3.1 Constructors

Constructors are special methods that are used to initialise a new object of a class. Constructors have the same name as the class; the name of the Student class's constructor is Student, the name of the Lecturer class's constructor is Lecturer, etc. If you don't define any constructor for your class, the compiler will supply a default constructor to the class that assigns a default value to the instance variable of a class's object. This was seen in unit 1, where the default value of a String instance variable was seen to be null and that of integer was seen to be 0. During object instantiation, the *new* keyword is used to invoke the constructor, in order for it to assign default values to the object's instance variable. If you don't want the instance variables of an object to take on default values, you must create your own constructor.

3.1.1 Default Constructor

When an object of a class SetGetClass2 is created, its instance variables FirstName and Score are respectively initialised to null and 0 by default. This initialisation is as a result of the default constructor provided by Java. It is always invoked during object instantiation with the *new* keyword as shown below:

Classname objectname=new Classname();

The empty parenthesis after the *Classname* indicates a call to the class's constructor without argument which is often the default constructor. The default value a constructor provides depends on the data type of an object's instance variables. Data types are divided into two categories; *primitive types* and *reference types.* The primitive types are *Boolean, byte, char, short, int, long, float* and *double*. All non primitive types are reference types, so classes which specify the types of objects are reference types. A String datatype is also a reference type. Primitive type instance variables of *char, short, int, long, float* and *double* are

initialised to *0* and variables of type *Boolean* are initialised to false by the default constructor. Reference type instance variables are initialised to null. This is why the *FirstName* and *score* instance variables of the SetGetClass are respectively initialised to *null* and *0* in the program output.

3.2 Creating Constructors

Constructors can be created to specify custom initialisation of objects of a class. For example, a programmer might want to specify a first name for the SetGetClass object of the *programming example 6* in unit 1. When the object is created as follows:

SetGetClass setgetclass= new SetGetClass("Rebecca");

In this case, the argument "*Rebecca*" is passed to the SetGetClass object's constructor and it is used to initialise the first name. The statement above requires that the class provides a constructor with a String parameter. A constructor is like a method but it differs from a method with the following two attributes:

- A constructor has no return type. It cannot even return void.
- A constructor must have the same name as the class.
- Methods must have a return type, even if it is not returning any value, it must return void. Methods usually don't have the same name with a class. The syntax for creating a constructor is;

```
public classname ([parameter])
{
Constructor body
}
```

The constructor has a public access modifier in order to make it accessible in other classes. This is so because most times, an object is always instantiated in another class and since a constructor is always invoked during object instantiation. A constructor has the same name with the class. A constructor may or may not have parameters. This is why *parameter* is in square bracket in the syntax showing that it is optional. A constructor that has no parameter is called a *no argument constructor* and the one that has parameter is called an *argument constructor.* The *no argument constructor* is different from the *default constructor* because it may not return the default value of an object's instance variable, the value it returns depends on the constructor's body.

3.2.1 Creating a No Argument Constructor

A no argument constructor is a constructor that is invoked with arguments. Such a constructor simply initialises the object as specified in the constructor's body. The next example creates a class that has a non default no argument constructor.

Example 2.2.1

Create a class that has instance variables of String and Integers which can respectively store a student's first name and age. In the class, create a constructor that assigns non default values to these variables.

Program Input:

```
public class StudentClass {
private String firstname;
private int age;
public StudentClass()
{
firstname="Grace";
 age=25;
 }
 public void setValues(String myname,int myage)
{
firstname=myname;
age=myage;
}
public String getFirstname()
{
return firstname;
}
public int getAge()
{
return age;
}
}
```

// A program that uses a non default no argument constructor to change the default values of instance variables.

Program Analysis

Two private instance variables of type String and int with names *firstname* and *age* respectively were used. The non default no argument constructor was created immediately after the instance variable declaration. The instance variables are initialised in the constructor's

body with values *Grace* and *25*. With this, it is expected that when the constructor is invoked, the default value of *firstname* and *age* will be Grace and 25 respectively. Other methods were also created to assign new values to and retrieve values from the instance variables (the set and get methods). This allows the user to enter inputs for *firstname* and *age*.

3.2.2 Making a Constructor Behave Like a Default Constructor

A constructor can be created to make it behave like a default constructor. To do this, in the body of the constructor, the instance variables will be initialised to their default values or the body of the constructor will be made empty. Our next two examples explain this.

Example 2.2.2

Modify the StudentClass to make its constructor behave like a default constructor.

Program Input:

```
public class StudentClass2 {
private String firstname;
private int age;
public StudentClass2()
{

}
public void setValues(String myname,int myage)
{
firstname=myname;
age=myage;
}
public String getFirstname()
{
return firstname;
}
public int getAge()
{
   return age;
}
}
```

// A class that has a non default constructor that behaves like a default constructor.

Program Analysis

The only difference in this program compared to StudentClass is the constructor body. The constructor body is empty; this makes instance variables of the class to be initialised to their default values. The next example tests this class to see the value that will be returned when an object of StudentClass2 is created.

Example 2.2.3

Create an executable class to an instantiate object of StudentClass2 and allow users to enter their first name and age. Display the values of the object's instance variables before and after user's input.

Program Input:

```
import  java.util.Scanner;
public  class StudentClass2Test
{
 public static void main(String[] args)
   {
 Scanner myinput=new Scanner(System.in);
 StudentClass2 stdclass2=new StudentClass2();
 String name;
 int age;
 System.out.println("The initial students's name is:
"+stdclass2.getFirstname());
   System.out.println("The initial students's age is:
"+stdclass2.getAge());
System.out.println("Enter your name");
name=myinput.nextLine();
System.out.println("Enter your age");
age=myinput.nextInt();
stdclass2.setValues(name,age);
System.out.println("The real name is: "+stdclass2.getFirstname());
System.out.println("The real age is: "+stdclass2.getAge());
}
}
// A program to test StudentClass2.
```

Program Output:

```
run:
The initial students's name is: null
The initial students's age is: 0
Enter your name
```

Grace
Enter your age
25
The real name is: Grace
The real age is: 25
BUILD SUCCESSFUL (total time: 16 seconds)

Program Analysis:

The new thing that will be discovered in this program when compared to *StudentClassTest* is in the output. Here, the initial values of the instance variables *firstname* and age are respectively null and 0. This is because of the non default constructor created in the *StudentClass*2 that behaves like a default constructor.

3.2.3 Creating an Argument Constructor

An argument constructor is a constructor that is invoked with arguments. Like a method, it specifies in its parameter list the data it requires to perform its task. When you create a new object of a class, this data is placed in the parenthesis that follows the class name i.e. the constructor of the class is called with an argument. Our next example shows how an argument constructor is created.

Example 2.2.4

Create a class that has a String instance variable and an int instance variable, in the class, create a constructor that has parameters that uses these instance variables as its data.

Program Input:

```
public class StudentClass3 {
   private String firstname;
   private int age;
   public StudentClass3(String myname,int myage)
   {
      firstname=myname;
      age=myage;
   }
public String getFirstname()
{
   return firstname;
}
public int getAge()
{
```

```
   return age;
}
}
// A class that shows how an argument constructor is created.
```

Program Analysis

The constructor of this class has a parameter that accepts a String value and an int value as data, the constructor body sets the instance variable of the class to the values in the parameter. This works like the set method, so there may not be any need to include the set method. That is why the set method is not included in the program. The get methods retrieve the values of the instance variables. The next example creates an object of Student3Class to see how the class's constructor works.

Example 2.2.5

Create an executable class and instantiate two objects of the StudentClass3 and display the values of their instance variables.

Program Input:

```
public class StudentClass3Test
{
     public static void main(String[] args)
   {
     StudentClass3 std1class3=new StudentClass3("Grace",25);
     StudentClass3 std2class3=new StudentClass3("Rebecca",38);
     System.out.println("The first students's name is:
"+std1class3.getFirstname());
     System.out.println("The first students's age is:
"+std1class3.getAge());
     System.out.println("The second students's name is:
"+std2class3.getFirstname());
     System.out.println("The second students's age is:
"+std2class3.getAge());
   }
  }
// An executable class that tests the StudentClass3.
```

Program Output:

```
run:
The first students's name is: Grace
The first students's age is: 25
The second students's name is: Rebecca
The second students's age is: 38
BUILD SUCCESSFUL (total time: 6 seconds)
```

Program Analysis

This is the first time we will be creating more than one object of a class. Two objects of StudentClass2 were instantiated and a constructor call was invoked on them. The constructor has two arguments that accept a first name and an age. Because each object has its own instance variable in memory, that is why we are able to create two objects with different instance variable. The instance variables for the first object are Grace and 25 while the instance variables for the second object are Rebecca and 38 as displayed in the output. These instance variables are set by the constructor during the constructor call.

3.3 Constructor Overloading

As you know, you can declare your own constructor to specify how objects of a class should be initialised. In this section, we will explain how multiple constructors can be used to initialise objects of a class in different ways. These constructors are called *overloaded constructors.* To overload constructors, simply provide multiple constructor declarations with different signatures. The compiler differentiates signatures by the number of parameters, the types of parameters and the order of the parameter types in each signature. This will be further explained in the next example.

Example 2.2.6

Create a time class that has 3 private instance variables hour, minute and second. Your class should have overloaded constructors that can be used to manipulate these instance variables.

Program Input:

```
public class TimeClass {
   private int hour;
   private int minute;
   private int second;
   public TimeClass()
   {
   }
   public TimeClass(int thehour)
   {
        hour=thehour;
   }
    public TimeClass(int thehour,int theminute)
   {
         hour=thehour;
```

```
        minute=theminute;
    }
     public TimeClass(int thehour, int theminute, int thesecond)
    {
        hour=thehour;
        minute=theminute;
        second=thesecond;
    }
public int getHour()
{
    return hour;
}
public int getMinute()
{
    return minute;
}
public int getSecond()
{
    return second;
}
public String displayTime()
{
    return
String.format("%02d:%02d:%02d",getHour(),getMinute(),getSecond());
}
}
// A time class that three time instance variables with constructors to
manipulate these variables differently.
```

Program Analysis

The time class has four constructors; the no argument constructor that does not accept any time parameter, the one argument constructor that accepts only the hour time parameter, the two argument constructors that accept the hour and the minute time parameters and the three argument constructor that accepts the hour, minute and the second time parameters. All these constructors assign values to the time instance variables in the constructor's body. The getMethods retrieve the values of the three time instance variables. The *displayTime* method returns a String value that shows each of the time parameter in two digits separated by ":"

3.4 Destructors (Tidying Up)

Destructors are used to tidy up the computer's memory. They are used to delete unused objects from the memory. The local variables created

within the body of a method are automatically deleted once the control is outside the method. This is not so for dynamically created objects. Objects remain in memory until they are explicitly deleted. A destructor is a special method typically used to perform cleanup after an object is no longer needed by the program. C++ supports destructors, but JAVA does not support destructors. JAVA supports another mechanism for returning memory to the operating system when it is no longer needed by an object. This mechanism is called *garbage collection*.

The garbage collector is a part of the runtime system that runs in a low-priority thread reclaiming memory that is no longer needed by objects used by the program. An object becomes *eligible* for garbage collection in JAVA when there are no reference variables that reference the object. The garbage collector makes use of a method *finalise* to perform its housekeeping. The *finalise* method is called by the garbage collector to perform termination housekeeping on object just before the garbage collector reclaims the object's memory. Method *finalise* does not take parameters and has return type void. A problem with method *finalise* is that the garbage collector is not guaranteed to execute at a specified period. Infact, the garbage collector may never execute before a program terminates. Thus, it is not certain that method *finalise* will be called all the time. For this reason, the *finalise* method is rarely used.

4.0 CONCLUSION

A constructor is used to initialise an object of a class when the object is created. Constructors can specify parameters but cannot specify return types. If no constructor is provided by a class, the compiler provides a default constructor with no parameters. When a class has a default constructor, its instance variables are initialised to their default values. Variables of types *char*, *byte*, *short*, *int*, *long*, *float* and *double* are initialised to 0, variables of *Boolean* type are initialised to false and reference variables are initialised to null.

If a constructor is provided by a class, the default constructor will not be called. Overloaded constructors enable objects of a class to be initialised in different ways. The compiler differentiates overloaded constructors by their signatures.

7.0 SUMMARY

In this unit, the following were discussed:

- Constructors are special methods that are used to initialise a new object of a class. During object instantiation, the *new* keyword is

used to invoke the constructor, in order for it to assign default values to the object's instance variable.

- All non primitive types are reference types, so classes which specify the types of objects are reference types. A String datatype is also a reference type. Primitive type instance variables of *char, short, int, long, float* and *double* are initialised to 0 and variables of type *Boolean* are initialised to false by the default constructor.
- A constructor is like a method but it differs from a method with the following two attributes:

- a constructor has no return type. It cannot even return void.
- a constructor must have the same name as the class.
- a constructor that has no parameter is called a *no argument constructor* and the one that has parameter is called an *argument constructor.*

- An argument constructor is a constructor that is invoked with arguments. Like a method, it specifies in its parameter list the data it requires to perform its task.
- When multiple constructors is used to initialise objects of a class in different ways, these constructors are called *overloaded constructors.*

6.0 TUTOR-MARKED ASSIGNMENT

Review Exercise

i. What is the purpose of the keyword *new*? Explain what happens when this keyword is used in an application.
ii. What is a default constructor? How is an object's instance variables initialised if a class has only a default constructor?
iii. Can a non default constructor and a default constructor be used in a class?
iv. Differentiate between a default constructor and a no argument constructor? Differentiate between a non default constructor and a method?
v. What is constructor overloading?
vi. Explain the concept behind constructor overloading.
vii. What are destructors?
viii. Explain the concept behind garbage collection.

Programming Exercise

i. Create a class called DateClass that includes three private instance variables. These variables are the year, month and day parameter of the DateClass. Your class should have a constructor

that initialises the three instance variables and assumes that the values provided are correct. Provide a *set* and *get* method for each instance variable. Provide a method displayDate that displays the year, month and day separated by forward slashes (/). Write a test application name DateClasssTest that demonstrates class Date's capabilities.

ii. Create a class called FreshStudentClass that includes four private instance variables- *firstname* that can accept a student's first name, *matricno* that can accept a student's matric number, *deptname* that can accept a students' department name and *paid* that states whether a student have paid his bills or not. Instance variable paid is a *Boolean* type that accepts true or false. Create four constructors in this class that manipulates this instance variable differently.

The FreshStudentClass should be tested to demonstrate the constructors' capabilities.

7.0 REFERENCES/FURTHER READING

Deitel Java *How to Program* (7th ed.).

Mary Campione & Kathy Walrath. The Java ™ Object-Oriented Programming for the Internet.

Wrox Beginning Java ™ 2, JDK ™(5th ed.).

UNIT 3 STATIC BEHAVIOURS

CONTENTS

1.0 INTRODUCTION

The methods, attributes and constructors that have been described so far are all related to instantiated objects of classes. In each case, these members have defined how the individual objects behave and how they maintain and manipulate their own state. Sometimes you will want to create behaviour that is not linked to any individual object; instead it will be available to all instances and to objects of other classes. This is where *static members* become useful.

In this unit, we will discuss the static behaviour of some class members – *methods and instance variables.* You will see that at times, you don't need to create an object of a class before you invoke the class's method, the method can be directly called on the class and not the object. Such methods are called *static methods.* You will also see that some predefined classes have static methods which can be invoked directly on the classes. It is common for some predefined classes to contain convenient static methods to perform their tasks. We will also discuss why method *main* is declared static. We discussed in unit one that every object of a class has its own copy of all the class's instance variable in memory i.e. there's a one-one relationship between the object of a class and its instance variables. In this unit, you will learn that in certain cases, only one copy of a particular variable should be shared by all objects of a class. A static field called a class variable (not an instance variable) is used in such a case. You will also learn in this unit, how the import statement can be used to import static members of a predefined class.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- explain how static methods and fields are associated with an entire class rather than specific instances of the class
- identity and use some common Math methods available in Java API
- create and use static methods
- identify and use some implicitly imported static methods
- import and use some static methods of a class
- list, create and use the static fields of a class.

3.0 MAIN CONTENT

3.1 Static Behaviours

Two members of a class can have the same static properties and they are the methods of the class and the fields of the class. When the fields of a class have a static behaviour, it is called a class variable. When these static members of a class are used, the object of the class need not be instantiated to make members carry out their work.

3.2 Static Methods

Although most methods execute in response to method calls on specific objects, this is not always the case in other methods. Sometimes, a method performs a task that does not depend on the contents of any object. Such a method applies to the class in which it is declared as a whole and is known as a *static method*, it can also be called a *class method.*

3.2.1 Creating Static Methods

The way a static method is created is not too different from the way normal methods are created. The only difference is the inclusion of the *static* keyword. The syntax for creating a static method is shown below:

```
accessmodifier static returntype methodname([parameter])
{
Method body;
}
```

The keyword *static* usually comes before the return type in the method's declaration and the method may or may not accept parameters. Static

methods are usually called by specifying the name of the class in which the method is declared, followed by a dot (.) and the method name as shown below:

Classname.methodname(arguments);

This saves the programmer the stress of instantiating the object of a class before the class's methods can be called. Our next examples explain how static methods can be created and called in programmes.

Example 2.3.1

Create a non executable class called FactorialClass that has a one argument method and two argument methods respectively called computeFactorial and computePermutation. The computeFactorial method computes the factorial of a number and the computePermutation method computes the permutation of two numbers. The two methods should be declared as static, the computeFactorial method should be used in the computePermutation method, thus it should be declared as private.

Program Input:

```
public class PermutationClass {
   private static int computeFactorial(int number)
   {
      int result=1;
      for(int i=1;i<=number;i++)
      {
         result=result*i;
      }
      return result;
   }
   public static int computePermutation(int number1,int number2)
   {
      int result;
      int diff;
      diff=number1-number2;
result=PermutationClass.computeFactorial(number1)/PermutationClas
s.computeFactorial(diff);
      return result;
   }
}
// A program that has two static methods that compute the factorial and
permutation of numbers.
```

Program Analysis:

The class has a static private method *computeFactorial* that computes the factorial of a number. The method is private because it is only been used within the class to compute the permutation of two number. The *computePermutation* method uses the static factorial to compute the permutation of two numbers. Note the way the computeFactorial method was used; it was prefixed with the class name and a dot (.).
Our next example uses the static *computePermutation* method to compute the permutation of two numbers.

Example 2.3.2

Create an executable class that accepts two numbers from the user; the computePermutation method should be called to compute the permutation of two numbers.

Program Input:

```
import java.util.*;
public  class PermutationClassTest
{
    public static void main(String[] args)
    {
        Scanner myinput=new Scanner(System.in);
   int num1,num2, result;
   System.out.println("Enter the first number");
   num1=myinput.nextInt();
   System.out.println("Enter the second number");
   num2=myinput.nextInt();
  result= PermutationClass.computePermutation(num1,num2);
 System.out.println(num1+" permutation "+num2+" is "+result);
    }
   }
// A program that uses the static computePermutation method of class
PermutationClass to compute the permutation of two numbers
```

Program Output:

```
run:
Enter the first number
5
Enter the second number
3
5 permutation 3 is 60

BUILD SUCCESSFUL (total time: 10 seconds)
```

Program Analysis:

An object of Scanner class was called to accept user's input, the static *computePermutation* method of the *PermutationClass* was called to compute the permutation of the two entered numbers. This was done without instantiating an object of PermutationClass.

3.2.2 Static Predefined Methods

There are some readymade static methods that can be used to achieve special tasks. To use such methods, the *import* statement will be used to import the class (if necessary) and the class's name will then be called directly on the object when the method is used. Some classes do not need to be imported when their static methods are to be accessed. The packages of such classes are implicitly imported by the compiler. A common example of such packages is the *Java.lang* package. The package has a class *Math* that has some static methods that enable you to perform common mathematical calculations. Some of these static methods are discussed below and our next programming example shows how they work.

Table 3.1: One Argument Static Methods of the Math Class

Method	Description	Example
abs(x)	The absolute value of x	*abs(2.5)=2.5, abs(-2.5)=2.5*
cos(x)	The cosine of x in radians	*cos(0.7)=0.7648*
sin(x)	The sine of x in radians	*sin(0.7)=0.6442*
tan(x)	The tangent of x in radians	*tan(0.7)=0.8422*
sqrt(x)	The square root of x in radians	sqrt(900)=30
ceil(x)	Rounds x to the smallest integer not less than x	*ceil(2.6)=3, ceil(-2.6)=-2*
floor(x)	Rounds x to the largest integer not greater than x	*floor(2.6)=2, floor(-2.6)=-3*

Table 3.2: Two Argument Static Methods of the Math Class

Method	Description	Example
max(x,y)	Finds the larger of x and y	*max(2,5)=5, max(-2,-5)=-2*
min(x,y)	Finds the smaller of x and y	min(2,5)=2, min(-2,-5)= -5
pow(x,y)	Finds x raise to the power of y	*pow(2,5)=32, pow(5,2)=25*

The first category of methods accepts only one argument. The second category has two arguments. Care must be taken in the way the

arguments are placed when the *pow* method is used. If the arguments are not placed in their correct position, a wrong answer may be returned. From the table above, you can see that *pow(2,5)* and *pow(5,2)* returns different values. The first argument is the base and the second one is the index. This is further explained in the next example.

Example 2.3.3

Create an executable class that asks the user to input a number, the number should be passed as argument to all the methods discussed and the value of the computation should be displayed.

Program Input:

```
import java.util.*;
public class MathClassTest1
{
   public static void main(String[] args)
   {
      Scanner myinput=new Scanner(System.in);
   double num1;
   System.out.println("Enter a number");
   num1=myinput.nextDouble();
    System.out.println("The absolute value of "+num1+" is
"+Math.abs(num1));
    System.out.println("The cosine of "+num1+" in radians is
"+Math.cos(num1));
  System.out.println("The sine of "+num1+" in radians is
"+Math.sin(num1));
  System.out.println("The tangent of "+num1+" in radians is
"+Math.tan(num1));
  System.out.println("The square root of "+num1+" is
"+Math.sqrt(num1));
  System.out.println("The ceiling of "+num1+" is "+Math.ceil(num1));
System.out.println("The floor of "+num1+" is "+Math.floor(num1));
   }
  }
// A class that calls some static methods of class Math to perform some
mathematical computations.
```

Program Output:

```
Enter a number
1.5
The absolute value of 1.5 is 1.5
The cosine of 1.5 in radians is 0.0707372016677029
```

```
The sine of 1.5 in radians is 0.9974949866040544
The tangent of 1.5 in radians is 14.101419947171719
The square root of 1.5 is 1.224744871391589
The ceiling of 1.5 is 2.0
The floor of 1.5 is 1.0
BUILD SUCCESSFUL (total time: 6 seconds)
```

Program Analysis:

An object of the Scanner class accepts user input and each of the static methods of class Math was called to make some computations. No object of the Math class was created to do this but the class name and a dot was prefixed to all these methods because they are static methods.

3.2.3 The Main Method

You may be wondering why the main method is always declared as static whenever it is used, you will know why it is always like that in this section. Whenever Java programmes are written, the Java compiler translates the Java source code into bytecodes, which are the codes understandable by the Java Virtual Machine (JVM). These bytecodes are executed by the Java Virtual Machine. A virtual machine is a software application that simulates a computer, but hides the underlying operating system and hardware from programmes that interact with the virtual machine. When the Java Virtual Machine (JVM) is executed with the Java command, the JVM attempts to invoke the main method of the class you specify; when objects of the class have been created. Declaring method main as static allows the Java Virtual Machine to invoke the main method without creating an instance of the class.

3.3 Static Fields

Every object has its own copy of the entire instance variables of the class. In certain cases, only one of a particular variable should be shared by all objects of a class. A static field is used in such an instance and it is usually called a *class variable.* A static field or a class variable has general information that is to be shared by all objects of the class. The syntax for creating a static field is;

accessmodifier static datatype fieldname;

A class's static members can be public or private. A class's public static members can be accessed through a reference to any object of a class, or by qualifying the member name with the class name and a dot (.). A class's private static fields can be accessed only through methods of the class. Declaring a class field to be static makes the field to exist even

when no objects of the class exist, they are available as soon as the class is loaded into memory at execution time. To access a public static field when no objects of the class exist (and even when they do), prefix the class name and a dot (.) to the static members. Private static members can only be accessed within the class. When they are used, a public static method must be provided and the method must be called by qualifying its name with the class name and a dot (.).

There are two major static fields in the Math Class of Java.lang, these fields are *PI* and *E*. These fields represent commonly used mathematical constants, to use them, prefix their names with the class name which is Math and a dot (.). Thus, we have *Math.PI* and *Math.E*. The constant *Math.PI* is the ratio of a class's circumference to its diameter and its value is *3.14159265358979323846*. The constant *Math.E* is the base value for natural logarithms and its value is *2.7182818284590452354*. These fields are declared in the class Math with the modifier *public*, *final* and *static*. Making them *public* allows other programmers to use these fields in their own classes. They are declared *final* because their values never change. Making them *static* allows them to be accessed through the class name Math and a dot (.) separator.

The difference between an instance variable and a class variable is that with an instance variable, the object of a class maintains its own copy of attribute, each object (instance) of the class has a separate instance of the variable in memory. Class variables are fields for which each object of a class does not have a separate instance of the field and such variables are declared as static and they are thus called static variables. When objects of a class containing static fields are created, all the objects of that class share one copy of the class's static fields. Class variables and instance variables make up the fields of a class. The next examples explain how static variables are used.

Example 2.3.4

Create an executable class that allows a user to enter a number that represents the radius of a circle, use the Math.PI static field of the Math class to compute the perimeter and the area of the circle.

Program Input:

```
import java.util.*;
  public class CircleTest
{
   public static void main(String[] args)
   {
      Scanner myinput=new Scanner(System.in);
```

```
    double radius;
    System.out.println("Enter the radius of a circle");
    radius=myinput.nextDouble();
     System.out.println("The perimeter of a circle with radius "+radius+"
is "+(2*Math.PI*radius));
System.out.println("The area of a circle with radius "+radius+" is
"+(Math.PI*radius*radius));
    }
   }
```

// A program that uses the static field PI of the Math class to calculate the perimeter and the area of a circle.

Program Output:

```
run:
Enter the radius of a circle
14
The perimeter of a circle with radius 14.0 is 87.96459430051421
The area of a circle with radius 14.0 is 615.7521601035994
BUILD SUCCESSFUL (total time: 4 seconds)
```

Program Analysis:

An object of Scanner class was instantiated to accept input that represents the radius of a circle from a user and the *Math.PI* was used to calculate the area and perimeter of the circle.

3.4 Static Imports

A static import declaration enables you to refer to imported static members as if they were declared in the class that uses them because the class name and a dot(.) will not be required any more. A static import has two forms; the *single static import* and the *static import on demand.* The former imports a particular static member whereas the later imports all static members of a class. The syntax for using the single static import is:

import static packagename.classname.staticmembername;

In the syntax, *packagename* is the package of the class (e.g Java.lang), *classname* is the name of the class (e.g Math) and *staticmembername* is the name of the static field or method (e.g cos or PI). The syntax for using the static import on demand is:

import static packagename.classname.;*

In the syntax, the *packagename and classname* represents the package of the class and the name of the class respectively. The asterisk (*) denotes that all static members of the specified class should be available for use in the program. It should be noted that static import declarations import only static class members. The next example shows how the static import is used and its effect on a class.

Example 2.3.5

Import all static methods of class Maths explicitly and use some of these methods to perform some computations without the class name and the dot(.).

Program Input:

```
import java.util.*;
   import static java.lang.Math.*;
 class StaticImportClassTest
{
    public static void main(String[] args)
    {
   Scanner myinput=new Scanner(System.in);
   double num1,num2,num3;
   System.out.println("Enter a number");
   num1=myinput.nextDouble();
   System.out.println("Enter another number");
   num2=myinput.nextDouble();
   System.out.println("Enter the last number");
   num3=myinput.nextDouble();
  System.out.println("The square root of "+num1+" is "+sqrt(num1));
System.out.println("The cosine of "+ num2+" is "+cos(num2));
System.out.println("The larger number between "+num2+" and
"+num3+" is "+max(num2,num3));
    System.out.println(num2+" raise to the power of "+num3+" is
"+pow(num2,num3));
    }
   }
// A class that explicitly imports all the static methods of class Math.
```

Program Output:

```
run:
Enter a number
49
Enter another number
2
```

```
Enter the last number
3
The square root of 49.0 is 7.0
The cosine of 2.0 is -0.4161468365471424
The larger number between 2.0 and 3.0 is 3.0
2.0 raise to the power of 3.0 is 8.0
BUILD SUCCESSFUL (total time: 11 seconds)
```

Program Analysis:

All the static methods in the Math class were imported explicitly and when those methods were called, they were no prefixed with the class name and a dot(.) as if they were created inside the class.

4.0 CONCLUSION

A class may contain static methods to perform common tasks that do not require an object of the class. A static method is called by specifying the name of the class in which the method is declared followed by a dot(.) and the method name. A static variable represents class wide information that is shared among all objects of the class. Static members exist even when no objects of the class exist; they are available as soon as the class is loaded into memory at execution time. To access a private static member when no objects of the class exist, a public static method must be provided. A static import declaration enables programmers to refer to imported static members without the class name and a dot (.). A single static import declaration imports one static member, and a static import on demand imports all static members of a class.

7.0 SUMMARY

In this unit, the following were discussed:

- When methods directly called on the class and not the object, such methods are called *static methods.* A static field called a class variable (not an instance variable) is used in such a case.
- The keyword *static* usually comes before the return type in the method's declaration and the method may or may not accept parameters. Static methods are usually called by specifying the name of the class in which the method is declared.
- Some classes do not need to be imported when their static methods are to be accessed. The packages of such classes are implicitly imported by the compiler. A common example of such packages is the *Java.lang* package.

- A virtual machine is a software application that simulates a computer, but hides the underlying operating system and hardware from programmes that interact with the virtual machine.
- When the Java Virtual Machine (JVM) is executed with the Java command, the JVM attempts to invoke the main method of the class you specify; when objects of the class have been created.
- A class's static members can be public or private. A class's public static members can be accessed through a reference to any object of a class, or by qualifying the member name with the class name and a dot (.).
- The difference between an instance variable and a class variable is that with an instance variable, the object of a class maintains its own copy of attribute, each object (instance) of the class has a separate instance of the variable in memory.

6.0 TUTOR-MARKED ASSIGNMENT

Review Exercise

i. Differentiate between a public and a private access modifier?
ii. Differentiate between an instance variable and a static field?
iii. What is the value of x after each of the following statement is executed:

a. x=Math.abs(-7.5);
b. x=Math.floor(-2.7);
c. x=Math.ceil(6.3);
d. x=Math.flooe(4.1);
e. x=Math.ceil(-2.5);
f. x=Math.pow(4,6);
g. x=Math.pow(7,2);
h. x=Math.ceil(-Math.abs(-5+Math.floor(-3.2)));

Programming Exercise

Write a program that works like a clock. Your class should be given the name *Time* and it should have three private static fields *hour, minute* and *second* that respectively represents the hour, the minute and the second of a time. Whenever an object of time is created, it should be assumed that 20 ticks have occurred, thus the second should increase by 20, when it reaches 60, the minute should be increased by 20 and when the minute reaches 60, the hour should increase by one. Test your time class and ensure that it is working perfectly.

7.0 REFERENCES/FURTHER READING

Deitel Java *How to Program* (7th ed.).

Wrox Beginning Java ™ 2, JDK ™ (5th ed.).

UNIT 4 INHERITANCE AND COMPOSITION

CONTENTS

1.0 INTRODUCTION

This unit discusses an important OOP concept that promotes software reusability. This concept is called *inheritance.* Inheritance allows a new class to be created by using an existing class's members and manipulating them with new or modified capabilities. Inheritance saves the time programmers would have otherwise used re-inventing the wheels. It has increased the possibility that system will be effectively implemented. Inheritance allows classes to associate with one another. When creating a class, instead of declaring completely new members, you can let the new class inherit the members of an existing class. In such an instance, the existing class is called the *superclass* and the new class is called the *subclass.*A subclass can be a superclass for future subclasses and a superclass can be a subclass for some superclasses.

Software development has shown that significant amounts of code deal with closely related special cases. When the developer is preoccupied with special cases, the details can obscure the big picture. With object-oriented programming, the programmer focuses on commonalities among objects in the system rather than on the special cases. New classes can inherit from classes in class libraries and thus the development of more powerful, abundant and economical software will be facilitated. Inheritance is of two types- *single and* multiple inheritances. In single inheritance, a class is derived from one direct superclass whereas in a multiple inheritance, a class is derived from more than one direct superclass. Java does not support a multiple inheritance.

Another OOP concept which will be discussed in this unit is Composition. Composition allows a class to have references to objects of other classes as members. An example of composition in a real life sense is an object of AlarmClock class, this object needs to know the current time and the time when it is supposed to sound its alarm, so it is reasonable to include two references to Time objects as members of the AlarmClock object.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- explain how inheritance promotes software reusability
- describe the concepts of superclasses and subclasses
- list the properties of a class that inherits fields and methods from another class
- outline how the object of a class can reference members of another class (composition).

3.0 MAIN CONTENT

3.1 Inheritance

Inheritance allows classes to associate with themselves that is, it promotes association within classes. This is one of the properties of a class. There is always a hierarchical structure with inheritance. This hierarchy is represented by classes and superclasses. A superclass is a more general class whereas a subclass is a more specific class.

3.2 Inheritance Properties

An example of a superclass and a subclass is a vehicle and a car respectively. This is an example of inheritance relationship. Inheritance is an *is-a* relationship whereby an object of a subclass can also be treated as an object of its superclass e.g. a car (a subclass) is an object of vehicle (a superclass). A car is a specific type of vehicle but it is incorrect to say that every vehicle is a car because a vehicle could be a lorry, trailer, bicycle or even a motorcycle. The set of objects represented by a superclass is typically larger than the set of objects represented by any of its subclasses because every subclass object is an object of its superclass and one superclass can have many subclasses. Example, the superclass vehicle represents all vehicles, including cars, trucks, motorcycles, boats, etc. whereas a subclass lorry represents a smaller more specific subset of vehicles.

In some cases, a class can be a superclass and also a subclass. For example, in the Shape class, A shape can be two dimensional or three dimensional, thus shape is the superclass for two dimensional shapes and three dimensional shapes, two dimensional shapes and three dimensional shapes are the subclasses for shape. However, a two dimensional shape can be a circle, square, triangle, thus it (two dimensional shape) becomes a superclass for circle, square and triangle. A three dimensional shape can be a sphere, cube or tetrahedron, thus it (three dimensional shape) becomes a superclass for sphere, cube, and tetrahedron. In this example, a two dimensional shape are subclasses of shape and superclasses of circle, square, triangle and sphere, cube, tetrahedron respectively.

It is possible to treat superclass objects and subclass objects similarly-their commonalities are expressed in the members of the superclass. Objects of all classes that inherits a common superclass can be treated as objects of that superclass i.e (such objects have an *is-a* relationship with the superclass). However, superclass objects cannot be treated as objects of their subclasses e.g. all cars are vehicles but not all vehicles are cars. Sometimes, a subclass can inherit methods that it does not need, in such an instance, the method can be customised in order to make it useful to the subclass i.e. the subclass will override the superclass method with an appropriate implementation.

3.2.1 Protected Members

Members can have access modifier *protected* in their classes in order to make them accessible by the subclass of that class. In our previous units, it was discussed that a class's public members are accessible wherever the program has a reference to an object of that class or one of its subclasses. A class's private members are accessible only from within the class itself. A superclass's private members are accessible only from within the class itself. A superclass's private members are not inherited by its subclasses. When the members of a superclass are declared as protected, the members of its subclasses and members of other classes in the same package can also access those members.

All public and protected superclass members retain their original access modifiers when they become members of the subclass (i.e. the public members of the superclass become the public members of the subclass and protected members of the superclass become the protected members of the subclass).

Subclass methods can refer to public and protected members inherited from the superclass simply by using the members' names. When a subclass modifies a superclass method, the superclass can be accessed

from the subclass by preceding the superclass method with keyword *super* and a dot (.) separator. This is called *method overriding.*

Methods of a subclass cannot directly access private members of their superclass. A subclass can change the state of the private superclass instance variables only through non-private methods provided in the superclass and inherited by the subclass. Our next example shows the concept of superclasses and subclasses.

Example 2.4.1

In a training Institute, the pre-requisite for being a member is to write three exams (mathematics, English and Science) and score at least a credit. There is an elective course (vocational study) that is optional for the aspiring members. Write a non executable class that has all the appropriate variables and methods that determines members' grades and gives comment showing their first names, last names, their grades and comment whether they are bonafide member of the institution or not. The scoring for each grade is shown below:

Grade A – 70 and above
Grade B- 60-69
Grade C-50-59
Grade D- 40-49
Grade F- 39 and below.

Note: Design the class in such a way that it will be used easily by members offering the elective course.

Program Input:

```
public class Member {
protected String firstname;
protected String lastname;
protected int mathscore;
protected int englishscore;
protected int sciencescore;
public Member (String first, String last, int course1,int course2, int
course3)
  {
    firstname=first;
    lastname=last;
    mathscore=course1;
    englishscore=course2;
    sciencescore=course3;
  }
```

```
public double  getAverage()
{
  return (mathscore+englishscore+sciencescore)/3;
}
public char getGrade()

{
   double avg;
   int approx;
   char grade=0;
   avg =getAverage();
   approx=(int)(avg);
   switch(approx/10)
   {
      case 10:
      case 9:
      case 8:
      case 7:
       grade='A';
         break;
      case 6:
         grade='B';
               break;
      case 5:
         grade='C';
         break;
      case 4:
         grade='D';
         break;
      case 3:
      case 2:
      case 1:
         grade='F';
         break;
   }
      return grade;
}
public void getComment(char grade)
{
   if (grade=='A'||grade=='B'||grade=='C')
      System.out.println(firstname+" "+lastname+": "+"A bonafide
member");
   else
 System.out.println(firstname+" "+lastname+": "+"Not a bonafide
member");
}
```

```
}
// A superclass Member that has five protected members.
```

Program Analysis:

The class has five protected fields – *firstname*, *lastname*, *mathscore*, *englishscore* and *sciencescore* that respectively store the first name, last name, mathematics score, English language score and science score of intending members. The variable are declared as *protected* because we will create a class ElectiveMember which is a subclass of this class, protected makes all these five variables accessible to members of this subclass. The remaining lines of codes are fundamental programming concepts which we have either discussed here or you have a prior knowledge of.

3.2.2 Data Hiding

Declaring the fields of a class as protected is not a good software development procedure. Doing so may make the subclass of that class to modify the protected members, which can invariably change the services provided by the class. The best software development procedure is to declare the fields of a class as private and then make them available to other classes through the public methods of the class. When this is done, other classes do not have a direct access to fields of a class, they can only access the public methods of the class. This procedure is called *data hiding*. The next example shows how the Member class can be rewritten and used by his subclass to promote good software development procedure.

Example 2.4.2

From the example above, rewrite the Member class in a way that promotes data hiding.

Program Input:

```
public class Member2 {
   private String firstname;
   private String lastname;
   private int mathscore;
   private int englishscore;
   private int sciencescore;
   public Member2 (String first, String last, int course1,int course2, int
course3)
   {
      firstname=first;
```

```
        lastname=last;
        mathscore=course1;
        englishscore=course2;
        sciencescore=course3;
    }
public double  getAverage()
{
  return (mathscore+englishscore+sciencescore)/3;
}
public char getGrade()

{
    double  avg;
    int approx;
    char grade=0;
    avg =getAverage();
    approx=(int)(avg);
    switch(approx/10)
    {
        case 10:
        case 9:
        case 8:
        case 7:
         grade='A';
            break;
        case 6:
            grade='B';
                  break;
        case 5:
            grade='C';
            break;
        case 4:
            grade='D';
            break;
        case 3:
        case 2:
        case 1:
            grade='F';
            break;
    }

        return grade;
}
public void getComment(char grade)
{
    if (grade=='A'||grade=='B'||grade=='C')
```

```
      System.out.println(firstname+" "+lastname+": "+"A bonafide
member");
   else
       System.out.println(firstname+" "+lastname+": "+"Not a bonafide
member");
}
}
//  A modified version of the Member class that promotes data hiding.
```

Program Analysis:

The only new thing introduced is that all the fields that were declared as protected before are now declared as private. This is to enhance data hiding.

Example 2.4.3

Using the Member2 class as subclass, create a subclass that provides the functions that can handle the necessary computations for students taking the elective course.

Program Input:

```
public class ElectiveMember2 extends Member2{
private int vocscore;
public ElectiveMember2(String first,String last, int course1,int course2,
int course3, int course4)
 {
  super(first,last,course1,course2,course3);
  vocscore=course4;
   }
   public double getAverage()
   {
      return super.getAverage()+(vocscore/4);
   }
}
//  A subclass (ElectiveMember2) for the superclass Member2
```

Program Analysis:

The class has a new name that inherits from the modified Member class. In the getAverage method, a call was made to the public method of the superclass to return the average of scores in maths, English and science, a quarter of the score in vocational studies is then added. This is because this class no longer has a direct access to these three fields just because they have been declared as private.

3.3 Composition

People often confuse composition for inheritance and vice versa. This is because these two concepts allow class association. However, there is a striking difference between inheritance and composition. Inheritance is an *is-a* relationship whereas composition is an *has-a* relationship. In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass, e.g. a car (a subclass) is a vehicle (superclass). By contrast, in an *has-a* relationship, an object has reference to other objects e.g. a car has a steering wheel (i.e. a car object has a reference to a steering wheel object). Composition allows a class to have references to objects of other classes as members. It is also a form of software reuse. Next examples explain composition. In these examples, three classes will be created, the first class is the Date class and this class has three private instance variables of date parameters- *day*, *month* and *year*.

The second class is the Student class and it has three private variables pertaining to a student- *firstname, lastname*, and *birthdate.* Birthdate is an object of the Date class, thus, a student object will have a reference to members of the Date class because a student must have a birth date. The third class tests the Student class to further explain the concept of composition.

Example 2.4.4

Create a class Date that has all the three date parameters- *day, month* and *year*. This class should have a method that displays date in a meaningful format.

Program Input:

```
public class Date {
   private int day;
   private int month;
   private int year;
   public Date(int theday, int themonth, int theyear)
   {
      day=theday;
      month=themonth;
      year=theyear;
   }
   public  String toString()
   {
      return
String.format("%02d%s%02d%s%d",day,"/",month,"/",year);
   }
}
// A date class that displays date.
```

Program Analysis:

The three date parameters – day, month and year are declared private to promote data hiding. A new method toString was called to display the date in a meaningful format. This method is inherited from the implicit direct superclass of Date called Object. Like it was said earlier all classes have a direct or indirect superclass called Object. Method toString is one of the most prominent methods of the Object class. The next example shows how the object of a class can reference the variables of a class. You will create a Student class, since every student is expected to have a date of birth; you will reference the Date class in the class.

Example 2.4.5

Create a class Student that has a field birthdate which is of the date type. The class should have methods that can display the first name, last name and the birth date of students.

Program Input:

```
public class Student {
 private String firstname;
   private String lastname;
   private Date birthdate;
   public Student(String first, String last, Date dateOfBirth)
   {
      firstname=first;
      lastname=last;
      birthdate=dateOfBirth;
   }
   public void getComment()
   {
      System.out.println("Student's name: "+firstname+" "+lastname);
      System.out.println("Birth Date: "+birthdate.toString());
   }
}
// A Student class that references the Date type as one of its fields.
```

Program Analysis:

The field *birthdate* is of the Date class, thus, it is a date datatype. The constructor also takes as part of its argument a date data type *dateOfbirth* since every student is expected to have a date of birth which is a date data type. When the toString method is called on a date object,

the date is displayed in the format specified by the Date class in its toString method.

Example 2.4.6

Test the Student class created in the previous example with an executable class.

Program Input:

```
public class StudentTest {
public static void main(String[] args) {
        Date birthdate=new Date(13,01,1990);
       Student student=new Student("Grace","Omotoso", birthdate);
       student.getComment();
   }
// A program to test the Student class that has a data type variable.
```

Program Output:

```
run:
Student's name: Grace Omotoso
Birth Date: 13/01/1990
BUILD SUCCESSFUL (total time: 1 second)
```

Program Analysis:

An object of Date class was first created and later passed to the constructor during object instantiation. The getComment method of the Student class was then called to display the first name, last name and birth date of students.

4.0 CONCLUSION

The direct superclass of a subclass (specified by the keyword *extends* in the first line of a class declaration) is the superclass from which the subclass inherits. An indirect superclass of a subclass is two or more levels up in the class hierarchy from that subclass. Thus, the object class is either a direct superclass for a class (a class that does not explicitly inherit from another class) or an indirect superclass for a class (a class that explicitly inherits from another class). A subclass is more specific than its superclass and represents a smaller group of objects. Every object of a subclass is also an object of that class's superclass. However, a superclass object is not an object of its class's subclasses.

A superclass's public members are accessible wherever the program has a reference to an object of that superclass or one of its subclasses. A superclass's private members are accessible only within the declaration of the superclass. A superclass's protected members have an intermediate level of protection between public and private access. They can be accessed by members of the superclass by members of its subclasses and by members of other classes in the same package. When a subclass method overrides a superclass method, the superclass method can be accessed from the subclass if the superclass method is preceded by the keyword *super*. A class can have references to objects of other classes as members. Such a capability is called *composition* and it is sometimes referred to as an *has-a* relationship.

5.0 SUMMARY

In this unit, we have learnt the following:

- Inheritance is an *is-a* relationship whereby an object of a subclass can also be treated as an object of its superclass e.g. a car (a subclass) is an object of vehicle (a superclass).
- It is possible to treat superclass objects and subclass objects similarly- their commonalities are expressed in the members of the superclass.
- Objects of all classes that inherits a common superclass can be treated as objects of that superclass i.e (such objects have an *is-a* relationship with the superclass).
- Members can have access modifier *protected* in their classes in order to make them accessible by the subclass of that class.
- A superclass's private members are not inherited by its subclasses. When the members of a superclass are declared as protected, the members of its subclasses and members of other classes in the same package can also access those members.
- Composition allows a class to have references to objects of other classes as members. It is also a form of software reuse.
- A superclass's private members are accessible only within the declaration of the superclass. A superclass's protected members have an intermediate level of protection between public and private access.

6.0 TUTOR-MARKED ASSIGNMENT

Review Exercise

i. Mention and explain the two forms of software reuse discussed in this unit
ii. Differentiate between public, private and protected variables? Which of the variables promotes good software engineering?
iii. Differentiate between inheritance and composition?
iv. Mention the two types of inheritance we have. Which of this form does Java support.
v. There are classes that can act either as a direct superclass of an indirect superclass to all classes. What is the name of this class and mention one of the most prominent methods of this class?

Programming Exercise

In some organisations, some employees are given commission in addition to their salary based on their sales and commission rate. Create a class called Commission Employee that uses the Employee class as a superclass. Provide two new private fields. *commisionrate* and *sales.* Create a method that shows the earnings of a Commission Employee. Test your class with an executable class.

7.0 REFERENCES/FURTHER READING

Deitel Java *How to Program* (7th ed.).

SAMS Teach Yourself Borland JBuilder ™ 2 in 21 Days.

Wrox Beginning Java ™ 2, JDK ™ (5th ed.).

Zbigniew M.S. Java: Practical Guide for Programmers.

UNIT 5 POLYMORPHISM

CONTENTS

1.0 INTRODUCTION

In this unit, we will discuss a very useful OOP concept called *polymorphism* that allows software reuse. It enables programmers to program in the general rather than in the specific. Polymorphism enables programmers to write programmes that process objects that share the same superclass in a class hierarchy as if they were objects of the superclass. This significantly helps to simplify programming. With polymorphism, programmers can design and implement systems that are easily extensible, new classes can be added with little modification to the general portions of the program.

People often find difficulty in differentiating between polymorphism and inheritance, at the end of this unit; you will understand the difference between these two OOP concepts. You will also learn some OOP terminologies and concepts like interfaces, abstract classes and abstract methods.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- explain the concept of polymorphism
- use overridden methods to effect polymorphism
- distinguish between abstract and concrete classes
- design abstract methods to create concrete abstract classes
- explain how polymorphism makes systems extensible and maintainable
- determine an object's type at execution.

3.0 MAIN CONTENT

3.1 Polymorphic Behaviours and Examples

An example of polymorphic behaviour is a programme that simulates the movement of several types of Vehicles for a mechanical study. Classes *car, bicycle* and *boat* represent the three types of vehicles under investigation; assuming that each of the classes has a superclass *Vehicle* that contains a method *move* and maintains a vehicle current position. Each subclass of Vehicle implements method *move.* The program maintains an array of references to object of the various Vehicle subclasses. To simulate the vehicles' movements, the programme sends each object of Vehicle the same message once per second- the message is called *move.* When this message is received by the objects, each specific type of vehicle represents a move message in a distinct way. The program issues the message *move* to each vehicle object as a group but each object knows how to modify its position appropriately for its specific type of movement.

Polymorphism is the act of designing a class in a way that the objects of the class behave in different ways. With polymorphism, programmers can rely on various objects of a class to do the right thing (i.e. do what is appropriate for that type of object) in response to the same method call. Just like in our example, the same message (in this case, *move*) sent to various objects have "*many forms*" of results; this is where the term polymorphism is derived.

Another example of polymorphism is with *Quadrilaterals.* A Quadrilateral class can have subclasses *Square, Parallelogram* and *Trapezium.* Suppose the Quadrilateral class has a method *getArea* that computes the area of a quadrilateral because a quadrilateral has an area, when the three objects of the Quadrilateral subclasses are called, they all inherit the *getArea* method but they will return different values even though they are using the same method.

With polymorphism, the same method and signature can be used to cause different actions to occur, depending on the type of object on which the method is invoked.

Polymorphism promotes extensibility because software that invokes polymorphic behaviour is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the base system.

3.2 Abstract Classes and Concrete Classes

Not all classes can be used for object instantiations, most times, such classes acts as a superclass and their main work is to make other classes inherit their methods and to share a common design. Such classes are called *abstract* classes. While abstract classes cannot be used for object instantiation, *concrete* classes are classes that can be used for object instantiation. The classes discussed so far in all our earlier examples are concrete classes. Concrete classes provide implementations of every method they declare whereas abstract classes contain at least one method that does not have implementation.

When polymorphism is discussed, inheritance has to come to play, thus, there is usually a superclass. A superclass that can behave in many forms is thus created. This superclass may contain at least one method that does not provide the actual method implementation; such a superclass is called an *abstract superclass.* Abstract superclasses are too general to create real objects – they specify only what is common among subclasses and since we need to be more specific before we can create objects, abstract classes are not used for object instantiation. Although, object instantiation cannot be done with abstract superclasses, however, like concrete classes, they can be used to declare variables that can hold references to objects of any concrete class derived from them. These variables are used to manipulate the objects of subclasses polymorphically. Just like concrete classes, an abstract class can have static methods inside it and the name of the abstract class can be used to invoke the static methods declared in them. A class can be made abstract by declaring it with the keyword *abstract* as shown below:

public abstract class classname

3.3 Abstract Methods

An abstract method is a method that does not provide implementations for its action. Such a method is declared like a normal method in addition to the keyword *abstract.* A class that contains any abstract method must be declared as an abstract class even if the class contains some concrete (non abstract) methods. When a concrete subclass inherits from an abstract superclass, the subclass must provide concrete implementations of each of the superclass's abstract methods, failure to do this, results in compilation errors.

Constructor and static methods cannot be declared as abstract. Since constructors are not inherited, an abstract constructor can never be implemented. Although, static methods are inherited, they are not associated with particular objects of the classes that declare the static

methods. Since abstract methods are meant to be overridden so that they can process objects based on their types, it would not be proper to declare a static method as abstract. The next examples make use of the concept of abstract classes, abstract methods and inheritance.

Case Study:

A company pays its employees on a weekly basis. The employees are *hourly-employees*, *salaried-employees* and *commission-employees.* Hourly-employees are paid by the hour and receive overtime pay for all hours worked in excess of forty hours. The overtime pay per hour is one – half of their normal pay per hour. Salaried-employees are paid a fixed weekly salary regardless of the number of hours worked. The commission-employees are only paid a percentage of their sales. The company wants you to implement a Java application that performs its payroll calculation polymorphically.

Example 2.5.1

Create an abstract superclass called Employee that each of the employee class can inherit to perform its payroll calculations.

Program Input:

```
public abstract class Employee {
   private String firstname;
   private String lastname;
   private String employee_id;
   public Employee(String first, String last, String id)
   {
      firstname=first;
      lastname=last;
      employee_id=id;
   }
   public String toString()
   {
      return String.format("%s %s\n%s %s",firstname,lastname,
"employee_id: ",employee_id);
   }
   public abstract double earnings();
}
// An abstract class that acts as a superclass for all employee
categories.
```

Program Analysis:

The class is declared as abstract in the first line, this tells that it has at least one method that has no implementation. The class contains three private instance variables *firstname, lastname* and *employee_id,* since an employee is expected to have a first name, a last name and an identity number. The constructor body sets all these three instance variables. The toString method is inherited from the implicit direct superclass Object and it returns a string value that displays the first name, last name and employee id of all employees. Method earnings is declared abstract since it does not have implementation code, this is because different categories of employee have earn different amounts. When subclasses of employee extend the class, they must all provide the implementation code for the method earning because this is part of the contract they will sign with the abstract superclass before inheriting it.

Example 2.5.2

Create a class for the categories of employees that are paid hourly.

Program Input:

```
public class HourlyEmployee extends Employee{
   double hours;
   double rate;
   public HourlyEmployee(String first, String last, String id, double
thehours, double therate)
   {
      super(first,last,id);
      hours=thehours;
      rate=therate;
   }
   public double earnings()
   {
      if(hours<=40)
      return hours*rate;
      else
         return (40*rate)+((hours-40)*1.5*rate);
   }
   public String toString()
   {
      return String.format("\nHourly employee: %s \n%s%.2f\n%s%.2f",
super.toString(),"Hours worked: ",hours,
            "Wages per hour: ",rate);
   }
}
// A class for the hourly paid employees.
```

Program Analysis:

The class has two private variables *hours* and *rate*, since an hourly-employee is paid based on the hours worked and the rate of wages per hour. The constructor has five arguments, three of the arguments are set by calling the superclass constructor and the remaining two are set newly. Method earning returns the amount an hourly paid employee will earn, overtime is also considered. The overtime is one-half the wages per hour for each overtime hour. The method toString has a call to the superclass, this displays the first name, last name and employee *id*, and other parameters are also included to make the displayed result complete.

Example 2.5.3:

Create a class for the salaried-employee.

Program Input:

```
public class SalariedEmployee extends Employee {
   private double weeklysalary;
   public SalariedEmployee(String first,String last, String id, double
weeklysal )
   {
      super(first,last,id);
      weeklysalary=weeklysal;
   }
public double earnings()
{
   return weeklysalary;
}
public String toString()
{
   return String.format
         ("\nSalaried employee: %s\n%s %.2f",super.toString(),"Weekly
Salary:",weeklysalary);
}
}
// A class for the salaried employee
```

Program Analysis:

The class has a private instance variable salary, since a salaried-employee is paid a fixed salary irrespective of the hours worked. The constructor has four arguments, three of the arguments are set by calling the superclass constructor and the remaining one is set newly. Method

earning returns the amount salaried-employee will earn which is basically the value of the instance variable salary. The method toString has a call to the superclass, this displays the first name, last name and employee id, and other parameters are also included to make the displayed result complete.

Example 2.5.4

Create a class for the commissioned-employee.

Program Input:

```
public class CommissionEmployee extends Employee{
private double grossales;
private double commissionrate;
public CommissionEmployee(String first,String last,String id, double
sales, double rate)
{
   super(first,last,id);
   grossales=sales;
   commissionrate=rate;
}
public double earnings()
{
   return grossales*commissionrate;
}
public String toString()
{
   return String.format("\nCommission employee:
%s\n%s%.2f\n%s%.2f", super.toString(),"Gross Sales: ",grossales,
         "Commission rate: ",commissionrate);
}
}
// A class for the commission-employees
```

Program Analysis:

The class has two private variables *grossales* and *commissionrate*, since a commission-employee is paid based on the gross sales and the commission rate. The constructor has five arguments, three of the arguments are set by calling the superclass constructor and the remaining two are set newly. Method earning returns the amount a commission-employee will earn. The method toString has a call to the superclass, this displays the first name, last name and employee id, and other parameters are also included to make the displayed result complete.

3.4 Downcasting and Dynamic Binding

A superclass reference can be used to invoke only the methods declared in the superclass, attempting to invoke a subclass only methods through a superclass give a compilation error. If a program needs to perform a subclass-specific operation in a subclass object referenced by a superclass variable, the program must first cast the superclass reference to a subclass reference through a technique known as *downcasting.* Downcasting enables a program to invoke subclass methods that are not in the superclass. The syntax for downcasting is:

subclassname subclassobject = new (subclassname) superclassobject

An object of a subclass is an object of its superclass just like all cars are vehicles. An object of a superclass may not be an object of a subclass just like all vehicles are not cars. Downcasting allows an object of a superclass to behave like an object of a class, thus the methods of a subclass can be called through a superclass reference. Dynamic binding is the process through which the type (class) of an object is determined at execution time rather than at compilation time. It is also called *late binding. N*ext examples explain the concept of downcasting and dynamic binding.

Example 2.5.5

For the current pay period, the company has decided to reward the salaried-commission employees by adding 5% to their base salaries. The company wants you to implement this without altering the classes you have already because it's going to be a temporal process.
Clue: *Use an executable class*

Program Input:

```
public class Main {
   public static void main(String[] args) {
      HourlyEmployee hourlyemployee=
            new
HourlyEmployee("Grace","Omotoso","08/021",16.75,40);
      SalariedEmployee                    salaryemployee=new
SalariedEmployee("John","Kufor","06/017",800);
                  CommissionEmployee commissionemployee=
                        new
CommissionEmployee("Victor","Michael","05/002",10000,0.06);
      BasePlusCommissionEmployee basepluscommissionemployee=
```

```
            new
BasePlusCommissionEmployee("Rebecca","Olufunke","07/001",5000,0.
04,300);
      System.out.println("Employees processed Polymorphically");
      Employee employees[]=new Employee[4];
      employees[0]=hourlyemployee;
      employees[1]=salaryemployee;
      employees[2]=commissionemployee;
      employees[3]=basepluscommissionemployee;
     for(Employee currentemp: employees)
      {
         System.out.println(currentemp);
         if (currentemp instanceof BasePlusCommissionEmployee)
         {
            BasePlusCommissionEmployee
emp=(BasePlusCommissionEmployee)currentemp;
            emp.setSalary(1.05*emp.getSalary());
               System.out.printf("new base salary with 5%% increase is:
#%,.2f\n",
                     emp.getSalary());
         }
         System.out.printf("earned #%,.2f\n",currentemp.earnings());
      }
// An executable class that processes the objects of Employee subclasses
polymorphically.
```

Program Output:

```
run:
Employees processed Polymorphically
Hourly employee: Grace Omotoso
employee_id:  08/021
Hours worked: 16.75
Wages per hour: 40.00
earned #670.00

Salaried employee: John Kufor
employee_id:  06/017
Weekly Salary: 800.00
earned #800.00

Commission employee: Victor Michael
employee_id:  05/002
Gross Sales: 10000.00
Commission rate: 0.06
earned #600.00
```

```
Base-salaried commission employee:
Commission employee: Rebecca Olufunke
employee_id:  07/001
Gross Sales: 5000.00
Commission rate: 0.04
Salary: 300.00
new base salary with 5% increase is: #315.00
earned #515.00
BUILD SUCCESSFUL (total time: 3 seconds)
```

Program Analysis:

An array of Employee type is used to process the objects of the various subclasses polymorphically. Since we are more particular about the BasePlusCommissionEmployee, the program uses an advanced for loop to iterate through the various object, if an object is of the BasePlusEmployee class, it is given a special consideration. The operator *instanceof* determines if an object is of a particular class. Thus, if an object is of the BasePlusCommission class, the object is downcasted so that the methods of the subclass will be called from a superclass reference. The salary is reset using the setSalary method of the class and the new salary is returned by the getSalary method.

3.5 Final Methods and Classes

It was discussed in unit three that a variable declared final cannot be modified after they are initialised- Such variables represent constant values. It is also possible to declare methods, method parameters and classes with the final modifier. A method that is declared final in a superclass cannot be overridden in a subclass. Methods that are declared private are implicitly final because they cannot be used outside the class. A final method's declaration can never change, so all subclasses inheriting the final methods of a superclass use the same implementation and calls to final methods are resolved at compile time. This concept is known as *static binding.*

3.6 Interfaces

Interfaces are used for assigning common functionality to possible unrelated classes. This allows objects of unrelated classes to be processed polymorphically. Interfaces define and standardise the ways in which things such as people and systems can interact with one another. Interfaces specify *what* and not *how* e.g. the controls of a radio serve as an interface between radio users and a radio's internal components. The interface specifies what operations a radio must permit users to perform but does not specify how the operations are performed.

An interface is used when unrelated classes need to share common methods and constants. This allows objects of unrelated classes to be processed polymorphically.

To use an interface, a concrete class must specify that it implements the interface and must declare each method in the interface with the signature specified in the interface declaration. A class that does not implement all the methods of the interface is like signing an agreement with the computer that states "*I will declare all the methods specified by the interface or I will declare my abstract class*". Although, Java does not support a multiple inheritance, whereby a class inherits from multiple classes, it however allows a class to inherit from a superclass and implement more than one interface. A class makes use of an interface by using the *implements* keyword as shown below:

public class classname implements interface

To implement more than one interface, use a comma-separated list of interface after the keyword *implements* in the class declaration as shown below:

public class classname implements $interface_1$*,* $interface_2$*, …*

All objects of a class that implement multiple have the *is-a* relationship with each implemented interface type. Next examples explain how an interface can be used to make objects of unrelated classes to be processed polymorphically.

Case Study:

Suppose that the company involved in our first case study wishes to perform several accounting operations in a single amountPayable application- in addition to calculating the earning paid to each employee, the company must also calculate the payment due on each of several invoices.

Example 2.5.6

Develop an application that can determine payments for employees and invoices

Clue: Create an interface called Payment that contains method getPaymentAmount that returns a double amount that must be paid for an object of any class that implements the interface.

Program Input:

```
public interface Payment {
   public double getPaymentAmount();
}
// An interface that get the amount of any payable object
```

Program Analysis:

The program creates an interface called Payment. Inside the interface is an empty method *getPaymentAmount.* Any class that implements this interface must provide implementation for this method, if otherwise, the class should be declared abstract.

Example 2.5.7

Create a class Invoice that implements the Payment interface. This class should show the product number and description, it should also show the product quantity and the price of product. Method *getPaymentAmount* should be implemented in this class to calculate the amount on the invoice.

Program Input

```
public class Invoice implements Payment{
   private String productId;
   private String productDesc;
   private int quantity;
   private double price;
   public Invoice(String id, String desc, int qty, double theprice)
   {
      productId=id;
      productDesc=desc;
      quantity=qty;
      price=theprice;
   }
   public double getPaymentAmount()
   {
      return quantity * price;
   }
   public String toString()
   {
      return String.format("%s\n%s: %s (%s)\n%s: %d\n%s: #%.2f",
            "Invoice","Product
id",productId,productDesc,"Quantity",quantity,"Price per item",
            price);
```

```
    }
}
// A class that implements the Payment Interface.
```

Program Analysis:

The first line of the program has the *implement* keyword. This shows that it is making use of an interface called *Payment.* The *getPaymentAmount* of this class is implemented because the class is not declared as *abstract,* it is implicitly signing an agreement that it will provide implementations for all the methods of the interface.

Example 2.5.8

Create a class that uses the interface *Payment* that can be used as a superclass for any of the categories of employees discussed earlier.

Program Input:

```
public abstract class Employee2 implements Payment{
   private String firstname;
   private String lastname;
   private String employee_id;
   public Employee2(String first, String last, String id)
   {
      firstname=first;
      lastname=last;
      employee_id=id;
   }
   public String toString()
   {
      return String.format("%s %s\n%s %s",firstname,lastname,
"employee_id: ",employee_id);
   }
}
//  An abstract class that implements the Payment interface
```

Program Analysis:

The class is abstract because the method *getPaymentAmount* has not been used and hence no implementation is provided for it.

Example 2.5.9

Create a subclass for this new Employee2 superclass for the salaried-employee category. This method should implement the *getPaymentAmount* of the *Payment* interface.

Program Input:

```
public class SalariedEmployee extends Employee2{
    private double weeklysalary;
   public SalariedEmployee(String first,String last, String id, double
weeklysal )
   {
      super(first,last,id);
      weeklysalary=weeklysal;
   }
public double getPaymentAmount()
{
   return weeklysalary;
}
public String toString()
{
   return String.format
         ("\nSalaried employee: %s\n%s %.2f",super.toString(),"Weekly
Salary:",weeklysalary);
}
}
// A subclass for the Employee2 class that uses interface Payment.
```

Program Analysis:

The class is a concrete class. Even though the *implement* keyword was not used in the first line, since it is a subclass of the Employee2 class that implements the *Payment* interface, the class indirectly implements the interface. It must provide implementations for the *getPaymentAmount* method used in the interface since it is not declared as abstract.

Example 2.5.10

Test all your classes with an executable class by polymorphically manipulating objects of Invoice and Employee2.

Program Input:

```
public class Main {
   public static void main(String[] args) {
      Invoice invoice1=new Invoice("0101","PS2",5,500);
      Invoice invoice2=new Invoice ("0201","Optical",7,700);
      SalariedEmployee salariedemployee1=new
SalariedEmployee("Rebecca","Olufunke","03/042",120000);
      SalariedEmployee salariedemployee2=new
SalariedEmployee("Grace","Omotoso","05/095",50000);
      Payment payable[]=new Payment[4];
      payable[0]=invoice1;
      payable[1]=invoice2;
      payable[2]=salariedemployee1;
      payable[3]=salariedemployee2;
      System.out.println("Employees and invoices processed
polymorphically");
      for(Payment currentpayable:payable)
      {
         System.out.printf("%s\n%s: %.2f\n",
currentpayable.toString(),"Payment due",
               currentpayable.getPaymentAmount());
      }
   }
}
// A class to test the objects of the classes that implement the Payment
interface.
```

Program Output:

```
run:
Employees and invoices processed polymorphically
Invoice
Product id: 0101 (PS2)
Quantity: 5
Price per item: #500.00
Payment due: 2500.00
Invoice
Product id: 0201 (Optical)
Quantity: 7
Price per item: #700.00
Payment due: 4900.00

Salaried employee: Rebecca Olufunke
employee_id:  03/042
Weekly Salary: 120000.00
```

Payment due: 120000.00

Salaried employee: Grace Omotoso
employee_id: 05/095
Weekly Salary: 50000.00
Payment due: 50000.00
BUILD SUCCESSFUL (total time: 1 second)

Program Analysis:

No new thing was introduced in the program. Careful examination of each line of code makes you understand better.

4.0 CONCLUSION

With polymorphism, programmers can design and implement systems that are easily extensible. New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. In some cases, it is useful to declare classes for which you never intend to instantiate objects. Such classes are called *abstract classes.* Because they are used only as superclasses in inheritance hierarchies, they are referred to as *abstract superclasses.* The primary purpose of an abstract class is to provide an appropriate superclass from which other classes can inherit and thus share a common design. Classes that can be used to instantiate objects are called *concrete classes* and such classes declare implementations of every method they declare.

Abstract methods do not provide implementations. A class that contains an abstract method must be declared as an abstract class even if that class contains some concrete methods. Constructors and static methods cannot be declared abstract. Including an abstract method in a superclass forces every direct subclass of the superclass to override the abstract method in order to become a concrete class. The *instanceof* operator can be used to determine whether a particular object's type has the *is-a* relationship with a specific time.

The *is-a* relationship applies only between a subclass and its superclass, not vice-versa. However, downcasting allows a superclass object behave like its subclass object. A method that is declared final in a superclass cannot be overridden in a subclass.

Interfaces define and standardise the ways in which things such as people and system can interact with one another. To use an interface, a concrete class must specify that it implements the interface and must declare each interface method with the signatures specified in the

interface declaration. A class that does not implement all the interface's methods is an abstract class and must be declared abstract.

5.0 SUMMARY

In this unit, the following were discussed:

- Abstract superclasses are too general to create real objects; they specify only what is common among subclasses and since we need to be more specific before we can create objects, abstract classes are not used for object instantiation.
- While abstract classes cannot be used for object instantiation, *concrete* classes are classes that can be used for object instantiation.
- An abstract method is a method that does not provide implementations for its action. Such a method is declared like a normal method in addition to the keyword *abstract.*
- Although, static methods are inherited, they are not associated with particular objects of the classes that declare the static methods. Since abstract methods are meant to be overridden so that they can process objects based on their types, it would not be proper to declare a static method as abstract.
- A superclass reference can be used to invoke only the methods declared in the superclass- attempting to invoke a subclass only methods through a superclass gives a compilation error.
- A method that is declared final in a superclass cannot be overridden in a subclass. Methods that are declared private are implicitly final because they cannot be used outside the class.

6.0 TUTOR-MARKED ASSIGNMENT

Review Exercise

1. How does polymorphism enable promote software extensibility?
2. Differentiate between an abstract class and a concrete class
3. What is an interface? Explain the concept of interfaces in OOP context.
4. Explain what you understand by the term downcasting.
5. Differentiate between dynamic binding and static binding.
6. Differentiate between the inheritance hierarchies designed for inheriting interface and the ones designed for inheriting implementations.

Programming Exercise

1. Implement the shape hierarchy shown below. Each TwoDimensionalShape should contain method getArea to calculate the area of the two-dimensional shape. Each ThreeDimensional shape should have methods getArea and getVolume to calculate the surface area and volume, respectively, of the three dimensional shape.
2. Create a program that uses an array of shape references to objects of each concrete class in the hierarchy. The program should print a text description of the object to which each array element refers. Also, in the loop that processes all the shapes in the array, determine whether each shape is a TwoDimensionalShape or a ThreeDimensionalShape. If it is a TwoDimensionalShape, display its area. If it is a ThreeDimensionalShape, display its area and volume.

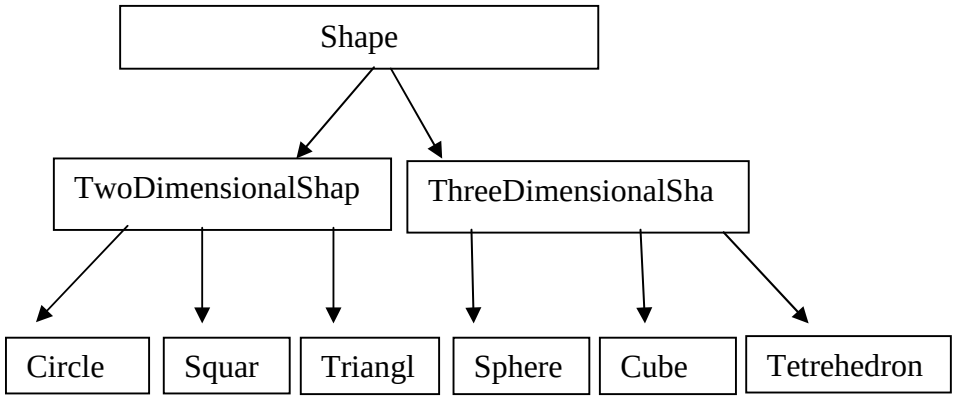

Fig. 5.1: A Shape Hierarchy for Programming Exercises

7.0 REFERENCES/FURTHER READING

Beginning Java ™ 2, JDK ™ (5th ed.).

Deitel Java *How to Program* (7th ed.).

MODULE 3 OVERLOADING

Unit 1 Methods and Method Overloading
Unit 2 Basic Operators Overloading
Unit 3 Logical Operator Overloading
Unit 4 Overloading True and False
Unit 5 Conversion Operator Overloading and Indexers

UNIT 1 METHODS AND METHOD OVERLOADING

CONTENTS

1.0 INTRODUCTION

Most computer programmes that solve real-world problems are much larger than the programmes presented in module two. The best way to develop and maintain a large program is to construct it from small, simple pieces called *modules.* Although, methods was introduces in our previous module, it will be studied at greater length in this unit. In this unit, you will see how method declaration can be used to facilitate the design, implementation, operation and maintenance of large programmes. You will also learn how Java is able to keep track of which method is currently executing, how local variables of methods are maintained in memory and how a method knows where to return after it completes execution.

Many of the classes you will use or create while developing applications will have more than one method of the same name. This technique is called *method overloading* and it is used to implement methods that perform similar tasks for arguments of different types or for different numbers of arguments. For some problems, it is useful to have a method

call itself; such a method is called a *recursive method.* You will learn the concepts of recursion in this unit.

2.0 OBJECTIVES

At the end this unit, you should be able to:

- list the mechanisms for passing information between methods
- differentiate between parameters and arguments
- use random-number generation to implement game playing applications
- define method overloading
- outline the concept of recursion
- identify recursive methods.

3.0 MAIN CONTENT

3.1 Methods

Methods are one of the modules that exist in Java. The three kinds of modules that exist in Java are *methods, classes* and *packages.* Methods (called functions or procedures in other languages) allow programmers to modularise a program by separating its tasks into self-contained units. The statements in the method bodies are written only once, are reused from perhaps several locations in a program and are hidden from other methods. The following are the motivations for modularising a program into methods:

- The divide and conquer approach- This makes program development more manageable by constructing programmes for small, simple pieces.
- Software reusability- This allows existing methods to be used as building blocks to create programmes. Programmers write programmes from standardised methods rather than building customised codes.

Code Maintainability: Methods prevent programmers from writing redundant codes that involves code repetition. Dividing a program into meaningful methods makes the program easier to debug and maintain.

A method is *invoked* (i.e., made to perform its designated task) by a *method call.* The method call specifies the name of the method and may provide information (as *arguments*) that the called method requires to perform its task. When the method call completes, the method either returns a result to the *calling method* (or *caller*) or simply returns control to the calling method. A common analogy for this is the hierarchical

form of management. A boss (the calling method or caller) asks a worker (the *called method*) to perform a task and report back (i.e., *return*) the results after completing the task. The boss method does not know *how* the worker method performs its designated tasks. The worker may also call other worker methods, and the boss will be unaware of these calls. The hiding of implementation details promotes good software engineering.

3.2 Creating and Using Methods

Methods often require pieces of information to perform their tasks. The pieces of information are called *parameters.* Most times, methods use more than one parameter. When methods are created, the *data type, number of parameters* and the *order of parameters* are to be put into consideration. These three factors are called method signature. When methods are to be called (after creation), the pieces of data passed into them are called *arguments.* The arguments used by a called method must have the same signature as the parameters used during the method creation. Our next examples show how methods with multiple parameters are declared in a program and the mechanism for passing information between methods. The syntax for creating a method is;

accesslevel retuntype methodname([parameters])
{
Method body
[return value];
}

Note: Anything contained inside square bracket [] is optional.

The accesslevel describes how other methods in the class will be able to access this method. The *access level* can either be *public or private.* A *private accesslevel* makes the method inaccessible by others methods of the class. A *public access level* makes the method accessible by other methods of the class. *Returntype* describes the data type the method will return to its caller. Return type can be any valid data type. A method that has a return type of *int* returns an integer to its caller, while a method that has a return type of *String* value returns a String value to its caller. Sometimes a method may not return any value to its caller. Such methods have a return type of *void.*

The *methodname* is any valid variablename. The method may or may not have parameters depending on the method action. If a method has parameters, when such method is to be called, the signature of the arguments must be equal to the specified signature in the called method. Just like classes, methods have a method body that begins and ends

respectively with a left brace "{"and a right brace "}". A method may or may not return a value. A method that has a return type of *void* does not return any value to its caller, thus, the second to the last line is absent i.e. *return value.*

Note: For methods that do not have a return type of void, the last line before the closing brace in the method should be the return value. All programming examples in this Course book are compiled using the Java Netbeans.

Example 3.1.1

Create an executable class that accepts two numbers from users, the class should have a method that finds the difference of the two numbers; the method should be called to display the difference of the two numbers.

Program Input:

```
import java.util.Scanner;
public class Main {
   public static void main(String[] args) {
      double num1,num2;
      Scanner myinput=new Scanner(System.in);
    System.out.println ("Enter the first number");
    num1=myinput.nextDouble();
    System.out.println("Enter the second number");
    num2=myinput.nextDouble();
    System.out.println("The difference between "+num1+" and
"+num2+" is "+getDifference(num1,num2));
   System.out.println("The difference between "+num2+" and
"+num1+" is "+getDifference(num2,num1));
   }
   public static double getDifference(double number1,double number2)
   {
      return number1-number2;
   }
}
//  A program that has a method that computes the difference of two
numbers.
```

Program Output:

```
run:
Enter the first number
5
Enter the second number
3
The difference between 5.0 and 3.0 is 2.0

The difference between 3.0 and 5.0 is -2.0

BUILD SUCCESSFUL (total time: 10 seconds)
```

Program Analysis:

The program uses an executable class and the *java.util* package was used to import the scanner class. An object of Scanner *myinput* was created to accept inputs from users. The input expected from users are of double types but in our output we entered integer values and no error was returned, this is because the compiler was able to implicitly convert the integer values to double values, you can see in the output that *5* and *3* entered by the user have been respectively changed to *5.0* and *3.0* by the compiler. This process is called *argument promotion* and it will be discussed in the next section. The method getDifference was declared as static because it was used in the static main method. The method has two parameters *number1* and *number2* both are of the double data type. When the method was called in the main method, two arguments *num1* and *num2* were passed into the method. It should be noted that the number of parameters and their datatypes in the method is equivalent to the number of arguments and datatypes when the method was called. Supplying any data other than the one specified in the method parameters results to compilation error.

The order of arguments should also be put into consideration. It will be discovered from the output that changing the order of the method arguments gives different results. getDifference(5,3)=2.0 whereas getDifference(3,5)=-2.0. This is because in the method body, the second argument was to be subtracted from the first argument. During program execution, control first goes to the main method, when a method call occurs, control goes to the method and the action specified in the method is carried out. If it is a method that returns a value, the value is returned to the previous program segment before the method call, if it is a method that does not return a value, control is returned back to the previous program segment. For example, in our programming example, the getDifference method was called in the output segment of the main method, this makes control to go to the method body carrying along the

method arguments *num1* and *num2* and replacing them with the parameters *number1* and *number2* and subtracting num2 from num1. Since the method returns a value, this value is passed back to the output segment of the main method. The next example illustrates how information can be passed between methods.

Example 3.1.2

Modify the programming example 3.1.1 to embed the getDifference methods. The program should allow users to input three numbers, the difference of the first number and the last two numbers should be returned.

Program Input:

```
import java.util.Scanner;
public class Main {

   public static void main(String[] args) {
      double num1,num2,num3;
      Scanner myinput=new Scanner(System.in);

    System.out.println ("Enter the first number");
    num1=myinput.nextDouble();
    System.out.println("Enter the second number");
    num2=myinput.nextDouble();
    System.out.println("Enter the third number");
    num3=myinput.nextDouble();
    System.out.println("The difference between "+num1+" " +"and the
difference of " +
          num2+"  and  "+num3+"  is
"+getDifference(num1,getDifference(num2,num3)));
   }
   public static double getDifference(double number1,double number2)
   {
      return number1-number2;
   }
}
//  A program to describe how information is passed between methods.
```

Program Output:

```
run:
Enter the first number
9
Enter the second number
```

```
6
Enter the third number
4
The difference between 9.0 and the difference of 6.0 and 4.0 is 7.0
BUILD SUCCESSFUL (total time: 22 seconds)
```

Program Analysis:

The program is not far different from the one in example one, only that the getDifference methods were embedded into each other in the output segment of the main method. The outermost method is first executed, this accepts 6 and 4 and returns 2 to the outermost method, the outermost method uses the returned value as its second arguments. The argument for the method is now 9 and 2 which then returns 7 as its result.

There are three ways to call a method, they are:

- Using a method name by itself to call another methods of the same class e.g. in our programming example 3.1.1, getDifference(num1,num2).
- Using a variable that contains a reference to an object followed by a dot (.) and the method name to call a method of the referenced object. This was discussed in unit one of module two.
- Using the class name and a dot (.) to call a static method of a class. This was discussed in unit three of module two.

3.2.1 Argument Promotion and Casting

Argument promotion is an important feature of method calls. It is the process of converting an argument's value to the type that the method expects to receive in its corresponding parameter e.g. a program can call the square root method of the Math class with an integer argument, although, the method expects to receive a double argument, it does not still return an error. The method declaration parameter list makes Java to convert the int values 5 and 3 to 5.0 and 3.0 before passing the values to getDifference. Some conversions may lead to compilation errors if Java's promotion rules are not satisfied. The promotion rules specify which conversions are allowed, these are the conversions that can be performed without losing data. An int value can be converted to a double value without changing its value just like 9 and 9.0 have the same value. Converting a double value to an int value truncates the fractional part of the double value- thus part of the value will be lost.

However, there are cases where programmers may intentionally want information to be lost, in such cases; the java compiler requires the

programmer to use a *cast* operator to explicitly force the conversion to occur. This makes the programmer take control from the compiler by saying he knows the conversion might cause loss of information but for his purpose, he is okay with the information loss. This is called *casting. N*ext example creates a class that has methods that enables number conversion from one base to another.

Example 3.1.3

Develop a non executable class that can be used for converting numbers from one base to another. Modularise your program as much as possible with methods so that the problem will be easier to solve.

Program Input:

```
public class NumberConversion {
   public int toBaseTen (String number,int base)
   {
      int num=0, result=0;
      for(int i=0;i<number.length();i++)
      {
        char val=number.charAt(i);
          if (isNumber(val)==true)
           num= Integer.parseInt(number.charAt(i)+"");

          else
              num=giveDecValue(val);
        result+=num*(int)Math.pow(base, (number.length()-1-i));
      }
      return result;
      }
   public boolean isNumber(char value)
   {

if(value=='0'||value=='1'||value=='2'||value=='3'||value=='4'||value=
='5'
              ||value=='6'||value=='7'||value=='8'||value=='9')
          return true;
      else
          return false;
   }
   public int giveDecValue(char number)
   {
      int hold=0;
      switch(number)
      {
```

```
        case 'A':
            hold=10;
            break;
        case 'B':
            hold=11;
            break;
        case 'C':
            hold=12;
            break;
        case 'D':
            hold=13;
        case 'E':
            hold=14;
            break;
        case 'F':
            hold=15;
            break;
    }
    return hold;
}
public char giveHexaValues(int number)
{
    char hold=' ';
    switch(number)
        {
            case 10:
                hold='A';
                break;
            case 11:
                hold='B';
                break;
            case 12:
                hold='C';
                break;
            case 13:
                hold='D';
                break;
            case 14:
                hold='E';
                break;
            case 15:
                hold='F';
                break;
        }
    return hold;
}
```

```
    public String BaseTentoOtherBases(int number, int base)
    {
        int rem;
        String hold="",result="";
        rem=number%base;
        if(rem<10)
        hold+=rem;
        else
            hold+=giveHexaValues(rem);
        number=number/base;
        while(number>0)
        {
        rem=number%base;
        if(rem<10)
        hold+=rem;
        else
          hold+=giveHexaValues(rem);
        number=number/base;
        }
        for(int i=hold.length()-1;i>=0;i--)
        {
            result+=hold.charAt(i);
        }
        return result;
    }
}
//  A class for number conversion from one base to another.
```

Program Analysis:

The program was written based on the concept introduced in our discussion of methods, careful analysis of each line of codes makes you understand better.

Example 3.1.4

Test your number conversion class by accepting three inputs from users, the first input is the number to convert, the second input is the base of the number and the third input is the base you want to convert the number to.

Program Input:

```
import java.util.Scanner;
 public class NumberConversionTest {
   public static void main(String[] args) {
```

```
      Scanner myinput=new Scanner(System.in);
      String number;
      int initialbase, finalbase;
      System.out.println("Enter the number you want to convert");
      number= myinput.next();
      System.out.println("Enter the initial base of the number");
     initialbase=myinput.nextInt();
     System.out.println("Enter the base you want to convert the number
to");
      finalbase=myinput.nextInt();
        NumberConversion nc=new NumberConversion();
     System.out.println(number+" in base "+initialbase+" to
"+finalbase+" is "+
            nc.BaseTentoOtherBases(nc.toBaseTen(number, initialbase),
finalbase));
   }
}
//  An executable class that tests the non executable NumberConversion
class.
```

Program Input:

```
run:
Enter the number you want to convert
110011
Enter the initial base of the number
2
Enter the base you want to convert the number to
16
110011 in base 2 to 16 is 33
BUILD SUCCESSFUL (total time: 15 seconds)
```

```
run:
Enter the number you want to convert
A4B5
Enter the initial base of the number
16
Enter the base you want to convert the number to
10
A4B5 in base 16 to 10 is 42165
BUILD SUCCESSFUL (total time: 40 seconds)
```

3.2.2 Random Number Generation

This section takes us to a brief and interesting common type of programming application which is a game of chance using random

number generation. The element of chance can either be introduced through the object of class Random in the *java.util* package or through the static method random of the Math class. In this section, we will be using the Random class from the *java.util* package to simulate a game of chance. A new random number generator can be created by instantiating an object of the Random class as follows:

Random random=new Random();

The random number generator can be used to generate random *Boolean, byte, float, double, int, long and Gaussian* values but only the *int* values will be discussed in this section. There is a method that can be called on the random generator to make it generate integer values. This method is the *nextInt* method. It may or may not accept arguments. If no argument is supplied, then the method generates a random *int* value from *-2,147,483,648* to *-2,147,483,647*. The values returned by *nextInt* are pseudorandom numbers – a sequence of values produced by a complex mathematical calculation. The calculation uses the current time of day (which changes constantly) to seed the random-number generation such that each execution of a program yields a different sequence of random values.

The range of values produced directly by method *nextInt* often differs from the range of values required in a particular Java application e.g. a program that simulates the toss of a coin might only require 0 and 1, and a method that simulates the rolling of a die might require numbers 1 to 6. For such cases, class Random provides another version of method *nextInt* that accepts integer argument, and returns a value from 0 up to the number before the number specified in the argument. For instance *random.nextInt(4)* returns generates number one of numbers 0 to 3.

The argument 4 in the example is called the scaling factor- which represents the number of unique values that *nextInt* should produce (In this case four- 0, 1, 2, 3). To use the random generator to generate numbers that can appear on a die, we can have the following declaration;

int face= 1 + random.nextInt(6);

random.nextInt(6) generates numbers 0 - 5, addition of 1 make the variable *face* to take numbers 1 – 6.

3.3 Method Overloading

Method overloading is the process through which methods of the name are declared in the same class. In this process, there seems to arise some naming conflicts, but the signature of the methods resolves this. A

method signature is the sets of parameters of the method that basically comprises the *number of parameters,* the *data types of parameters* and the *order* of the parameters. When an overloaded method is called, the java compiler is able to select the exact method that is called by checking the *number, data types* and the *order* of arguments supplied to the called method.

Method overloading is commonly used to create methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments. An important aspect to take note of in method overloading is that method calls cannot be distinguished by their return types. The method factors that differentiate between overloaded methods are their signatures only. Overloaded method declarations with identical signatures cause compilation errors even if the return types are different. The next example explains the concept of method overloading.

Example 3.1.5

Write a program that uses a method *minimum* to find the minimum of three numbers. The method should be overloaded as three methods. The first method accepts three integer values as its arguments and returns an integer, the second method accepts three double values as its parameters and returns double, and the third method accepts an integer value and two double values as its argument and returns double. Users should be asked to enter three integer values and three double values. The overloaded methods should accept these variables as applicable.

Program Input:

```
import java.util.Scanner;
public class OverloadedMethods {
  public static int minimum(int number1, int number2, int number3)
  {
     int min=number1;
     if(number2<=min)
       min=number2;
     if(number3<=min)
       min=number3;
     return min;
  }
  public static double minimum(double number1, double number2,
double number3)
  {
     double min=number1;
     if(number2<=min)
        min=number2;
```

```
      if(number3<=min)
         min=number3;
      return min;
   }
    public static double minimum(double number1, double number2, int
number3)
   {
      double min=number1;
      if(number2<=min)
         min=number2;
      if(number3<=min)
        min=number3;
      return min;
   }
    public static void main(String[] args) {
       Scanner myinput=new Scanner(System.in);
       int intnum1=3,intnum2=7, intnum3=6;
       double doublenum1=5.0, doublenum2=-2.3, doublenum3=-7;
       System.out.println("The minimum of
"+intnum1+","+intnum2+","+intnum3+" is "+
            minimum(intnum1,intnum2,intnum3));
       System.out.println("The minimum of
"+doublenum1+","+doublenum2+","+doublenum3+" is "+
            minimum(doublenum1,doublenum2,doublenum3));
       System.out.println("The minimum of
"+doublenum1+","+doublenum2+","+intnum3+" is "+
            minimum(doublenum1,doublenum2,intnum3));
    }
}
//  A program that has overloaded methods.
```

Program Output:

```
run:
The minimum of 3,7,6 is 3
The minimum of 5.0,-2.3,-7.0 is -7.0
The minimum of 5.0,-2.3,6 is -2.3
BUILD SUCCESSFUL (total time: 2 seconds)
```

Program Analysis:

There are three static methods in the program; these methods are declared static because they are used inside the static main method. The methods have the same and different signatures, when the methods are called; the compiler is able to call the right method by comparing the

signature of the arguments of the called method to that of the parameters of the method itself.

3.4 Recursive Methods

A recursive method is a method that calls itself. A recursive method can be called either directly or indirectly through another method. Recursive problem solving approaches have a number of elements in common. When a recursive method is called to solve a problem, the method is only capable of solving the simplest cases. If the method is called with a base case, it returns a result, if it is called with a more complex problem, the method divides the problem into conceptual pieces; a piece that the method knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem but must be a slightly simpler or smaller version of it. Because this new problem looks like the original problem, the method calls a fresh copy of itself to work on the smaller problem, this is called a *recursive call* and it works like a method call. Our next examples use the concept of recursion to find the factorial of numbers and to generate Fibonacci series up to a specified number of terms.

Example 3.1.6

Using recursion, write a method that finds the factorial of numbers. Your method should be used in an application that finds the factorial of the first 10 numbers.

Program Input:

```
public class RecursiveFactorial {
   public static long  factorial(int number)
   {
      if (number<=1)
         return 1;
      else
         return number*factorial(number-1);
   }
      public static void main(String[] args) {
      for (int i=1;i<=10;i++)
      {
         System.out.println(i+"!\t"+factorial(i));
      }
   }
}
//  A program that uses recursion to compute factorial of numbers.
```

Program Output:

```
run:
1!      1
2!      2
3!      6
4!      24
5!      120
6!      720
7!      5040
8!      40320
9!      362880
10!     3628800

BUILD SUCCESSFUL (total time: 3 seconds)
```

Program Analysis:

The factorial method first solves the base case by specifying that factorial of one or any number less than one should be zero, the complex case is then solved by dividing the problem into two pieces which later divides itself until the base case is reached. The next example we will look at uses recursion to generate Fibonacci series.

4.0 CONCLUSION

Experience has shown that the best way to develop and maintain a large program is to construct it from several small, simple pieces or modules. This technique is called *divide* and *conquer*. Methods allow programmers to modularise a program by separating its tasks into self-contained units. The statements in a method are written only once and hidden from other methods. Using existing methods as building blocks to create new programmes is a form of software reusability that prevents programmers from repeating codes within a program. When a method is called, the program makes a copy of the method's argument values and assigns them to the method's corresponding parameters, which are created and initialised when the method is called. When program control returns to the point in the program where the method was called, the method parameters are removed from memory. A method can return at most one value, but the returned value can be a reference to an object that contains many values.

An important feature of method calls is *argument promotion*- converting an argument's value to the type that the method expects to receive in its corresponding parameters. In cases where information may be lost due to conversion, the Java compiler requests programmers to use a cast

operator to explicitly force the conversion to occur. Random numbers in a range can be generated with the formula.

Number=shiftin value +randomNumberGenerator.nextInt(scaling factor);

Where *shiftvalue* specifies the first number in the desired range of consecutive integers and *scaling factor* specifies how many numbers are in the range. A recursive method calls itself directly or indirectly through another method. When a recursive method is called to solve a problem, the method is capable of solving only the simplest case(s) or base case(s). If it is called with a base case, the method returns a result. If a recursive method is called with a more complex problem than a base case, it divides the problem into two conceptual pieces- a piece that the method knows how to do and a piece that it does not know how to do. It then further divides the latter piece into two different pieces and continually does that until the base case is finally used to solve the large problem.

5.0 SUMMARY

In this unit, we have discussed the following:

- Methods are one of the modules that exist in Java and the three kinds of modules that exist in Java are *methods, classes* and *packages.*
- Methods (called functions or procedures in other languages) allow programmers to modularise a program by separating its tasks into self-contained units.
- A method is *invoked* (i.e., made to perform its designated task) by a *method call.* The method call specifies the name of the method and may provide information (as *arguments*) that the called method requires to perform its task.
- Argument promotion is the process of converting an argument's value to the type that the method expects to receive in its corresponding parameter.
- Method overloading is commonly used to create methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.
- A recursive method is a method that calls itself which can be called either directly or indirectly through another method.

6.0 TUTOR-MARKED ASSIGNMENT

Review Exercise

1. A method is invoked with a _____
2. The_____statement in a called method can be used to pass the value of an expression back to the calling method.
3. The keyword_____ indicates that a method does not return a value.
4. List the three ways through which methods can be called?
5. An object of a class _____ produces random numbers
6. Define method overloading?

Programming Exercise

1. Write a method *IntegerPower(base, exponent)* that returns the value of base exponent, for example, *IntegerPower(3, 4) = 3 * 3 * 3 * 3.* Assume that the exponent is a positive integer, and base is an integer. Method *IntegerPower* should use for or while to control the calculation. Do not use any Math library methods. Use the method in an application that reads integer values for base and exponent and performs the calculation with the method *IntegerPower.*

2. Implement the following methods:

a. Method *Celsius* returns the Celsius equivalence of a Fahrenheit temperature using the formalar; *Celsius = 5.0/9.0 * (Fahrenheit – 32);*

b. Method *Fahrenheit* returns the Fahrenheit equivalence of a Celsius temperature using the formular; Fahrenheit=9.0/5.0*(Celsius + 32);

c. Use the methods from parts (a) and (b) to write an application that enables the user to either enter a Fahrenheit temperature and displays the equivalent Celsius temperature or to enter a Celsius temperature and display the Fahrenheit equivalence.

3. Write an application that displays a table of binary, octal and hexadecimal equivalence of the decimal numbers in the range I through 128.

4. A palindrome is a string that is spelt the same way forward and backward. Some examples of palindromes are “radar”, “pap” etc. Write a recursive method that returns Boolean value true if the string stored in the array is a palindrome and false, if otherwise.

7.0 REFERENCES/FURTHER READING

Deitel Java *How to Program* (7th ed.).

Wrox Beginning Java TM 2, JDK TM(5th ed.).

UNIT 2 BASIC OPERATORS OVERLOADING

CONTENTS

1.0 INTRODUCTION

Object- Oriented Programming concept allows programmers to overload operators, not just methods. This can be done by defining `static` methods using the `operator` keyword. Being able to overload basic operators like `+`, `-`, `*` and so on for classes lets programmers use those classes with those operators, just as if they were types built into the program. Although Java does not support operator overloading, we will look at some other programming languages that support operator overloading in order to make our discussion on object- oriented programming complete.

2.0 OBJECTIVES

At the end this unit, you should be able to:

- list the concept of operator overloading
- identify some object oriented programming languages that support operator overloading
- explain why Java does not support operator overloading.

3.0 MAIN CONTENT

3.1 Basic Operator Overloading

Operator overloading is simply the process of adding operator functionality to a class. This allows you to define exactly how the operator behaves when used with your class and other data types. This can be standard uses such as the ability to add the values of two vectors, more complex mathematics for multiplying matrices or non-arithmetic functions such as using the + operator to add a new item to a collection or combine the contents of two arrays. Multiple overloaded versions of operators may also be created to provide different functionality according to the data types being processed, in a similar manner to the varying signatures of method overloading.

Java does not support operator overloading but an exception was made for String by using the "+" symbol to concatenate string literals. Although, this is out of the control of developers as it is a built- in feature of Java. Developers have no control over operator overloading in Java. Java does not support operator overloading because the designers of Java wanted to keep Java codes simple and found that operator overloading made code more complex and difficult to read. There are other methods available to achieve the same functionality as operator overloading, you could create methods in a class named plus (), minus (), multiply (), etc... The designers of Java must have decided that operator overloading in Java was more of a problem than it was worth.

Some Object- Oriented languages allow you to overload an operator. C#/ C++ are examples of such languages. Operator overloading allows you to change the meaning of an operator. For example, when most people see a plus sign, they assume it represents addition. If you see the equation, you expect that X would contain the value 11. And in this case, you would be correct.

X = 5 + 6;

However, there are times when a plus sign could represent something else. For example, in the following code:

- *String firstName = "Joe", lastName = "Smith";*
- *String Name = firstName + " " + lastName;*

You would expect that Name would contain Joe Smith. The plus sign here has been overloaded to perform string concatenation which is commonly used in Java but it is a built- in feature of Java that cannot be controlled by programmers.

Many languages allow the programmer to redefine, either partially, or wholly, the definition of basic operators (+, -, etc.). This can be extremely useful as it allows the programmer to use operators on new data types in a way that makes sense. For scientist, the classic example is complex numbers. Most languages don't support complex numbers as intrinsic data types but with operator overloading they can be used and the resulting code is clear to read since Java doesn't allow operator overloading, the programmer must use methods which makes the code more verbose and harder to read and maintain.

The following codes create a vector class that can be used to manipulate values that have two coordinates, e.g. 2x+6y, through operator overloading, we can perform the basic mathematical operations on members of the vector class.

```
public class Vector
{
private int _x, _y;

public Vector(int x, int y) { _x = x; _y = y; }

public int X
{
   get { return _x; }
   set { _x = value; }
}

public int Y
{
   get { return _y; }
   set { _y = value; }
}
}
```

3.2 Binary Operator Overloading

The first type of operator to consider is the *binary operator*, so named because they require two values to work with. These include the simple arithmetic operators such as +, -, *, / and %. In C#, the syntax for overloading binary operators is:

public static *result-type* operator *binary-operator* (
    op-type operand,
    op-type2 operand2)

This initially appears to be a rather complex declaration but in fact is quite simple. The declaration starts with *public static* as all operators must be declared as such. Other scopes are not permitted and neither are non-static operators. The *result-type* defines the data type or class that is returned as the result of using the operator. Usually this will be the same type as the class that it is being defined within. However, that need not be the case and it is perfectly valid to return data of a different type. The *operator* keyword is added to tell the compiler that the following *binary-operator* symbol is an operator rather than a normal method. This operator will then process the two *operand* parameters, each prefixed with its data type *(op-type* and *op-type2*). As least one of these operands must be the same type as the containing class.

3.2.1 Creating the Addition (+) Operator

The syntax for binary operators can now be used to create a new addition operator for the Vector class. This operator will simply add the X and Y elements of two vectors together and return a new vector containing the result. Add the following to the Vector class to provide this functionality. Note that a new vector is created rather than adjusting one of the operands. This is because the operands are *reference-types* and the original values should not be updated in this case.

```
public static Vector operator +(Vector v1, Vector v2)
{
   return new Vector(v1.X + v2.X, v1.Y + v2.Y);
}
```

We can now test the vector's new operator by modifying the program's main method. The following program instantiates two vector objects, adds them together and outputs the values of the resultant vector's X and Y properties.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);
   Vector v2 = new Vector(0, 8);

   Vector v3 = v1 + v2;
   Console.WriteLine("({0},{1})", v3.X, v3.Y);     // Outputs "(4,19)"
}
```

3.2.2 Creating the Subtraction (-) Operator

Addition is a *commutative* operation. This means the order of the two operands can be swapped without affecting the outcome. In the case of

subtraction, this is not the case so it important to remember that the first operand in the declaration represents the value to the left of the operator and the second operand represents the value to the right. If these are used incorrectly, the resultant value will be incorrect. Using this knowledge we can add a subtraction operator to the Vector class:

```
public static Vector operator -(Vector v1, Vector v2)
{
   return new Vector(v1.X - v2.X, v1.Y - v2.Y);
}
```

To test the new operator, modify the Main method as follows and execute the program

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);
   Vector v2 = new Vector(0, 8);

   Vector v3 = v1 - v2;

   Console.WriteLine("({0},{1})", v3.X, v3.Y);     // Outputs "(4,3)"
}
```

3.2.3 Creating the Multiplication (*) Operator

The last binary operator that will be added to the Vector class is multiplication. This operator will be used to scale the vector by multiplying the X and Y properties by the same integer value. This demonstrates the use of operands of a different type to the class they are defined within.

```
public static Vector operator *(Vector v1, int scale)
{
   return new Vector(v1.X * scale, v1.Y * scale);
}
```

To test the multiplication operator, adjust the Main method again

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);

   Vector v2 = v1 * 3;

   Console.WriteLine("({0},{1})", v2.X, v2.Y);     // Outputs "(12,33)"
}
```

In the operator code for the multiplication operator, the Vector is the first operand and the integer the second. This means that the order used

in the multiplication statement must have the vector at the left of the operator and the integer value to the right. Changing the order of the operands in the Main method will cause a compiler error.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);

   Vector v2 = 3 * v1;

   Console.WriteLine("({0},{1})", v2.X, v2.Y);     // Does not compile
}
```

If the class must support both variations of multiplication, both must be declared in the code. This provides the benefit of allowing the order of operands change the underlying function. To provide the second variation of multiplication, add the following code to the Vector class. Afterwards, the program will execute correctly.

```
public static Vector operator *(int scale, Vector v1)
{
   return new Vector(v1.X * scale, v1.Y * scale);
}
```

3.3 Unary Operator Overloading

Unary operators are those that require a single operand. These include the simple increment (++) and decrement (--) operators. To declare a unary operator, the following syntax is used:

- public static *result-type* operator *unary-operator* (*op-type operand*)

This syntax is almost identical to that used for binary operators. The difference is that only one operand is declared. The operand type must be the same as the class in which the operator is declared.

3.3.1 Creating the Increment and Decrement Operator

Using the syntax defined above, we can now add the increment and decrement operators to the Vector class. Note that there is only a single definition for each. There is no way to differentiate between prefix and postfix versions of the operator so both provide the same underlying functionality. To declare the two operators, add the following code to the Vector class. Each increments or decrements both affect X and Y properties for Vector objects.

```
public static Vector operator ++(Vector v)
{
   v.X++;
   v.Y++;
   return v;
}

public static Vector operator --(Vector v)
{
   v.X--;
   v.Y--;
   return v;
}
```

To test these operators, update and execute the Main method:

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);

   v1++;
   Console.WriteLine("({0},{1})", v1.X, v1.Y);     // Outputs "(5,12)"

   v1--;
   Console.WriteLine("({0},{1})", v1.X, v1.Y);     // Outputs "(4,11)"
}
```

3.3.2 Creating the Negation Operator

The last arithmetic unary operator to be considered in this unit is the *negation operator*. This is the unary version of subtraction used to identify a negative version of a value. We can add this operator using the following code:

```
public static Vector operator -(Vector v)
{
  return new Vector(-v.X, -v.Y);
}
```

To test the negation operator, update the Main method and run the program.

```
static void Main(string[] args)
        {
                Vector v1 = new Vector(4, 11);
                Vector v2 = -v1;
                        Console.WriteLine("({0},{1})", v2.X, v2.Y);     //
Outputs "(-4,-11)"
        }
```

4.0 CONCLUSION

In this unit, you learnt that it is possible to change the way basic operators like +, - , *, / etc behave in order to suit the programmers work. Java does not support this but methods can be used to make Java behave like that. Java was designed not to support operator overloading because it makes Java codes complex but since Java was designed to make programming easier, the feature was embedded into Java. Some Object- oriented languages like C#/ C++ support the basic operator overloading it.

8.0 SUMMARY

In this unit, we have discussed the following:

- Operator overloading is the process of adding operator functionality to a class which allows one to define exactly how the operator behaves when used with your class and other data types.
- Java does not support operator overloading but an exception was made for String by using the "+" symbol to concatenate string literals.
- In the operator code for the multiplication operator, the Vector is the first operand and the integer the second.
- Most languages don't support complex numbers as intrinsic data types but with operator overloading they can be used and the resulting code is clear to read since Java doesn't allow operator overloading, the programmer must use methods which makes the code more verbose and harder to read and maintain.
- The *operator* keyword is added to tell the compiler that the following *binary-operator* symbol is an operator rather than a normal method. This operator will then process the two *operand* parameters, each prefixed with its data type *(op-type* and *op-type2)*.

9.0 TUTOR-MARKED ASSIGNMENT

1. What is operator overloading?
2. How does operator overloading differ from method overloading
3. Why does Java not support operator overloading?
4. Is there a way through which Java methods can be manipulated to make it behave as if it supports method overloading? If yes, explain how.
5. Mention a programming language that supports method overloading.

7.0 REFERENCES/FURTHER READING

The basics of Java Programming for Scientists.

www.blackwasp.co.uk/CSharpObjectOriented.aspx

UNIT 3 LOGICAL OPERATOR OVERLOADING

CONTENTS

1.0 INTRODUCTION

In this unit, the overloading of the logical operators is described including how to enable short-circuit Boolean logical operators. All programmes are written in C sharp because it enables overloading of operators.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- describe logical operator
- explain operator overloading
- list the boolean AND operator
- outline overloaded boolean logical operator
- analyse short-circuit operator.

3.0 MAIN CONTENT

3.1 The Logical Operators

There are four logical operators that can be directly overloaded for a class. These are the NOT operator (!), the AND operator (&), the OR operator (|) and the exclusive OR or XOR operator (^). The short-circuit operators (&& and ||) cannot be overloaded directly. However, if the class meets certain conditions the short-circuit operators are enabled automatically.

In this unit, a Vector class will be used. The Vector class represents a two-dimensional vector and already overloads arithmetic operators and the true and false operators. The Vector class will be updated to include overloading of the four logical operators for Boolean operators.

Here is a sample Vector class

```
[DefaultMemberAttribute("Item")]
[SerialisableAttribute()]
public class Vector : ICloneable, ISerialisable, IDisposable
```

3.2 Overloading the Binary Boolean Logical Operators

The Boolean binary logical operators are generally used in conditional processing operations and thus return a Boolean value, either true or false. To create an overloaded version of these operators, the syntax is the same as for other binary overloads, except that the return value is generally of the bool type.

```
public static bool operator logical-operator (
   op-type operand,
   op-type2 operand2
)
```

At least one of the operands must be of the same type as the containing class. This means that by providing the correct signature, you can permit logical operations between two classes of differing types.

3.2.1 Creating the Boolean AND Operator (&)

The three binary Boolean operators can be added to the Vector class using the syntax described above. When determining the results of each, a vector will be deemed to equate to true if either of the co-ordinates is non-zero. If both the X and Y properties are zero, the vector will be

considered 'false'. Add the following operator overload to the Vector class to implement the AND function:

```
public static bool operator &(Vector v1, Vector v2)
{
   bool v1flag = !((v1.X == 0) && (v1.Y == 0));
   bool v2flag = !((v2.X == 0) && (v2.Y == 0));
   return v1flag & v2flag;
}
```

Analysis:

The code evaluates each vector's co-ordinates to create a Boolean representation in the 'flag' variables. An AND operation is then performed on the two flags and the result returned. You can test the code by changing the Main method of the program as follows and executing the console application.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(0, 0);
   Vector v2 = new Vector(10, 0);
   Console.WriteLine(v1 & v1);          // Outputs "False"
   Console.WriteLine(v1 & v2);          // Outputs "False"
   Console.WriteLine(v2 & v1);          // Outputs "False"
   Console.WriteLine(v2 & v2);          // Outputs "True"
}
```

3.2.2 Creating the Boolean OR (|) and XOR (^) Operators

Using similar code, we can add the Boolean OR and XOR operators to the Vector class. Add the following two operator overloads to the class to implement the two logical functions.

```
public static bool operator |(Vector v1, Vector v2)
{
   bool v1flag = !((v1.X == 0) && (v1.Y == 0));
   bool v2flag = !((v2.X == 0) && (v2.Y == 0));
   return v1flag | v2flag;
}
public static bool operator ^(Vector v1, Vector v2)
{
   bool v1flag = !((v1.X == 0) && (v1.Y == 0));
   bool v2flag = !((v2.X == 0) && (v2.Y == 0));
   return v1flag ^ v2flag;
}
```

Again, these operators can be tested using a modified Main method:

```
static void Main(string[] args)
{
   Vector v1 = new Vector(0, 0);
   Vector v2 = new Vector(10, 0);

   Console.WriteLine(v1 | v1);          // Outputs "False"
   Console.WriteLine(v1 | v2);          // Outputs "True"
   Console.WriteLine(v2 | v1);          // Outputs "True"
   Console.WriteLine(v2 | v2);          // Outputs "True"

   Console.WriteLine(v1 ^ v1);          // Outputs "False"
   Console.WriteLine(v1 ^ v2);          // Outputs "True"
   Console.WriteLine(v2 ^ v1);          // Outputs "True"
   Console.WriteLine(v2 ^ v2);          // Outputs "False"
}
```

3.3 Overloading the Unary Boolean Logical Operator

Only one unary Boolean logical operator exists. This is the NOT operator (!) that switches a Boolean value between true and false. When overloaded using the unary syntax below, the value returned should be the opposite of the Boolean representation of the object. As with other unary operators, the type of the operand provided must be the same as the class that the declaration appears within.

```
public static bool operator !(op-type operand)
```

3.3.1 Creating the Boolean NOT Operator (!)

The NOT operator for the Vector class will examine the contents of the X and Y properties. As described above, if both co-ordinates are zero, the Boolean value for the object is false. However, as this is the NOT operator, we will perform the check and return true only when both values are zero. Add the following code to the class to provide the NOT operator:

```
public static bool operator !(Vector v)
{
   return ((v.X == 0) && (v.Y == 0));
}
```

To test that the operator is working correctly, modify the Main method as follows:

```
static void Main(string[] args)
{
   Vector v1 = new Vector(0, 0);
   Vector v2 = new Vector(10, 0);

   Console.WriteLine(!v1);          // Outputs "True"
   Console.WriteLine(!v2);          // Outputs "False"
}
```

3.3.2 Enabling the Short-Circuit Operators

The short-circuit operators provide an additional variant of the AND and OR operators. In each case, the left-hand operand in the operation is examined in isolation first. When evaluating an AND where the first operand evaluates to false, the result of the operation will be false regardless of the value of the second operand. Similarly, when evaluating an OR operation where the first operand evaluates to true, the result will always be true. In these special cases, the right-hand operand is not evaluated at all, potentially improving performance. The short-circuit operators cannot be overloaded directly. However, if two conditions are met in the class then the short-circuit operators are automatically made available. The two conditions are as follows:

- The class must overload the normal logical operators (& and |) with the operation returning a value of the same type as the containing class. Each parameter of the operator must also be of the type of the containing class.
- The true and false operators must be overloaded.
- When the operator is invoked, the true or false operators are used to determine the status of the first operand. If this guarantees an outcome from the operation, the result is returned immediately. When the state of the first operand does not force an outcome, the AND or OR operator is used for the two values to determine the result.

3.3.3 Adding the Short-Circuit Operator Pre-Requisites to the Vector Class

The Vector class already includes overloaded true and false operators. To enable the short-circuit operators, one need to add the correct signature for the AND and OR operators. As these must return a Vector result that can be evaluated as either true or false according to the results, this will return (0,0) if the operation equates to false and (1,1)

for true. Modify the code for the existing & and | operator overloads as follows:

```
public static Vector operator &(Vector v1, Vector v2)
{
   bool v1flag = !((v1.X == 0) && (v1.Y == 0));
   bool v2flag = !((v2.X == 0) && (v2.Y == 0));

   if (v1flag & v2flag)
   {
      return new Vector(1, 1);
   }
   else
   {
      return new Vector(0, 0);
   }
}

public static Vector operator |(Vector v1, Vector v2)
{
   bool v1flag = !((v1.X == 0) && (v1.Y == 0));
   bool v2flag = !((v2.X == 0) && (v2.Y == 0));
   if (v1flag | v2flag)
   {
      return new Vector(1, 1);
   }
   else
   {
      return new Vector(0, 0);
   }
}
```

You can test that the short-circuit operators are correct by modifying the Main method of the program. However, as the operators do not return a Boolean value, an if statement is required in order to test the results of the operations. This is demonstrated in the following code.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(0, 0);
   Vector v2 = new Vector(10, 0);

   if (v1 && v2)
   {
      Console.WriteLine("v1 && v2 = true");
   }
```

```
   else
   {
      Console.WriteLine("v1 && v2 = false");  // Outputs "v1 && v2 =
false"
   }
      if (v1 || v2)
   {
      Console.WriteLine("v1 || v2 = true");  // Outputs "v1 || v2 = true"
   }
   else
   {
      Console.WriteLine("v1 || v2 = false");
   }
}
```

4.0 CONCLUSION

Operator overloading refers to the ability to define a new meaning for an existing (built-in) "operator". The list of "operators" includes mathematical operators (+, -, *, /, ++, etc), relational operators (<, >, ==, etc), logical operators (&&, ||, etc.), access operators ([], ->), assignment operator (=), stream I/O operators (<<, >>), type conversion operators and several others. While all of these operators have a predefined and unchangeable meaning for the built-in types, all of these operators can be given a specific interpretation for different classes or combination of classes.

Vector class is used to represent a real vector. Its methods can be used to perform vector operations and data manipulation.

8.0 SUMMARY

In this unit, we have discussed the following:

- There are four logical operators that can be directly overloaded for a class. These are the NOT operator (!), the AND operator (&), the OR operator (|) and the exclusive OR or XOR operator (^).
- The Boolean binary logical operators are generally used in conditional processing operations and thus return a Boolean value, either true or false.
- The NOT operator for the Vector class will examine the contents of the X and Y properties. If both co-ordinates are zero, the Boolean value for the object is false.

- When evaluating an AND where the first operand evaluates to false, the result of the operation will be false regardless of the value of the second operand.
- The Vector class already includes overloaded true and false operators. To enable the short-circuit operators, one need to add the correct signature for the AND and OR operators.
- Operator overloading refers to the ability to define a new meaning for an existing (built-in) operator.

9.0 TUTOR-MARKED ASSIGNMENT

1. Write a program to overload the Boolean OR(!) and AND operators.
2. Due to the recent administration in an organisation, some commission-employees are to be given a base salary in addition to their commission. This has to be included in the payroll urgently. Create a class that handles this new set of employees.

7.0 REFERENCE/FURTHER READING

BlackWasp (2006). C#Object-Oriented Pragramming Tutorial. Website-www.java2s.com

UNIT 4 OVERLOADING TRUE AND FALSE

CONTENTS

1.0 INTRODUCTION

To continue the discussion on overloading, this unit describes overloading of the true and false operators, allowing an object to be used in conditional processing. The true and false keywords have many uses in C#. One of these uses is as a pair of operators within a class that allow the class to represent its own state as either true or false. The determination of the result is implemented in rules coded by the programmer. This effectively gives an implicit conversion of a type to a Boolean value that can then be used in conditional processing scenarios such as with the *if* statement or the conditional operator (?). This unit uses Vector class to demonstrate overloading bahaviour.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- list true and false overloaded operator
- define true and false operator
- outline the true and false overloading
- identify a vector operator.

3.0 MAIN CONTENT

3.1 Adding True and False Operators

The default behaviour of any class is to provide no support for the true and false operators. This means that if an attempt is made to evaluate an object of such a class as a Boolean the code will fail to compile. This can be demonstrated by using a Vector object as the condition in an *if*

statement. Change the Main method of the VectorDemo program as follows and attempt to compile it to see the error.

```
static void Main(string[] args)
{
   Vector test = new Vector(4, 3);
   if (test)
   {
      Console.WriteLine("True");
   }
   else
   {
      Console.WriteLine("False");
   }
}
```

3.2 Overloading True and False

The syntax for overloading the true and false operators is similar to that of other unary operators. Two limitations exist. Firstly, the return value must be a Boolean. Secondly, it is invalid to overload only one of the two operators; if the true operator is overloaded than so must be false and vice versa.

```
public static bool operator true(op-type operand)
{
   // Evaluation code
}
public static bool operator false(op-type operand)
{
   // Evaluation code
}
```

3.2.1 Adding True and False to the Vector Class

The Vector class can now be updated to include overloaded versions of the true and false operators. In the case of vectors, the object will be deemed to be 'true' when either of the vector's X or Y properties is non-zero. If the X and Y values are both zero, the object will evaluate as false. This is simply implemented by adding the following code:

```
public static bool operator true(Vector v)
{
   if ((v.X != 0) || (v.Y != 0))
   {
      return true;
```

```
    }
    else
    {
        return false;
    }
}
public static bool operator false(Vector v)
{
    if ((v.X == 0) && (v.Y == 0))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Analysis:

Now that the two operators have been added, you should be able to compile and execute the Main method described above. As the vector's X and Y co-ordinates are not zero, the code outputs the text *"True"*. If you modify the Vector's declaration as follows, the code will output "*False"* instead.

```
static void Main(string[] args)
{
    Vector test = new Vector(0, 0);
    if (test)
    {
        Console.WriteLine("True");
    }
    else
    {
        Console.WriteLine("False");        // Outputs "False"
    }
}
```

The following are the examples of true and false operator overloading:

Overloading true and false Operator

```
 public class MyType
   {
      public static bool operator true ( MyType e )
```

```
{
   return  ( e == null ) ? false : e.b;
}

public static bool operator false ( MyType e )
{
   return  ( e == null ) ? true : !e.b;
}

public bool b;

public MyType( bool b )
{
   this.b = b;
}

public static void Main( string[] args )
{
   MyType myTrue = new MyType( true );
   MyType myFalse = new MyType( false );
   MyType myNull = null;

   if ( myTrue )
   {
      System.Console.WriteLine( "true" );
   }
   else
   {
      System.Console.WriteLine( "false" );
   }

   if ( myFalse )
   {
      System.Console.WriteLine( "true" );
   }
   else
   {
      System.Console.WriteLine( "false" );
   }

   if ( myNull )
   {
      System.Console.WriteLine( "true" );
   }
   else
   {
```

```
            System.Console.WriteLine( "false" );
         }
      }
   }
```

Output:

```
True
False
False
```

True and False Operator for Complex

```
using System;
public struct Complex
{
   public Complex( double real, double imaginary ) {
      this.real = real;
      this.imaginary = imaginary;
   }

   public override string ToString() {
      return String.Format( "({0}, {1})", real, imaginary );
   }
   public double Magnitude {
      get {
         return Math.Sqrt( Math.Pow(this.real, 2) +
Math.Pow(this.imaginary, 2) );
      }
   }

   public static bool operator true( Complex c ) {
      return (c.real != 0) || (c.imaginary != 0);
   }

   public static bool operator false( Complex c ) {
      return (c.real == 0) && (c.imaginary == 0);
   }

   private double real;
   private double imaginary;
}

public class MainClass
{
   static void Main() {
```

```
      Complex cpx1 = new Complex( 1.0, 3.0 );
      if( cpx1 ) {
         Console.WriteLine( "cpx1 is true" );
      } else {
         Console.WriteLine( "cpx1 is false" );
      }

      Complex cpx2 = new Complex( 0, 0 );
      Console.WriteLine( "cpx2 is {0}", cpx2 ? "true" : "false" );
   }
}
```

Output:

```
cpx1 is true
cpx2 is false
```

Overload True and False for Two Dimension

```
using System;

class TwoDimension {
  int x, y;

  public TwoDimension() {
    x = y = 0;
  }
  public TwoDimension(int i, int j) {
    x = i;
    y = j;
  }

  // Overload true.
  public static bool operator true(TwoDimension op) {
    if((op.x != 0) || (op.y != 0))
      return true; // at least one coordinate is non-zero
    else
      return false;
  }

  // Overload false.
  public static bool operator false(TwoDimension op) {
    if((op.x == 0) && (op.y == 0))
      return true; // all coordinates are zero
    else
      return false;
  }
```

```
  // Overload unary --.
  public static TwoDimension operator --(TwoDimension op)
  {
    // for ++, modify argument
    op.x--;
    op.y--;

    return op;
  }

  // Show X, Y, Z coordinates.
  public void show()
  {
    Console.WriteLine(x + ", " + y);
  }
}

class MainClass {
  public static void Main() {
    TwoDimension a = new TwoDimension(5, 6);
    TwoDimension b = new TwoDimension(10, 10);
    TwoDimension c = new TwoDimension(0, 0);

    Console.Write("Here is a: ");
    a.show();
    Console.Write("Here is b: ");
    b.show();
    Console.Write("Here is c: ");
    c.show();
    Console.WriteLine();

    if(a)
      Console.WriteLine("a is true.");
    else
      Console.WriteLine("a is false.");

    if(b)
      Console.WriteLine("b is true.");
    else
      Console.WriteLine("b is false.");

    if(c)
      Console.WriteLine("c is true.");
    else
      Console.WriteLine("c is false.");
```

```
    Console.WriteLine();

    Console.WriteLine("Control a loop using a TwoDimension object.");
    do {
      b.show();
      b--;
    } while(b);

  }
}
```

Output:

```
Here is a: 5, 6
Here is b: 10, 10
Here is c: 0, 0

a is true.
b is true.
c is false.

Control a loop using a TwoDimension object.
10, 10
9, 9
8, 8
7, 7
6, 6
5, 5
4, 4
3, 3
2, 2
1, 1
```

4.0 CONCLUSION

The true and false operators can be overloaded, to allow a class to represent its own state as true or false. You can overload the `true` or `false` operators if you are defining a specialised boolean value. Operator overloading allows us to define/redefine the way operators work with our classes and structures. This allows programmers to make their custom types look and feel like simple types such as *int* and string. It consists of nothing more than a method declared by the keyword operator and followed by an operator. There are three types of overload able operators called unary, binary, and conversion. Not all operators of each type can be overloaded.

5.0 SUMMARY

In this unit, the following were discussed:

- What the default behaviour of any class does is to provide no support for the true and false operator, Which means if an attempt is made to evaluate an object of such a class as a Boolean, the code will fail to compile
- Two limitations that exist in overloading are: first, the return value must be a Boolean; secondly, it is invalid to overload only one of the two operators; if the true operator is overloaded than so must be false and vice versa.
- The true and false operators can be overloaded, to allow a class to represent its own state as true or false.

6.0 TUTOR-MARKED ASSIGNMENT

1. Describe how you can enable the short- circuit operator.
2. Write a program to add the true and false operator.

7.0 REFERENCES/FURTHER READING

Design Patterns: Elements of Reusable Object-Oriented Software.

Erich Gamma, Richard Helm, Ralph Johnson & John M. Vlissides (2006).

Iain D. Craig (2003). Object-Oriented Programming Languages: Interpretation.

Nancy M. Wilkinson (2006). An Informal Approach to Object-Oriented Development.

Timothy Budd (2005). Understanding Object-Oriented Programming with Java.

UNIT 5 CONVERSION OPERATOR OVERLOADING AND INDEXERS

CONTENTS

1.0 INTRODUCTION

In order to complete our discussion on operation overloading, this unit describes the process for overloading conversion operators to allow implicit and explicit casting between data types. To understand conversion operator, we will make use of casting. Casting permits a value of one type to be converted to another data type so that it can be used in a calculation or method or in any other situation where the value's current data type is unsuitable.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- create an explicit conversion
- differentiate between implicit and explicit conversion
- create an indexer
- explain how to add indexers to main class.

3.0 MAIN CONTENT

3.1 Conversion of Data Types

The conversion of data types provided by casting can be either implicit or explicit. An implicit cast occurs automatically when an implicit conversion operator has been defined and the two data types are compatible. An explicit cast is used when the two data types are not entirely compatible and requires the source type to be prefixed with a cast operator. This operator is the desired data type enclosed in parentheses ().

The following code sample shows a comparison of conversion using implicit and explicit casts for basic numeric data types. Note the requirement for the cast operator for the second conversion.

```
unit integer = 100;
long longInteger;
// Implicit cast
longInteger = integer;
// Explicit cast
integer = (unit)longInteger;
```

When you create a new class, it does not possess the capabilities to be cast to other data types. Each possible cast must be defined within the class and declared as either implicit or explicit. As can be seen in the above example code, the implicit cast can be easier to read and can provide neater code. However, it is not immediately apparent to the reader that a conversion is occurring and so implicit casting can hide problems that would be more apparent if an explicit cast operator were used.

As a rule of thumb, implicit casting should only be used where there is no risk of data loss or an exception being thrown. Explicit casting should be used in all other situations. This is demonstrated in the above code. Implicit casting from the smaller integer to the larger integer is risk-free as all possible values can be converted. For conversion from the large signed integer to the smaller unsigned integer, there is a risk that the value will lose its sign or be too large to be converted, so explicit casting is more appropriate. There are some restrictions to overloading the conversion operators. The key restrictions are:

- You may not create operators that convert a class to the object data type. Conversion to object is provided automatically to permit boxing and unboxing.

- You may not create operators that convert a class to a defined interface. If conversion to an interface is required, the class must implement the interface.
- You may not create operators that convert from a base class into a class derived from that base class.

This unit uses the Vector class. The Vector class represents a two-dimensional vector and already overloads arithmetic operators, true and false operators, logical operators and relational operators.

3.2 Creating an Implicit Conversion Operator

The implicit and explicit cast operators are unary operators and, as such, are overridden using a similar syntax as other basic unary operators. The following syntax is for the implicit conversion operator:

public static implicit operator result-type(op-type operand)

The result-type is the data type for the return value of the operation, i.e. the target type for the cast. The *op-type* is the data type for the operand that is to be converted. One of the two data types must be the same as the class in which the declaration is made.

3.2.1 Adding Double Conversion to the Vector Class

Using the syntax described above, we will now add an implicit conversion operator to the Vector class. This operator will cast a vector to a double-precision floating point number representing the length of the vector. The length calculation is already present as a property of the class so we will reuse this functionality. To add the new conversion operator, add the following code to the Vector class:

```
public static implicit operator double(Vector v)
{
   return v.Length;
}
```

You can test the new conversion operator by modifying the Main method of the program:

```
static void Main(string[] args)
{
   Vector v = new Vector(5, 5);
   double d = v;
   Console.WriteLine(d);          // Outputs 7.07106781186548
}
```

3.3 Creating an Explicit Conversion Operator

The syntax for creating an explicit conversion operator is similar to that of the implicit version. Only the implicit keyword is changed. The syntax is therefore:

public static explicit operator result-type(op-type operand)

3.3.1 Adding Single Conversion to the Vector Class

We will use the explicit version of the conversion operator syntax to allow the vector to be cast as a single-precision floating point value representing the vector's length. The choice of explicit operator is due to the loss of accuracy of the number when the length is converted from a double to a float. To add the new conversion operator, add the following code to the Vector class:

```
public static explicit operator float(Vector v)
{
   return (float)v.Length;
}
```

The new conversion can be tested using the updated Main method below. Note the loss of accuracy that occurs when the vector's length is cast.

```
static void Main(string[] args)
{
   Vector v = new Vector(5, 5);

   float f = (float)v;
   Console.WriteLine(f);
}
// Outputs 7.071068
```

Note: The use of the (float) operator is required in this example as the conversion has been defined as explicit. Removing this cast operator causes a compiler error.

3.4 Indexers

A class that has an indexer can be used in a similar manner to an array. Objects of the class can use array-style notation to present multiple values. Most C# programmers first use an indexer when working with an array. An array is used to store a number of similar, related variables

under the same name, with each variable accessed using an index number provided in square brackets. For example:

thirdItem = items[3];

Indexer is use often to incorporate the square bracket notation for new classes. This may be because the class is used to store related information in a similar manner to an array, or simply because the index number can be useful in a calculation or lookup. Adding an indexer to a class provides this functionality.

3.4.1 Creating an Indexer

The simplest type of an indexer is the one-dimensional. A one-dimensional indexer accepts a single value between the square brackets when used. The standard syntax used to declare the indexer is similar to that used to define the get and set accessors of a property. However, instead of defining a property name, the accessors are declared for this[] as follows:

```
public data-type this[index-type index-name]
{
   get {}
   set {}
}
```

In the syntax definition, data-type determines the type of information that will be returned when the indexer is queried and the type that will be required when setting a value. Index-type specifies the data type of the indexer itself. This permits declaration of indexers that are not based upon integer values, allowing similar functionality to that of a Hash table for example. The index-name is the variable containing the index value that can be used during processing of the get and set accessors. The get accessor is required for an indexer and must return a value of the type data-type. The set accessor is defined for writeable indexers and is omitted if a read-only variant is desired.

3.4.2 Creating a New Array-Like Class

To demonstrate the use of an indexer, in this unit, we will create a new class that behaves like a simple array of string variables. Unlike a standard array that only permits zero-based indexing, the new class will provide an integer-based array for which the programmer can specify the upper and lower boundaries using a constructor during instantiation.

3.4.2.1 Adding the Class Variables

The new array-like class requires three private variables. Two integer values will hold the upper and lower boundaries. An array of strings will also be required to store the items added to the *MyArray* class. This will be a zero-based array with the same length as the created *MyArray* object. The indexer will interpret the index number supplied and map it against this underlying array forget and set operations. To add the class level variables, add the following code to the *MyArray* class' code block:

```
int _lowerBound;
int _upperBound;
string[] _items;
```

3.4.2.2 Adding the Constructor

The constructor for the new class will accept two integer parameters that define the upper and lower boundaries. These values will be stored in the two associated class variables. Using these boundaries, the length of the underlying array can be calculated and the array can be initialised accordingly. To create the constructor, add the following code to the class:

```
public MyArray(int lowerBound, int upperBound)
{
   _lowerBound = lowerBound;
   _upperBound = upperBound;
   _items = new string[1 + upperBound - lowerBound];
}
```

Note: To simplify this example validation, checks have been omitted. In a real program, you would want to validate that the upper boundary is larger than the lower boundary. Other validation checks in the code above have also been removed for clarity.

3.4.2.3 Adding the Indexer

Now that the preparation work is complete, we can add the indexer to the class. For this simple array-like class, the indexer accepts a single integer parameter containing the index of the string that is being read from or written to. This index needs to be adjusted to correctly map to the underlying data before returning the value from the array or writing the new value into the array. The code to add the indexer is shown below. Note that as with property declarations, the set accessor uses the

'value' variable to determine the value that has been assigned by the calling function:

```
public string this[int index]
{
   get
   {
      return _items[index - _lowerBound];
   }
   set
   {
      _items[index - _lowerBound] = value;
   }
}
```

Analysis:

The new class can be tested using the Main method of the program. Open the Program class and add the following code to create a new instance of *MyArray* and to populate it with values.

```
static void Main(string[] args)
{
   MyArray fruit = new MyArray(-2, 1);
   fruit[-2] = "Apple";
   fruit[-1] = "Orange";
   fruit[0] = "Banana";
   fruit[1] = "Blackcurrant";
   Console.WriteLine(fruit[-1]);      // Outputs "Orange"
   Console.WriteLine(fruit[0]);       // Outputs "Banana"
}
```

3.4.3 Creating a Multidimensional Indexer

Indexers are not limited to a single dimension. By including more than one index variable in the square brackets of the indexer declaration, multiple dimensions may be added. For example, to declare a two-dimensional indexer the syntax is as follows:

```
public data-type this[index-type1 index-name1, index-type2 index-
name2]
{
   get {}
   set {}
}
```

It is also possible to overload indexers in a similar manner to overloading methods. Several declarations for this [] can be included, each with a different set of parameters, or signature. We can demonstrate this by adding a second indexer to the *MyArray* class. For simplicity, the following code creates an overloaded read-only indexer.

```
public string this[int word, int position]
{
   get
   {
      return _items[word - _lowerBound].Substring(position, 1);
   }
}
```

To test the second indexer, modify the Main method as follows:

```
static void Main(string[] args)
{
   MyArray fruit = new MyArray(-2, 1);
   fruit[-2] = "Apple";
   fruit[-1] = "Orange";
   fruit[0] = "Banana";
   fruit[1] = "Blackcurrant";
   Console.WriteLine(fruit[-1, 0]);    // Outputs "O"
   Console.WriteLine(fruit[0, 2]);     // Outputs "n"
}
```

3.4.4 Creating an Indexer with no Underlying Array

As mentioned at the beginning of this unit, there is no requirement that the indexer is linked to an underlying array or collection. Sometimes, it is useful to use the indexer to refer to a row from a database table, a position within a text file or XML document or simply to perform a calculation. The following indexer, added to the *MyArray* class, uses no lookup at all. Instead, the floating-point value passed to the indexer is simply squared:

```
public float this[float toSquare]
{
   get
   {
      return toSquare * toSquare;
   }
}
```

This can be tested with a final change to the Main method:

```
static void Main(string[] args)
{
   MyArray fruit = new MyArray(0, 0);
   Console.WriteLine(fruit[5F]);       // Outputs 25
}
```

4.0 CONCLUSION

Conversion operators are those that involve converting from one data type to another through assignment. There are implicit and explicit conversions. Implicit conversions are those that involve direct assignment of one type to another. Explicit conversions are conversions that require one type to be casted as another type in order to perform the conversion. Conversions that may cause exceptions or result in loss of data as the type is converted should be handled as explicit conversions.

An indexer allows objects of the class to use array-style notation to present multiple values. Indexers are not limited to a single dimension. By including more than one index variable in the square brackets of the indexer declaration, multiple dimensions may be added. This time, the indexer will have two integer parameters. The first will determine which item from the private array will be looked up. The second parameter will determine a position within the string and return the character at that position.

5.0 SUMMARY

In this unit, the following were discussed:

- An implicit cast occurs automatically when an implicit conversion operator has been defined and the two data types are compatible. An explicit cast is used when the two data types are not entirely compatible and requires the source type to be prefixed with a cast operator.
- As a rule of thumb, implicit casting should only be used where there is no risk of data loss or an exception being thrown. Explicit casting should be used in all other situations.
- For conversion from the large signed integer to the smaller unsigned integer, there is a risk that the value will lose its sign or be too large to be converted, so explicit casting is more appropriate.
- The implicit and explicit cast operators are unary operators and, as such, are overridden using a similar syntax as other basic unary operators.

6.0 TUTOR-MARKED ASSIGNMENT

Describe what is meant by conversion operator overloading and write a program that will overload a conversion operator.

7.0 REFERENCES/FURTHER READING

Erich Gamma, Richard Helm, Ralph Johnson & John M. Vlissides (2006). Design Patterns: Elements of Reusable Object-Oriented Software.

Iain D. Craig (2003). Object-Oriented Programming Languages: Interpretation.

Nancy M. Wilkinson (2006). An Informal Approach to Object-Oriented Development.

Richard Wiener. Object-Oriented Introduction to Data Structures Using Eiffel [FACSIMILE] (Paperback).

Timothy Budd (2005). Understanding Object-Oriented Programming with Java.