

Table of Contents

Page

UNIT it INTRODUCTION TO OBJECT-ORIENTED CONCEPTS

1.0	Introduction	
2.0	Objectives	2
3.0	Fundamental Concepts of Object-Oriented Technology	3
3.1	The Object Model	4
3.2	Object-Oriented Programming	5
4.0	Conclusion	ii
5.0	Summary	ii
6.0	Tutor Marked Assignment	11
7.0	References	11

UNIT 2: MORE OBJECT ORIENTED CONCEPTS

1.0	Introduction	12
2.0	Objectives	
3.1	Encapsulation	12
3.2	Inheritance	14
3.3	Polymorphism	15
4.0	Conclusion	17
5.0	Summary	17
6.0	Tutor Marked Assignment	17
7.0	References & Other Resources	

UNIT 3: MORE OBJECT ORIENTED CONCEPTS

1.0	Introduction	18
2.0	Objectives	18
3.0	Relationships	18
3.1.1	A-Kind-Of Relationship	18
3.1.2	Is-A Relationship	20
3.1.3	Part-Of Relationship	20
3.1.4	Has-A Relationship	21
3.2	Inheritance	21
3.3	Multiple Inheritance	24
3.4	Abstract Classes	26
4.0	Conclusion	28
5.0	Summary	28
6.0	Tutor Mark Assignment	28
7.0	References And Other Resources	28

UNIT 4: A SURVEY OF PROGRAMMING TECHNIQUES

1.0	Introduction	29
-----	--------------	----

20	Objectives	29
31	Unstructured Programming	30
3.2	Procedural Programming	30
33	Modular Programming	31
34	An Example With Data Structures	32
35	Modular Programming Problems	37
36	Object-Oriented Programming	39
4.0	Conclusion	39
5.0	Summary	39
6.0	Tutor Marked Assignment	39
7.0	References & Other Resources	39

UNIT 5: SOFTWARE ENGINEERING

1.0	Introduction	40
20	Objectives	40
3.1	Software Product, Components And Characteristics	40
3.2	Software Engineering Concepts	42
33	Documentation Of The Software Product	44
3.4	Software Process And Models	45
4.0	Conclusion	48
5.0	Summary	48
6.0	Tutor Marked Assignment	49
7.0	References & Other Resources	49

UNIT 6: SOFTWARE QUALITY CONCEPTS

1.0	Introduction	50
2.0	Objectives	50
3.1	Important Qualities Of Software Product And Process	50
3.2	Principles Of Software Engineering	60
4.0	Conclusion	64
5.0	Summary	64
6.0	Tutor Marked Assignment	64
7.0	References And Other Resources	64

UNIT 7: LANDMARKS OF OBJECT-ORIENTED ANALYSIS AND DESIGN

1.0	Introduction	65
2.0	Objectives	65
31	Origin Of Object-Oriented Design (OOD)	65
3.2	Two Different Cultures	66
33	COD Approaches	70
34	Analysis of COD Approaches	73

4.0	Conclusion	75
5.0	Tutor Marked Assignment	75
6.0	References and Other Resources	75

UNIT 8: OBJECT-ORIENTED ANALYSIS AND DESIGN

1.0	Introduction	76
2.0	Objectives	76
3.1	Object-Oriented Analysis And Design	76
3.2	Role Of Ooad In The Software Life Cycle	77
3.3	Ooad Methodologies	79
4.0	Conclusion	84
5.0	Summary	85
6.0	Tutor Marked Assignment	85
7.0	References And Other Resources	85

UNIT 9: OBJECT-ORIENTED SOFTWARE DESIGN

1.0	Introduction	86
2.0	Objectives	86
3.1	Software Engineering Trends	86
3.2	Object-Oriented Approach	99
4.0	Conclusion	100
5.0	Summary	100
6.0	Tutor-Marked Assignment	100
7.0	References And Other Resources	100

UNIT 10: OBJECT-ORIENTED PROGRAMMING IN JAVA

1.0	Introduction	101
2.0	Objectives	101
3.1	A Checklist	107
3.2	Creating Your First Application	107
3.3	Creating Your First Applet	109
3.4	About The Java Technology	111
4.0	Conclusion	111
5.0	Summary	111
6.0	Tutor Marked Assignment	111
7.0	References & Other Resources	111

UNIT 11: A CLOSER LOOK AT THE "HELLOWORLD" SAMPLE

1.0	Introduction	112
2.0	Objectives	112
3.1	The "Hello World" Application	112
3.2	The "Hello World" Applet	118
3.3	Solving Common Compiler and Interpreter Problems	123
4.0	Conclusion	125
5.0	Summary	125

6.0	Tutor Marked Assignment	125
7.0	References & Other Resources	125

UNIT 12: OBJECT-ORIENTED PROGRAMMING CONCEPTS IN JAVA

1.0	Introduction	126
2.0	Objectives	126
3.1	What Is An Object	126
3.2	What Is A Message	129
3.3	What Is A Class	130
3.4	Object Vs. Classes	132
3.5	What Is Inheritance	132
3.6	What Is An Interface	134
4.0	Conclusion	135
5.0	Summary	135
6.0	Tutor Marked Assignment	135
7.0	References & Other Resources	135

UNIT 13: TRANSLATING CONCEPTS INTO CODE

1.0	Introduction	136
2.0	Objectives	136
3.1	Clickme Applet	136
3.2	Objects In The Clickme Applet	137
3.3	Classes In The Clickme Applet	138
3.4	Messages In The Clickme Applet	139
3.5	Inheritance In The Clickme Applet	140
3.6	Interfaces In The Clickme Applet	140
3.7	Api Documentation	142
4.0	Conclusion	142
5.0	Summary	142
7.0	References And Other Resources	142

UNIT 14: JAVA LANGUAGE BASICS I (VARIABLES AND OPERATORS)

1.0	Introduction	143
2.0	Objectives	143
3.1	Variables	144
3.1.4	Variable Initialization - 150	
3.2	Operators	151
4.0	Conclusion	159
5.0	Summary	159
6.0	Tutor Marked Assignment	159
7.0	References & Other Resources	160

UNIT 15: JAVA LANGUAGE BASICS 2 (EXPRESSIONS AND STATEMENTS)

1.0	Introduction	161
2.0	Objective	161
3.1	Expression	161
3.2	Statements	164
3.3	Blocks	165
3.4	Control Structures	165
4.0	Conclusion	175
5.0	Summary	175
6.0	Tutor Marked Assignment	175
7.0	References & Other Resources	176

UNIT 16: OBJECT-ORIENTED PROGRAMMING IN C++

1.0	Introduction	177
2.0	Objectives	177
3.1	History of C++	177
3.2	Programming Using C++	178
4.0	Conclusion	183
5.0	Summary	183
6.0	Tutor Marked Assignment	183
7.0	References & Other Resources	183

UNIT 17: C++ LANGUAGE BASICS

1.0	Introduction	184
2.0	Objectives	184
3.1	VARIABLES	184
3.2	DEFINING A VARIABLE	188
3.3	CONSTANTS	193
4.0	Conclusion	197
5.0	Summary	197
6.0	Tutor Marked Assignment	198
7.0	References And Other Resources	198

UNIT 18: C++ EXPRESSIONS AND STATEMENTS

1.0	Introduction	199
2.0	Objectives	199
3.1	Statements	199
3.2	Operators	201
3.3	The If Statement	205
3.4	Logical Operators	211
4.0	Conclusion	214
5.0	Summary	214
6.0	Tutor Marked Assignment	215
7.0	References & Other Resources	215

UNIT 19: CLASSES I

1.0	Introduction	216
2.0	Objectives	216
3.1	Classes and that/Member	216
3.2	Defining An Object	218
3.3	Classes Versus Objects	218
3.4	Accessing Class Members	218
3.5	Access Modifiers	219
3.6	Implementing Class Methods	223
4.0	Conclusion	226
5.0	Summary	226
6.0	Tutor Marked Assignments	226
7.0	References And Other Resources	227

UNIT 20: CLASSES II

1.0	Introduction	228
2.0	Objectives	228
3.1	Constructors And Destructors	228
3.2	Structures	242
4.0	Conclusion	243
5.0	Summary	243
6.0	Tutor Marked Assignment	243
7.0	References & Other Resources	243

UNIT 1: INTRODUCTION TO OBJECT-ORIENTED CONCEPTS**TABLE OF CONTENTS**

1.0	Introduction
2.0	Objectives
3.0	Fundamental Concepts of Object-Oriented Technology
3.1	The Object Model
3.2	Object-Oriented Programming
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	References

1.1 Introduction

This unit is a -general introduction to the fundamental concepts of object-oriented technology. The unit introduces the concepts of objects, classes and methods and the role they play in object-oriented software.

2.0 Objectives

By the end of this unit, you should be able to

- Define objects, classes and methods and identify their relationships
- Explain the concept of Object-oriented programming
- Identify Object-oriented languages

3.0 The Object Paradigm**3.1.1 What is an Object?**

An object is a "black box" which receives and sends messages. A black box actually contains code (sequences of computer instructions) and data (information which the instructions operate on).



An Object

An object is a .software unit composed of a set of data and a set of operations for accessing and processing that data. Traditionally, code and data have been kept apart. For example, in the C language, units of code are called functions, while units of data are called structures. Functions and structures are not formally connected in C. A C function can operate on more than one type of structure, and more than one function can operate on the same structure.

Not so for object-oriented software! In object-oriented programming, code and data are merged into a single indivisible thing -- an object. This has some big advantages, as you'll see in a moment. But first, why use the "black box" metaphor for an object? A primary rule of object-oriented programming is this: as the user of an object, you should never need to peek inside the box!

Why shouldn't you need to look inside an object'? For one thing, all communication to it is done via messages. The object which a message is sent to is called the receiver of the message. Messages define the interface to the object. Everything an object can do is represented by its message interface. So you shouldn't have to know anything about what is in the black box in order to use it.

*How are
you?*

I'm fine!

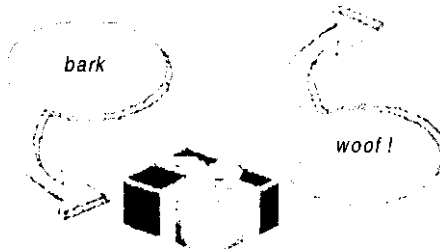


And not looking inside the object's black box doesn't tempt you to directly modify that object. If you did, you would be tampering with the details of how the object works. Suppose the person who programmed the object in the first place decided later on to change some of these details? Then you would be in trouble. Your software would no longer work correctly! But so long as you just deal with objects as black boxes via their messages, the software is guaranteed to work, providing access to an object only through its messages while keeping the details private is called information hiding (technically speaking, encapsulation).

Why all this concern for being able to change software? Because experience has taught us that software changes. A popular adage is that "software is not written, it is re-written". And some of the costliest mistakes in computer history have come from software that breaks when someone tries to change it.

3.12 Classes

How are objects defined? An object is defined via its class, which determines everything about an object. Objects are individual instances of a class. For example, you may create an object called Spot from class Dog. The Dog class defines what it is to be a Dog object, and all the "dog-related" messages a Dog object can act upon. All object-oriented languages have some means, usually called a factory, to "manufacture" object instances from a class definition.



Spot

You can make more than one object of this class, and call them Spot, Fido, Rover, etc. The Dog class defines messages that the Dog objects understand, such as "bark", "fetch", and "roll-over".

3.1.3 Methods

You may also hear the term method used. A method is simply the action that a message carries out. It is the code, which gets executed when the message is sent to a particular object.

A method can be applied to (or invoked on) a specific object or the class itself. To be more precise a method consists of the following:

- (i) a n a m e
- (i i) a (possibly empty) list of input names, called parameters or arguments, and their types
- (i i i) an (optional) output type, called the return type (if the return type is 'void', then it means that there is no return type)
- (i v) a body of executable. The body of executable code is called the implementation of the method.

Arguments are often supplied as part of a message. For example, the "fetch" message might contain an argument that says what to fetch, like "the-stick".

Or the "roll-over" message could contain one argument to say how fast, and a second argument to say how many times.

Exercise 1.1

Identify the Object, Class and Method in this: I have two Balls, a football and volleyball. The balls could bounce fast or slow and could roll left or right.

Answer:

Objects	Volleyball, Football
Class	Ball
Methods	Bounce (argument fast/slow) Roll (argument left/right)

32 Object Oriented Programming

3.2.1 What is Object-Oriented Programming?

Since the early days of computing, programmers have looked for ways to manage the complexity of programming computers. Because a computer's central processing unit (CPU) works by fetching and executing simple instructions from memory, early computer programs were a sequence of such machine instructions that had to be loaded into memory through a set of switches or through a numeric keypad.

Assembly language improved the situation by enabling us to use mnemonic names for the machine instructions and symbolic names for memory locations. A translator called the assembler converts assembly language programs into machine code. Assemblers soon started providing special commands or directives that allowed programmers to group basic data items into structures with assigned names. With a program's tasks broken down into procedures and data organized into structures, assembly language provided reasonable programming facilities. However, the close connection between assembly language and the machine code means that assembly language forces you to think of the program in terms of the machine instructions that the underlying CPU can execute.

Higher-level languages such as FORTRAN, BASIC, Pascal, and C largely eliminate the close ties to the CPU's machine instructions by providing standard data types such as integers, floating-point

numbers, and character strings that can be used in expressions and statements-each of which is translated by a compiler into many machine instructions. Most recent applications are written in a combination of these high-level languages and some assembly language. The accepted programming style has been to organize related data items using programming constructs such as Pascal RECORDs or C STRUCTs and then treat the resulting block of data as a single unit. Once the data structures are laid out, the application is written as a collection of procedures that manipulate these structures.

Although the traditional "design the data structures and write the functions to manipulate them" approach to programming has served us well, there is no denying that the complexity of software is increasing in keeping with more powerful computer hardware. With ever-increasing hardware capabilities such as faster CPUs, better graphics, and easier networking, users have come to expect software to have greater functionality. Users now routinely expect programs to include features such as a window-based graphical user interface, transparent access to data stored in mini- or mainframe computers, and the ability to work in a networked environment. Faced with this complexity, more programmers are starting to use object-oriented programming (OOP). OOP is a new way of organizing code and data that promises increased control over the complexity of the software development process.

Object-oriented programming, is nothing new; its underlying concepts are data abstraction, inheritance, and polymorphism (explained in later units). All three have been around for quite some time (for example, in languages such as SIMULA67 and Smalltalk). What is new is the increasing interest in OOP among programmers in general.

The term object-oriented programming (OOP) is widely used, but experts cannot seem to agree on its exact definition. However, most experts agree that OOP involves defining abstract data types (ADT) representing complex real-world or abstract objects and organizing your program around the collection of ADTs with an eye to exploiting their common features. The term data abstraction refers to the process of defining ADTs: inheritance and polymorphism refer to the mechanisms that enable you to take advantage of the common characteristics of the ADTs-the objects in OOP. These terms are clearly explained later in the course.

The term abstract data type, or ADT for short, refers to a programmer-defined data type together with a set of operations that can be performed on that data,

Before you jump into OOP, take note of two points. First, Use of object-oriented technology is only a method of designing and implementing software. It does not impart anything to a finished software product that the user can see. However, as a programmer implementing the software, you can gain significant advantages by using object-oriented methods, especially in large software projects. Because OOP enables you to remain close to the conceptual, higher-level model of the real-world problem you are trying to solve, you can manage the complexity better than with approaches that force you to map the problem to list the features of the language. You can take advantage of the modularity of objects and implement the program in relatively independent units that are easier to maintain and extend. You can also share code among objects through inheritance.

Exercise 1.2

Mention three underlying concepts of object oriented programming
Answer

Data Abstraction, Inheritance, Polymorphism

Practical Examples

If you wanted to add two numbers, say, 1 and 2, in an ordinary, non-object-oriented computer language like C (don't worry --you don't need to know any C to follow this), you might write this:

```
a=1;b=2;
c=a+b;
```

This says,

"Take a, which has the value 1, and b, which has the 2, and add them together using the C language's built-in addition capability. Take the result, 3, and place it into the variable called c." Now, here's the same thing expressed in Smalltalk, which is a pure object-oriented language:

```
a := 1.
b := 2.
c := a+b.
```

Wait a minute. Except for some minor notational differences, they look exactly the same ! Okey, it is the same, but meaning is dramatically different.

In Smalltalk, this says,

"Take the object a, which has the value 1, and send it the message which included the argument b, which, in turn, has the value 2. Object a. receives this message and performs the action requested,. which is to add the value of the argument to yourself. Create a new object, give this the result, 3, and assign this object to c."

This seems a more complicated way of accomplishing exactly the same thing! So why bother?

The reason is that objects greatly simplify matters when the data get more complex. Suppose you wanted a data type called list, which is a list of names. In C, list would be defined as a structure.

```
struct list (  
<definition of list structure data here> ,
```

```
list a, b, c;  
a = "John Jones";  
b = "Suzy Smith";
```

Let's try to add these new a and b in the C language:

Guess what? This doesn't work. The C compiler will generate an error when it tries to compile this because it doesn't know what to do with a and b. C compilers just know how to add numbers. Period. But a and b are not numbers. One can do the same thing in Smalltalk, but this time, list is made a class, which is a subclass of the built-in Smalltalk class called "String":

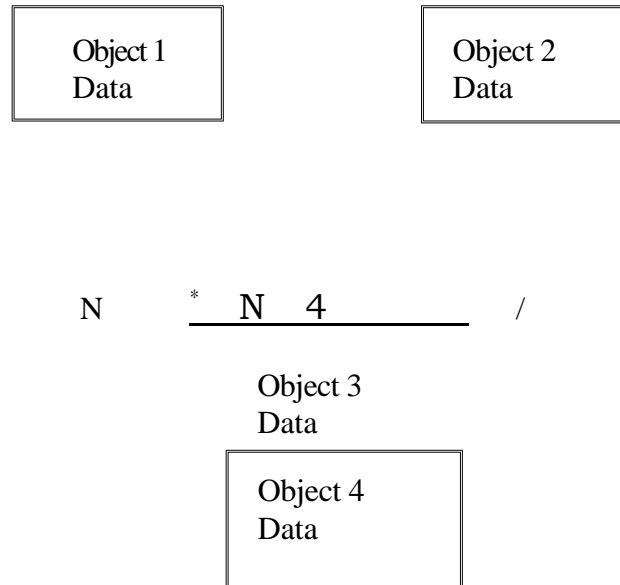
```
a := List from String : 'John Jones'.  
b := List from String : 'Suzy Smith'.  
c := a + b.
```

The first two lines simply create List objects a and b from the given strings. This now works, because the list class was created with a method which specifically "knows" how to handle the message. For example. it might simply combine the argument with its own object by sticking them together with a comma separating them (this is done a single line of Smalltalk I:). So c will have the new value:

```
'John Jones, Suzy Smith'
```

3.2.1 Object-Oriented Programming Languages

Object-oriented programming is a programming method that combines data and instructions into a self-sufficient "object". In an object-oriented program, the design is represented by objects. Objects have two sections, fields (instance variables) and methods. Fields represent what an object is. Methods represent how an object is used. These fields and methods are closely tied to real world characteristics and use of the object. An object is used by means of its methods. The following figure shows a view of object-oriented programming.



Atka'
° u_
ci.....0cA
\\ 40/0- cr
stkr

A pro rammin jjsupports the
concepts of data abstraction, class, and more advanced concepts
 such as inheritance and polymorphism.

There are almost two dozens major object-oriented programming languages in use today. But the leading commercial Object-Oriented languages are far fewer in number. These are:

C'++
 Smalltalk
 Java

C++
 C++ is an object-oriented version of C. It is compatible with C (it is actually a superset), so that existing C code can be incorporated into

C++ has become so popular that most 'lbw C **compilers are actually C/C++ compilers. However, to take full advantage of object-oriented** pedgranthdtig; ohs mustrogram(and think!) in.C+4_tuot C. This can 6fr6n)be tt 1najor problem for experiencedjaprogrammers. Many tortigrraraitetsithr1k4h8Yeteleadingibteisiti battistateadJare only using a atturIFpattiethelingthigestibjeeNortented pasiehr.

tenicriv
%lama . mlitai'
, "ft to ;I lit A) Lou qjp.
&naked/ fall(pamobjecfratridesnedllangteWhikuflo- Makes
lintiaticalicannottanisbot easurrenfast expautiorandarnall code size,
Smalltalk makes none. It uses run-lima trindinpmarrhich means that
nothing about the type of an object need be known before a Smalltalk

Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. A rich class library that can be easily reused via inheritance is one reason for this. Anitlitsansihson

- R/ bi ait44hrtsJithat itixtreanades neluvibendalsilyrstaisde onhurak,classes niamtdviittitioniy Atatriltritefinedit illuartibefolgrapprutritrobjapbasariented, ihhoillttigratntAtittlinuanotaingthyfripatibeirgtogatinlahn Ci/I•gastriented waters, as with C-1+. For this 'reason, Smalltalk general's-hies longer to Master than C++. But most of this time is actually spent learning

0/9)-titnithdktt allsichihsisibasamosantidsclizathe Ssnalltailplanuage
differs somewhat from one implementation teinoth.alhainust
Populartidnurieroialilbaleols" ofStnalinikatar'. 4tri 3101111

tbsoft **iterstscna**

Visual Age from IBM

Java

Java is the latest, flashiest object-oriented language. It has taken the software world by storm due to its close ties with the Internet and Web browsers. It is designed as a portable language that can run on any web-enabled computer via that computer's Web browser. As such, it offers great promise as the standard Internet and Intranet programming language.

Java is a curious mixture of C++ and Smalltalk. It has the syntax of C++, Making it easy (or difficult) to learn, depending on your experience. But it has improved C++ in some important areas. For one thing, it has no pointers, low-level programming constructs that make for error-prone programs. Like Smalltalk, it has garbage collection, a feature that frees the programmer from explicitly allocating and de-allocating memory. And it runs on a Smalltalk-style virtual machine, software built into your web browser, which executes the same standard compiled Java byte codes no matter what type of computer you have.

Java development tools are being rapidly deployed, and, are available from such major software companies as IBM, Microsoft, and Symantec.

2. Object-oriented programming

Object-oriented programming is a new and powerful model for writing computer software. Objects are "black boxes" which send and receive messages. This approach speeds the development of new programs, and, if properly used, improves the maintenance, reusability and modifiability of the software.

Object-oriented programming requires a major shift in thinking by programmers; however, the C++ language offers an easier transition via C, but it still requires an object-oriented design approach in order to make proper use of this technology. Smalltalk offers a mature object-oriented environment, with more rapid development time and greater flexibility and power. Java promises much for Web-enabling Object-oriented programs.

5.0 Summary

In this unit you have learnt about objects and how they are defined through classes. You also learnt about the underlying concepts of object-oriented programming and some examples of object-oriented languages. We will build on these concepts as we proceed in the course.

6.0 Tutor Marked Assignment

TV 1

- 1 Describe the following terms
 - Object
 - Class
 - Method

- 2 Describe Object-oriented programming

7.0 References & Other Resources

Terry Mantic What is **Object-Oriented StilliOrt Copyright1995**l: 199611y
Software Design ConsultantmLL.CAlltigidstammetlit.A uT

Nabajyoti **Saticalcati, Object-Oriented ProgramamingtittoC4+, Prentice-Han of India PriVatetintited; New Delhi — 110 001, 2001.**

Object-oriented programming is a programming paradigm that uses objects and classes to represent data and behavior. It is a paradigm that uses objects and classes to represent data and behavior. It is a paradigm that uses objects and classes to represent data and behavior.

- Object-oriented programming is a programming paradigm that uses objects and classes to represent data and behavior.
- Object-oriented programming is a programming paradigm that uses objects and classes to represent data and behavior.
- Object-oriented programming is a programming paradigm that uses objects and classes to represent data and behavior.
- Object-oriented programming is a programming paradigm that uses objects and classes to represent data and behavior.

Object-oriented programming is a programming paradigm that uses objects and classes to represent data and behavior. It is a paradigm that uses objects and classes to represent data and behavior. It is a paradigm that uses objects and classes to represent data and behavior.

Object-oriented programming is a programming paradigm that uses objects and classes to represent data and behavior. It is a paradigm that uses objects and classes to represent data and behavior. It is a paradigm that uses objects and classes to represent data and behavior.

Object-oriented programming (OOP) is a programming paradigm that uses objects and classes to organize and structure code. It is a way of thinking about and organizing the components within the computer. OOP is a programming paradigm that uses objects and classes to organize and structure code. It is a way of thinking about and organizing the components within the computer.

UNIT 2: MORE OBJECT ORIENTED CONCEPTS

TABLE OF CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.1 Encapsulation
- 3.2 Inheritance
- 3.3 Polymorphism
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor Marked Assignment.
- 7.0 References & Other' Resources

it 1:-1 i Introdaditeft

In the previous unit you have learned about some basic object-oriented concepts. In this unit you will learn about some more object-oriented concepts.

2.0 Objecdvves

After completing this unit, you should be able to:

- Explain Encapsulation
- Explain PolSmorphism
- Explain Inheritance

3.1 Encapsulation

Encapsulation is hiding parts of program that do internal things, and by their nature shouldn't be of concern to anybody from outside. Why good to hide anything? Because of two reasons:

- (i) You warn user of your code what he can use safely, and what is not of his concern.
- (ii) You can change (fix errors and enhance) what you hid, without influencing- public part. User continues to use your code in the same manner, but it works better.

Imagine you are writing piece of software that does some common task (graphic library for example)., you are in role of programmer writing for other programmers, and your relation with these programmers is similar to relation between them and end, users. Although, end users don't know (nor They should. the program is *made*, they, are perfectly capable to use it efficiently through it's user interface. The same is true with programmers that use your library - don't waste their time 'by making them know internal details of your code. Just provide efficient "user interface" and let them use your code only through that interface.

Maybe what you are writing is huge. There is lot of auxiliary/internal code. Line count is 10000+. Of course you are constantly developing it - fixing bugs and adding new features. Put yourself in the place of programmers who are using your code. Whenever you make a change, these poor fellows have to check if their code is compatible with the "new version" of your code. If you are selfish person. YOU might ask: "Why, that should, be my concern?". Well, IT IS YOUR CONCERN: It's Perfectly possible :(especially if you are not working in a team) that "other programmers" are actually, ypii. If you are member of the team - remember, team wins, not an individual player!

The solution is to divide constant from misting part. Every library/class/module... has part which has form that it has because it's logical and natural to be that way, and because communication with the outer world is conducted ,through it. The "communication channel" is ,permanent. Only the way messages that come through it are processed changes from time to time.

That constant communication channel is called *interface*. The rest is *Implementation*

Classic programming tools, mainly modules and, in a lower level, blocks (begin end in Pascal, { } in C-like languages), give fair set of possibilities to use encapsulation. OOP, however, has comprehensive model of encapsulation related to classes.

Every element (method) of one class can have one of these three levels of visibility:

1. **Public** elements are completely visible from outside of the class. They make interface to the outside world.
2. **Private** elements are visible only from methods of the class itself. In some languages, their visibility is extended to module in which class resides. They make implementation.

3 Protected elements are something between I and 2. To the outside world, they act same as the private elements, but they are completely visible from inherited child classes. They have public behaviour for these classes. In some languages protected elements visibility is extend to module in which class resides. Although, strictly speaking, protected elements belong to implementation, they should be considered as interface to child classes.

Let's make one Delphi class:

```

TTriangle = class
private
  X,Y,A: integer;
protected
  procedure Initialize;
public
  procedure Place (x — arg- arg: integer);
  procedure Draw;
published
end;

```

Just Initialize. Place and Draw, methods can access X, Y and A. No outer function can access X, Y and A. Initialize can be called from child classes. Finally Place and all methods from child classes. Finally Place and Draw can be called from any place in which class itself is visible.

In addition, Delphi permits access to private and protected members from the same unit. Java has similar behaviour for packages. C++ is more rigorous (and, as the matter of fact, has no real modules) and there is no such visibility of private and protected members, but there is so-called friend mechanism. Delphi supports fourth, published specified, which has role in visual programming. But this **isnot** our topic in this point.

3.2 Inheritance

One of the most important, (and yet not completely solved) problems of programming states: "How not to re-invent a wheel again?". Differently said, problem is: how to write something that can be used more than one time. I admit, sounds weird.

However, anybody with intense Programming experience, wrote similar (or same!) code in different programs, or even the same program.

Why?

There are several reasons - one is traditional laziness of computer programmers. Simply, it was easier to ad-hoc do exactly what has to be done, exactly where it has to be done, than to try to extract a global solution that can be later specialized for concrete cases. Such "copy-and-paste" programming (meaning copying pre-existing code to needed place and changing it slightly) is a way to finish the program fast and to ~~work~~ ^{not} work fast. But, the word "fast" doesn't mean "robust". In this case, word "fast" means "nightmare" for maintenances and enhancement.

Not to blank only programmers, languages too didn't support "abstract" way of thinking that would allow you to think using, concepts close to problem. not using concepts close to machine, such as bytes and pointers.

In classic "procedural", or "structure" programming; there are techniques to extract a code that handles some general task (although some old dBase-like languages lack even these capabilities). The most important technique of that kind are "procedures" (or "functions" or "sub-routines"): The second Important technique are "modules" (Similar terms: units, "packages". even "compilation Unit"); Modules appear in three roles: 1) encapsulation 2) code reuse achieved thru procedures contained in modules and 3) physical organization a module is (in most languages) a file.

It seems this, Solves the need for Serime; but Slim long track (and in the large programs), We are napped lit thinking on the machine level, not problem level. •

There are better, Object-oriented solutions which don't negate classic techniques, but organize them on a higher level. Key stitution is **libtrilance**•

Inheritance_ is the way to change or atend already existing class .(parent — C, Lss) by Making it's child (inherited; derived) class and writing otvly 'changed or cilhanre4/ pia in flat classed 'everything the (pin of.dul patent that . ~~doesn't~~ need telie'cluniged), cliildolaktrithimits torn the parent /hem is an important fact child remains linked to it's parent - when parent is changed, 'child changes a Well (in its inherited Mt). sa if you fix a **wit** "Within the parent, You autosnally fixed that bug in all children. If you 'enhance parent you diktat: id all children.

Let's say we *need* bunch of different Objects /that art used in the same way. Typical example. widely used in OOP literature, are geometric shapes.

Wouldn't he nice' ~~to fill~~ ^{to fill} settledita stnachire With square*: triangles.

Circles,... and call DI-011' I (In all of them, without further complications (such as explicit type checking)).

Traditional solution is use Of classical structures (records in Pascal) or unions that contain data-type identification field. This approach suffers front two weaknesses.

1. Bad abstraction and non-optimal use of space because all types have to be placed in one format. If we rationalized use Of Space. that usually means we "flattened" the types too much. and they are no more a good representation of the problem we are trying to solve. In the other hand, if we wanted to achieve an optimal abstraction level, there is probably lot of wasted space because physical formats of types are very different Of course, tricks are always available (Such pointers and dynamic memory allocation) but that implicates a different set of problems.

Non-standardized type identification fields. You have to memorize what value identities what type can help, but Simply it's not an optimal solution.

Polymorphism is ability of objects to act depending ,a their run-time type. Without inheritance, polymorphism makes no sense. Without inheritance it's impossible to pave object of one (child) class placed instead of another (parent) class. So, with inheritance, we need a mechanism to deal with awareness of the nm-time (not the compile-time) class of the particular it's , placed instead of different compile-time declared class.

Polymorphism is such mechanism.

-Polymorphism is extremely , important for effective use of inheritance. Frequently, it is used to "persuade; methods of ,k;•arent, class to use redefined _tnelbods of the child class. A good example can be seen intim inheritance.

The other Situation is what I wrote the beginning of this page we have a number of objects with the ,same interface, but different,beHiayourk . The solution is to create one base class that will contain ell,common Methods. 'these methods should be virtual. After that. from &Chase class, we'll inherit all needed classes. Then, we'll declare an array (or list or,whatever) whose elements are declared as instances of the base clasi, litie in 'the run-titrie instances of child classes bc Naceol,IPtead.,,

Virtual methods m:44141104 awake* polytpoR

In some language Smalltalk) all methods are ,vittual - polymorphism is always ensures no other version is called (no matter that other version exists

in a child class, and a method is called for instance of the child class). You should be very careful when deciding do you need polymorphic or non-polymorphic behaviour for particular method.

4.0 Conclusion

According to Ejaime Structure, Programming language has two aspects;

- First is technical part of telling the machine what to do.
- Second is conceptual tool that you use in thinking how* solve a problem.

These concepts you have learned in this unit are not only contained in every major object-oriented language but represent excellent tool ,ofl abstract thinking.

5.0 Summary

You have learned about the major concepts of object-oriented programming in this unit; these concepts will continue to get, claim and clutter 10 you as , • you continue in the course.

121

6.0 Tutor Marked Assignment

fin,141111:1

1 Explain fully any of the following terms.

1. Inheritance
2. Encapsulation
3. Polymorphism

7.0 References and Other Resources

www.bepp.8m.com/Englishoop/aboutoop.htm . This link contains information about the basic concepts of object-oriented programming and the various languages that support it.

The following links provide information about the various languages that support object-oriented programming. The links are provided for your reference and are not intended to be a comprehensive list of all the languages that support object-oriented programming.

Unit 3: More Object Oriented Concepts

UNIT 3: MORE OBJECT ORIENTED CONCEPTS

TABLE OF CONTENTS

1.0	Introduction	
2.0	Objectives	
3.0	Relationships	
3.1.1	A-Kind-Of Relationship	
3.1.2	Is-A Relationship	
3.1.3	Part-Of Relationship	
3.2	Inheritance	
3.3	Multiple Inheritance	
3.4	Abstract Classes	
4.0	Conclusion	
5.0	Summary	
6.0	Assignment	
7.0	References And Other Resources	

1.0 Introduction

Whereas the previous units introduce the fundamental concepts of object-oriented programming, this unit presents more details about the object-oriented idea. We will use a pseudo language to describe most of the concepts described in this unit.

2.0 Objectives

By the end of this unit, you should be able to

- Explain types of relationships of classes
- Describe Inheritance and Multiple Inheritance
- Define Super class and Subclass
- Describe Abstract Classes

3.1 Relationships

3.1.1 A-Kind-of relationship

Consider you have to write a drawing program. This program would allow drawing of various object such as points, circles, rectangles, triangles and many more. For each object you provide a class definition. For example, the point class just defines a point by its coordinates:


```
class Point [
  attributes:
  Int x, y
```

```
  methods:
  sexX(int newX)
  getX ()
  setY(int newY)
  getY ()
```

'I • 1.F. tie21

·f', 1 Jr': I,cd, d

You continue defining classesiofeyotutdrawing prd 4-m with a class to describe circles:A circle defines, a centerrtioint mil a radius:

```
class Circle [
```

'!titir4.-I

```
  attributes:
```

```
  int x, y,
  radius
```

iplintontskri / E. I./.

```
  .1
```

tiftl Pfirj sd 1

v, I ;T(inethadgf e;,1

WI setX(iut netyX)ro

11540

lt)t.)• #WP1115

```
  getX ()
```

```
  isetY(ist itewY)_
```

'vim/.

H&K

```
  getY104
```

IV ne'.

/Le'

I

```
  'setRadiustnewRadius)
```

.f

Y EfttSi'

```
  (getRadintrOf (I'
```

3/01¹, U

<⁴1§

./ to

Jut, /

Comparing both class definitions we can observe the following:

2,114,0116

10011 202

t1 hit: rd..); lit !Pun' S.F. -stin2,1T1

Both cfttgstigikatOutWal anti) eltititiOtrri0 an* VitIn the class Point these elementrAiesavitottheipestitiowef thepMk4. Iii the case class Circle the describe the circle's center. Thus. .v and, have the same uteanirig in bOttuchuse)r.itThey:ditsalatrihe position of their associated objec_t_by defining a point.

Both classes offer the saAe

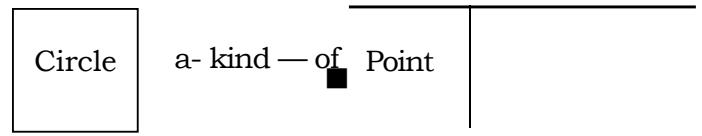
- the value of the two data elements x and y.

- Class Circle "adds" a *new* data element radius and corresponding access methods.

Knowing the properties of class Poino9.afffiscifityitfireas a point plus a radius and methods to access it. Thus. a Circle is "a-kind-or¹mpoitll. lovy,cvor, a is jcsotutti.ytAat, uvisa'tveRiplized". We il.ITIRC5 this f¹⁴1¹⁴4th f s??9- 4ritqffe 0.¹₀ e lila

f: "1:11

•:/nri

Figure 3.1: Illustration of "a-kind-of relationship.

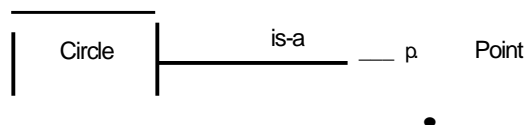
In this and the following- figures, classes are drawn using rectangles. Their name always starts with an Uppercase letter. The arrowed line indicates the direction of the relation, Hence, it is to be read as "Circle is a-kind-of Point."

3.1.2 Is-A relationship

The previous relationship is used at the class level to describe relationships between two similar classes. If we create objects of two such classes we refer to their relationship as an "is-a" relationship.

Since the class Circle is a kind of class Point, an instance of Circle, say a circle, is a point. Consequently, each circle behaves like a point. For example, you can move points in x direction by altering the value of x. Similarly, you move circles in this direction by altering their x value.

Figure 3.2 illustrates this relationship. In this and the following figures, objects are drawn using rectangles with round corners. Their name only consists of lowercase letters.

Figure 3.2: Illustration of "is-a" relationship.

3.13 Part-Of relationship

You sometimes need to be able to build objects by combining them out of others. You already know this from procedural programming, where, you have the structure or record construct to put data of various types together.

Let's come back to our drawing program. You already have created several classes for the available figures. Now you decide that you want to have a special figure which represents your own logo which consists of a circle and a triangle. (Let's assume, that you already have defined a class Triangle.) Thus, your logo consists of two parts or the circle and triangle are part-of your logo:

```
class Logo {
  attributes:
    Circle circle
    Triangle triangle
```

```
methods:
```

```
  set (Point where)
```

This is illustrated in Figure 3.3.

Figure 3.3: Illustration of "part-of" relationship.



3.1.4 Has-A relationship

This relationship is just the inverse version of the part-of relationship. Therefore we can easily add this relationship to the part-of illustration by adding arrows in the other direction (Figure 3.4).

Figure 3.4: Illustration of "-has-a" relationship.

Circle	Part-of	Part a	Tangle
	h a s - a t	h is- a	

3.2 Inheritance

*1111 inheritance we are able to make use of the a-kind-of and is-a relationship. As described there, classes which are a-kind-of another class share properties of the latter. In our point and circle example, we can define a circle which inherits from point:

class Circle inherits from Point:

```
  attributes:
    int radius
```

11 **get-Raditisa**

Class Circle inherits all data elements and methods from point. There is no need to define them twice: We just use already existing and well-known data and method definitions.

On the object level we are now able to use a circle just as we would use a point, because a circle is-a point. For example, we can define a circle object and set its center point coordinates:

```
Circle acircle
acircle.setX (1) /* Inherited 'from' ['hint */
acircle.setY(2)
acircle.setRUDing(3) /* Added by Circle /
```

"Is-a" also implies, that we can use .a circle everywhere where a point is expected. For example, you can write a/unction or method, say move(), which should move a point in x direction:

```

move (Point apoint, int deltax)
    apoint.setX(apoint.getX() + deltax)
015.39t4410lit$fl10l12,atpoint; you can use4thir futittinnWith a Circle
rpowpf t      yqiisipepter point and; tiChce,!thewhole cutl

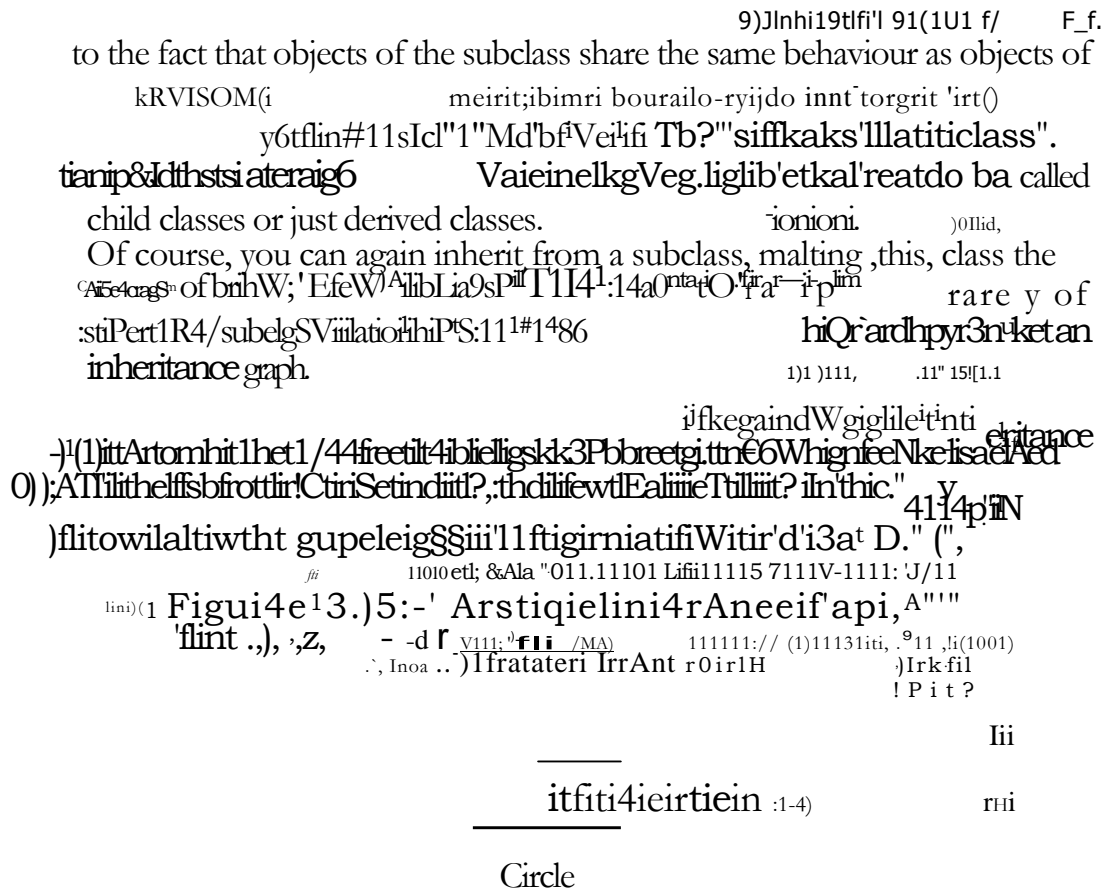
Circle acircle
    •!ti-r
. . .
Move (acircle,10)      /* Move circle by moving */
    - -1* its_center_point *1 -

```

Let's try to formalize the term "inheritance":

Definition (Inheritance) Inheritance is the mechanism which allow a C^{iAsig} A to inherit properties of a class B. we say "A inheres/1' from $AlgeCtlf$ class^{cl} A tht gl, 119 YilacceSs Ito attributes and methods! of class' 13 without the need to redefine, lhenn. The following definition defines' two terms With which we are able to refer to participating classes Nyhen they use inheritance.

Definition (Super class Subclass) If class A inherits, from class 13, then B is called super class of A. A is called succors of 13. Objects of a subclass can be used where objects of the corresponding super class are expected. This is due



In some literature you also find illustrations where the arrowed lines are used just the other way around. The direction in which the arrowed line is used, depends on how the corresponding author has decided to understand it.

Anyway, within this unit, the arrowed line is always directed towards the superclass.

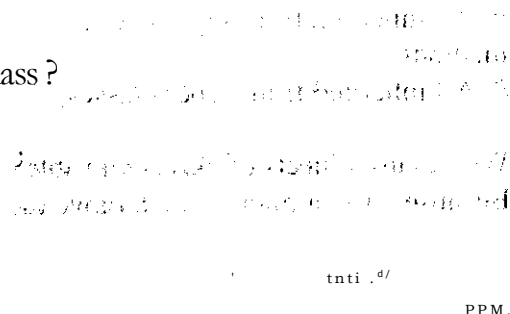
In the following sections an unmarked arrowed line indicates "inherit-from".

Exercise 2.1

What is another name for a subclass?

Answer

Derived class



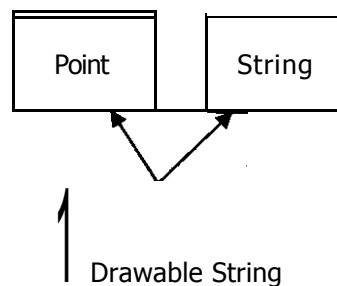
3.3 Multiple Inheritance

One important object-oriented mechanism is multiple inheritance. Multiple inheritance does not mean that multiple subclasses share the same superclass. It also does not mean that a subclass can inherit from a class which itself is a subclass of another class.

Multiple inheritance means that one subclass can have more than one superclass. This enables the subclass to inherit properties of more than one superclass and to "merge" their properties.

As an example, consider again our drawing program. Suppose we already have a class `String` which allows convenient handling of text. For example, it might have a method to append other text. In our program we would like to use this class to add text to the possible drawing objects. It would be nice to also use already existing routines such as `move` — to move the text around. Consequently, it makes sense to let a drawable text have a point which defines its location within the drawing area. Therefore we derive a new class `DrawableString` which inherits properties from `Point` and `String` as illustrated in Figure 3.6:

Figure 3.6: Derive a drawable string which inherits properties of `Point` and `String`.



In our pseudo language we do this by simply separating the multiple superclasses by comma:

```

class DrawableString inherits from Point, String
attributes:
/* All inherited from super classes */
methods:
/* All inherited from super classes */
  
```

We can use objects of class `DrawableString` like both points and strings. Because as we have a point we can move them around

```

DrawableString dstring
Move (dstring, 10)...
  
```

Since it is a string, we can append other text to them: `cistring.append("The red brown fox ...")` Now it's time for the definition of multiple inheritances: Definition (Multiple Inheritance) If class A inherits from more than one class, ie. A inherits from B1, B2, ..., Bn, we speak of multiple inheritance. This may introduce naming conflicts in A if at least two of its super classes define properties with the same name.

The above definition introduce naming conflict which occur if more than one super class of a subclass use the same name for either attributes or methods. For an example, let's assume, that class `String` defines a method which sets the string to a sequence of "X" characters. The question arises, what should be inherited by `Drawable String`? The `Point`, `String` version or none of them?

These conflicts can be solved in at least two ways:

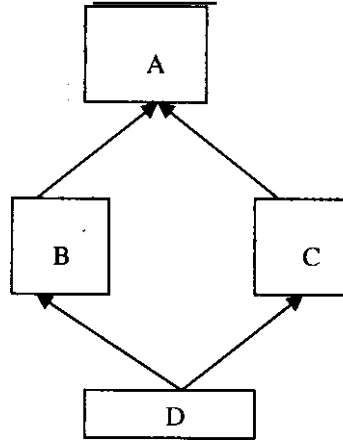
- The order in which the super class is provided define which property will be accessible by the conflict causing name. Others will be "hidden".

The subclass must resolve the conflict by providing a property with the name and by defining how to use the ones from its super classes.

The first solution is not very convenient as it introduces implicit consequences depending on the order in which classes inherit from each other. For the second case, subclasses must explicitly redefine properties which are involved in a naming conflict.

A special type of naming conflict is introduced if a class **D** multiply inherits from two super classes **B** and **C** which themselves are derived from one superclass **A**. This leads to an inheritance graph as shown in Figure 3.7.

Figure 3.7: A name conflict introduced by a shared superclass of super classes used with multiple inheritance.



The question arises what properties class D actually inherits from its super classes B and C. Some existing programming language solve this special inheritance graphs by deriving D with

The properties of A plus

The properties of B and C without the properties they have inherited from A.

Consequently, D cannot introduce naming conflicts with names of class A. However, if B and C add properties with the same name, D runs into a naming conflict.

Another possible solution is, that D inherits from both inheritance paths. In this solution, D owns two copies of the properties of A: one is inherited by B and one by C.

Although multiple inheritance is a powerful object-oriented mechanism the problems introduced with naming conflicts have lead several authors to "doom" it. As the result of multiple inheritance can always be achieved by using (simple) inheritance some object-oriented languages even don't allow its use. However, carefully used, under some conditions multiple inheritance provides an efficient and elegant way of formulating things.

3.4 Abstract Classes

With inheritance we are able to force a subclass to offer the same properties like their super classes. Consequently, objects of a subclass behave like objects of their super classes.

Sometimes it make sense to only describe the properties of a set of objects rollout knowing the actual behaviour before land. In our drawing program example, each object should provide a method to draw itself on the drawing area. However, the necessary steps to draw an objects depends on its represented shape. For example, the drawing routine of a circle is different from the drawing routine of a rectangle.

Let's call the drawing nettled print(). To force every drawable object to include SLICK method, we define a class Drawable object from wilily every other class in our example inherits general properties of drawable objects:

```
abstract class Drawable object {
  attributes:

  methods:
  setX (int newX)
  getX 0
  setY (in newY)
  getY 0
  print o)
```

We introduce the new keyword abstract here. It is used to express the fact that derived classes must redefine" the properties to fulfill the desired functionality. THUS from the abstract class' point of view, the properties are only specified but not fully classified. The full definition including the semantics of the properties must be provided by derived classes.

Now, every class in our drawing program example inherits properties from the general drawable object class. Therefore, class foiml changes to:

```
class Point inherits from Drawable object { attributes:
  int x, y

  methods:
  setX(int newX)
  getX ( )
  setY (int newY)
  getY ( )
  print() /* Redefine for Point */
```

We .re now able to force every drawable object to have a method called print which should provide functionality to draw the object within the driven;., area. The superclass of all draw able objects, class Drainable object, does not provide any functionality for drawing itself. It is not intended to create objects from it. This class rather specifies properties which must be defined

by every derived class. We refer to this special type of classes as abstract classes:

Definition (Abstract Class) .- 1 class A is called abstract class if it is only used to create each object. Derived classes must define the properties of A.

Abstract classes allow us to structure our inheritance graph. However, we actually don't want to create objects from them: we only want to express common characteristics of a set of classes.

4.0 Conclusion

Classes are related one way or the other. You have learnt about different types of class relationships in this unit. The concept of inheritance and multiple inheritance among related classes have also been discussed in this unit.

5.0 Summary

In this unit we have used a pseudo language to describe the concept of oriented language like C++ or Java which will be discussed later in the course.

6.0 Tutor Marked Assignment

- Q1. Describe with diagrams three types of relationships
- Q2. Define the following terms
 - a. Inheritance
 - b. Multiple Inheritance
 - c. Abstract Class

References And Other Resources

1. Peter Miller - Introduction to Object Oriented Programming in C++
Globewide Network Academy (GNA)

UNIT 4: A SURVEY OF PROGRAMMING TECHNIQUES

TABLE OF CONTENTS

1.0	Introduction
2.0	Objectives
3.1	Unstructured Programming
3.2	Procedural Programming
3.3	Modular Programming
3.4	An Example With Data Structures
3.5	Modular Programming Problems
3.6	Object-Oriented Programming
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	References & Other Resources

• Introduction

This unit is a short survey of programming techniques. We use a simple example to illustrate the particular properties and to point out their main ideas and problems. Roughly speaking, we can distinguish the following learning curve of someone who learns to program:

- Unstructured programming,
- Procedural programming,
- Modular programming and
- Object-oriented programming.

This unit is organized as follows. Sections 3.1 to 3.3 briefly describe the first three programming techniques. Subsequently, we present a simple example of how modular programming can be used to implement a singly linked list module (section 3.4). Using this we state a few problems with this kind of technique in section 3.5. Finally, section 3.6 describes the fourth programming technique.

20 Objectives

By the end of this unit, you should be able to

- Explain unstructured programming, procedural programming, modular programming and object-oriented programming
- Identify some problems with modular programming technique

Unstructured Programming

Usually, people start learning programming by writing's small and simple programs consisting only of one main program, Here "main program" stands

UNIT 4: A SURVEY OF PROGRAMMING TECHNIQUES

TABLE OF CONTENTS

1.0	Introduction
2.0	Objectives
3.1	Unstructured Programming
3.2	Procedural Programming
3.3	Modular Programming
3.4	An Example With Data Structures
3.5	Modular Programming Problems
3.6	Object-Oriented Programming
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	References & Other Resources

1.0 Introduction

This unit is a short survey of programming techniques. We use a simple example to illustrate the particular properties and to point out their main ideas and problems. Roughly speaking, we can distinguish* the following learning curve of someone who learns to program:

- Unstructured programming,
- Procedural programming,
- Modular programming and
- Object-oriented programming.

This unit is organized as follows. Sections 3.1 to 3.3 briefly describe the first three programming techniques. Subsequently, we present a simple example of how modular programming can be used to implement a singly linked list module (section 3.4). Using this we state a few problems with this kind of technique in section 3.5. Finally, section 3.6 describes the fourth programming technique.

20 Objectives

By the end of this unit, you should be able to

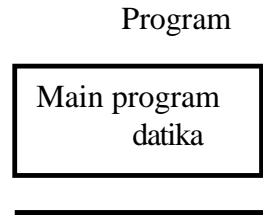
- Explain unstructured programming, procedural programming, modular programming and object-oriented programming
- Identify some problems with modular programming technique

(34) Unstructured Programming

Usually, people start learning programming by writing's small and simple programs consisting only of one main program. Here "main program" stands

for --a sequence of commands or instructress which modify data which is global throughout the whole program. We can illustrate this as shown in Fig. 3.1.

Figure 3.1: Unstructured programming. The main program directly operates on global data.



As you should know, this programming technique provide tremendous disadvantages once the program gets sufficiently large. For example, if the same statement sequence is needed at different locations within the program, the sequence must be copied. This has led to the idea to extract these sequences, name them and offering a technique to call and return from these procedures.

3.2

Procedural Programming

With procedural programming you are able to combine returning sequences of a procedure. After the last sequence is processed, flow of control proceeds right after the position where the call was made (Fig. 3.2).

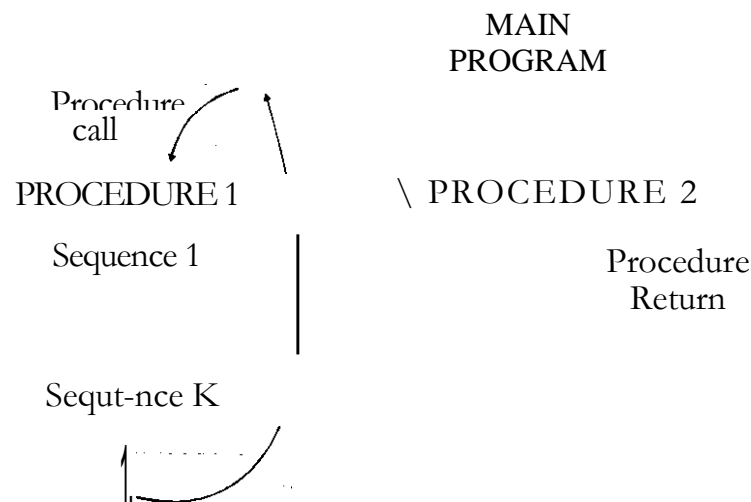


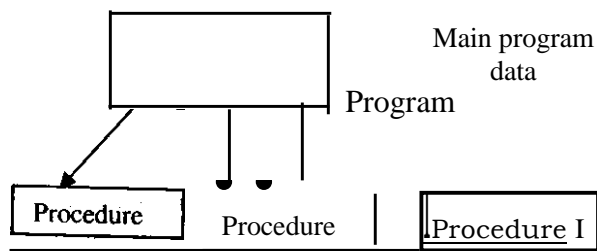
Figure 3.2: Execution of procedures. After processing flow of controls proceed where the call was made.

With introducing **pharmacies** as well as procedures (subprocedures) programs can now be written more structured and error free. For example, if a procedure is correct, every time it is used it produces correct results.

Consequently, in cases of errors you can narrow your search to those places which are not proven to be correct.

Now a program can be viewed as a sequence of procedure calls. The main program is responsible to pass data to the individual calls, the data is processed by the procedures and, once the program has finished, the resulting data is presented. Thus, the flow of data can be illustrated as a hierarchical graph, a tree, as shown in Fig. 3.3 for a program with no subprocedures.

Figure 3.3: Procedural programming. The main program coordinates calls to procedures and hands over appropriate data as parameters.



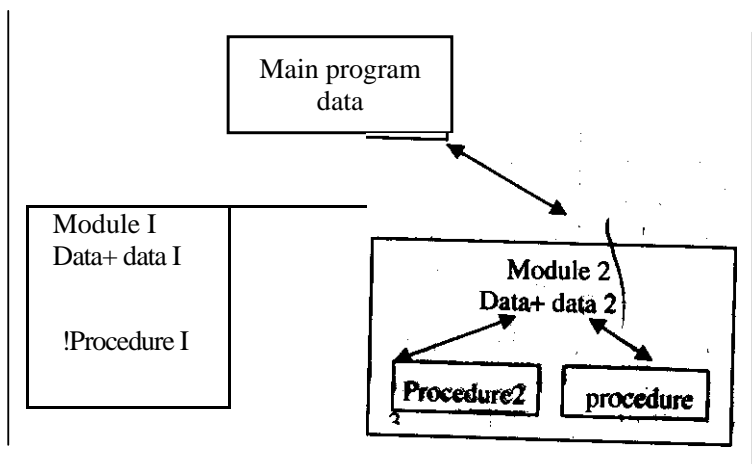
To sum up: Now we leave a single program which is divided into small pieces in other programs, they must be separately available. For that reason, modular programming allows grouping of procedures into modules.

3.3

Modular Programming

With modular programming, procedures of a common functionality 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 (Fig. 3.4).

Figure 3.4: Modular programming. The main program coordinates and calls to procedures in separate modules and hands over appropriate data as parameters.



Each module can have its own data. This allows each module to manage an internal slate 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.

3.4 An Example with Data Structures

Programs use data structures to store data. Several data structures exist, for example lists, trees, arrays, sets, bags or queues to name a few. Each of these data structures can be characterized by their instructure and their access methods.

3.4.1 Handling Single Lists

You all know singly linked lists which use a very simple structure, consisting of elements which are strung together, as shown in Fig. 3.5).

Figure 3.5: Structure of a singly linked list.



Singly linked lists just provides access methods to uppencl a new element to their end and to delete the element at the front. Complex data structures might use already existing ones. For example a queue can be structured like a singly linked list. However, queues provide access methods to put a data element at the end and to get the first data element (first in first-out (FIFO) behaviours).

We will now present an example which we use to present some design concepts. Since this example is just used to illustrate these concepts and problems it is neither complete nor optimal. Suppose you want to program a list in a modular programming language such as C or Modula-2. As you believe that lists are a common data structure, you decide to implement it in a separate moldier. Typically, this requires you to write two Ides: the interface definition and the implementation file. Within this unit we will use a very simple pseudo code which you should understand immediately. Let's assume, that comments are enclosed in `/* */`. Our interface definition might then look similar to that below:

* Interface definition for a module which implements
 * a singly linked list for storing data of any type. '

```
MODULE Singly-Linked-List-1
BOOT, list_initialize();
BOOT, list_append(ANY data);
BOOT, list_delete();
list_end();
ANY list_getFirst();
ANY list_getNext();
BOOT, list_isEmpty();
END Singly-Linked-List-I
```

Interface definitions just describe what is available and not how it is made available. YOU hide the information of the implementation in the implementation tile. This is a fundamental principle in software engineering, so let's repeat it: YOU hide information of the actual implementation (information hiding). This enables you to change the implementation, for example to use a faster but more memory consuming algorithm for storing elements without the need to change other modules of your program: The calls to provided procedures remain the same.

The idea of this interface is as follows: Before using the list one has to call list_initialize to initialize variables local to the module. The following two procedures implement the mentioned access methods append and delete. The delete procedure needs a more detailed discussion. Function list_delete takes one argument data of arbitrary type. This is necessary since YOU wish to use your list in several different environments, hence, the type of the data elements to be stored in the list is not known beforehand. Consequently, you have to use a special type ANY which allows to assign data of any type to it. The third procedure list_end() needs to be called when the program terminates to enable the module to clean up its internally used variables. For example you might want to release allocated memory.

With the next two procedures list_getFirst() and list_getNext() a simple mechanism to traverse through the list is offered. Traversing can be done using the following in-loop:

```
ANY data;
```



```

* Interface definition for a module which implements
* a singly linked list for storing data of any type.
*/

```

```

MODULE Singly-Linked-List-1
BOOT, list_initialize();
BOOT, list_append(ANY data);
BOOT, list_delete();
list_end();
ANY list_getFirst();
ANY list_getNext();
BOOT, list_isEmpty();
END Singly-Linked-List-1

```

Interface definitions just describe what is available and not how it is made available. YOU hide the information of the implementation in the implementation tile. This is a fundamental principle in software engineering, so let's repeat it: YOU hide information of the actual implementation (information hiding). This enables you to change the implementation, for example to use a faster but more memory consuming algorithm for storing elements without the need to change other modules of your program: The calls to provided procedures remain the same.

The idea of this interface is as follows: Before using the list one has to call list_initialize to initialize variables local to the module. The following two procedures implement the mentioned access methods append and delete. The delete procedure needs a more detailed discussion. Function list_remove takes one argument data of arbitrary type. This is necessary since YOU wish to use your list in several different environments, hence, the type of the data elements to be stored in the list is not known beforehand. Consequently, you have to use a special type ANY which allows to assign data of any type to it. The third procedure list_end() needs to be called when the program terminates to enable the module to clean up its internally used variables. For example you might want to release allocated memory.

With the next two procedures list_getFirst() and list_getNext() a simple mechanism to traverse through the list is offered. Traversing can be done using the following loop:

```

ANY data;

```

```

data <- list getFirst();
WHILE data IS VALID DO
Do Something(data);
data <- list_getNext();

END

```

Now you have a list module which allows you to use a list with any type of data elements. But what, if you need more than one list in one of your programs?

3.-1.2 Handling Multiple Lists

YOU decide to redesign your list module to be able to manage more than one list. YOU therefore create a new interface description which now includes a definition for a list handle. This handle is used in every provided procedure to uniquely identify the list in question. Your interface definition file of your new list module looks like this:

```

* A list module for more than one list.
*/

```

```

MODULE Singly-Linked-List-2

```

```

DECLARE TYPE list handle t;

```

```

list_handle_t list-create();

```

```

list_destroy(list handle _t this);

```

```

  BOOL      list_append(list handle _t this, ANY data);

```

```

  ANY      list get First(list handle t this);

```

```

  ANY      list_getNext(list handle _t this);

```

```

  1-300L    list is Empty(list handle t this);

```

```

END      Singly-Linked-List-2;

```

renresents your list handle. We do not specify, how this handle is acually represeered or even implemented. 'Thu liso hick the implementation details of this t- ie in your iniplernemation file. Note

the difference to the previous version where you just li ICIC functions or procedures, respectively. Now you also hide information for an user defined data type called list-handle- t.

You use list create () to obtain a handle to a new thus empty list. Every other procedure now contains the special parameter This which just identifies the list in question. All procedures now operate on this handle rather than a module global list.

Now you might say, that you can create list objects. Each such object can be defined to operate on this handle.

3.5

Modular Programming Problems

The previous section shows, that you already program with some object-oriented concepts in mind. However, the example implies some problems which we will outline now.

3.5.1 Explicit Creation and Destruction

In the example every time you want to use a list, you explicitly have to declare a handle and perform a call to list create to obtain a valid one. After the use of the list you must explicitly call list destroy with the handle of the list you want to be destroyed. If you want to use a list within a procedure, say, food you use the following code frame:

```
PROCEDURE fool) BEGIN
list-handle t rnyList;
myList <- list create();

/* Do something with myList
* ..
list destroy(myList);

END
```

Let's compare the list with other data types, for example an integer. Integers are declared within a particular scope (for example within a procedure). Once you've defined them, you can use them. Once you leave the scope (for example the procedure where the integer was defined) the integer is lost. It is automatically created and destroyed. Some compilers even initialize newly created integers to a specific value, typically 0 (zero).

Where is the difference to list "objects"? The lifetime of a list is also defined by its scope, hence, it must be created once the scope is entered and destroyed once it is left. On creation time a list should be similar to the definition of an integer. A code frame for this would look like this:

```
PROCEDURE fool) BEGIN

list handle t myList; /* List is created and initialized */

/* Do something with the myList */

END /* myList is destroyed */
```

The advantage is, that now the compiler takes care of calling initialization and termination procedures as appropriate. For example, this ensures that the list is correctly deleted, returning resources to the program.

3.5.2 Decoupled Data and Operations

Decoupling of data and operations leads usually to a structure based on the operations rather than the data: Modules group common operations (such as those list ... () operations) together. You then use these operations by providing explicitly the data to them on which they should operate. The resulting module structure is therefore oriented on the operations rather than the actual data. One could say that the defined operations specify the data to be used.

In object-orientation, structure is organized by the data. You choose the data representations which best fit your requirements. Consequently, your programs get structured by the data rather than operations. Thus, it is exactly the other way around: Data specifies valid operations. Now modules group data representations together.

3.5.3 Missing Type Safety

In our list example we have to use the special type ANY to allow the list to carry any data we like. This implies, that the compiler cannot guarantee for type safety. Consider the following example which the compiler cannot check for correctness:

```
PROCEDURE fool) BEGIN
```

```

Some Data Type data;
SomeOtherType data2;
List_handle t myList;

myList <- list_create();
list_append(myList, data);
list_append(myList, data2); /* Oops */

```

END

It is in your responsibility to ensure that your list is used consistently. A element. However, this implies more overhead and does not prevent you from knowing what you are doing.

What we would like to have is a mechanism which allows us to specify on always the same, whether we store apples, numbers, cars or even lists. Therefore it would be nice to declare a new list with something like:

```

list_handle_KApple> list; /* a list of apples */
list handle t<Car> list2; /* a list of cars */

```

The corresponding list routines should then automatically return the correct data types. The compiler should be able to check for type consistency.

3.5.4 Strategies and Representation

The list example implies operations to traverse through the list. Typically a cursor is used for that purpose which points to the current element. This implies a traversing strategy which defines the order in which the elements of the data structure are to be visited.

For a simple data structure like the singly linked list one can think of only one traversing strategy. Starting with the leftmost element one successively visits the right neighbors until one reaches the last element. However, more complex data structures such as trees can be traversed using different strategies. Even worse, sometimes traversing strategies depend on the particular context in which a data structure is used. Consequently, it makes sense to separate the actual representation or shape of the data structure from its traversing strategy.

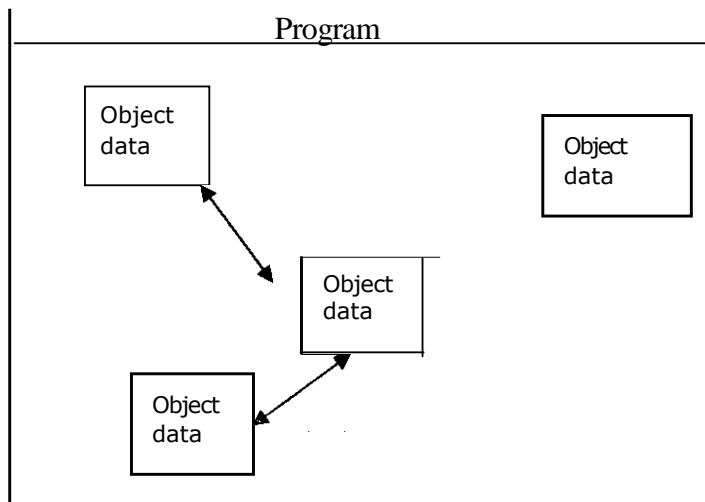
What we have shown with the traversing strategy applies to other strategies as well. For example insertion might be done such that an order over the elements is achieved or not.

3.6

Object-Oriented Programming

Object-oriented programming solves some of the problems just mentioned. In contrast to the other techniques, we now have a web of interacting objects, each keeping its own state (Fig. 3.6).

Figure 3.6: Object-oriented programming. Objects of the program interact by sending messages to each other.



Consider the multiple lists example again. The problem here with modular use the procedures of the module to modify each of your handles.

In contrast to that, in object-oriented programming we would have as many list objects as needed. Instead of calling a procedure which we must provide with the correct list handle, we would directly send a message to the list object in question. Roughly speaking, each object implements its own module allowing for example many lists to coexist.

Each object is responsible to initialize and destroy itself correctly. Consequently, there is no longer the need to explicitly call allocation or termination procedure.

4.0 Conclusion

You might ask: So what isn't object-oriented technique. Just a fancier modular programming technique? You were right, if this would be the case.

all about object-orientation. Fortunately. It is not. Front The previous units additional features of object-orientation were discussed which makes object - oriented programming a new programming technique.

5.0 Summary

In this unit you have had a comparison of pro-ramming techniques and **I** am sure you now know why object oriented programming technique stands shoulder high

6.0 Tutor Marked Assignment

Qi. Describe the four programming techniques discussed in this unit.

7.0 References and Other Resources

Peter Muller - introduction to Object Oriented Programming in C++ Globe wide
Network Academy (GNA)

UNIT 5: SOFTWARE ENGINEERING

TABLE OF CONTENTS

1.0	Introduction
2M	Objectives
3.1	Software Product, Components And Characteristics
3.2	Software Engineering Concepts

- 33 Documentation Of The Software Product ,
- 3.4 Software Process And Models
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor Marked Assignment
- 7.0 References & Other Resources

1.0 INTRODUCTION

This unit focuses on Software Product, Component and Characteristics. This unit discusses in details the evolution process of Software Engineering life cycle. A sample waterfall model is also discussed in this unit. Software Engineering concepts and its phases are also included in this unit. The documentation part is also included in this unit, Software Documentation is a continuous and parallel activity in development process.

2.0 OBJECTIVES

After going through this unit, you should be able to:

- Define Software Product Component and characteristics.
- Explain what is documentation of Software Product.
- Describe what is Software Process and what is Software Life Cycle.
- Describe a Generic View of Software Engineering.

3.1 SOFTWARE PRODUCT, COMPONENTS AND CHARACTERISTICS

We are in an information age, one in which the management of the information resource of organization is of vital importance. Business information systems are systems that use these resources to convert data into information in order to improve productivity. Business information systems usually are composed smaller systems, called subsystems. Computer hardware and software are important resources that support information systems and subsystems.

Systems analysis is a general terms that refers to an orderly, structured process life cycle-methodology. Four phases - study, design, development, and operation - make up the life -cycle of computer-related business systems. A systems analyst is a person who performs systems analysis during any or all of the life-cycle phases. The systems analyst not only analyzes information system problems, but also synthesizes new systems to solve these problems.

The four information eras are: the Early Era (1940-1955), the Growing Era (1955- 1965), the Refining Era (1965-1980), and the Maturing Era (1980-). The Early Era concentrated on hardware, and human machine communication was very difficult. The growing Era improved this communication through the introduction of English- like programming languages; however, techniques for managing Computer related projects were lacking. During the Refining Era, explosive growth occurred in the development of large (midi and maxi) and small (micro and mini) computer systems and in their applications. Developments in microelectronics technology contributed significantly to this growth. Throughout most of the Refining Era, in spite of a proliferation of applications, difficulties were encountered in using computer to solve business problems. However, toward the end of this era, a structured system analysis process-called the life-cycle methodology - came into increasing use as a means of developing usable business information systems. Structured techniques for the analysis, design, and development of computer-related information systems are now enhanced in the maturing Era. These techniques are used to develop information systems in applications areas such as distributed, data processing, the animated office, and management-decision support. This is an era in which information is acknowledged as an important corporate resource. The systems analyst assumes an important role in managing the information resources of the corporation.

The computer-based business system also contains hardware components; however, its most significant characteristics is a software end-product. Software may be defined as a collection of programs or routines that facilitates the use of a computer. This definition includes operating systems, which facilitate the general use of computers, and application programs, which are written to solve specific problems. The latter is the end-product associated with a computer-based information system. Software, in contrast with hardware, does not possess attributes that can readily be observed and measured from concept to end product. The software end-product is information. Although it may be stored or printed on a physical medium, ' such as a magnetic disk, a reel of tape, or a sheet of paper, information is transient and fragile compared with hardware. •

- Many of the past difficulties in developing effective computer-based business
- systems stemmed not only from belated efforts to apply management controls, but also from failure to recognize that techniques applicable to the development of hardware end-products could not be applied without modification to the development of software .ead-morla However, as a result of experience gained from large government and commercial software projects in the latter part of the 1960s and throughout the 1970s, ***e concept**

of life-cycle management was adapted to fit the development of computer based business systems.

The key to modifying the life-cycle concept for the management of software projects was the recognition that, although supporting documentation accompanies a physical product throughout its development, documentation is the software product.

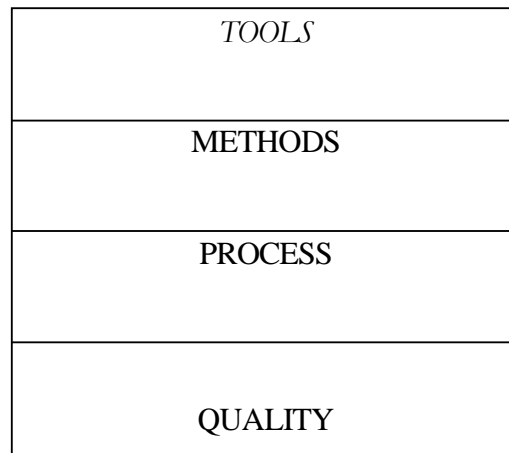


Figure 5.1 : Software engineering layers

3.2 Software Engineering Phases

Life-Cycle Phases and the Life-Cycle Manager:

The life-cycle methodology for developing complex systems is modular, top-down procedure. In the study phases, modules that describe the major functions to be performed by the system are developed. The procedure is called top-down because in successive phases the major modules are expanded into additional, increasingly detailed, modules. Powerful graphic tools have been developed to structure the top-down design and development phase activities in detail. For the present, we can summarize the principal tasks associated with each of the phases of the life cycle of a computer-based business system.

The life cycle of a computer-based system exhibits distinct phases. These are:

32.1 ' the Study Phase

This is the phases during which a problem is identified, alternate , system required to design the system. Task performed in the study phase are grossly analogous to

determining that a shelter from the elements is needed, and deciding that a two-bedroom house is a more appropriate shelter than a palace, a cave, or other possible selections.

3.2.2 The Design Phase

In this phase the detailed design of the system selected in the study phases is decided on in the ,study phase. In the case of a computer-based business system, design phase activities include the allocation of resources to equipment tasks, personnel tasks, and computer program task. In the design phase, the technical specifications are prepared for the performance of all allocated tasks.

3.2.3 The Development Phase

This is the phase in which the computer-based system is constructed from the development phase. All necessary procedure, manuals, software specifications, and other documentation are completed. The staff is trained, and the complete system is tested for operational readiness. This is analogous to the actual construction of our two-bedroom house from the plans prepared in its design phase.

3.2.4 The Operation Phase

In this phase, the new system is installed or there is a changeover from the old system to the new system. The new system is operated and maintained. Its performance is reviewed, and changes in it are managed, the operation phase is analogous to moving into and living in the house that we have built. If we have performed the activities of the preceding phases adequately, the roof should not leak.

All of the activities associated with each life-cycle phase must be performed, managed, and documented. Hence, we now define systems analysis as the performance, management, and documentation of the activities related to the four life-cycle phases of a computer based business system. **SimiAarly, we ;tow can identify the system analyst as the individual who is responsible for the performance of systems analysis for at or,a portion of the phases of**

the life cycle of a business system. The analyst is, in effect, a lifecycle manager.

3.3 DOCUMENTATION OF THE SOFTWARE PRODUCT

The accumulation of documentation parallels the life-cycle performance and management review activities. Documentation is not a task accomplished as a "wind up" activity; rather, it is continuous and cumulative. The most essential documents are called baseline specification (that is, specifications to which change can be referred). There are three baseline specifications:

1. Performance specification: It is completed at the end of the study phase, and describing in the language of the user exactly what the system is to do. It is a "design to" specification.
2. Design specification: It is completed at the end of the design phase, and describing in the language of the programmer (and others employed in actually constructing the system) how to develop the system. It is a "but to".
3. System specification: It is completed at the end of the development phase and containing all of the critical system documentation. It is the basis for all manuals and procedures, and it is an "as built" specification.

The design specification evolves from the performance specification, and the system specification evolves from the design specification. Since these documents are the only measurable evidence that progress is being made toward the creation of a useful software end-product, it is not possible to manage the lifecycle process without them. Thus, documentation is not only the "visible" software end-product, but also the key to the successful management of the life cycle of computer-based business systems.

Exercises 1.1

Which of the following is not an example of program documentation?

- (a) Source code
- (b) Object code
- (c) Specification
- (d) Identifier Names

Answer: (b)

3.4 Software Development Cycle

"tin" *4";;I:144:1:14*;;1144t)"

34.1 Software Development Cycle

From the inception of an idea for a software system, until it is implemented and delivered to a customer, and even after that, the system undergoes gradual development and evolution. The software is said to have a life cycle composed of several phases. Most of these phases result in the development of either a part of the system or something associated with the system, such as a test plan or user manual. In the traditional life cycle model, called the "waterfall model," each phases has well-defined starting "and ending points, with clearly identifiable deliverables to the next phase.

A sample waterfall life cycle model comprises .the phases, similar to described in next sections.

3:¹.2 **Requirements analysis and specification**

Requirements analysis is usually the first phase of large-scale software development project. It is undertaken after a feasibility study has been performed to define the precise costs and benefits of a software system. The purpose of this phase is to identify and document the exact requirements for the system. The customer, the developer, a marketing organization or any combination of the three may perform such study. In cases where the requirements are not clear e.g., for a system that has never been defined, more interaction is required between the user and the developer. The requirements at this stage are in end-user terms. Various software engineering methodologies advocate that this phase must also produce user manuals and system test plans.

3.4.3 Design and specification

Once the requirements for a system have been documented, software engineers design a software system to meet them. This phase is , sometime split into two sub-phases: architectural or high-level design and detailed design. High-level design deals with overall module structure and organization, rather than the details of the modules. The high level design is refined by designing each module in detail (detailed design). Separating the requirements analysis phase from the design' phase is instance of a fundamental "what/how" dichotomy that we encounter quite often in Computer science. The general principle involves makings a clear distinction between what the problem is and how to solve the problem. In this case, the **requirement phase** attempts to specify what the problem is. There are usually many ways that the requirements may be met, including some solutions that do

not involve the use of computers at all. The purpose of the design phase is to specify a particular software system that will meet the stated requirements. Again there are usually many ways to build the specified system. In the coding phase, which follows the design phase, a particular system is coded to meet the design specification.

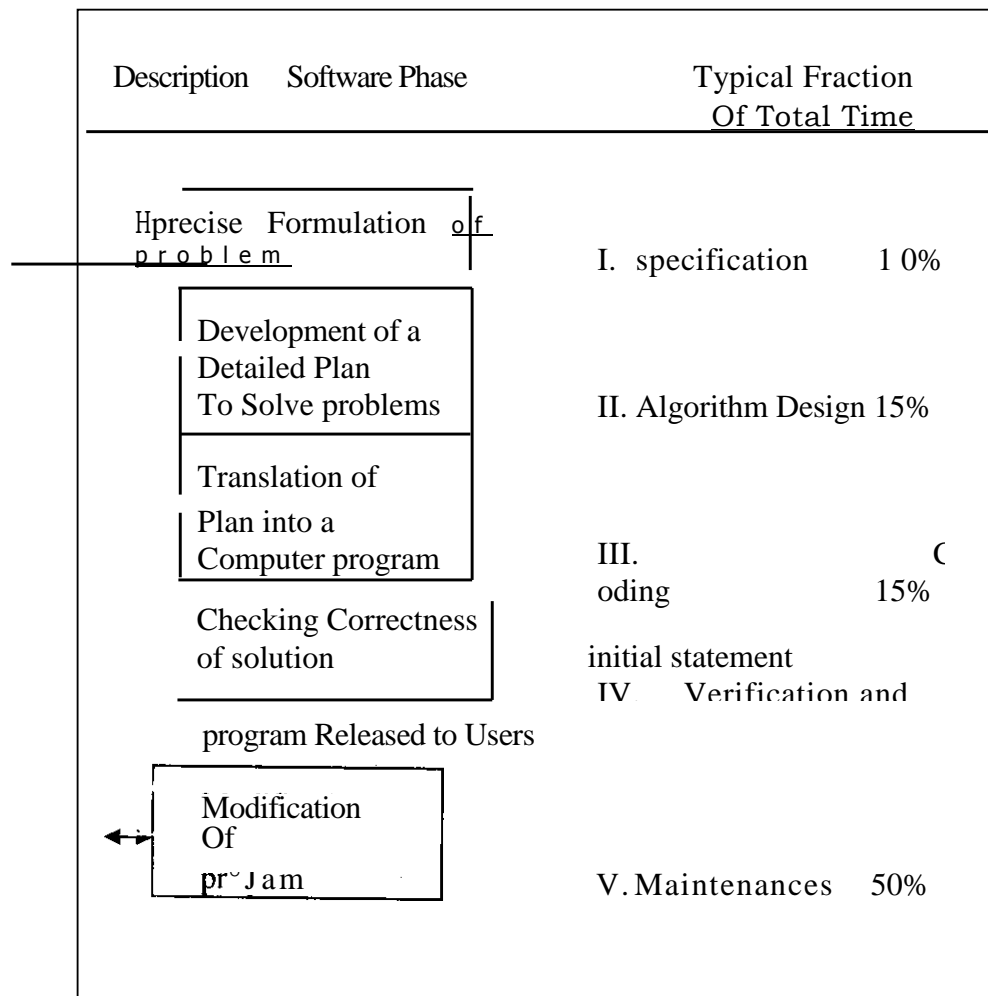


Figure 5.2: Software Life Cycle

3.4.4 Coding and module testing

This is the phase that produces the actual code that will be delivered to the customer as the running system. The other phases of the life cycle may also develop code, such as prototypes, tests, and test drivers, but these are for use by the developer. Individual modules developed in this phase are also tested before being delivered to the next phase.

- Integration and system testing: All the modules that have been developed before and tested individually are put together (integrated) in this phase and tested as a whole system.
- Delivery and maintenance: Once the system passes the entire test, it is delivered to the customer and enters the maintenance phase. Any modifications made to the system after initial delivery are usually attributed to this phase.

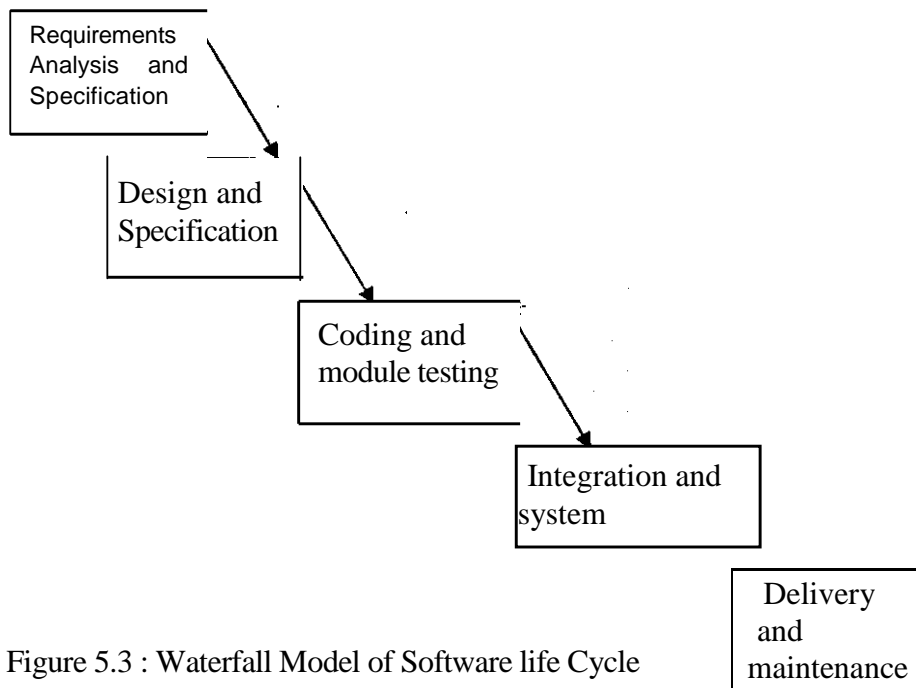


Figure 5.3 : Waterfall Model of Software life Cycle

Exercise 1.2

Which of the following is (are) among the legitimate purposes of software documentation?

- I To assist in maintaining and modification.
- II To describe the capabilities of the program.
- III To provide the user with instructions..

- a) I only
- b) II and III only
- c) III only
- d) I, II and III

Answer: (d)

40 Conclusion

As presented above, the phases give a partial, simplified view of the conventional waterfall software life cycle. The process may be decomposed into a different set of phases, with different names, different purpose, and different granularity. Entirely different life cycle schemes may even be proposed, not based on a strictly phased waterfall development. For example, it is clear that if any tests uncover defects in the system, we have to go back at least to the coding phase and perhaps to the design phase to correct some mistakes. In general, any phase may uncover problems in previous phases this will necessitate going back to the previous phases and redoing some earlier work. For example, if the system design, phase uncovers inconsistencies or ambiguities in the system requirements, the requirements analysis phase must be revisited to determine what requirements were really intended.

Another simplification in the above presentation is that it assumes that a phase is completed before the next one begins. In practice, it is often expedient to start a phase before a previous one is finished. This may happen, for example, if some data necessary for the completion of the requirement phase will not be available for some time or it might be necessary because the people ready to start the next phases are available and have nothing else to do

Most books on software engineering are organized according to the traditional software life cycle model, each, section or chapter being devoted to one phase. Once mastered, the software engineer can apply these principles in all phases of software development, and also in life cycle models that are not based on phased development, as discussed above. Indeed, research and experience over the past decade have shown that there is a variety of life cycle models and that no single one is appropriate for all software systems.

5.0 Summary

In this unit, you have learnt the techniques applicable to the development of software products. The phases of software engineering and the use of baseline specifications in the software process were also discussed. The next unit focuses on Software Performance.

6.0 Tutor Marked Assignment

- 1 The life cycle of a computer-based system exhibits distinct phases.
Discuss
- 2 Describe the phases of a Software LA Cycle

7.0 REFERENCES AND OTHER RESOURCES

Management of Information Systems, Unit 2, Indira
Gandhi National Open University

Software Engineering - A Practitionees, Approach by ROGER S.
PRESSMAN: McGraw Hill International Edition.

UNIT 6: SOFTWARE QUALITY CONCEPTS

TABLE OF CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.1 Important Qualities Of Software Product And Process
- 3.2 Principles Of Software Engineering
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor Marked Assignment
- 7.0 References And Other Resources

1.0 Introduction

The goal of any engineering activity is to build a product. For example, the aerospace engineer builds an airplane. The product of software engineer is a software system. But the difference between software product and other product is that it is modifiable. This quality makes software quite different from other products such as cars. In this unit, we will first examine the important software qualities and then discuss software engineering principles.

2.0 Objectives

After going through this unit, you will be able to:

- List various qualities of software product;
- Discuss various qualities of software product
- Explain principles of software engineering

3.1 Important Qualities Of Software Product And Process

There are many important qualities Of software products. Some of these qualities are applicable both to product and to the process used to produce the product.

The user wants the software product to be reliable and user-friendly. The designer of the software wants it to be maintainable, portable and extensible. In this unit, we will consider all these qualities.

3.1.1 Correctness

A program is functionally correct if it behaves according to the . specification of function it should provides (called functional icluirements specifications). It is common simply to use the term

term specification implies functional requirement specifications. We will follow this convention when the context is clear.

The definition of correctness assumes that a specification of the system is available and that it is possible to determine unambiguously whether or not a program meets the specifications. With most current software systems, no such specification exists. If a specification does exist, it is usually written in an informal style using natural language. Such a specification is likely to contain many ambiguities. Regardless of these difficulties with current specifications, however, the definition of correctness is useful. Clearly, correctness is a desirable property for software systems.

Correctness is a mathematical property that establishes the equivalence between the software and its specification. Obviously, we can be more systematic and precise in assessing correctness depending on how rigorous we are in specifying functional requirements. Correctness can be assessed through a variety of functional requirements. Correctness can be assessed through a variety of methods, some stressing an experimental approach (e.g. testing), others stressing an analytic approach (e.g. formal verification of correctness). Correctness can also be enhanced by using appropriate tools such as high-level languages, particularly those supporting extensive static analysis. Likewise, it can be improved by using standard algorithms or using libraries of standard modules, rather than inventing new ones.

3.1.2 Reliability

Informally, software is reliable if the user can depend on it. The specialized literature on software reliability defines reliability in terms of statistical behaviour: the probability that the software will operate as expected over a specified time interval.

Correctness is an absolute quality; any deviation from the requirements makes the systems incorrect, regardless of how minor or serious are the consequences of the deviation. The notion of reliability is on the other hand, releases along with a list of known bugs. User of software takes it for granted that Release I of a product is buggy. This is one of the most striking symptoms of the immaturity of the software engineering field as an engineering discipline.

In classic engineering disciplines, a product is not released if it has bugs. You do not expect to take delivery of an automobile, along with a list of shortcomings or a bridge with a warning not to use the railing. Design errors are extremely rare and worthy of news headlines. A bridge that collapses may even cause the designers to be prosecuted in court.

On the contrary, software design errors are generally treated as unavoidable. Far from being surprised with the occurrence of software errors, we expect them. Whereas with all other products the customer receives a guarantee of reliability, with software we get a disclaimer that the software manufacturer is not responsible for any damages due to product errors. Software engineering can truly be called an engineering discipline only when we can achieve software reliability comparable to the reliability of other products.

3.1.3 Robustness

A program is robust if it behaves reasonably, even in circumstances that were not anticipated in the requirements specification - for example, when it encounters incorrect input data or some hardware malfunction. A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command would not be robust. It might be correct, though, if the requirements specification does not state what the action should be upon entry of an incorrect command. Obviously, robustness is a difficult quality to define; after all, if we could state precisely what we should do to make an application robust, we would be able to specify its reasonable behaviour completely. Thus, robustness would become equivalent to correctness.

The amount of code devoted to robustness depends on the application area. For example, a system written to be used by novice computer users must be more prepared to deal with ill-formatted input than an embedded system that receives its input from a sensor - although, if the embedded system is controlling the space shuttle or some life-critical devices, then extra robustness is advisable.

In conclusion, we can see that robustness and correctness are strongly related without a sharp dividing line between them. If we put a requirement in the specification, its accomplishment becomes an issue of correctness; if we leave it out of the specification, it may become an issue of robustness. The borderline between the two qualities is the specification of the system. Finally, reliability comes in because not

all incorrect behaviours signify equally serious problems; some incorrect behaviours may actually be tolerated.

Correctness, robustness, and reliability also apply to the software. A process is ; robust, for example, if it can accommodate changes, in the environment, such as a new release of the operating system. When transfer of half the employees to another location ;, is reliable if it consistently leads to the production of high-quality products. In many engineering disciplines, considerable research is devoted to the discovery, of reliable processes.

3.1.4 User **Friendliness**

A software system is user friendly if its human user find it to use. This definition ;, reflects the subjective ;nature of user fervidness. An application that is used by novice programmers qualifies a user friendly by virtue of ;different properties than an application that is used by expert programmers. For example, a novice user may appreciate verbose messages, while an experienced user grows to dates and ignore them. Similarly, a nonprogrammer may appreciate the use of menus, while a programmer may be more comfortable with typing a command.

The user interface is an important component of user friendliness. A software system that presents the novice user a set of one-letter commands is not user friendly. On the other hand, a keystrokes rather than

a fancy window interface through which he has to navigate to get to the command that's he knew all along he wanted to execute, will not be appropriate. ;

There is more to user friendliness, however, than the user interface. For example, an embedded software system does not have a human user interface. Instead, it interacts with hardware and perhaps other software systems. In this case, ;, the user friendliness is reflected in the ease with which the system can be configured and adapted to the hardware environment.

In general, the user friendliness of a system depends on the consistency of its user and operator interfaces. Clearly, however, the other qualities mentioned above such as correctness and performance - also affect user friendliness. A software system that produces wrong answers is not friendly, ; regardless of how fancy its user interface is.

Also, a software system that produces answers more slowly than the user requires is not friendly even if the answers are displayed in colour.

User friendliness is also discussed under the subject human factors. Human factors or human engineering plays a major role in many engineering disciplines. For example, automobile manufacturers devote significant effort to deciding the position of the various control knobs on the dashboard. Television manufacturers and microwave oven makers also try to make their products easy to use. User interface decisions in these classical engineering fields are made, not randomly by engineers, but only after extensive study of user needs and attitude by specialists in fields such as industrial design or psychology.

Increasingly, ease of use in many of these engineering disciplines is achieved. Television set, he or she can operate almost any other television set. The significant current research and development activity in the area of standard user interface for software systems will lead to more user-friendly systems in future.

3.1.5 Verifiability

A software system is verifiable if its properties can be verified easily. For example the correctness or the performance of a software system are properties we would be interested in verifying. Verification can be performed either by formal analysis methods or through testing. A common technique for improving verifiability is the qualities such as performance: correctness.

Modular design, disciplined coding practices, and the use of an appropriate programming language all contribute to verifiability.

Verifiability is usually an internal quality, although it sometimes becomes a measurable quality also. For example, in many security-critical applications the customer requires the verifiability of certain properties. The highest level of the security standard for a trusted computer system requires the verifiability of the operation system's kernel.

3.1.6 Maintainability

The term software maintenance is commonly used to refer to the modification that are made to a software system after its initial release.

Maintenance used to be viewed as merely ^{bug} fixing, and it was distressing to discover that so much effort was being open on fixing defects. Studies have shown, however, that the majority of time spent on maintenance is in fact spent on enhancing the product with features that were not in the original specifications or were stated incorrectly.

Maintenance is indeed not the proper word to use with software. First, as it is used today, the terms cover a wide range of activities, all having to do with modifying an existing piece of software in order to make an improvement. A term that perhaps captures the essences of this process better is software evolution. Second, I other engineering products, such as computer hardware or automobiles or washing machine, maintenance refers to the upkeep of the product in response' to the gradual deterioration of parts due to extended use of the product. Fox, example, transmissions are oiled and air filters are dusted and periodically changes. To use the word maintenance with software gives the wrong connotation because software does not wear out. Unfortunately, however, the term is used so widely that we will continue using it.

There is evidence that maintenance costs exceed 60% of the total cost of software. To analyze the factor that affect such costs, it is customary to divide software maintenance into three categories, corrective, adaptive and perceptive maintenance.

3.1.7 Reusability

Reusability is akin to resolvability. In product evolution, we modify a product to build a new version of that same product. Reusability appears to be more applicable to software components than to whole product. But it certainly seems possible to build products that are reusable

A good example of a reusable product is the UNIX shell. The UNIX shell is a command language to be used both interactively and in batch. The ability to start a new shell with a file containing a list of shell commands allows us to write program scripts - in the shell command language. We can view the program as a new product that uses the shell as a component. The UNIX environment in fact supports the reuse of any of its commands, as well as the shell, in building powerful utilities.

Scientific libraries are best known reusable components. Several large FORTRAN libraries have existed for many years. User can buy these

and use them to build their own products, without having to reinvent or recode well-known algorithms. Indeed, several companies are devoted to producing just such libraries.

Another successful example of reusable packages is the recent development of windowing system such as Microsoft Window, X ' windows or Motif, for the development of user interface.

Unfortunately, while reusability is clearly an important tool for reducing software production costs, example of software reuse in practice is rather rare.

Reusability is difficult to achieve a posteriori, therefore, one should strive for reusability when software components are developed. One of the more promising techniques is the use of object-oriented design, which can unify the qualities of resolvability and reusability.

So far. We have discussed reusability in the framework of reusable components, but the concept has boarder applicability it may occur at different levels and may affect both product and process. A simple and widely practiced type of reusability consists of the reuse of people, i.e. reusing their specific knowledge of an application domain of a development or target environment, and so on. This level is unsatisfactory, partially due to the turnover of software engineers: knowledge goes away with people and never becomes a permanent asset.

Another level of reuse may occur at the requirements level. When a new application is conceived, we may try to identify parts that are similar to parts used in a previous application. Thus, we may reuse parts of the previous requirement specification instead of developing an entirely new one.

As discussed above, further levels of reuse may occur when the application is designed, or even at the code level. In the latter case, we might be provide with expect claim that in the future new application by assembling together a set ready-made, off- the- shell component. Software companies will invest in the development of their own catalogues of reusable components so that the knowledge acquired in developing applications will not disappear as people leave, but will progressively accumulate in the catalogues. Other companies will invest their efforts in the production of generalized reusable components to be put on the marketplace for use other software producers.

Reusability applies to the software process as well. Indeed, the various software methodologies can be viewed as attempts to reuse the same process for building different products. The various life cycle models are also attempts at reusing higher level processes. Another example of reusability in a process is the replay approach to software maintenance. In this approach, the entire process is repeated when making a modification. That is, first the requirements are modified, and then the subsequent steps are followed as in initial product development.

Reusability is a key factor that characterizes the maturity of an industrial field. We see high degrees of reusability in such mature areas as the automobile industry and consumer electronic. For example, in the automobiles industry, the engines often reused from model to model. Moreover, a car is constructed by assembling together many components that are highly standardized and used across many models produced by the same industry. Finally, the manufacturing process is often reused. The low degree of reusability in software is a clear indication that the field must evolve to achieve the status of a well established discipline.

3.1.8 Portability

Software is portable if it can run in different environments. The term environment can refer to 'a hardware platform or a software environment such as a particular operating system. With the proliferation of different processors and operating systems, portability has become an important issue for software engineers.

More generally, portability refers to the ability to run a system on different hardware platforms. As the ratio of money spent on software versus hardware increases, portability gains more importance. Some software systems are inherently machine specific. For example, an operating system is written to control a specific computer, and a compiler produces code for a specific machine. Even in these cases, however, it is a possibility to achieve some level of portability. Again, UNIX is an example of an operating system that has been ported to many different hardware systems. Of course, the porting effort requires months of work. Still, we can call the software portable because writing the system from scratch for the new environment would require much effort than porting it.

For many applications, it is important to be portable across operating system. or, looked at another way the operauci., sy;lcm pnivkies tability acro, s hardware platforms.

3.1.9 Data Abstraction

Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details. Thus, abstraction is a special case of separation of concerns wherein we separate the concern of the important aspects from the concern of the unimportant details.

The programming language that we use are abstraction built on top of the hardware: they provide us with useful and powerful constructs so that we can write (most) programs ignoring such details as the number of bits that are used to represent numbers or the addressing mechanism. This helps us concentrate on the problem to solve rather than the way to instruct the machine on how to solve it. The programs we write are themselves abstractions. For example, a computerized payroll procedure is an abstraction over the manual procedure it replaces: it provides the essence of the manual procedure, not its exact details.

well defined procedure/ function in a single unit. This encapsulation forms a wall, which is intended to shield the data representation from computer users. There are two requirements for data abstraction facilities in programming language.

- (i) Data structure and operations as described is a single semantic unit.
- (ii) Data structure and internal representation of the data abstractions are not visible to the programmer. rather the programmer is presented with a well. defined procedural interface. Today most of the object oriented programming language support this feature.

3.1.0 Modularity

A complex system may be divided into similar pieces called modules. A system that is composed of modules is called modular. The main benefit of modularity is that it allows the principle of separation of concerns to be applied in two phases: when dealing with the details of each module in isolation (and ignoring details of other modules); and when

dealing with the overall characteristics of all modules and their relationship in order to integrate them into a coherent system. If the two phases are temporarily executed in the order mentioned, then we say that the system is designed bottom up; the converse denotes top-down design.

Modularity is an important property of most engineering processes and products. For example, in the automobiles industry, the construction of cars proceeds by assembling building blocks that are designed and built separately. Furthermore, parts are often reused from model to model, perhaps after minor changes. Most industrial processes are essentially modular, made out of work packages that are combined in simple ways (sequentially or overlapping) to achieve the desired result.

Modularity, however, is not only a desirable design principle, but permeates the whole of software production. In particular, there are three goals that modularity tries to achieve in practice: capacity of decomposing a complex system or composing it from existing modules, and of understanding the system in pieces.

The decomposability of a system is based on dividing the original problem top down into sub problems and then applying the decomposition to each sub problem recursively. This procedure reflects the well-known Latin motto divide (divide and conquer), which described the philosophy followed by the ancient Romans to dominate other nations: divide and isolate them first and conquer them individually.

The compensability of a system is based on starting bottom up from elementary components and proceeding to the finished system. As an example, a system for office automation may be designed by assembling together existing hardware components such as personal workstation, a network, and peripherals; system software such as the operating system; and productivity tools such as document processors, data bases and spreadsheets. A car is another obvious example of a system that is built by assembling components. Consider first the main subsystems in a car system, each of them, in turn, is made out of standard parts, for example, the battery, fuses, cables, etc. from the electrical system. When something goes wrong, defective components may be replaced by new ones.

Ideally, in software production we would like to be able to assemble new applications by taking modules from a library and combining

them to form the required product. Such modules should be designed with the express goal of being reusable. By using reusable components, we may speed up both the initial system construction and its fine-tuning. For example, it would be possible to replace a component by another that performs the same function but differs in computational resource requirements.

The capability of understanding each part of a system separately aids in modifying a system. The evolutionary nature of software is such that the software engineer is often required to go back to previous work to modify it. If the entire system can be understood only entirely, modifications are likely to be difficult to apply, and the result unreliable. When the need for repair arises, proper modularity helps confine the searches for the source of malfunction to single components.

To achieve modular compensability, decomposability, and understanding, modules must have high cohesion and low coupling.

A module has high cohesion if all elements are related strongly.

Elements of a module (e.g. statement, procedures, and declarations) are grouped together in the same modules for a logical reason, not just by chance; they co-operate in the same modules (e. g. modules A call a routine provided by module B or

accesses a variable declared by Module B) if two modules in a system to exhibit low coupling, because if two modules are highly coupled, it will be difficult to analyze, understand, modify, test, or reuse them separately.

Module structures with high cohesion and low coupling allow us to see modules as black boxes when the overall structure of a system is described and then deal with each module separately when the module's functionality is described or analyzed. This is just another example of the principle of separation of concerns.

3.2 Principles Of Software Engineering

Engineering disciplines have principles based on the laws of physics, biology, chemistry or mathematics. Principles are rules to live by, they represent the collected wisdom of many dozens of people who have learned through experience.

Because the product of software engineering is to physical, physical laws do not form a suitable foundation. Instead, software engineering has had to evolve its principles based solely on observation of thousands of projects. The following are probably the more important ones. A customer will not tolerate a poor-quality product, regardless of how you define quality. Quality must be quantified and mechanisms 'put into to deliver a product on time, even though its quality is poor, but this is correct only in the short term, it is suicide in the middle and long term. There is no trade-off to be made here. The first requirement must be quality.

However, there is no one definition of software quality. To developers, it might be elegant design or elegant code. To users, it might be good response time or high customers, it might be satisfying all their perceived and no-yet-perceived needs. The dilemma is that these definitions may not be compatible.

3.1 High-quality Software Possible

Although our industry is saturated with examples of software system that perform poorly, are full of bugs, or otherwise fail to satisfy user needs, there are counter example. Large software systems can be built with very high quality, but they carry a steep price tag - on the order of \$1,000 per line of code: one example is IBM's on-board flight software for the space shuttle: three million lines of code with less than one error per 10,000 lines.

Techniques that have been demonstrated to increase quality considerably include involving the customer, prototyping (to verify requirements before full-scale development), simplifying design, conducting inspections, and hiring the best people.

3.2.2 Give Products to Customers Early

No matter how hard you try to learn user's needs during the requirements phase, the most effective way to ascertain real needs is to give users a product and let them play with it. The conventional waterfall model delivers the first product after 99 percent of the development resources have been expended. Thus, the majority of customer feedback on need occurs after resource have been expended. Contrast this with an approach that you deliver a quick - and dirty prototype early in development, gather feedback, write a requirements specification, and then proceed with full-scale development. In this scenario, only five to twenty percent of development resources have been expended when customers first see the product.

3.2.3 Evaluate Design Alternatives

After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use an architecture simply because it was used in the requirements specification. After all, that architecture was selected to optimize the Understandability of the system's external behaviour. The architecture you want is the one that optimizes conformance with the requirements.

For example, architectures are generally selected to optimize , constructability, throughput, response time, modifiability, portability, interoperability, safety, functional requirements. The best way to do this is to enumerate a variety of software architectures, analyze (or simulate) each with respect to the goals, and select the best alternative. Some design methods result in specific architectures, so one way to generate a variety of architectures is to use a variety of methods.

3.2.5 Use an Appropriate Process Model

There are dozens of process models: waterfall, throwaway prototyping, incremental, spiral, operational prototyping, and so on. There is no such thing as a process model that works for every project. Each project must select a process that takes risks, application area, volatility of requirements, and the extent to which requirements are well-understood.

Study your project's characteristics and select a process model that makes the most sense. When building a prototype for example, choose a process that minimizes protocol, facilitates rapid development and does not worry about checks and balances. Choose the opposite when building a file critical product.

3.2 Minimize Intellectual Distance

Edsger Dijkstra defined intellectual distance as the distance between the real-world problem and the computerized solution to the problem. Richard Fairley has argued that the smaller the intellectual distance, the easier it is to maintain the software. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure. This is primary motivation for approaches such as objective-oriented design and Jackson System Development. But you can minimize intellectual distance using any design approach. Of

course, the real -world structure can vary as Jawed Siddiqi points out !Challenging Univemity Truths of Requirements Engineerit g, Mar. pp.1819). Different humans perceive different 'structm es when they examine the same real world and this construct quite different realities.

3.2.7 Good Management is More Important than Good Technology.

The best technology will mit compensate for poor management, and a good manager can produce great results even with meager resources. Successful software start -ups are not successful beecause they have great proaess of great tools (or great **products** for that matter!). Most are successful because of great management and great marketing.

Good management motivates people to do their best, but there are no universal right styles of management. Management style must be adapted to the situation. It is not uncommon for a successful leader to be an autocrat in one situation and a consensus based leader in another. Some styles are innate, others can be learnt.

3.2 People are the key to success

Highly skilled people with appropriate experience, talent and training are key. The right people with insufficient tools, languages, and, process will succeed. The wrong people with appropriate tool, language and process will probably fail (as will the right people with insufficient training or experience). When interviewing prospective employees, remember that there is no substitute for quality. Don't compare two people by saying, person x is better than person y but person y good enough and less expensive. Your can't have all superstars, but unless you hilly have **an overabundance**, hire them when you find them!

3.2.9 Follow with care

Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to you environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping - all these, might increase quality, decrease cost, and increase **user satisfbction**.

However, only those organizations that can take advantage of them will reap the rewards. The potential Of such techniques is often oversold, and beneffts are by no means guaranteed or universal. You can't afford to ignore a new technology. But don't believe the•

inevitable hype associated with it. Read carefully. Be realistic with respect to payoffs and risks. And run experiments before you make a major commitment.

3.2.10 Take responsibility

When a bridge collapses we ask, what did the engineers do wrong? When software fails we rarely ask this. When we do, the response is, I was just following the **15** steps of this method, or My manager made me do it or The Schedule left insufficient time to do it right. The fact is that many engineering disciplines the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.

There are no excuses: If you develop a system, it is your responsibility to do it right. Take that responsibility. Do it right, or don't do it at all.

4.0 Conclusion

Software engineering deals with the applications of engineering principles to the building of software products. To arrive at a set of engineering principles, one has to select a set of qualities that characterize the products. In this unit we presented a set of qualities for software product. At the end, we discussed several principles which should be applied in designing software products that achieve these qualities.

5.0 Summary

In this unit you have learned about qualities to consider in software products and processes. You have also learnt about Software Engineering principles. In the next unit you will learn about issues that relate to software performance.

6.0 Tutor Marked Assignment

Discuss seven (7) of the important qualities of software products

2 Outline the principles of Software Engineering discussed in this unit.

7.0 References and Other Resources

Management of Information Systems, Unit 2,
Indira Gandhi National Open University

Software Engineering - A Practitioner's Approach by ROGER S.
PRESSMAN: McGraw Hill International Edition.

UNIT 1 LANDMARKS OF OBJECT-ORIENTED ANALYSIS AND DESIGN

6

TABLE OF CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.1 Origin Of Object-Oriented Design (OOD)
- 3.2 Two Different Cultures
- 3.3 COD Approaches
- 3.4 Analysis of OOD Approaches
- 4.0 Conclusion
- 5.0 MI or Marked Assignment
- 6.0 References and Other Resources

LO Introduction

Like many other concepts, the object-oriented concept did not, come up overnight. Object-oriented design has come a long way. In this unit; some of the important landmarks of 'object-oriented design will be discussed

20 Objectives

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

- describe the origin of Object-oriented design
- differentiate between the two cultures in the "object-oriented community"

31 Origin of Object-Oriented Design (OOD)

Work on what was to become "structured design" began in the early 1960s. Structured design, as a well defined and named concept, did not achieve appreciable visibility until the publication of an article in the IBM Systems Journal in 1974. Five years later, Prentice Hall published a book by Larry Constantine and Edward that introduced structured design to the masses.

Even before 1979, variations on Constantine's version of structured design began to appear. After 1979, software engineers could select from an ever increasing variety of approaches to structured design, e.g., Goltz, 1984], Ward and Mellor. • 1985], Hatley and Pirbhai, 1987], and Marca and McGowan, 1988]. It should be obvious to even a casual observer that there is more than one way to accomplish a "structured design." Let

Like structured design, the term "object-oriented design (OOD) means different things to different people. For example, OOD has been used to imply such things as:

- The design of individual objects, and/or the design of the individual methods contained in those objects
- the design of an inheritance (specialization) hierarchy of objects,
- the design of a library of reusable objects , and
- the process of specifying and coding of an entire object-oriented application.

What many people consider to be the first object-oriented programming language - Smalltalk - was introduced in 1966. The term "object-oriented," however, did not come into use until around 1970, with many people crediting Alan Kay as the person who coined the term.

3.2 Two Different Cultures

Part of the problem is the diversity within the so-called "object-oriented community." Many object oriented people seem to focus primarily on programming language issues. They tend to cast all discussions in terms of the syntax and semantics of their chosen object-oriented programming language. These people find it almost impossible to discuss any software engineering activity (e.g., analysis, design, and testing) without direct mention of some specific implementation language.

Outside of producing executable "prototypes," people who emphasize programming languages seldom have well-defined techniques for analyzing their client's problems or describing the overall architecture of the software product. A great deal of what they do is intuitive. If they have a natural instinct/intuition for good analysis or good design, their efforts on small-to-medium, non-critical projects can result in respectable software solutions.

Programming language people use the terms "analysis" and "design" in a very loose sense. Analysis can mean listening to their customer, making some notes and sketches, thinking about both the problem and potential solutions, and even constructing a few software prototypes. Design can mean the code-level design of an individual object, the development of an inheritance (specialization) hierarchy, or the informal definition and implementation of a software product (e.g., identify all the objects, create instances of the objects, and have the instances send messages to each other).

Another group of object-oriented people are interested in formality and rigor. To these people, software engineering is largely very systematic, repeatable, and transferable. They view object-oriented software engineering as primarily an engineering process with well-defined, coordinated tasks, and well-

defined deliverables. The quality of The 'resulting products (and the process its(.lf) can be evaluated in a quantitative, as well as qualitative, manner.

For members of this second camp, "object-oriented programming" (OOP) is primarily a coding activity, and "object-oriented design" did not exist until about 1980. The programming language people, on the other hand, often lay claim to all things object-oriented, including object-oriented design. Even though a well-defined, quantifiable, transferable and repeatable process for object-oriented design did not exist until the early 1980s in their minds OOD has existed at least since there were object-oriented programming languages - a process many people date from 1966, when Simula was first introduced.

As you might guess, there are significant cultural differences between these two groups of object-oriented people. For example; some of 'those that emphasize rigor and formality view the 'programming language people as chaotic, overly error prone, wasteful; and largely unpredictable. On the other hand, some of the programming language people consider the "formality" and "rigor" to be mere window dressing - at best adding nothing to the quality of the final product, and 'at worst increasing the cost or development while simultaneously delaying the delivery and lowering the quality of the resulting software product.

Even if one takes into account the different perspectives described above, there are still significant variations within each of these perspectives. Consider inheritance, i.e., the process whereby an object acquires characteristics from another object. A few object-oriented programming languages allow an object to inherit directly from only one other object (single inheritance). Other languages allow an object to inherit characteristics from more than one object (multiple inheritance). Someone who defines OOD in programming language terms may choose to include or exclude Multiple inheritance from the 'design process based partly whether or not their particular language supports the concept.

Other issues that arise in the programming language camp include the mixing of data and objects, the ability to have program elements that are not encapsulated within any object, and the use of exceptions, parameterized classes, nested classes, and concurrency.

Within the "formality and rigor" group there is also a significant amount of diversity. Some try to portray object-oriented methods as slightly recast structured approaches (old wine in new bottles). Others advocate a more data-driven style, e.g., the extensive use of entity-relationship diagrams and other data modeling techniques. Still others have successfully

blended object-oriented thinking and rigorous software engineering. In effect, they have integrated the two without losing the benefits of either.

At most everyone that advocates a formal approach views object-oriented design as only one part of the software development life-cycle. It may be preceded by such activities as object-oriented analysis and feasibility studies, and followed by object-oriented programming (coding). Those accomplishing object oriented design will also be expected to interact with testing, quality assurance, and management personnel. Only if the software problem is small, and of relatively low risk, will object-oriented design be the first life-cycle activity, and even then it will be followed by object-oriented programming.

The word "formal" is sometimes reserved for mathematical (or logically rigorous) approaches to software development. Examples of such approaches include Vienna Development Methodology (VDM), OBJ and

its derivatives, Z, and C.A.R. Hoare's communicating sequential processes. In this unit, we will use the word "formal" more in the sense of "a well-defined, step-by-step, repeatable process with accompanying guidelines, and defined deliverables.")

3.2.1 Programming Language (Non-Formal) Views of OOD

We will use the term "non-formal" to describe approaches to OOD that are not well-defined, step-by-step, or repeatable, such as those that emphasize the design of individual objects, specialization (inheritance) hierarchies, and libraries of objects. A survey of such approaches indicates that there may indeed be some repeatable rigor (and some sage advice) given for these approaches, but they are severely lacking when it comes to defining the software architecture of large/critical systems.

Non-formal OOD approaches usually exhibit many of the following characteristics:

- There is an overriding emphasis on identifying the objects that will make up the application - almost to the exclusion of understanding the overall application. For example, one non-formal approach defines the OOD approach as "identify and create the classes, create instances from the classes, and then have the instances send messages to each other."
- Because of the above, non-formal approaches are often bottom-up in nature. Specifically, one identifies, defines, and implements pieces of the solution, and then merges these pieces into a final or partial solution. (Note that a bottom-up approach need not be

chaotic, and may be entirely appropriate for small and/or non-critical, problems.)

- Non-formal approaches frequently address issues related to the internal design of individual objects and almost never discuss strategies for the design of the software architecture for the overall application.
- Often there is a blurring of the distinction between the design of an individual object and the design of the application at hand. "Specifically, it is not unusual in such approaches to see objects containing application-specific information. (This, unfortunately, reduces the reusability of such objects and decreases the overall reliability of the application.)

3.2.2 Life-Cycle Views of OOD

We now Shift our attention to those who view object-oriented design as one process among several in the development of software products. This view stipulates that, with the possible exception of very simple, non-critical pieces of software, several processes are involved in the creation of software products. These processes have commonly been referred to as "methodologies." (There are those who point out that the "-ology" suffix on methodology should mean "the study of methods," but we will use the term methodology as it is usually understood.)

There are many views as to how to partition the development part of the software product life-cycle. These views can be very simple (e.g., design, followed by coding, followed by testing) or fairly involved (e.g., including feasibility studies, requirements analysis, high-level external design, et cetera).

Virtually all life-cycle views of OOD assume the possibility of some form of analysis (establishing an understanding of the problem to be solved, sometimes coupled with a high-level external description of the solution). Only if the problem is very simple, will the "analysis phase" become optional. Life-cycle views very often consider "coding" to be separate from "design." For example, it is not uncommon to hear statements such as, "no code will be produced until the design is complete."

Software engineers have known for some time that the divisions between various life-cycle phases (e.g., between analysis and design, and between design and coding) are not sharp. Older approaches tended to emphasize the divisions by using different techniques, deliverables, and viewpoints. For example:

- Structured analysis emphasized understanding the way the client conducted business, data flow diagrams, and "the flow of data."
- Structured design, on the other hand, stressed developing an acceptable software system architecture, structure charts, and "the flow of control."

With both increased experience in software development techniques, and as life-cycle phases have become more blurred. Many people have observed that structured analysis, structured design, and structured programming had little in common other than a sense of formality and the adjective "structured." Object-oriented techniques (e.g., object-oriented analysis, object-oriented design, and object-oriented programming), however, tend to be much more consistent with each other.

E.W. Dijkstra introduced the term, and basic concepts behind, "structured programming" in 1969. Although structured programming focused primarily on coding activities, it accelerated a movement that led to the formalization of other life-cycle phases, e.g., structured design and structured analysis. Object-oriented programming has had a similar effect on life-cycle Phases.

One very important difference between the so-called "structured revolution" and the so-called "object oriented revolution" is the much higher degree of consistency among object-oriented life-cycle phases. For example, the concept of inheritance (specialization) is much the same in object-oriented requirements analysis, in object-oriented design, and in object-oriented programming.

3.3 COD Approaches

3.2.1 Non-Rigorous OOD Approaches

Dave Bulman ([Bulman, 1989]) and others have observed that the mere identification and creation of objects is not a substitute for "design." It is important to realize that, by "design," Bulman means the establishment of a system architecture. This includes not only the identification of system components (objects), but also the definitions of their interactions and interrelationships as well.

Russell J. Abbott ([Abbott, 1980] and [Abbott, 1983]) described an informal method for specifying the design of a software system. It involved writing a paragraph that described a solution to the problem

at hand, and then identifying the nouns and noun phrases as candidate objects. The verbs in the paragraph could then be analyzed to suggest methods (operations) encapsulated within the objects.

Grady Booch adopted the work of Abbott, as a mechanism for bringing out the software engineering features of Ada (e.g., [Booch, 1981], [Booch, 1982], [Booch, 1983a], and [Booch, 1983b]). Initially making only a few changes in Abbott's approach, Booch referred to his technique as "object-oriented design." To be sure, Booch was also influenced by Smalltalk, object-oriented computer hardware, and semantic data modeling. One could say that Booch was probably the first person to significantly popularize the term "object-oriented design." (It was not until the mid to late 1980s that people who focused primarily on object-oriented programming languages began using the term OOD with any frequency.)

3.3.2 Moderately Formal OOD Approaches

People who define approaches, guidelines, and techniques for various software engineering activities are commonly called "methodologists." People such as Grady Booch, Larry Constantine, and James Rumbaugh are examples of people who are often referred to as methodologists.

As both the methodologists and the users of their methodologies become more experienced, they cause mutations (clarifications, modifications, and extensions) to the methodologies. In fact, it can be

Interesting to follow the evolution of a particular methodology over time. gain experience. Thinking on object oriented approaches. Realizing that object-oriented thinking is not limited to design and coding, Booch began to refer to his approach as "object-oriented development."

, Some OOD approaches are strongly influenced by particular programming languages. Although the proponents of "responsibility-driven design" do not think of the approach as specific to a particular programming language, its Smalltalk roots are fairly obvious.

As object-oriented technology increases in popularity, people who formerly advocated structured techniques have begun to modify their approaches to encompass object-oriented thinking. The degree of modification varies from author to author.

One technique for creating a "new" methodology is to take one or more important principles from older approaches and then to recast . new ideas (e.g., object-orientation) in terms of these principles. The Hierarchical Object-Oriented Design (HOOD) stresses the importance of a top-down approach to software engineering, and places object-orientation within that framework. The General Object-Oriented Development (GOOD) approach advocated by Ed Seidewitz and Mike Stark stressed the importance of understanding a problem in more traditional terms (e.g., data modeling) before moving to an object oriented perspective.

There are other views on OOD that have evolved from fairly unusual perspectives. [Jochem et al., 1989] describes an approach that was influenced by computer integrated manufacturing (CIM). James Rumbaugh ([Blaha et al., 1988] and [Rumbaugh et al., 1991]) advocates an approach that is more closely based on data modeling than on object-oriented thinking. Peter Coad ([Coad and Yourdon, 1991]) describes a "multi-component, multi-layered" approach that is fairly unique.

3.3 3 Mixed Paradigms

There are methodologists that suggest mixing object-oriented approaches with other approaches, and giving each approach equal weighting. [Bewtra et al., 1990] suggests combining object-oriented technology with functional programming ([Backus, 1978] and [Backus, 1982]). [Pendley, 1989] describes a combination of object-oriented thinking and information engineering ([Finkelstein, 1989], [Martin, 1989], [Martin, 1990a], and [Martin, 19906]). Stream and formal object-oriented specification techniques are advocated in [Toetenel et al., 1990].

3.3 4 Modifying Other Approaches to Encompass ,Object-Oriented Thinking

When moving from an older way of doing things to a newer way, it is seldom advisable to "throw out" everything connected with the old way. One often used strategy is to enlarge the older way so that it can encompass some or all of the aspects of the newer. Some authors suggest mechanisms for keeping much of traditional structured/functional-decomposition thinking while addressing object- , oriented concerns.

3.33 Different Paradigms in the Same Life-Cycle

Experience has shown that simply attempting to integrate object-oriented thinking into the more traditional methodologies (e.g., structured) is a mistake. The major problem is that of localization, i.e., the placing of related items in close physical proximity to each other. Functional approaches, for example, tend to localize information around functions, whereas object-oriented approaches tend to localize information around objects. A functional decomposition "front end" to an object-oriented process, in effect, breaks up objects and scatters their parts. Later, these parts must be retrieved and delocalized around objects.

There have been quite a number of attempts to reconcile the output of a non-object-oriented process with the input requirements of an OOD process, but none of these scenarios are as clean and easy as using an ' object-oriented approach from the very beginning of the software life-cycle.

3.4 Analysis of OOD Approaches

Since object-oriented programming has been with us for more than a quarter of a century, and OOD proper has been around for over a decade, it is not unusual that a number of attempts have been made to analyze the OOD process. While some of these analyses reveal potential problems with particular OOD approaches, none have advocated avoiding an object-oriented approach.

Comparisons

Comparisons of methodologies have been around since the 1970s. The first significant comparison of OOD with Structured Analysis/Structured Design and Jackson System Development was done under the auspices of General Electric ([Boehm-Davis and Ross, 1984]). This study gave the same problem to three different groups of people, and had them all implement solutions using the same programming language.

When compared with the other solutions, the researchers found that the OOD solutions:

- were simpler (using control-flow complexity and numbers of operators and operands as metrics),
- were smaller (using lines of code as a metric),
- took less time to develop, and
- were better suited to real-time problems.

Seine comparisons have been very informal (e.g., [Boyd, 1987] and Pamea, 19(-11), while others have been flawed because the authors did not fully understand what OOD was.

009 Techniques

A number of authors, while not describing complete OOD methodologies, have described techniques that can be used in the OOD process. [Beck and . Cunningham, 1989], for example, describes Class responsibility-Collaboration (CRC) cards. The idea is to create one CRC card for every class involved in the problem or solution. The responsibilities (method interfaces) of the object are documented on one area of the card and the coilaborations (other objects with which the object must interact) are placed on .mother area.

[13: rue and Kwiatkowski, 1986] describe a graphical means of representing obiccts in an OOD process. In the same vein, [Coleman et al., 1992] de cribes a variation on state charts aHarel, 1987] and [Harel et al., 1987], i.e.. 'object charts." (Loomis et al., 1987] defines a graphical technique that is in tile OOD approach advocated by James Kmbaugh.

Experiences With OOD

accounts of the experiences of those who have used the technology. [Chedghey et al., 1987] describes an attempt to use a formal methodology (Vienna Development Method, e.g., [Jones, 1986]) with 00D. [Meyer et al., 19N9] talks about the experience of using an approach to OOD that mixes functional decomposition with more traditional object-oriented techniques. " [Dovis and Irving, 1989] presents a discussion of one of the most common uses of 00D, i.e., for real-time systems. [Vlissides and Linton, 1988] describes the use of OOD for the creation of a graphics application.

Metrics

Evcn before Tom Glib wrote his landmark book on software metrics ([Glib, 19771), people had been interested in measuring software. While there are now numerous metrics for measuring non-object oriented software (e.g., [A rthur, 1985], [Card and Glass, 1990], [Conte et], [Dreger, 1989], [Ejiogu,

19"1_1, [Grady and Caswell. 1987], and [Jones, 19911). there are relatively fcii discussions on metrics for object-oriented software.

Kul Livebearer and his colleagues have written a number of articles on asi-ssinr the quality of the design of an individual object (e.g., [Livebearer an Riel, 1989]). However, until very recently, there has not been much else

pul dished. [Chidambaram and Kemmerer, 1991] is probably the most comprehensive article to date on assessing the quality of an object-oriented

Computer Aided Software Engineering For OOD

Computer aided software engineering (CASE), once considered a luxury, is becoming increasingly necessary in today's software engineering arena. With the rising interest in object-oriented software engineering, it is only natural to ask. "Where are the CASE tools for object-oriented technology?"

One of the major obstacles to OOD CASE tools is the wide variety of approaches. Some OOD methodologists, or their organizations, have put out their own CASE tools, e.g., Rationale's ROSE(tm) and Object International's OGATool(tm). Some CASE vendors have chosen to automate the approaches of several different methodologists, e.g., Mark V Systems, Ltd.'s Object.

ker(tm) and Photoset Inc.'s Paradigm Plus(tm). However, it will be some time before the object-oriented technology market becomes as focused as the so-called structured technology marketplace.

4.0 Conclusion

Object-oriented technology has many different dimensions, viewpoints, and implementation strategies. Those considering using OOD on a project have many different options from which to choose. However, the "right choice" will require careful research.

6.0 Tutor Marked Assignment

1. Describe briefly the origin of Object-Oriented Design
2. Discuss the two views of Object-Oriented Design

7.0 References and Other Resources

Object-Oriented Design By Edward V. Berard

UNIR 8: OBJECT-ORIENTED ANALYSIS AND DESIGN**TABLE OF CONTENTS**

1.0	Introduction
2.0	Objectives
3.1	Object-Oriented Analysis And Design
3.2	Role Of OOAD In The Software Life Cycle
3.3	OOAD Methodologies
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	References And Other Resources

1.0 Introduction

Most new client/server application development tools emphasize object-oriented (OO) features. This implies that an object - oriented analysis and design (OOAD) methodology should be an effective guide to applying these tools to business problems. But with the numerous methodologies available for OOAD, how do you choose the correct one? How do you get started, and how do you ascertain whether your approach is as efficient as it should be? And how do you avoid the pitfalls? OOAD can provide wonderful benefits, but believing that a new methodology will solve all your problems is like believing in Utopia.

In this unit we will discuss a number of Object-oriented Analysis and Design methodologies.

2.0 Objectives

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

- describe the origin of Object-oriented design
- differentiate between the two cultures in the "object-oriented

community"

3.1 Object-Oriented Analysis and Design

A survey of available literature shows that the applications competing for "object honors" are written mostly in C++ or Smalltalk, and have a heavy manufacturing, engineering, aerospace, or scientific focus. That the major OOAD methodologies are best for such problems, however, does not guarantee that they are also best for typical business client/server applications. You need to determine which methodology is best suited for your particular application, and what to do if no methodology matches your •

requirements precisely. (Using OOAD often requires adaptations to methodologies.)

3.2 The Role of OOAD in the Software Life Cycle

To understand what's right and wrong with OOAD, you need to know where OO methodologies fit into the software life cycle. These methodologies do not replace traditional approaches (such as data flow, process flow, and state transition diagrams); they are important new additions to the toolkit.

According to Donald Firestone in his book Dictionary of Object Technology (SIGS Books, 1995), analysis is "the development activity consisting of the discovery, modeling, specification and evaluation of requirements," while OO analysis is "the discovery, analysis and specification of requirements in terms of objects with identity that encapsulate properties and operations, message passing, classes, inheritance, polymorphism and dynamic binding." Firestone also states that OO design is "the design of an application in terms of objects, classes, clusters, frameworks and their interactions."

In comparing the definition of traditional analysis with that of OOAD, the only aspect that is really new is thinking of the world or the problem in terms of objects and object classes. A class is any uniquely identified abstraction (that is, model) of a set of logically related instances that share the same or similar characteristics. An object is any abstraction that models a single thing, and the term "object" is synonymous with instance. Classes have attributes and methods. For an object class named Customer, attributes might be Name and Address, and methods might be Add, Update, Delete, and Validate. The class definition defines the Customer class attributes and methods, and a real customer such as "XYZ Corp." is an instance of the class. If you have different kinds of customers, such as residential customers and commercial customers, you can create two new classes of customers that are descendants of the Customer class. These descendants use inheritance to gain access to all of the Customer class attributes and methods, but can override any of the ancestor attributes and methods, as well as contain any required new attributes and methods.

There are three types of relationships between classes: inheritance, aggregation, and association. Inheritance (also referred to as generalization/specialization) is usually identified by the phrase "is a kind of." For example, Student and Faculty are both a kind of Person and are therefore inherited from the Person class. Aggregation is identified by the as with a product that contains parts. If neither of the first two relationships applies; but the objects are clearly related (for

example, an employee is associated with a company), then the relationship is association.

An abstract class is a class that has no instances, and is used only for inheritance. A concrete class is a class that can be instantiated, that is, that can have direct instances.

All of the major OOAD methodologies have a similar basic view of objects, classes, inheritance, and relationships. The drawing notation is slightly different in each; the real differences in the methodologies are more subtle.

Table 1 (below) summarizes the major OOAD methodologies. When you're choosing a methodology, it is important to consider not only the methodology's features, but also the cost of using it, the types of problems to which it is best suited, its limitations, and the training available. When used in typical initial attempts to develop client/server applications using OOAD methodologies, all of the methodologies suffer from the same basic flaws:

- an overemphasis on the OO approach in general, even though another approach might be better for some parts of the problem
- an overemphasis on the problem domain object model during the analysis phase
- analysis diagrams and output formats that end users may find difficult to understand
- difficulty in the methodology's ability to describe complex analysis problems
- a lack of emphasis on the underlying system architecture
- an inability to understand the limitations of either 4GL OO languages or of beginning OO developers

Table 1. Major OOAD Methodologies

Analysis Methodology	Proprietary	Type Scope	Strengths	Primary Cited Applications / Markets
Booch	No	Ternary	Complex, Rich, Pragmatic	Design All
Coad and Yourdon	No	Unary	Simple, Limited, Pragmatic	Analysis Client/Server
Fusion	No	Ternary	Complex, Rich, Pragmatic	Full Life Cycle All

Jacobson Objectory	Yes Ternary	Complex, Rich	Full Life Cycle	All
LBMS c.j.;flfl	Yes Ternary	Middle of the Road. Pragmatic	Full Life Cycle	Client/Server
Rumbaugh	No Ternary	Complex, Rich, Somewhat Pragmatic	Analysis & Design	All, but heavily Embedded. Real-Time
Shlaer and Mellor OOA	No Ternary	Complex, Rich	Design	Embedded, Real-Time

*- Every object methodology tells you to start with the object model, not the data model. there are at least four problems with this approach.

- The data model often exists before the object model.
- The analyst may rightly be more comfortable building the data model before the object model.
- A good object model should be able to map to any data model. For me, it is usually a requirement in complex systems that an object's attributes can map to one or more tables in one or more databases.
- A good abstract object model of the problem domain may not be easy to implement in the chosen language or development tool.

3.3 OOAD Methodologies

OOAD methodologies fall into two basic types. The ternary (or three-pronged) type is the natural evolution of existing structured methods and has three separate notations for data, dynamics, and process. The unary type asserts that because objects combine processes (methods) and data, only one notation is needed. The unary type is considered to be more object-like and easier to learn from scratch, but has the disadvantage of producing output from analysis that may be impossible to review with users.

Thompson's is concerned with events and states, and generally uses state transition diagrams. Process modeling or functional modeling is concerned with processes that transform data values, and traditionally uses techniques such as data flow diagrams.

In the following sections, I describe the methodologies of Booch, Coad and Yourdon, Fusion, Jacobson, LBMS, Rumbaugh, and Shlaer and Mellor. There are several other methodologies that I don't discuss, and if you are

Jacobson	Yes	Ternary	Complex, Rich	Full Life Cycle	All
LBMS	Yes	Ternary	Middle of the Road. Pragmatic	Full Life Cycle	Client/Server
Rumbaugh	No	Ternary	Complex, Rich, Somewhat Pragmatic	Analysis & Design	All, but heavily Embedded. Real-Time
Shlaer and Mellor	No	Ternary	Complex, Rich	Design	Embedded, Real-Time

4. Every object methodology tells you to start with the object model, not the data model. there are at least four problems with this:

- The data model often exists before the object model.
- The analyst may rightly be more comfortable building the data model first; the object model.
- A good object model should be able to map to any data model. For me, it is; usually a requirement in complex systems that an object's attributes can map to one or more tables in one or more databases.
- A good abstract object model of the problem domain may not be easy to implement in the chosen language or development tool.

3.3 OOAD Methodologies

OOAD methodologies fall into two basic types. The ternary (or three-pronged) type is the natural evolution of existing structured methods and has three separate notations for data, dynamics, and process. The unary type asserts that because objects combine processes (methods) and data, only one notation is needed. The unary type is considered to be more object-like and easier to learn from scratch, but has the disadvantage of producing output from analysis that may be impossible to review with users.

Dynamic modeling is concerned with events and states, and generally uses state transition diagrams. Process modeling or functional modeling is concerned with processes that transform data values, and traditionally uses techniques such as data flow diagrams.

In the following sections, I describe the methodologies of Booch, Coad and Yourdon, Fusion, Jacobson, LBMS, Rumbaugh, and Shlaer and Mellor. There are several other methodologies that I don't discuss, and if you are

interested in learning more about them, I strongly recommend Ian Graham's book, *Object-Oriented Methods* (Addison-Wesley, 1994), which does an excellent job of both describing and comparing available methodologies.

3.3.1 Booch

Grady Booch's approach to 00AD is one of the most popular, and is supported by a variety of reasonably priced tools ranging from Visio to Rational Rose. Booch is the chief scientist at Rational Software, which produces Rational Rose. (Now that James Rumbaugh and Ivar Jacobson have joined the company, Rational Software is one of the major forces in the 00AD world.)

Booch's design method and notation consist of four major activities and six notations, as shown schematically in Table 2.

While the Booch methodology covers requirements analysis and domain analysis, its major strength has been in design. However, with Rumbaugh and Jacobson entering the fold, the (relative) weaknesses in analysis are disappearing rapidly. I believe that Booch represents one of the better developed 00AD methodologies, and now that Rational Rose is moving away from its previous tight link with C++ to a more open approach that supports 4GLs such as PowerBuilder, the methodology's popularity should increase rapidly.

Table 2. The Steps in Booch's Methodology

Steps Notations

Logical structure	Class Diagrams Object Diagrams
Physical structure	Module Diagrams Process Diagrams
Dynamics of State Transition Diagrams Classes	
Dynamics of Timing Diagrams Instances	

For systems with complex rules, state diagrams are fine for those with a small number of states, but are not usable for systems with a large number of states. Once a single-state transition diagram has more than

eight to 10 states, it becomes difficult to manage. For more than 20 states, state transition diagrams become excessively unwieldy.

W- 3.3.2 Coad and Yourdon

Coad and Yourdon published the first practical and reasonably complete books on OOAD (Object Oriented Analysis and Object-Oriented Design, Prentice-Hall, 1990 and 1991, respectively). Their methodology focuses on analysis of business problems, and uses a friendlier notation than that of Booch, Shlaer and Mellor, or the others that focus more on design.

In Coad and Yourdon, analysis proceeds in five stages, called SOSAS:

- **Subjects:** These are similar to the levels or layers in data-flow diagrams and should contain five to nine objects.
- **Objects:** Object classes must be specified in this stage, but Coad and Yourdon provide few guidelines for how to do this.
- **Structures:** There are two types: classification structures and composition structures. Classification structures correspond to the inheritance relationship between classes. Composition structures define the other types of relationships between classes. Coad and Yourdon do not deal as well as Rumbaugh, Jacobson, and several other methodologies do with these structures.
- **Attributes:** These are handled in a fashion very similar to that in relational analysis.
- **Services:** The identification of what other methodologies call methods or operations.
- In design, these five activities are supplanted by and refined into four components:
 - **problem domain component:** classes that deal with the problem domain; for example, Customer classes and Order classes
 - **human interaction component:** user-interface classes such as window classes
 - **task management component:** system-management classes such as error classes and security classes
 - **data management component:** database access method classes and the like

Although Coad and Yourdon's methodology is perhaps one of the easiest OO methodologies to learn and get started with, the most common complaint is that it is too simple and not suitable for large projects. However, if you adhere to a premise that you should use those pieces of a methodology that work, and add other parts from

other methodologies as required, Coad and Yourdon's methodology is not as limiting as its critics claim.

3.3.3 Fusion

In 1990, Derek Coleman of Hewlett-Packard led a team in the U.K. to develop a set of requirements for 00AD, and conducted a major survey of methods in use at HP and elsewhere. The chief requirement was a simple methodology with an effective notation.

The result was Fusion, Which Coleman and others developed by borrowing and adapting ideas from other methodologies. They incorporated some major ideas from Booch, Jacobson, Rumbaugh, and others, and explicitly rejected many other ideas from these methodologies.

Coleman did not use some of the major components of Rumbaugh and Shlaer and Mellor in Fusion, because the components were not found to be useful in practice. Some writers have called this encouraging and remarkable, and consider it indirect proof that excessive emphasis on state models comes from Rumbaugh and Shlaer and Mellor's telecommunications and real time system backgrounds.

Fusion's pragmatic approach seems to hold considerable potential for client/server applications, but this methodology is not being marketed as aggressively as most of the other methodologies.

3.3.4 Jacobson: Objectors and DOSE

Although Jacobson's full 00AD methodology, Objectory, is proprietary (to use it you must buy consulting services and a CASE tool, 00SE, from Rational Software), it is probably the most serious attempt by an 00AD tool vendor to support the entire software development life cycle. Jacobson is considered to be one of the most experienced 00 experts for applying 00 to business problems such as client/server applications.

Jacobson's Object-Oriented Software Engineering (00SE) is a simplified applications. According to Jacobson: "You will need the complete ... description which, excluding large examples, amounts to more than 1200 pages" (Object-Oriented Systems Engineering, Addison-Wesley, 1992).

Object modeling and many other OO concepts in Objectory and OOSE are similar to OO concepts in other methodologies. The major distinguishing feature in Jacobson is the use case. A use-case definition consists of a diagram and a description of a single interaction between an actor and a system; the actor may be an end user or some other object in the system. For example, the use-case description of an order entry application would contain a detailed description of how the actor (the user) interacts with the system during each step of the order entry, and would include descriptions of all the exception handling that might occur.

3.3.5 LBMS SE00

Systems Engineering OO (SE00) is a proprietary methodology and toolkit from the U.K.-based company LBMS, which has its U.S. headquarters in Houston. SE00 is tightly integrated with Windows 4GLs such as PowerBuilder, and is perceived to be a very pragmatic and useful tool, but this perception may be due in part to a stronger marketing effort than is often made for nonproprietary methodologies.

Because SE00 is proprietary, there is not as much detailed information available about it as there is about other methodologies, and it is somewhere between difficult and impossible to try it out just to compare it with the others.

The four major components of the SE00 methodology are: •
work-breakdown structures and technique

- an object modeling methodology
- GUI design techniques
- relational database linkages to provide ER modeling and 4GL-specific features

with non-OO approaches and then adapting to OO. A very positive aspect of this is the heavy focus on data management and data modeling. SE00 is intended to be object oriented while retaining the advantages of traditional data modeling. This makes the methodology well-suited for client/server database applications. SE00 is unique in treating data, triggers, and referential-integrity rules as a set of shared objects in a database. It treats a data model as a view of the shared objects, which also include constraints, rules, and dynamics (state transitions and so on). SE00 draws a clear line between shared objects and other objects, and regards the shared objects as important interfaces between subsystems. This technique allows a distinction, for example, between customer behavior shared by all applications and customer object behavior unique

to a single application. It is a technique with which a purist would quibble, but which is eminently practical.

3.3.6 Rumbaugh OMD

James Rumbaugh's methodology, as described in his book *Object-Oriented Modeling and Design* (Prentice-Hall, 1991), offers one of the most complete descriptions yet written of an OO analysis methodology. Although it is somewhat lacking in OO design and construction, it contains a large number of ideas and approaches that are of significant use to analysts and designers.

Analysis consists of building three separate models:

- the Object Model (OM): definition of classes, together with attributes and methods; the notation is similar to that of ER modeling with methods (operations) added
- the Dynamic Model (DM): state transition diagrams (STDs) for each class, as well as global event-flow diagrams
- the Functional Model (FM): diagrams very similar to data flow diagrams

3.3.7 Shlaer and Mellor OO Analysis

When Shlaer and Mellor OO analysis first came out in 1988, it represented one of the earliest examples of OO methodology and it has evolved very positively since then.

Originally an object-based extension of data modeling, the Shlaer and Mellor methodology starts with an information model describing objects, attributes, and relationships. (Note that this is more like a data model than an object model.) Next, a state model documents the states of objects and the transitions between them. Finally, a data-flow diagram shows the process model.

4.0 Conclusion

No one feature will make all software trivial to write, and no one architecture will be ideal for all problems. Creating good software will continue to be hard work. Each of these methodologies are very useful for particular , situations.

5.0 Summary

In this unit you have learned about the role of object-oriented analysis and design in the Software Life Cycle and you have been introduced to some of the major 00AD methodologies.

6.0 Tutor-marked Assignment

I Outline FOUR 00AD methodologies and briefly describe them.

7.0 References and Other Resources

Michael Gora, Object-Oriented Analysis and Design

UNIT 9: OBJECT-ORIENTED SOFTWARE DESIGN

TABLE OF CONTENTS

1.0	Introduction
2.0	Objectives
3.1	Software Engineering Trends
3.2	Object-Oriented Approach
4.0	Conclusion
5.0	Summary
6.0	Tutor-Marked Assignment
7.0	References And Other Resources

1.0 Introduction

For any real-world problem, you have to design the software - identify the objects and their interrelationships - before you can write any code to implement the objects in an object-oriented language such as C++. In fact, this early part of the software development process - the analysis and design phases - is much harder than the actual coding because there are no well-defined, step-by-step methods for accomplishing the job. The best you can do is to learn about the prevailing practices in object-oriented analysis and design and adapt them to your problem.

This unit provides a summary description of some of the current ideas in the field of object-oriented design. However, the unit does not attempt complete coverage of object-oriented design.

2.0 Objectives

After completing this unit, you should be able to;

- describe the sequence of activities that constitute software development
- identify methods and tools for accomplishing the analysis and design phases of the designing software
- enumerate the problems with top-down approach to software design
- describe the use of object-oriented techniques in software development process

3.1 SOFTWARE ENGINEERING TRENDS

Books on software engineering show the traditional lifecycle of software development in the form of a waterfall (see Figure 9.1), in which the development process follows a rigid sequence from analysis to design and to implementation and testing. Based on that waterfall process, here is an overview of a simplified view of the sequence of activities that constitute software

1. You begin development with the analysis phase by analyzing what the software must do and arriving at a complete, detailed description of the software's behavior in response to the possible set of inputs. You work with potential users of the software to find a definitive answer to the question: What does the software do?

The result of this step is a set of requirements for the software.

2. Next comes the design phase, during which you decide how to do what the user wants. Here your goal is to map the user's real-world description of the software into algorithms and data that can be implemented in a programming language. Typically, you might follow a top-down approach and repeatedly decompose the software's functions into a sequence of progressively simpler functions that are eventually implemented in the implementation phase.
3. In the implementation phase you define the data structures and write the code to implement the functions that were identified in the design. Finally, you have to test the software to verify that it conforms to the original specification as much as possible.

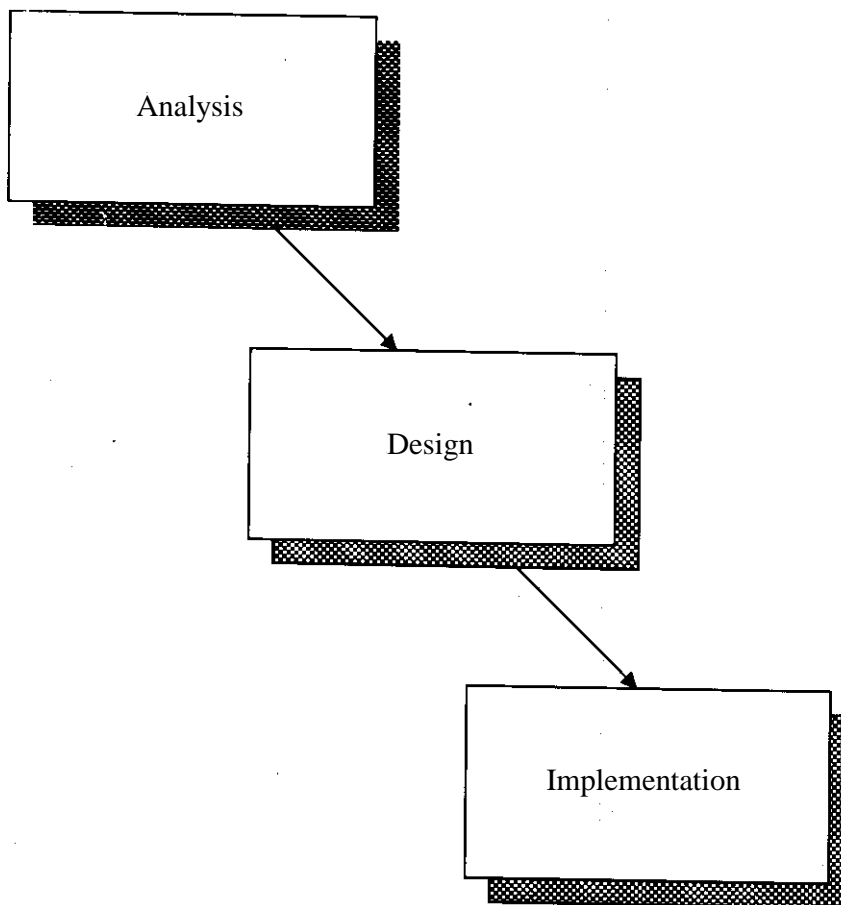
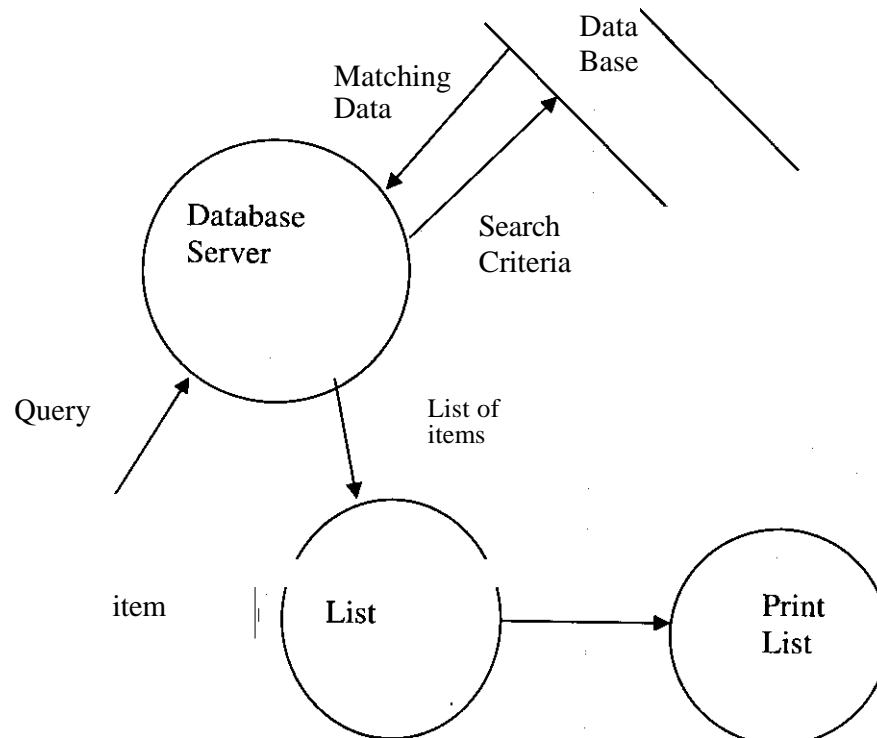


Figure 9.1 Traditional Life Cycle of Software

3.1.1 Prevailing Methods

Although there is no single, clear-cut, step-by-step approach to designing software, there are well-known methods and tools for accomplishing the analysis and design phases of the process.

Among the analysis methods, the most popular is structured analysis, attributed to Tom DeMarco, who built on prior work by Ed Yourdon and Larry Constantine. Structured analysis is concerned with the way data flows through the system. It generates a data flow diagram (DFD) and two textual descriptions: a data dictionary and a minispecification. The DFD is a diagramming notation that depicts the flow of data through the system and identifies the processes that manipulate the data. The data dictionary describes the data shown in the DFD, whereas the minispecification describes, in plain English, how the data is processed by each process. As an example, consider the problem of querying a database and printing a sorted list of the items retrieved by the database. Figure 9.2 shows a grossly simplified data flow diagram for this task. The diagram shows the major processes (functions), with arrows indicating the data being passed between functions.



Another type of diagram for representing a software system is the structure chart introduced by Larry Constantine in the 1970s as a replacement for flowcharts of earlier years. The structure charts are quite similar to the flowcharts, and they provide a notational means to represent the behaviour of the system being implemented. As shown in Figure 9.3, a structure chart starts with a single top-level module represented by a rectangle. That module represents the overall function of the system. From that topmost module, arrows fan out to subordinate modules that the topmost module invokes. The subordinate

modules, in turn, invoke other modules. Thus, the structure looks like an organization chart, reflecting the top-down nature of the design. Each arrow emanating from one module to another represents a function call as well as a data transfer. In an actual structure chart, a label next to each arrow identifies the data being transferred. There are other symbols as well as that indicate program's control flow, such as if statements and loops. Many commercially available computer-aided software engineering (CASE) tools support structure charts with symbols similar to those originally suggested by Larry Constantine.

Solve Problem I
 Part 1 Part 2 Part 3 Task A Task 6
 Library 1

Which problem?

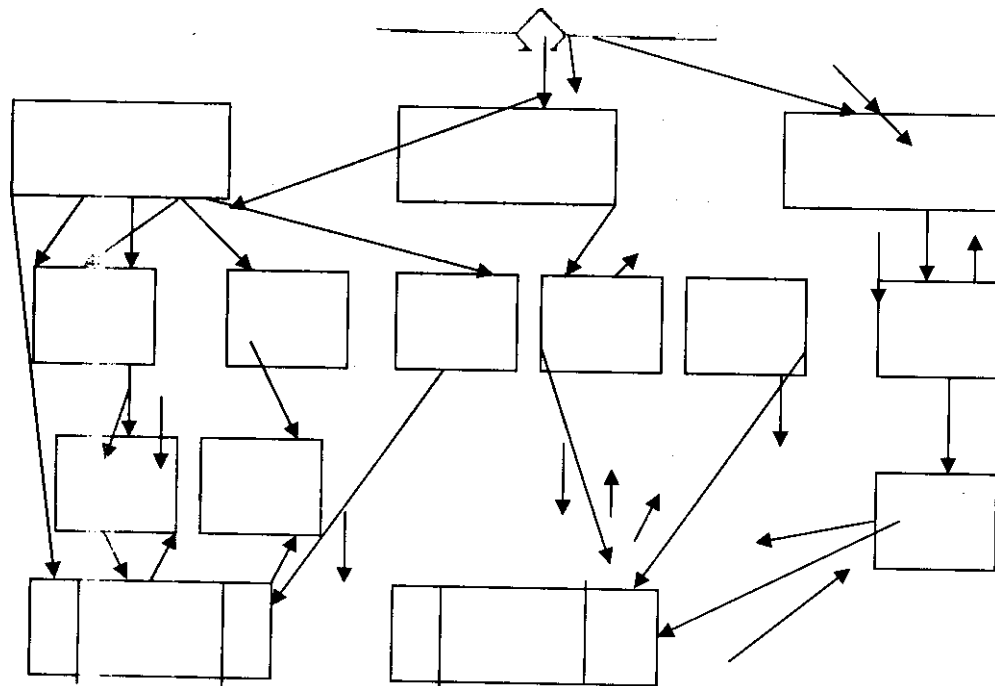


Figure 9.3 A typical structure chart

The design phase of the traditional development cycle is rather loosely defined. The goal of design is to refine the data flow diagrams and map the data into data structures using facilities of the programming language (for example, struct in C) that you will use to implement the software. Procedures are then defined to implement the modules that manipulate the data. The distinction between the analysis and design phases are often blurred and, usually, there are several iterations before you arrive at a design for the software.

f 3.1.2 Problems With The Top-Down Approach

There are several problems with the top-down approach so prevalent in structured techniques. As Bertrand Meyer points out, a top-down design

- does not allow for evolutionary changes in software.
- characterizes the system as having a single top-level function, which is not always true (in Meyer's words: "Real systems have no top").
-
- gives functions more importance than data, thus ignoring important characteristics of the data. • hampers reusability, because submodules are usually written to satisfy the specific needs of a higher-level module.

3.2 Object-Oriented Approach

In any realistic software project, post-implementation changes are all but evolutionary. Because we learn as we go along, our usual approach to a new programming task is to go through an iterative process of analyzing the problem, implementing it, and then refining the design. In other words, we develop prototypes or working models of the software. Grady Booch calls this the strategy of "analyze a little, design a little." He qualifies this by

- stating that it does not mean that you should design by trial and error. Instead, Booch advocates a design process that proceeds with a series of prototypes, each modeling an important aspect of the system and each selected with an
- end toward arriving at the complete functionality as the collection of prototypes grows.

The emerging object-oriented design (OOD) techniques reflect the evolutionary aspect of software development. The steps of analysis, design, and Implementation are still necessary, but the separation between them is blurred. Also, the approach in each phase is more closely tied to the objects in the real world problem being solved. The remainder of this unit briefly discusses several ways of using object oriented techniques in the software development process.

3.2.1 FINDING OBJECTS USING STRUCTURED ANALYSIS TOOLS

At the implementation level, object-orientation means encapsulating data structures with their functions and using the notion of "message-passing" (which may very well be implemented by function calls) to accomplish the tasks of a program. The question in object-oriented design is how to find the objects. For those already familiar with structured analysis, the answer may be in using the results of structured analysis to find the objects to be implemented using OOP techniques. This approach can be exploited by commercial software developers who are trying to introduce object-oriented technology but

have already invested in CASE tools that employ top-down analysis and design.

A class is the template that defines the data and functions common to a set of objects. Each object is an instance of its class. A class library is a collection of „lasses, usually meant for some related tasks such asnying objects or gyring them on disk

Figure 9.4 illustrates the steps for a simplified example of finding the objects for an index card file. Each card stores a name, address, and phone number. Conventional structured analysis will model the card file as follows:

ntify the requirements of the card filing system. In this case, the system must be capable of creating a new card deck, adding or deleting a card, fir Ting a card, and saving cards in a disk resident database.

Translate the requirements into a top-down structure in which the topmost module enables the user to pick one of several choices: create a new deck, save it, add a card, delete a card, or find a card. Each choice is handled by se; arate modules. The analysis also identifies the data items and how they are pr. .essed. Typically, you will use libraries of functions for specialized tasks such as displaying a card or organizing the deck of cards as a database.

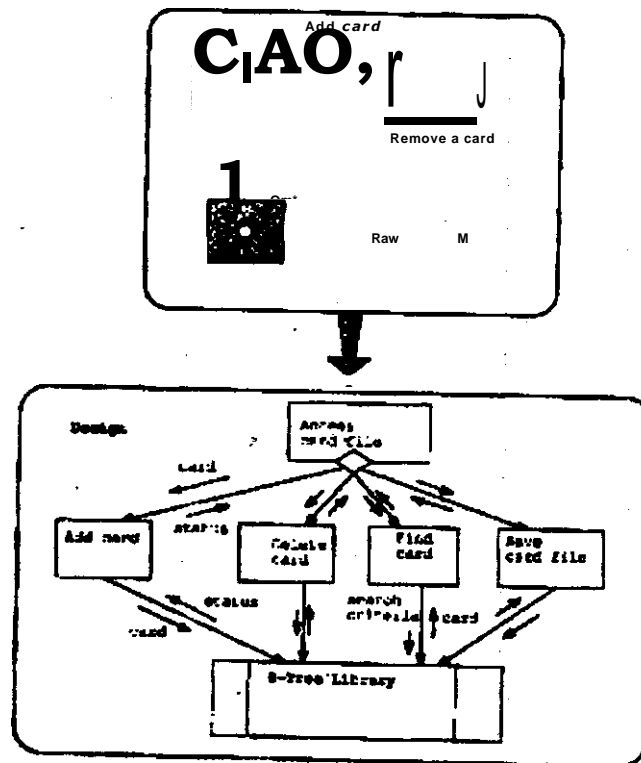


Figure 9.4 Designing a card file

You can use existing CASE tools in performing the analysis to determine the functional breakdown of the system and identify the necessary data items (from the data dictionary, for instance). At this point, instead of using the CASE tool to design the system, you can try to identify the object A from the data and the functions. For the card file, here is a partial description of how you might do this:

- Note that cards are manipulated by most modules identified by structured analysis. Thus, each card should be an object in the system. In C++ terminology, you might decide to define an Index card class, for instance, to encapsulate the data and functions necessary to model an index card.
- Because a card file is a collection of cards, there should be a way to maintain such a collection. You might use a container class (a class designed to hold objects) or simply use an array of Index Card objects for this.
- Each card has several strings to hold information such as the name and address. For this, you can use a String class that enables you to handle each string as an object. When a new card is created, you can create the necessary string objects.
- You can choose from among three options for obtaining the database that will store the cards. You can design a complete set of classes, implementing a database in an object-oriented manner. You can buy a commercially available library of classes and use it. Lastly, you can directly call functions from the programming interface to a commercial database that does not follow an object-based design. The problem with the last approach is that embedding calls to a database in the Index Card class, for instance, couples the card class too tightly to a specific database.

As you can see, the process of identifying the objects can be quite complicated, even for a relatively simple example such as the card file. Even so, there are some general guidelines for identifying the classes:

- (24
- 1. Look for data and related functions that operate on the data. Group them into a class.
 - If a class seems too specific, try to derive it from a more general-purpose class. The idea is to look for similarities among classes and to create a hierarchy in which the common features are in a base class. For example,

instead of implementing a circle and a rectangle shape, first define a generic shape class, then derive circle and rectangle classes from that generic shape. That way, a new shape, such as a triangle, can be derived easily from the same base.

- Use a bottom-up approach to design libraries of basic classes such as strings and collections. For example, If you design a database for the card file, it can be a general-purpose database class that can be reused in other projects as well.
- Avoid embedding in a class any code for displaying an object or storing it to the disk. Instead, use a separate class library for these tasks.

Unfortunately, as fuzzy as these guidelines are for identifying objects and their interrelationships, the material in most of the references listed at the end of this chapter is not any more specific. Luckily, this situation is bound to improve, because object-oriented design is the current topic of choice among many researchers and new design methods are gradually beginning to emerge. The following discussions summarize some important ideas that can help you gain insight into designing object-oriented software.

3L7 - i Notation For Objects

Diagrams are an essential part of any design, and object-oriented design is no exception. Unfortunately, there is no standard notation for representing objects and their interactions. Authors of books on object oriented techniques have used their own notations to denote objects. Recently other suggested notations have appeared in computer journals. Of these, Wasserman, Muller, and Pitcher have based their notation on a combination of structure charts and the notation that Booch uses for Ada packages - modules in the Ada programming language. Figure 5.5 shows a subset of Wasserman's notation that is adequate for representing objects. The notation developed by Wasserman and colleagues is used in a design technique termed object-oriented structured design (OOSD), which forms the basis of a useful design tool named Software Through Pictures developed and marketed by Interactive Development Environments, Inc.

3.2 .4 Hybrid Design Methods

One trouble with using structured analysis as a basis for object-oriented design is that the two approaches use radically different grouping of the functions in the system. As Bailin points out, structured analysis

groups functions together if, as a group, they constitute a higher-level , function. On the other hand, object-oriented analysis groups functions together on the basis of the data they operate on. Thus, in object-oriented approach, all functions operating on the same class of data fall in the same group.

This, however, does not mean that structured analysis is not useful. In fact, according to Larry Constantine, one of the pioneers of the structured approach, it may be more useful and even practical to use a mix of top-down functional analysis together with object-orientation. Here are three possible scenarios:

- An object-oriented system with top-down functional decomposition applied to each object's methods: You decompose the system into interacting objects, either by identifying the objects from the output of a CASE tool or by using one of the methods discussed later in the chapter. Then, you apply top-down structured analysis techniques to design the object's internal methods (member functions in C++ terminology). In this case, "
 - outwardly, the system appears object oriented, but inside each object there may be a small hierarchy of functions designed in a top down manner (see Figure 9.6).
- A top-down hierarchy of functions controlling the system that employs object-oriented modules for its functionality: As you can see from Figure 9.7, in this case you design the system using a
 - top-down approach, but you implement the modules by using a set of interacting objects.

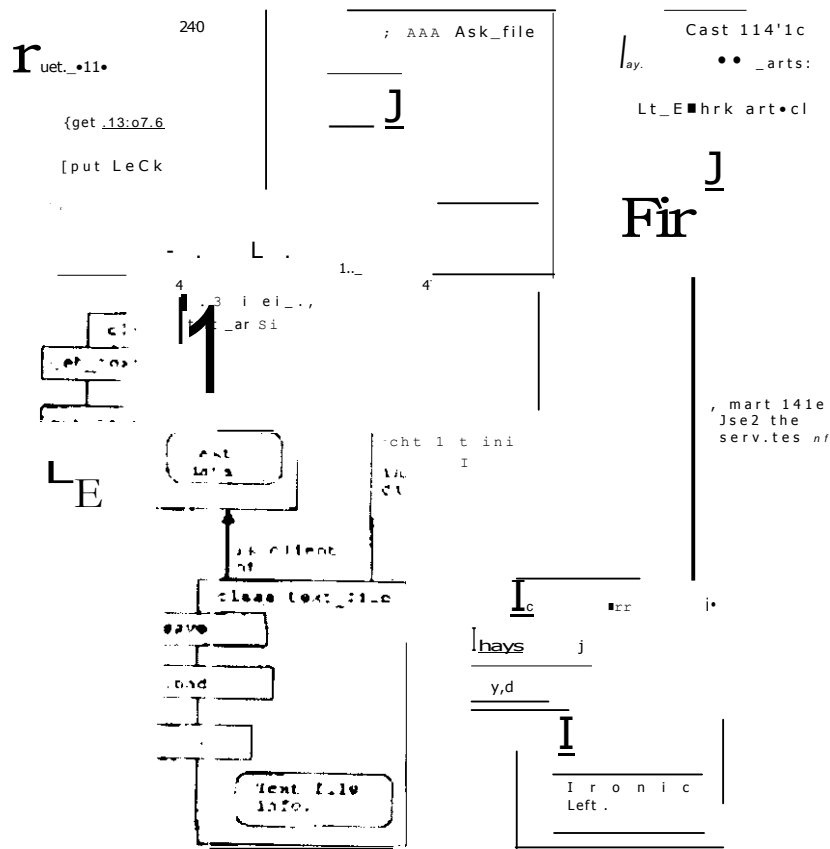


Figure 9.5 Notations for objects. and their interactions

A!:

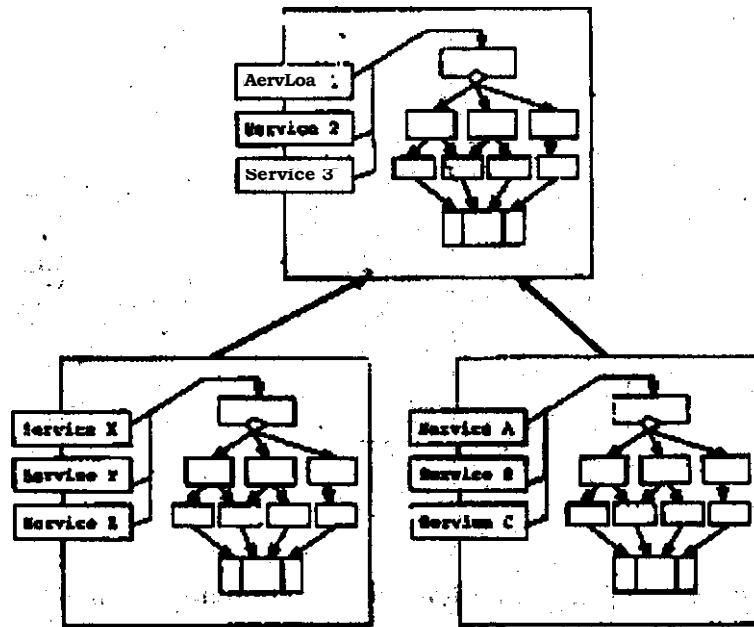
object-oriented system built on a traditional library of functions: This is a common case when you are building an object-oriented system using the facilities of a conventional library of functions such as Xlib - the C function libin Ty for the X Window System. As illustrated in

Figure 5.8, the objects call functions from the library to do their work, but the application is built on the objects.

3.2,5 Using Physical Metaphors: The Desktop

One way to design software is to find the right physical metaphor for the various features of the software you are designing. For example, the user interface in Apple Macintosh computers uses the desktop metaphor. You are supposed to view the display as desktop containing folders and files of all information laid out like paper. You can shuffle these "papers" around, keeping in view the one you are currently using. To use a document, you simply open it and the right application gets started. The desktop metaphor is extended to the

point that there is even a trashcan where you discard files and folders that you no longer need. As with a real world trashcan, you have to empty this trashcan before the items are actually deleted from the system's disk.



**Figure 945th1ect-oriented systemi with top-down design internal
to obiece**

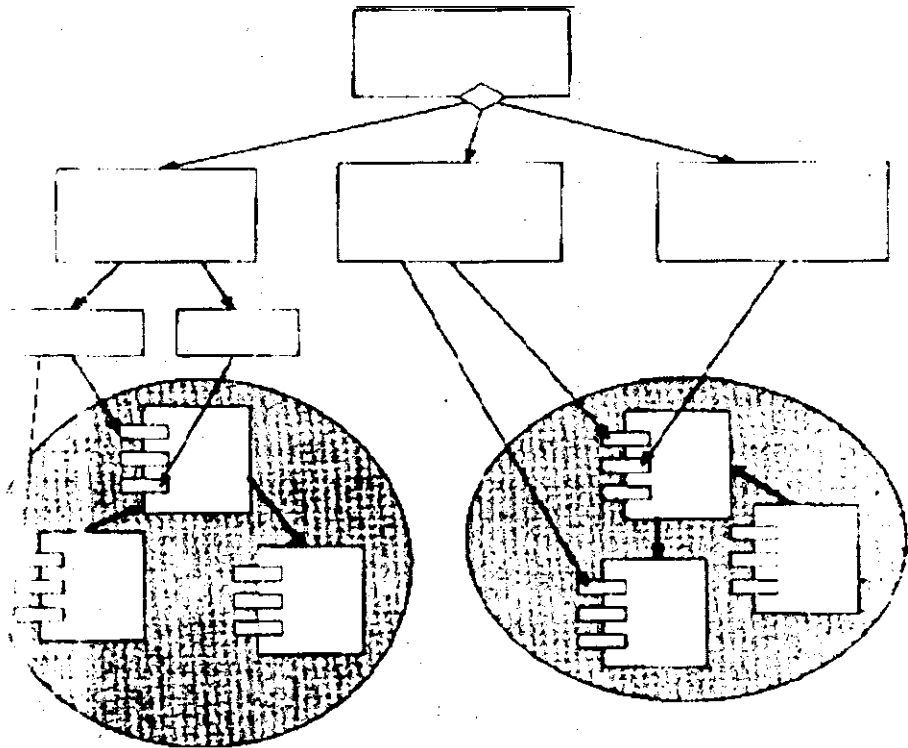


Figure 9.7 top-down design with functionality from objects

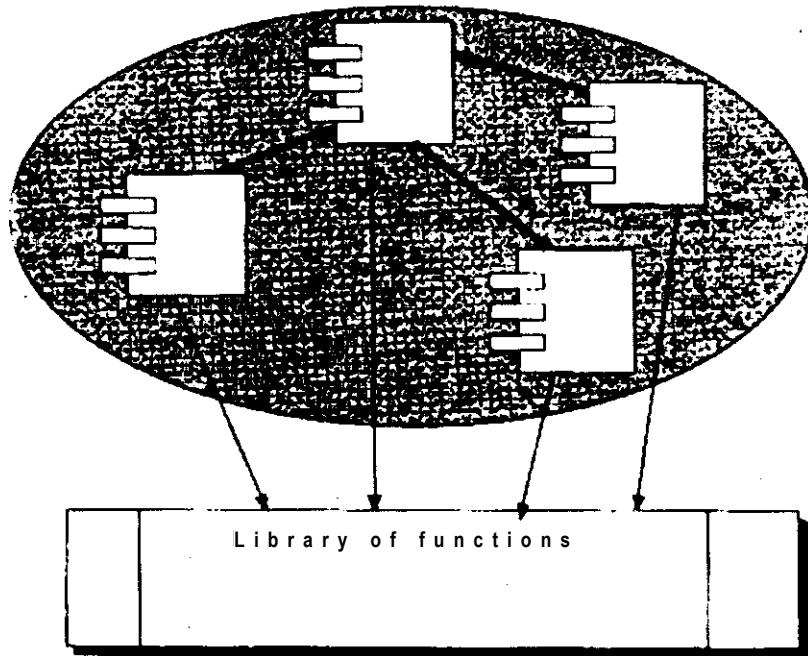


Figure 9.8 Object-oriented system built atop the functional layer

The advantage of using such physical metaphors is that your job as a designer metaphor. You do not necessarily have to use an object-oriented approach to exploit a metaphor, but physical metaphors lend themselves more readily to object-oriented organization. After all, the objects in the software can be the direct counterparts of the objects that are art of the physical metaphor.

The idea of using electronic circuits as a metaphor may seem natural for simulating actual integrated circuits (ICs), but you can use this metaphor to build user interfaces. In fact, the idea of software packaged as ICs was first used by Brad Cox, who also coined the term software-IC. Suppose you are using a windowing system such as the X Window system to implement a user interface. Here is a rough idea of how you might use the electronic circuit metaphor (see Figure 3.9) to implement the user interface:

- Think of each window as an IC with a number of input pins.
- Provide "connector" objects and functions to connect one pin to another. Use a linked list of connectors to handle multiple connections at a pin.
- Allow signals to be sent from output pins to input pins. This means that you need signal objects. Signals arriving at an input pin can be handled by calling a function inside the IC.
- As the user provides input with mouse or keyboard, send signals arriving at an input pin can be handled by calling a function inside the IC.
- As the user provides input with mouse or keyboard, send signals out on appropriate pins to perform the task requested by the user.

Depending on your application, this can turn out to be a useful metaphor. For example, you might exploit such a metaphor in an application with which the user interactively builds a graphical user interface. With such a metaphor, the user can select individual user interface components and connect their input and output pins to build, and interface. You could provide some means for exercising the interface and, when the user is satisfied, allow generation of code that implements the interface.

4.0 Conclusion

Object-oriented programming (OOP) refers to the implementation of programs using objects, preferably in an object-oriented programming language such as C++. Object-oriented analysis (OOA) refers to methods of specifying the requirements of the software in terms of real-world objects - their behaviour and their interactions. Object-oriented design (OOD), on the

other hand, turns the software requirements into specifications for objects and derives class hierarchies from which the objects can be created. OOD methods usually use a diagramming notation to represent the class hierarchy and to express the interaction among objects.

6.0 Tutor Marked Assignment

- 1 Define the following
 - a, Data Flow Diagram
 - b. Data Dictionary
 - c. Minispecification
- 2 Outline the problems with the top-down approach so prevalent in structured techniques as pointed out by Bertrand Meyer.

7.0 References And Other Resources

Object-oriented Programming in C++. Nabajyoti Bark Kati,
Prentice-Hall of India Private Limited, New Delhi - 110 001, 2001.

UNIT 10: OBJECT-ORIENTED PROGRAMMING IN JAVA

TABLE OF CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.1 A Checklist
- 3.2 Creating Your First Application
- 3.3 Creating Your First Applet
- 3.4 About The Java Technology
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor Marked Assignment
- 7.0 References & Other materials

1.0 Introduction

In this Unit you will start with a checklist of what you need to write your first Java Programme and explanations Of error messages you may encounter.

2.0 Objectives

After completing this Unit you should be able to;

- Identify what you need to write your first Java program
- Create an application
- Create an applet

3.1 A Checklist

To write your first program, you need:

1. **The Java™ 2 Platform, Standard Edition.** is can "be purchased on a CD-ROM. You can also download the SDK from <http://java.sun.com/j2se/1.4/download.html> and consult the installation instructions At <http://java.sun.com/j2se/1.4/install-windows.html>. (Make sure you download the SDK, not the JRE.)
2. **A text editor.** You can use any ASCII or Text Editor Notepad, the simple editor' included with the Windows platforms. To find Notepad, from the Start menu select Programs > Accessories > Notepad.

3.2 Creating Your First Application

Why Bytecodes are Cool

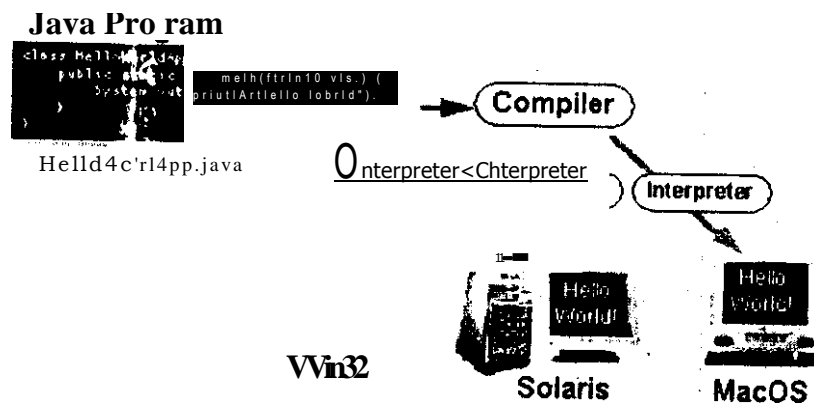
So, you've heard that With the Java programming language, you can "write once., run anywhere." This means that when you compile your program, you

don't generate instructions for one specific platform. Instead, you generate Java bytecodes, which are instructions for the Java Virtual Machine (Java VM). If your platform--whether it's Windows, UNIX, MacOS, or an Internet browser--has the Java VM, it can understand those bytecodes.

Your first program, Hello WORLDAPP!. To app, will simply display the greeting "Hello world!". To create this program, you will:

- **Create a source file.** A source file contains text, written in the Java ' programming language, that you and other programmers can understand. You can use any text editor to create and edit source files or use Notepad.
- **Compile the source file into a bytecode file.** The compiler, java, takes your source file and translates its text into instructions that the Java Virtual Machine (Java VM) can understand. The compiler converts these instructions into a bytecode file.

Run the program contained in the bytecode file. The Java interpreter installed on your computer implements the Java VM. This interpreter takes your bytecode file and carries out the instructions by translating them into instructions that your computer can understand.



a. Create a Source File.

To create a source file, you *have* two options:

You can save the HelloWorldApp.java file on your computer and avoid a lot of typing. Then, you can go straight to step b.

- **Or, you can follow these longer instructions:**

- 1. Start Notepad or any Text Editor. In a new document, type in the following code:**

```
/*g
 * The HelloWorldApp class implements an application that
 * displays "Hello World!" to the standard output
 */
public class HelloWorldApp {
    public static void Main(String args) {
        // Print "Hello World!"
        System.out.println("Hello World!");
    }
}
```

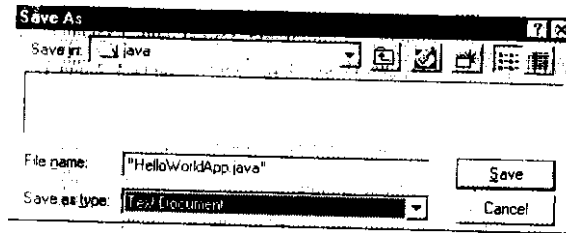
Be Careful When You Type

Type all code, comments, and file names **exactly** as shown. The Java compiler and interpreter are case sensitive, so you must capitalize consistently.

For example, "HelloWorldApp" is not the same as "helloworldapp"

- 2. Save this code to a file. If you are using Notepad, from the menu bar, select File > Save As. In the Save As dialog box:**
 - Using the Save in drop-down menu, specify the folder (directory) where you'll save your file. In this example, the directory is java on the C drive.
 - In the File name text box, type "HelloWorldApp.java", including the double quotation marks.
 - From the Save as type drop-down menu, choose Text

Document.



Click Save, and exit Notepad. b

Compile the Source File.

From the Start menu, select the MS-DOS Prompt application (Windows 95/98) or Command Prompt application (Windows NT). When the application launches, it should look like this:



The prompt shows your current directory. When you bring up the prompt for Windows 95/98, your current directory is usually WINDOWS on your C drive (as shown above) or WINNT for Windows NT. To compile your source code file, change your current directory to the directory where your file is located. For example, if your source directory is java on the C drive, you would type the following command at the prompt and press Enter:

cd c:\java

Now the prompt should change to C: \java>.

Note: To change to a directory on a different drive, you must type an extra command, as shown below:



As shown here, to change to the java directory **on** the D drive, you must reenter the drive, d:

If you enter `dir` at the prompt, yeas-should see your file.

```

MS-DOS Prompt
C:\java\dir

Volume in drive C is DB02
Volume Serial Number is F3C4-E800
Directory of C:\java

<DIR>          07-22-99 11:23:11
<DIR>          07-22-99 11:23:11
4.1.1.001 JAN  272 07-23-99 12:19: Htl oWorldApp.jav
HELLOW1 CIA    478 07-21-99 12:41: a HelloWorldApp.class
                750 bytes
                219,734,592 bytes free

C:\java>

```

Now you can compile. At the prompt, type the following command and press Enter: `Javac HelloWorldApp.java`

If your prompt reappears without error messages, congratulations. You have successfully compiled your program.

Error Explanation

Bad command or file name (Windows 95/98)

The name specified is not recognized as an **internal or external** command, operable program or batch file (*Windows NT*)

If you receive this error, Windows cannot find the Java compiler, *Javac*.

Here's one way to tell Windows where to find *Javac*. Suppose you installed the Java Software Development Kit in C: \jdk1.4. At the prompt you would type the following command and press **Enter**:

C: \jdk1.4\bin\javac HelloWorldApp.java

Note: If you choose this option, each time you compile **or** ran a program, you'll have to preCede your javac and java commands with C : \jdk1.4 \bin

The compiler **has generated a Java byteeadi file, HelloWorldApp.class**. At the **prompt, type dir to see the new file that was generated**

```

C:\java>dir
.1.101t.no in drive C i DPW
Directory of
          (Di 14)          (17-22-99 11:23p .
          67-22-99 11:23p
.11.LLOY 1 ail)          272  WI- 22-99 11 :29 HelloWorld' Jo  ua
          1
          2 nirCJ.>

```

V S . 1.14mitg:Vittitiiii,A1

Now that you have a class file, you can run your

program. **Run the program**

In the same directory, enter at the prompt:

Java HelloWorldApp

Now you should see:

result

Congratulations! Your program works.

Error Explanation

Exception in thread "main" Java. lang.NoClassDefFoundError:
HelloWorldApp

If you receive this error, Java cannot find your bytecode file,
HelloWorldApp . class.

One of the places Java tries to find your bytecode file is your current directory. So, if your bytecode file is in C : java, you should change your current directory to that. To change your directory, type the following command at the prompt and press **Enter**:

```
cd c: java
```

The prompt should change to C : \java>. If you enter "di?" at the prompt, you should see your. java and. class files. Now enter java HelloWorldApp again.

If you still have problems, you might have to change your CLASSPATH variable. To see if this is necessary, try "clobbering" the classpath with the following command:

set CLASSPATH=

Now enter java HelloWorldApp again. If the program works now, you'll have to change your CLASSPATH variable.

Exercise 1.1:

When you compile a program written in the Java programming language, the compiler converts the human-readable source file into platform-independent code that a Java Virtual Machine can understand. What is this platform-independent code called?

Answer : Bytecode.

33 Creating Your First Applet

HelloWorldApp is an example of a Java application, a standalone program. Now you will create a Java applet called HelloWorld, which also displays the greeting "Hello world!". Unlike HelloWorldApp, however, the applet runs in a Java-enabled Web browser such as Hot Java, Netscape Navigator, or Microsoft Internet Explorer.

To create this applet, you'll perform the basic steps as before: create a Java source file; compile the source file; and run the program.

a. Create a Java Source File.

Again, you have two options:

- You can save the files HelloWorld.java and Hello.html on your computer and avoid a lot of typing. Then, you can go straight to step b.

•

O

r, you can follow these instructions: **1. Start NotePad. Type the following code into a new document:**

```
import java.applet.*;
import java.awt.*;

/**
 * The HelloWorld class implementi an applet that
 * simply displays "Hello World!".
 */
```

```
*/  
  
        public class HelloWorld extends Applet  
            public void paint(Graphics g)  
            // Display "Hello World!"  
            g.drawString ("Hello world!", 50, 25);
```

Save this code to a file called HelloWorld.java.

- 2 You also need an HTML file to accompany your applet. Type the following code into a new Notepad document:**

```
<HTML> <HEAD> <TITLE>A  
Simple Program</TITLE>  
</HEAD> <BODY> Here is the  
output of my program:  
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>  
</APPLET>  
</BODY>  
</HTML>
```

Save this code to a file called Hello.html.

b. Compile the Source File.

At the prompt, type the following command and press Return: `javac HelloWorld.java`

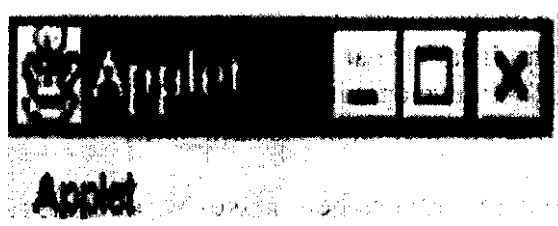
The compiler should generate a Java bytecode file, HelloWorld.class.

c. Run the Program.

Although you can view your applets using a Web browser, you may find it easier to test your applets using the simple application that comes with the Java TM Platform. To view the Hello World applet using appletviewer, enter at the prompt:

`Appletviewer Hello.html`

Now you should see:



Congratulations! Your applet works.

3.4 ABOUT THE JAVA TECHNOLOGY

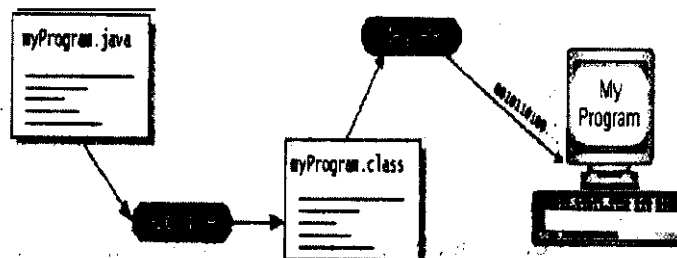
Java technology is both a programming language and a platform.

3.4.1 The Java Programming Language

The Java programming language is a high-level language that can be characterized by all of the **following** buzzwords:

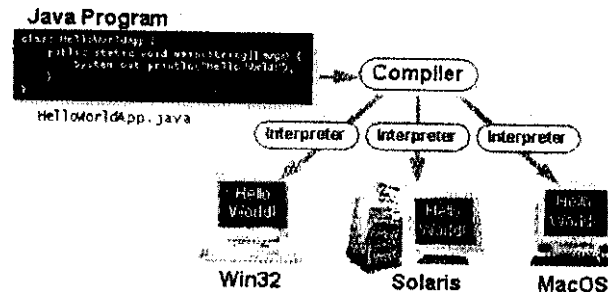
- | | |
|---------------|------------------------|
| Simple | • Architecture neutral |
| Object | • Portable |
| Distributable | • High performance |
| Interoperable | • Multithreaded |
| Robust | • Dynamic |
| Secure | |

With most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. With the compiler, first you translate a program into an intermediate language called Java bytecodes -the platform-independent codes interpreted by the interpreter on the Java platform. The interpreter parses and runs each Java bytecode instruction on the computer. Compilation happens just once; interpretation occurs each time the program is executed. The following figure illustrates how this works.



You can think of Java bytecodes as the machine code instructions for the Java Virtual Machine (Java VM). Every Java interpreter, whether it's a development tool or a Web browser that can run applets, is an implementation of the Java VMA

Java bytecodes help make "write once, run anywhere" possible. You can compile your program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the Java VM. That means that as long as a computer has a Java VM, the same program written in the Java programming language can run on Windows 2000, a Solaris workstation, or on an iMac.



3.4.2 The Java Platform

A platform is the hardware or software environment in which a program runs. 2000, Linux, Solaris, and MacOS. Most platforms can be described as a combination of the operating system and hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

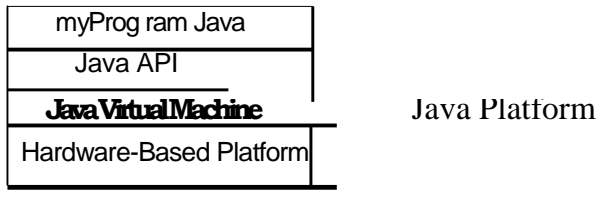
- The Java Virtual Machine (Java VM)
- The **Java** Application Programming Interface (Java API)

You've already been introduced to the Java VM. It's the base for the Java platform and **is** ported onto various hardware-based platforms.

The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into libraries

of related classes and interfaces; these libraries are known as *packages*.

The following figure depicts a program that's running on the Java platform. As the figure shows, the Java API and the virtual machine insulate the program from the hardware.



Native code is code that after you compile it, the compiled code runs on a specific hardware platform. As a platform-independent environment, the Java platform can be a bit slower than native code. However, smart compilers, well-tuned interpreters, and just-in-time bytecode compilers can bring performance close to that of native code without threatening portability.

4.0 Conclusion

Now you have learned how to create your own Java application and your applet. You have learnt how to create a source code file for your application or applet, compile the source code into a bytecode file and run the program contained in the bytecode file.

5.0 Summary

This unit as the title implies is a general introduction to the Java programming language and the Java platform. Subsequent units will go deeper into the characteristics, structure and use of the Java programming language.

6.0 References & Other Resources

www.java.sun.com/docs/books

UNIT 11: A CLOSER LOOK AT THE "HELLOWORLD" SAMPLE

TABLE OF CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.1 The "Hello World" Application
- 3.2 The "Hello World" Applet
- 3.3 Solving Common Compiler and Interpreter Problems
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor Marked Assignment
- 7.0 References & Other Resources

1.0 Introduction

The "Hello World" Application leads you through compiling and running a Java Programme. The unit also introduces some general Java techniques: how to define a class and how to use supporting classes and objects.

The "Hello World" Applet tells you how to compile and run an applet--a Java program to be included in HTML pages and executed in Java-enabled browsers. The lesson also introduces some general Java concepts and techniques: how to create a subclass, what packages are, and how to import classes and packages into a program.

This unit dissects the "Hello World" application and the "Hello World" applet that you've already seen.

2.0 Objectives

After completing this Unit you should be able to:

- Define a Class in Java
- Explain how the main method operates
- Describe a package
- Solve minor compiler and interpreter problems

3.1 The "Hello World" Application

Now that you've seen a Java application (and perhaps even compiled and run it), you might be wondering how it works and how similar it is to other Java applications. Remember that a Java application is a standalone Java program--a program written in the Java language that runs independently of any browser.

Here, again, is the code of the HelloWorld application

```

1/**
 * The HelloWorldApp class implements an application
 * that * simply displays "Hello World!" to the standard
 * output. *1
class HelloWorldApp {
    public static void main (String) { } arg) (
        system. Out. Println (" Hello world )!" // Display the string

```

3.1.1 Comments in Java Code

The **"Hello World"** application has two blocks of comments. The first block, at the top of the program uses `/**` and `*/` delimiters. Later, a line of code is explained with a comment that's marked by `//` characters. The Java language supports a third kind of comment, as well -- the familiar C-style comment, which is delimited with `/*` and

The bold characters in the following listing are comments.

```

/**
 * The HelloWorldApp class implements an application
 * that * simply displays "Hello World!" to the standard
 * output.

```

```

class HelloWorldApp (
    public static void main(String[] args) {
        System.out.println ("Hello World!");
        //Display the string.

```

The Java language supports three kinds of comments:

(i) `/* text */`

The compiler ignores everything from `/*` to `*/`.

(ii) `/**, documentation */`

This indicates a documentation comment (doc comment, for short). The JDK javadoc tool uses doc comments when preparing automatically generated documentation.

(iii) `// text` The compiler ignores everything from `//` to the end of the line.

3.1.2 Defining a Class

In the Java language, each method (function) and variable exists within a class or an object (an instance of a class). The Java language does not support global functions or variables. Thus, the skeleton of any Java program is a class definition.

The first bold line in the following listing begins a class definition block.

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
```

```
class HelloWorldApp {
```

```
    public static void main(Stringn args) {
        System. out. Println ("Hello World!"); //Display the string.
```

A class--the basic building block of an object-oriented language such as Java--is a template that describes the data and behavior associated with instances of that class. When you instantiate a class you create an object that looks and feels like other instances of the same class. The data associated with a class or object is stored in variables; the behavior associated with a class or object is implemented with methods. Methods are similar to the functions or procedures in procedural languages such as C.

Julia Child's recipe for rack of lamb is a real-world example of a class. Her (While both racks of lamb may "look and feel" the same, I imagine that they "smell and taste" different.)

A more traditional example from the world of programming is a class that represents a rectangle. The class would contain variables for the origin of the rectangle, its width, and its height. The class might also contain a method that calculates the area of the rectangle. An instance of the rectangle class would contain the information for a specific rectangle, such as the dimensions of the floor of your office, or the dimensions of this page.

In the Java language, the simplest form of a class definition is

```
class name f
```

```
. . .
```

The keyword `class` begins the class definition for class named `name`. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. The "Hello World" application has no variables and has a single method named `main`.

Exercise 2.1:

Which of the following is not a valid comment:

- a. `/** comment`
- , b. `/ * comment`
- c. `/* comment`
- d. `// comment`

Answer 2.1: c is an invalid comment.

3.1.3 The main Method

The entry point of every Java application is its main method. When you run an application with the Java interpreter, you specify the name of the class that you want to run. The interpreter invokes the main method defined within that class. The main method controls the flow of the program, allocates whatever resources are needed, and runs any other methods that provide the functionality for the application.

The first bold line in the following listing begins the definition of a main method.

*** The HelloWorldApp class implements an application that simply displays "Hello World!" to the standard output.**

```
class HelloWorldApp (
public static void main(String[] args) {
System.out.println ("Hello World!");
//Display the string.
```

Every Java application must, contain a main method whose signature looks like this: `public static void main(Stringn args)`

The method signature for the main method contains three modifiers:

- `public` indicates that the main method can be called by any object.

- static indicates that the main method is a class method.
- void indicates that the main method doesn't return any value.

3.1.31 How the main Method Gets Called

The main method in the Java language is similar to the main function in C and C++. When the Java interpreter executes an application (by being invoked upon the application's controlling class), it starts by calling the class's main method. The main method then calls all the other methods required to run your application.

method, the interpreter refuses to run your program and displays an error message similar to this:

```
In      class NoMain:      void main (String argv []) is
                                not defined
```

3.1.32 Arguments to the main Method

As you can see from the following code snippet, the main method accepts a single argument: an array of elements of type String.

```
public static void main(Stringn args)
```

This array is the mechanism through which the runtime system passes information to your application. Each String in the array is called a command-line argument. Command line arguments let users affect the

- operation of the application without recompiling it. For example, a sorting program might allow the user to specify that the data be sorted in descending order with this command-line argument:

-descending

The "Hello World" application ignores its command-line arguments, so there isn't much more to discuss here.

3.1.4 Using Classes and Objects

The other components of a Java application are the supporting objects, classes, method s, and Java language statements that you write to implement the application.

This section explains how the "Hello World" application uses classes and objects. If you aren't familiar with object-oriented concepts, then you might find this section confusing. If so, feel free to skip ahead to

the next unit (Object-oriented programming concepts) and don't forget to return to finish this section:

The "Helle World" application is about the simplest Java program you can write that actually does something. Because it is such a simple program, it doesn't need to define any classes except for HelloWorldApp. However, most programs that you write will be more complex and require you to write other classes and supporting Java code.

The "Hello Wpi ,cin application does use another class--the System class--that is part of the . API (application programming interface) provided with the Java environment. The System• class provides system-independent access to system-dependent functionality.

The bold code in the following listing illustrates the use of a class variable of the System class, and of an instance method.

In

* The helloWorldApp class implements an application that * simply displays "Hello World!" to the standard output.

```
class HelloWorldApp {
    public static void main(Stringn ergs)
    System.out.println("Hello World!");
    //Display the string.
```

3.141 Using a Class Method or Variable

Let's take a look at the first segment of the statement:

```
System.out.println("Hello World!");
```

The construct System. out is the full name of the out variable in the System class. Notice that the application never instantiates the System class and that out is referred to directly from the class name. This is because out is a class variable--a variable associated with the class rather than with an instance of the class. You can algid associate methods with a *class--class methods*.

To refer to class variables and methods, you join the class name and the name of the class method or class variable together with a period

3.1.42 Using an Instance Method or Variable

Methods and variables that are not class methods or class variables are known as instance methods and instance variables. To refer to instance methods and variables, you must reference the methods and variables from an object.

While `System.out` variable is a class variable, it refers to an instance of the `Print Stream` class (a class provided with the Java development environment) that implements the standard output stream.

When the `System` class is loaded into the application, it instantiates `Print Stream` and assigns the new `Print Stream` object to the `out` class variable. Now that you have an instance of a class, you can call one of its instance methods:

```
System.out.println ("Hello World!");
```

As you can see, you refer to instance methods and variables similarly to the way you refer to class methods and variables. You join an object reference (`out`) and the name of the instance method or variable (`println`) together with a period

The Java compiler allows you to cascade references to class and instance methods and variables together, resulting in constructs like the one that appears in the sample program:

```
System.out.println("Hello World!");
```

This line of code displays "Hello World!" to the application's standard output stream.

3.2 The "Helloworld" Applet

Now that you've seen a Java applet, you're probably wondering how it works. Remember that a Java applet is a program that adheres to a set of conventions that allows it to run within a Java-compatible browser.

Here again is the code for the "Hello World" applet.

```
import java.applet.Applet;  
import java.awt.Graphics;  
  
public class HelloWorld extends Applet (  
    public void paint(Graphics g)  
g.drawString("Hello world!", 50, 25);
```

3.2.1 Importing Classes and Packages

The code above starts off with two import statements. By importing classes or packages, a class can more easily refer to classes in other packages. In the Java language, packages are used to group classes, similar to the way libraries are used to group C functions.

The first two lines of the following listing import two classes used in the applet: Applet and Graphics.

```
import java.appletApplet;
import java.awtGraphics;
```

```
public class Hello World extends Applet (
public void paint(Graphics g)
g.drawString("Hello world!", 50,25);
{
```

If you removed the first two lines, the applet could still compile and run, but only if you changed the rest of the code like this:

```
public class HelloWorld extends
Java. applet. Applet
public void paint(java.awt.Graphics g)
    g.drawString("Hello world!", 50, 25);
```

As you can see, importing the Applet and Graphics classes lets the program refer to them later without any prefixes. The `Java. applet.` and `java. awt .` prefixes tell the compiler which packages it should search for the Applet and Graphics classes. Both the `java . app let` and `java . awt` packages are part of the core Java API -- API that every Java program can count on being in the Java environment. The `java. applet` package contains classes that are essential to Java applets. The `java . awt` package contains the most frequently used classes in the Abstract Window Toolkit (AWT), which provides the Java graphical user interface (GUI).

You might have noticed that the `Helloworldapp` example uses the `System` class without any prefix, and yet does not import the `System` class. The reason is that the `System` class is part of the `java. lang` package, and everything in the `Java. lang` package is automatically imported into every Java program:

Besides importing individual classes, you can also import entire packages. Here's an example:

```
import java. applet.*;
import java. awt.*;

public class HelloWorld extends Applet (
public void paint(Graphics g) {
g.drawString("Hello world!", 50, 25);
```

In the Java language, every class is in a package. If the source code for a class doesn't have a package statement at the top, declaring the package the class is in, then the class is in the default package. Almost all of the example classes in this unit are in the default package.

Within a package, all classes can refer to each other without prefixes. For example, the java. awt Component class refers to the java. awt Graphics class without any prefixes, without importing the Graphics class.

3.2.2 Defining an Applet Subclass

Every applet must define a subclass of the Applet class. In the "Hello World" applet, this subclass is called HelloWorld. Applets inherit a great deal of functionality from the Applet class, ranging from communication with the browser to the ability to present a graphical user interface (GUI).

The first bold line of the following listing begins a block that defines the HelloWorld class.

```
import java. applet. Applet;
import java. awt. Graphics;

public class HelloWorld extends Applet
public void paint(Graphics g) (
g.drawString("Hello world!", 50, 25);
```

The extends keyword indicates that HelloWorld is a subclass of the class whose name follows: Applet. If the term subclass means nothing

to you, you'll learn about it soon in Object-Oriented Programming Concepts in Java.

From the Applet class, applets inherit a great deal of functionality. Perhaps most important is the ability to respond to browser requests. For example, when a Java-capable browser loads a page containing an applet, the browser sends a request to the applet, telling the applet to initialize itself and start executing. ,

An applet isn't restricted to defining just one class. Besides the necessary Applet use a class, the application that's executing the applet first looks on the local host for the class. If the class isn't available locally, it's loaded from the location that the Applet subclass originated from.

3.2.3 Implementing Applet Methods

The HelloWorld applet implements just one method, the paint method. Every applet must implement at least one of the following methods: init, start, or paint. Unlike Java applications, applets do not need to implement a main method.

The bold lines of the following listing implement the paint method.

```
import java.applet.Applet;  
import java.awt.Graphics;  
  
public class HelloWorld extends Applet {  
    public void paint(Graphics g)  
    g.drawString("Hello world!", 50, 25);  
}
```

Every applet must implement one or more of the init, start, and paint methods.

Besides the init, start, and paint methods, applets can implement two more methods that the browser calls when a major event occurs (such as leaving the applet's page): stop and destroy. Applets can implement any number of other methods, as well.

Returning to the above code snippet, the Graphics object passed into the paint method represents the applet's onscreen drawing context. The first argument to the Graphics drawstring method is the string to draw onscreen. The second and third arguments are the (x,y) position of the lower left corner of the text onscreen. This applet draws the string "Hello world!" starting at location (50,25). The applet's

coordinate system starts at (0,0), which is at the upper left corner of the applet's display area.

3.2.4 Running an Applet

Applets are meant to be included in HTML pages. Using the `<APPLET>` tag, you specify (at a minimum) the location of the Applet subclass and the dimensions of the applet's onscreen display area. When a Java-capable browser encounters an `<APPLET>` tag, it reserves onscreen space for the applet, loads the Applet subclass onto the computer the browser is executing on, and creates an instance of the Applet Subclass.

The bold lines of the following listing comprise the `<APPLET>` tag that includes the "Hello World" applet in an HTML page.

```
<HTML>  <HEAD>  <TITLE>  A
Simple Program </TITLE>
</HEAD>
<BODY>
```

Here is the output of my program:

```
<APPLET      CODE='HelloWorld.class'      WIDTH=150
HEIGHT=25> </APPLET>
                </BODY>
                </HTML>
```

The above `<APPLET>` tag specifies that the browser should load the class whose compiled code is in the file named `HelloWorld.class`. The browser looks for this file in the same directory as the HTML document that contains the tag.

When the browser finds the class file, it loads it over the network, if necessary, onto the computer the browser is running on. The browser then creates an instance of the class. If you include an applet twice in one page, the browser loads the class file once and creates two instances of the class.

The `WIDTH` and `HEIGHT` attributes are like the same attributes in an `` tag: They specify the size in pixels of the applet's display area. Most browsers do not let the applet resize itself to be larger or smaller than this display area. For example, every bit of drawing that the "Hello World" applet does in its `paint` method occurs within the

150x25-pixel display area that the above <APPLET> tag reserves for it.

3.3 Solving Common Compiler and Interpreter Problems

If you're having trouble compiling your Java source code or running your application, this section might be able to help you. If nothing in this section helps, please refer to the documentation for the compiler or interpreter you're using.

Compiler Problems

Syntax Errors

If you mistype part of a program the compiler may issue a syntax error. The message usually displays the type of the error, the line number where the error was detected, the code on that line, and the position of the error within the code. Here's an error caused by omitting a semicolon (;) at the end of a statement:

```
testing.java:14: ';' expected.
System.out.println("Input has " + count + " chars."
```

1 error

Sometimes the compiler can't guess your intent and prints a confusing error message or multiple error messages if the error cascades over several lines. For example, the following code snippet omits a semicolon (;) from the bold line:

```
while (System.in.read() != -1)
    count++
System.out.println("Input has" + count + "chars.");
```

When processing this code, the compiler issues two error messages:

```
testing.java:13: Invalid type expression.
count++
```

```
testing.java:14: Invalid declaration.
System.out.println("Input has " + count + " chars.");
```

2 errors

The compiler issues two error messages because after it processes `count++`, the compiler's state indicates that it's in the middle of an expression. Without the semicolon, the compiler has no way of knowing that the statement is complete.

If you see any compiler errors, then your program did not successfully compile, and the compiler did not create a . class file. Carefully verify the program, fix any errors that you detect, and try again.

Semantic Errors

In addition to verifying that your program is syntactically correct, the compiler checks for other basic correctness. For example, the compiler warns you each time you use a variable that has not been initialized:

```
testing.java:13: Variable count may not have been initialized.  
    count++
```

```
testing.java:14: Variable count may not have been initialized.
```

```
System.out.println("Input has" + count + " chars.");
```

2 errors

Again, your program did not successfully compile, and the compiler did not create a . class file. Fix the error and try again.

Interpreter Problems

Can't Find Class

A common error of beginner Java programmers using the UNIX or Windows 95/NT JDK is to try to interpret the . class file created by the compiler. For example, if you try to interpret the file HelloWorldApp.class rather than the class HelloWorldApp, the interpreter displays this error message:

```
Can't find class HelloWorldApp.class
```

The argument to the Java interpreter is the name of the class that you want to use, not the filename.

The main Method Is Not Defined

The Java interpreter requires that the class you execute with it have a method named main, because the interpreter must have somewhere to begin execution of your Java application.

If you try to run a class with the Java interpreter that does not have a main method, the interpreter prints this error message:

```
In class classname:  
void main(String argv[]) is not defined
```

In the above message, class name is The name of the class that you tried to run.

Changes to My Program Didn't Take Effect

Sometimes when you are in the edit/debug/run cycle, it **appears that your** changes to an application didn't take effect -- a print statement **isn't printing**, for example. This is common when running Java applications **on MacOS** using Java Runner. If you recompile a . class file, you must quit **Jaya Runner** and bring it up again, since Java Runner does not reload classe\$.

4.0 Conclusion

In this unit you have learned how to define a class and how to use supporting classes and objects. You have also learned how to create a subclass, what packages are, and how to import classes and packages into a program.

5.0 Summary

You have learnt some general Java techniques in this unit, the subsequent units will throw more light on these techniques.

6.0 Tutor Marked Assignment

1. Change the "HelloWorldAvpdava" program so that it displays Hola Mundo! instead of Hello World!.
2. This modified version of HelloworldApp below has an error. Fix the error so that the program successfully compiles and runs. What was the error?

tic*

* The HelloWorldApp class implements an application that * simply displays "Hello World!" to the standard output.

*/

```
class HelloWorldApp
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
```

3. How do you run an applet ?

7.0 References & Other Resources

. www."java.sun.com/docs/books

UNIT 12: OBJECT-ORIENTED PROGRAMMING CONCEPTS IN JAVA**TABLE OF CONTENTS**

1.0	Introduction
2.0	Objectives
31	What Is An Object
3.2	What Is A Message
33	What Is A Class
3.4	Object Vs. Classes
35	What Is Inheritance
36	What Is An Interface
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	References & Other Resources

10 Introduction

If you've never used an object-oriented language before, you need to understand the underlying concepts before you begin writing code. You need to understand what an object is, what a class is, how objects and classes are related, and how objects communicate by using messages. This unit describes the concepts behind object-oriented programming.

20 Objectives

By the end of this unit, you should be able to

- Define an object, class and message
- Explain Inheritance
- Describe an Interface

31 What Is an Object?

Objects are key to understanding object-oriented technology. You can look around you now and see many examples of real-world objects: your dog, your desk, your television set, your bicycle.

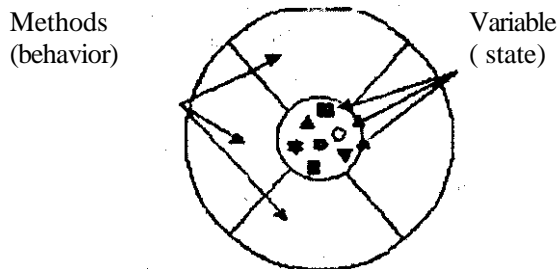
These real-world objects share two characteristics: They all have state and behaviour. For example, dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching, and wagging tail). Bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Software objects are modeled after re:[I-world objects in that they too have state and behaviour. A software object maintains its state in one or more variables. A variable is an item of data named by an identifier. A software object implements its behavior with methods. A method is a function (subroutine) associated with an object.

Definition: An object is a software bundle of variables and related methods.

You can represent rEal-world objects by using software objects. You might want to represent real-world dogs as software objects in an animation program or a real-world bicycle as a software object in the program that controls an electronic exercise bike. You can also use software objects to model abstract concepts. For example, an event is a common object used in GUI window systems to represent the action of a user pressing a mouse button or a key on the keyboard.

The following illustration is a common visual representation of a software object:



Everything that the software object knows (state) and can do (behaviour) is expressed by the variables and the methods within that object. A software object that modeled your real-world bicycle would have variables that indicated the bicycle's current state: its speed is 10 mph, its pedal cadence is 90 rpm, and its current gear is the 5th gear. These variables are formally known as instance variables because they contain the state for a particular bicycle object, and in object-oriented terminology, a particular object is called an instance.

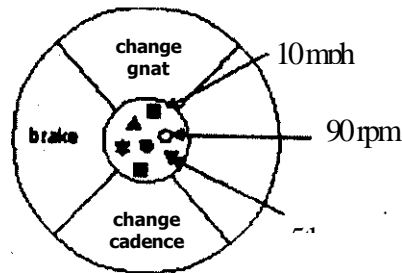
Exercise 3.1

Software objects are modeled after real-world objects in that they too have state behaviour.

Answer

A software object maintains its state in one or more variables. A software object implements its behavior with methods.

The following figure illustrates a bicycle modeled as a software object.



In addition to its variables, the software bicycle would also have methods to brake, change the pedal cadence, and change gears. (The bike would not have a method for changing the speed of the bicycle, as the bike's speed is just a side effect of what gear it's in, how fast the rider is pedaling, whether the brakes are on, and how steep the hill is.) These methods are formally known as instance methods because they inspect or change the state of a particular bicycle instance.

The object diagrams show that the object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. Packaging an object's variables within the protective custody of its methods is called encapsulation. This conceptual picture of an object—a nucleus of variables packaged within a protective membrane of methods—is an ideal representation of an object and is the ideal that designers of object-oriented systems strive for. However, it's not the

whole story. Often, for practical reasons, an object may wish to expose some of can specify one of four access levels for each of its variables and methods. The access level determines which other objects and classes can access that variable or method. Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

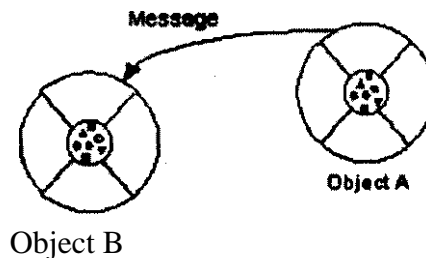
- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.

- Information hiding: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike to use it.

3.2 What Is a Message?

A single object alone is generally not very useful. Instead, an object usually appears as a component of a larger program or application that contains many other objects. Through the interaction of these objects, programmers achieve higher-order functionality and more complex behavior. Your bicycle hanging from a hook in the garage is just a bunch of titanium alloy and rubber; by itself, the bicycle is incapable of any activity. The bicycle is useful only when another object (you) interacts with it (pedal).

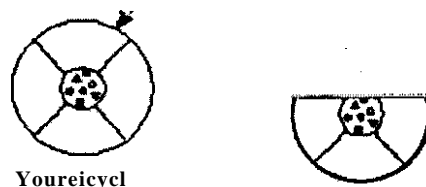
Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B



Sometimes, the receiving object needs more information so that it knows exactly what to do; for example, when you want to change gears on your bicycle, you have to indicate which gear you want. This information is passed along with the message as parameters.

The next figure shows the three components that comprise a message:

1. The object to which the message is addressed (Your Bicycle)
2. The name of the method to perform (change Gears)
3. Any parameters needed by the method (lower Gear)
change Gears(lowerGear)



These three components are enough information for the receiving object to perform the desired method. No other information or context is required.

Messages provide two important benefits.

- An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.
- Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

33 What Is a Class?

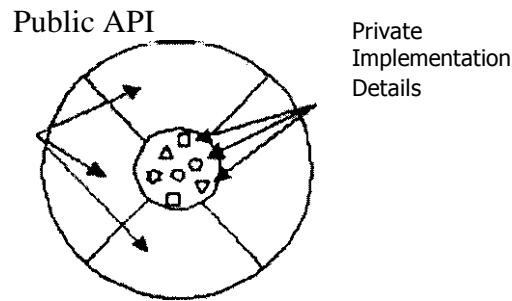
In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an instance of the class of objects known as bicycles. Bicycles have some state (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's state is independent of and can be different from that of other bicycles.

When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics, building many bicycles from the same blueprint. It would be very inefficient to produce a new blueprint for every individual bicycle manufactured.

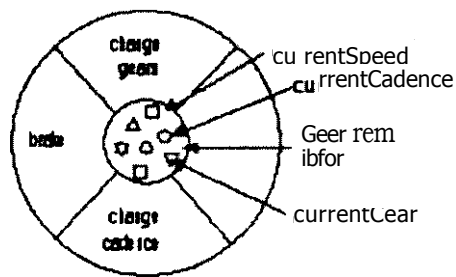
In object-oriented software, it's also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. Like the bicycle manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects. A software blueprint for objects is called a class.

/

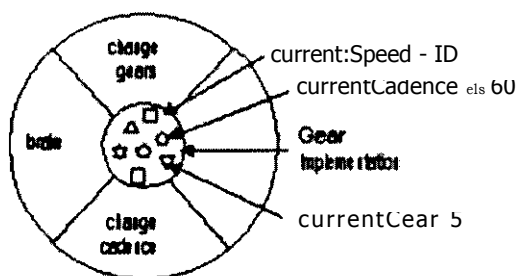
Definition: A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.



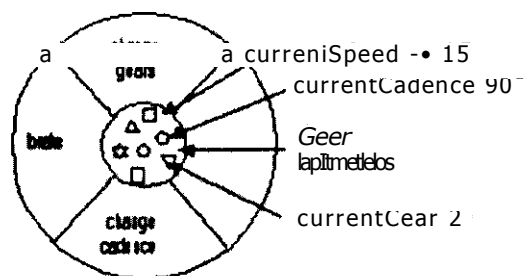
The class for our bicycle example would declare the instance variables necessary to contain the current gear, the current cadence, and so on, for each bicycle object. The class would also declare and provide implementations for the instance methods that allow the rider to change gears, brake, and change the pedaling cadence, as shown in the next figure.



After you've created the bicycle class, you can create any number of bicycle enough memory for the object and all its instance variables. Each instance gets its own copy of all the instance variables defined in the class.



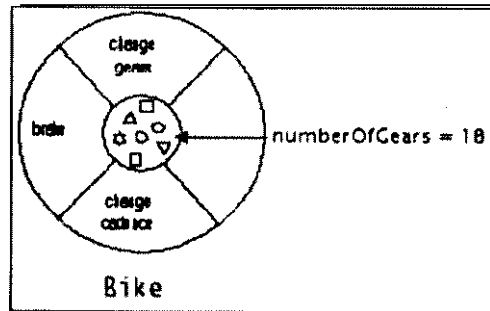
My Bike



YourBike

In addition to instance variables, classes can define class variables. A class variable contains information that is shared by all instances of the class. For example, suppose that all bicycles had the same number of gears. In this case,

defining an instance variable to hold the number of gears is inefficient; each instance would have its own copy of the variable, but the value would be the same for every instance. In such situations, you can define a class variable that contains the number of gears. All instances share this variable. If one object changes the variable, it changes for all other objects of that type. A class can also declare class methods. You can invoke a class method directly from the class, whereas you must invoke instance methods on a particular instance.



Your Bike

Class

Instance of a Class

34 Objects vs. Classes

You probably noticed that the illustrations of objects and classes look very similar. And indeed, the difference between classes and objects is often the source of some confusion. In the real world, it's obvious that classes are not themselves the objects they describe: A blueprint of a bicycle is not a bicycle. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it's also because the term "object" is sometimes used to refer to both classes and instances.

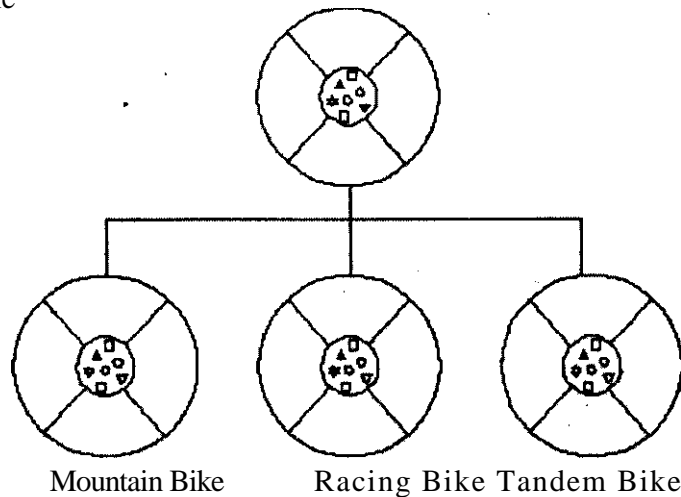
In the figures, the class is not shaded, because it represents a blueprint of an object rather than an object itself. In comparison, an object is shaded, indicating that the object exists and that you can use it.

35 What Is Inheritance?

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a penny-farthing is, if I told you it was a bicycle, you would know that it had two wheels, handle bars, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all kinds of bicycles. In object-oriented terminology, mountain bikes, racing bikes, and tandems are all subclasses of the bicycle class. Similarly, the bicycle class is the superclass of mountain bikes, racing bikes, and tandems. This relationship is shown in the following figure.

Bicycle



Each subclass inherits state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share behaviors: braking and changing pedaling speed, for example.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could use those new gears.

You are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more specialized its behavior.

The object class is at the top of class hierarchy, and each class is its descendant (directly or indirectly). A variable of type Object can hold a reference to any object, such as an instance of a class or an array. Object provides behaviors that are required of all objects running in the Java Virtual Machine. For example, all classes inherit Object's toString method, which returns a string representation of the object.

Inheritance offers the following benefits:

- Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.
- Programmers can implement super classes Called abstract classes that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

36 What Is an Interface?

In English, an interface is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people, and the protocol of behavior enforced in the military is the interface between people of different ranks. Within the Java programming language, an interface is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol (an agreed on behavior). In fact, other object-oriented languages have the functionality of interfaces, but they call their interfaces protocols.

The bicycle class and its class hierarchy defines what a bicycle can and cannot do in terms of its "bicycle ness." But bicycles interact with the world on other terms. For example, a bicycle in a store could be managed by an inventory program. An inventory program doesn't care what class of items it manages as long as each item provides certain information, such as price and tracking number. Instead of forcing class relationships on otherwise unrelated items, the inventory program sets up a protocol of communication. This protocol comes in the form of a set of constant and method definitions contained within an interface. The inventory interface would define, but not implement, methods that set and get the retail price, assign a tracking number, and so on.

To work in the inventory program, the bicycle class must agree to this protocol by implementing the interface. When a class implements an interface, the class agrees to implement all the methods defined in the interface. Thus, the bicycle class would provide the implementations for the methods that set and get retail price, assign a tracking number, and so on.

You use an interface to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship.
- Declaring methods that one or more classes are expected to implement.
- Revealing an object's programming interface without revealing its

class.

4.0 Conclusion

You now know what objects are. How objects communicate through messages and how objects differ from classes. You have learned how classes can be defined in terms of other classes through inheritance. You have also learned how unrelated classes communicate with each other in the Java language through interfaces.

5.0 Summary

In this unit, you have learned about object-oriented concepts. The next unit will explain how these concepts translate into Java codes.

6.0 Tutor Marked Assignment

1. Software objects interact and communicate with each other by sending messages to each other, Explain.
2. Differentiate between Objects and Classes.

7.0 References And Other Resources

www.java.sun.com/docs/books

UNIT 13: TRANSLATING CONCEPTS INTO CODE

TABLE OF CONTENTS

- 1.0 Introduction
- 2.0 Objectives
 - 3.1 Clickme Applet
 - 3.2 Objects In The Clickme Applet
 - 3.3 Classes In The Clickme Applet
 - 3.4 Messages In The Clickme Applet
 - 3.5 Inheritance In The Clickme Applet
 - 3.6 Interfaces In The Clickme Applet
 - 3.7 Api Documentation
- 4.0 Conclusion
- 5.0 Summary
- 7.0 References And Other Resources

1.0 Introduction

Now that you have a conceptual understanding of object-oriented programming let's look at how these concepts get translated into code.

This unit looks at a small applet, and shows you the code that creates objects, implements classes, sends messages, establishes a superclass, and implements an interface.

2.0 Objectives

By the end of this unit, you should be able to

- Identify objects in an applet
- Identify classes in an applet
- Identify messages in an applet
- Identify inheritance in an applet
- Identify interfaces in an applet

3.1 ClickMe Applet

The ClickMe applet is a relatively simple program and the code for it is short. However, if you don't have much experience with programming, you might find the code daunting. I don't expect you to understand everything in this program right away, and this unit won't explain every detail. The intent is to expose you to some source code and to associate it with the concepts and terminology you just learned.

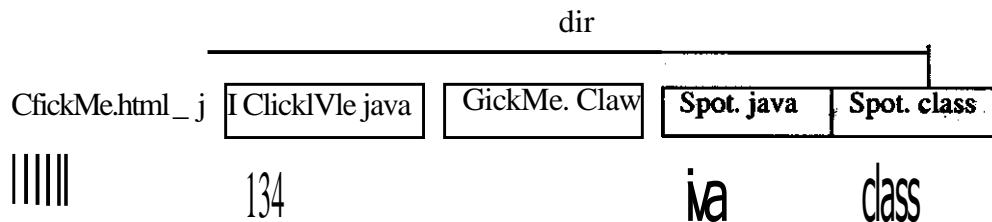
The Source Code and the Applet Tag for ClickMe

To compile this applet you need two source files: ClickMe . java and Spot. java.

To run the applet you need to create an html file with this applet tag in it:

```
<applet code="ClickMe.class"
      width="300" height=" 150">
</applet>
```

Then load the page into your browser or the appletviewer tool. Make sure all the necessary files are in the same directory.



Note: If you're having problems running this example:

- Make sure you copy both Spot. java and Click Me . Java and place them in the same directory
- If you can compile one file but not the other, you probably have a CLASSPATH problem. If "." (indicating your current directory) isn't in your class path, the ClickMe class can't find the Spot class, even though they are in the same directory. Try unsetting CLASSPATH and recompiling, like this:

- set CLASSPATH=
- javac ClickMe.java

If your problem is fixed, then you should modify your global CLASSPATH variable to add "." to it.

3.2 Objects in the ClickMe Applet

Many objects play a part in this applet. The two most obvious ones are the ones you can see: the applet itself and the spot, which is red on-screen.

The browser creates the applet object when it encounters the applet tag in the HTML code containing the applet. The applet tag provides the name of the class from which to create the applet object. In this case, the class name is `ClicicMe..`

The `Clicicme.applet` in turn creates an object to represent the spot on the screen. Every time you click the mouse in the applet, the applet moves the spot by changing the object's x and y location and repainting itself. The spot does not draw itself; the applet draws the spot, based on information contained within the spot object.

Besides these two obvious objects, other, nonvisible objects play a part in this applet. Three objects represent the three colors used in the applet (black, white, and red); an event object represents the user action of clicking the mouse, and so on.

33 **Classes in the ClickMe Applet**

Because the object that represents the spot on the screen is very simple, let's look at its class, named `Spot`. It declares three instance variables: `size` contains the spot's radius, `x` contains the spot's current horizontal location, and `y` contains the spot's current vertical location:

```
public class Spot {  
    //instance variables  
    public int size;  
    public int x, y;  
  
    //constructor  
    public Spot(int intSize)  
    size = intSize;  
    x = 1  
    y = 1;  
}
```

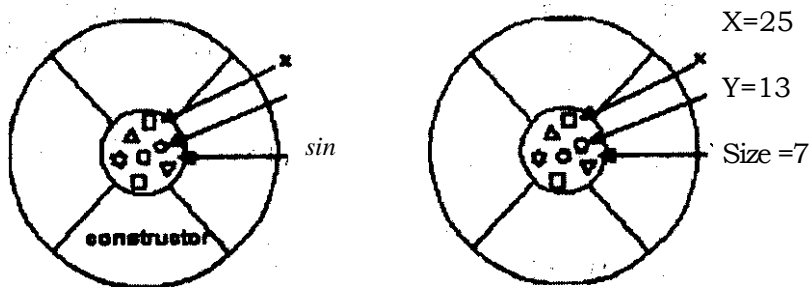
Additionally, the class has a constructor -- a subroutine used to initialize new objects created from the class. You can recognize a constructor because it has the same name as the class. The constructor initializes all three of the object's variables. The initial value of `size` is provided as an argument to the constructor by the caller. The `x` and `y` variables are set to -1 indicating that the spot is not on-screen when the applet starts up.

The applet creates a new spot object When the applet is initialized. Here's the relevant code from the applet class:

```
private Spot spot = null;
private static final int RADIUS = 7;

spot = new Spot(RADIUS);
```

The first line shown declares a variable named `spot` whose data type is `Spot`, the class from which the object is created, and initializes the variable to `null`. The second line declares an integer variable named `RADIUS` whose value is 7. Finally, the last line shown creates the object; `new` allocates memory space for the object. `Spot (RADIUS)` calls the constructor you saw previously and passes in the value of `RADIUS`. Thus the `spot` object's size is set to 7



The figure on the left is a representation of the `Spot` class. The figure on the right is a `spot` object.

3.4 Messages in the ClickMe Applet

As you know, object **A** can use a message to request that object **B** do something, and a message has three components:

1. The object to which the message is addressed
2. The name of the method to perform
3. Any parameters the method needs

Here are two lines of code from the Click Me applet:

```
g.setColor(Color.white);
g.fillRect(0, 0, getSize().width - 1, getSize().height - 1);
```

Both are messages from the applet to an object named `g`—a `Graphics` object that knows how to draw simple on-screen shapes and text. This object is

provided to the applet when the browser instructs the applet to draw itself. The first line sets the color to white; the second fills a rectangle the size of the applet, thus painting the extent of the applet's area white.

The following figure highlights each message component in the first message:

```
g. set Color (Color. White);
```

```
receiving method parameters
object      name
```

3.5 Inheritance in the ClickMe Applet

To run in a browser, an object must be an applet. This means that the object must be an instance of a class that derives from the Applet class provided by the Java platform.

The ClicicMe applet object is an instance of the ClickMe class, which is declared like this:

```
public class ClickMe extends Applet implements
Mouse Listener {
```

The extends Applet clause makes ClicicMe a subclass of Applet. ClickMe inherits a lot of capability from its superclass, including the ability to be initialized, started, and stopped by the browser, to draw within an area on a browser page, and to register to receive mouse events. Along with these benefits, the ClickMe class has certain obligations: its painting code must be in a method called paint, its initialization code must be in a method called init, and so on.

```
    public void init()
        // ClickMe's initialization code here

    public void paint(Graphics g)
        // ClickMe's painting code here
```

3.6 Interfaces in the ClickMe Applet

The ClickMe applet responds to mouse clicks by displaying a red spot at the click location. If an object wants to be notified of mouse clicks, the Java

platform event system requires that the object implement the Mouse Listener interface. The object must also register as a mouse listener.

The Mouse Listener interface declares five different methods each of which is called for a different kind of mouse event: when the mouse is clicked, when the mouse moves outside of the applet, and so on. Even though the applet is interested only in mouse clicks it must implement all five methods. The methods for the events that it isn't interested in are empty.

The complete code for the ClickMe applet is shown below. The code that participates in mouse event handling is bold:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ClickMe extends Applet implements MouseListener
private Spot spot = null;
private static final int RADIUS = 7;

public void init() {
addMouseListener(this);

public void paint(Graphics g) {
// draw a black border and a white background
g.setColor (Color.white);
g.fillRect 0, 0, getSize().width -1, getSize () height
    ;
    g.setColor(Color.black);
    g.drawRect(0, 0, getSize().width — 1, getl,
1) ;

    // draw the spot
    g. setColor (Color. Red;

    g.fillOval (spot.x - RADIUS,
spot.y - RADIUS,
(RADIUS * 2, RADIUS * 2);

    public void mousePressed(MouseEvent event)
    { if (spot == null) {
spot = new Spot(RADIUS);
```

```
spot.x = event.getX();  
spot.y = event.getY();  
repaint();
```

```
public void mouseClicked(MouseEvent event) {}  
public void mouseReleased(MouseEvent event) {}  
public void mouseEntered(MouseEvent event) {}  
public void mouseExited(MouseEvent event) {}
```

3.7 API Documentation

The ClickMe applet inherits a lot of capability from its superclass. To learn more about how ClickMe works, you need to learn about its superclass, Applet. How do you find that information? You can find detailed descriptions of every class in the API documentation, which constitute the specification for the classes that make up the Java platform.

The API documentation for the Java 2 Platform is online at [Java. sun. com](http://java.sun.com).

4.0 Conclusion

This discussion glossed over many details and left some things unexplained, but you should have some understanding now of what object-oriented concepts look like in code. You should now have a general understanding of the following:

- That a class is a prototype for objects
- That objects are created from classes
- That an object's class is its type
- How to create an object from a class
- What constructors are
- How to initialize objects
- What the code for a class looks like
- What class variables and methods are
- What instance variables and methods are

5.0 SumiiiiiiFy

In this unit, you have **learned** how to identify objects, classes, messages, inheritance, interfaces in an applet.

7.0 References & Other Resources

www.java.sun.com/docs/books

UNIT 14: JAVA LANGUAGE BASICS I (VARIABLES AND OPERATORS)

TABLE OF CONTENTS

1.0	Introduction
20	Objectives
31	Variables
3.1.4	Variable Initialization
32	Operators
4.0	Conclusion
5.0	Summary
6.0	Tutor Marked Assignment
7.0	References & Other Resources

1.0 Introduction

The BasicsDemo program that follows adds the numbers from 1 to 10 and displays the **result**.

```
public class BasicsDemo {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int current = 1; current <= 10;  
            current++) {  
            sum += current;  
            System.out.println ("Sum = " + sum);  
        }  
    }  
}
```

The output from this program is:

Sum = 55

Even a small program such as this uses many of the traditional features of the Java programming language, including variable, operators, and control structures. The code might look a little mysterious now. But this unit teaches what you need to know about the nuts and bolts of **the** Java programming language to understand this program.

20 Objectives

After this unit, you should be able to

- Define a variable.

- Describe data types
- List the different primitive data types in JAVA
- Determine the right data type to use for each variable declaration

31 Variables

An object stores its state in variables.

Definition: A *variable* is an item of data named by an identifier.

You must explicitly provide a name and a type for each variable you want to use in your program. The variable's name must be a legal identifier --an unlimited series of Unicode characters that begins with a letter. You use the variable name to refer to the data that the variable contains. The variable's type determines what values it can hold and what operations can be performed on it. To give a variable a type and a name, you write a variable declaration, which generally looks like this:

type name

In addition to the name and type that you explicitly give a variable, a variable has scope. The section of code where the variable's simple name can be used is the variable's scope. The variable's scope is determined implicitly by the location of the variable declaration, that is, where the declaration appears in relation to other code elements.

The MaxVariableDemo program, shown below declares eight variables of different

types within its main method. The variable declarations are bold:

```
public class MaxVariablesDemo
    public static void main(String args{

        // integers
        byte largestByte = Byte.MAX_VALUE;
        short largestShort = Short.MAX_VALUE;
        int largestInteger = Integer.MAX_VALUE;
        long largestLong = Long.MAX_VALUE

        // real numbers
        float largestFloat = Float.MAX_VALUE;
        double largestDouble = Double.MAX_VALUE
```

```

1/ other primitive types
char aChar
Boolean aBoolean = true;

// display them all
System.out.println("The largest byte value Is " +
largestByte);
System.out.println("The largest short value Is " +
largestShort);
System.out.println("The largest integer value Is " +
largestInteger);
System.out.println("The largest long value Is " +
largestLong);
System.out.println("The largest float value Is " +
largestFloat);
System.out.println("The largest double value Is
largestDouble);

if (Character.isUpperCase(aChar)) {
    System.out.println("The character "+ aChar + " is
Upper case
    } else {
        System.out.println("The character "+ aChar + "is
Lower case);
    }
    System.out.println("the value of aBoolean is " +
aBoolean);
}

```

The output from this program is:

```

The largest byte value is 127
The largest short value is 32767
The largest integer value is 2147483647
The largest long value is 9223372036854775807
The largest float value is 3.40282e+38
The largest double value is 1.79769e+308
The character S is upper case.
The value of aBoolean is true

```

- The following sections elaborate on the various aspects of variables, including data types, names, scope, initialization, and final variables. The MaxVariablesDemo program uses two items with which you might not yet be familiar and are not covered in this unit: several constants named MAX

VALUE and an if-else statement. Each MAX VALUE constant is defined in one of the number classes provided by the Java platform and is the largest value that can be assigned to a variable of that numeric type.

3.1.1 Data Types

Every variable must have a data type. A variable's data type determines the values that the variable can contain and the operations that can be performed on it. For example, in the MaxVariablesDemo program, the declaration `int largestInteger` declares that `largestInteger` has an integer data type (`int`). Integers can contain only integral values (both positive and negative). You can perform arithmetic operations, such as addition, on integer variables.

The Java programming language has two categories of data types: primitive and reference. A variable of primitive type contains a single value of the appropriate size and format for its type: a number, a character, or a boolean value. For example, an integer value is 32 bits of data in a format known as two's complement, the value of a `char` is 16 bits of data formatted as a Unicode character, and so on.

Variable Name Value

The following table lists, by keyword, all of the primitive data types supported by Java, their sizes and formats, and a brief description of each. The MaxVariablesDemo program declares one variable of each primitive type.

Primitive Data Type

Primitive Data Types

Keyword (integers)	Description	Size/Format
Byte	Byte-length integer	8-bit two's complement
Short	Short Integer	16-bit two's complement
Int	integer	32-bit two's complement
Long	Long integer	64-bit two's complement
(real numbers)		
Float	Single-precision floating point	32-bit IEEE 754
Double	Double-precision floating point	64-bit IEEE 754
(other types)		
Char	A single character	16-bit Unicode character
Boolean	A Boolean value (true or false)	True or false

Tip: In other languages, the format and size of primitive data types may depend on the platform on which a program is running. In

contrast, the Java programming language specifies the size and format of its primitive data types. Hence, you don't have to worry about system-dependencies.

You can put a literal primitive value directly in your code. For example, if you need to assign the value 4 to an integer variable you can write this:

```
int anInt = 4;
```

The digit 4 is a literal integer value. Here are some examples of literal values of various primitive types:

Examples of Literal Values and Their Data Types

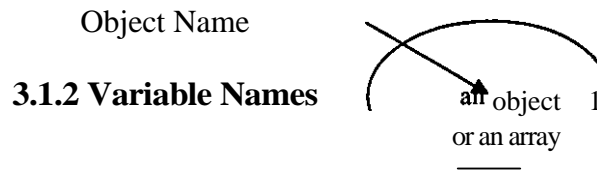
Literal	Data Type
178	int
8864L	Long
37.266	Double
37.266D	Double
87.363F	Float
26.77e3	Double
	Char
True	Boolean
False	Boolean

Generally speaking, a series of digits with no decimal point is typed as an integer as it cannot be confused with the digit '1'. A series of digits with a decimal point is of type double. You can specify a float by putting an 'f' or 'F' after the number. A literal character value is any single Unicode character between single quote marks. The two boolean literals are simply true and false.

Arrays, classes, and interfaces are reference types. The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable.

A reference is called a pointer, or a memory address in other languages. The Java programming language does not support the explicit use of addresses like other languages do. You use the variable's name instead.

E Reference



A program refers to a variable's value by the variable's name. For example, uses the name `largestByte`. A name, such as `largestByte`, that's composed of a single identifier, is called a simple name. Simple names are in contrast to qualified names, which a class uses to refer to a member variable that's in another object or class.

In the Java programming language, the following must hold true for a simple name:

1. It must be a legal identifier. An identifier is an unlimited series of Unicode characters that begins with a letter.
2. It must not be a keyword, a boolean literal (`true` or `false`), or the reserved word `null`.
3. It must be unique within its scope. A variable may have the same name as a variable whose declaration appears in a different scope. In some situations, a variable may share the same name as another variable if it is declared within a nested block of code. (We will cover this in the next section, Scope.)

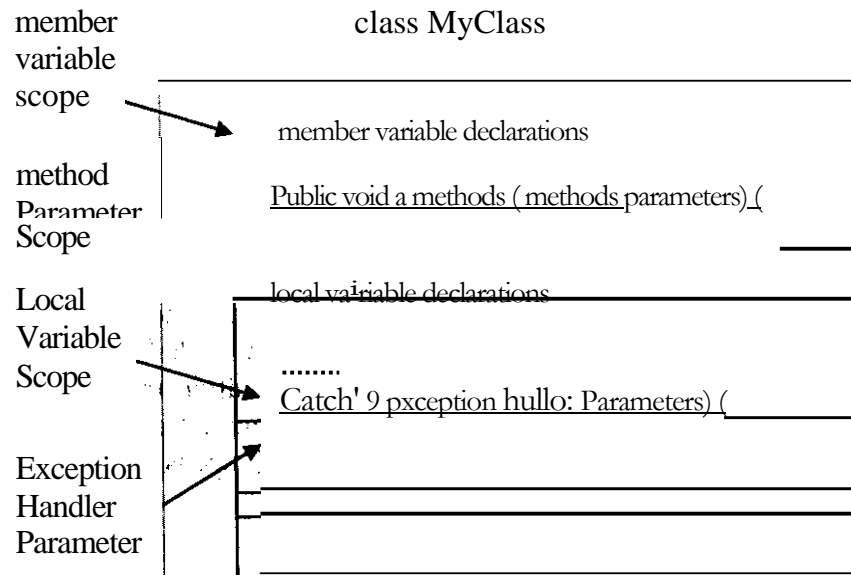
By Convention : Variable names begin with a lowercase letter, and class names words are joined together, and each word after the first begins with an uppercase letter, like this: `invisible`. The underscore character (`_`) is acceptable anywhere in a name, but by convention it is used only to separate words in constants (because constants are all caps by convention and thus cannot be case-delimited).

3.1.3 Scope

to by its simple name. Secondly, scope also determines when the system creates and destroys memory for the variable. Scope is distinct from visibility, which applies only to member variables and determines whether the variable can be used from outside of the class within which it is declared. Visibility is set with an access modifier.

The location of the variable declaration within your program establishes its scope and places it into one of these four categories:

- Member variable
- Local variable
- method parameter
- Member Variable Scope
- exception-handler parameter



A member variable is a member of a class or an object. It is declared within a class but outside of any method or constructor. A member variable's scope is the entire declaration of that class. However, the declaration of a member needs to appear before it is used when the use is in a member initialization expression.

You declare local variables within a block of code. In general, the scope of a local variable extends from its declaration to the end of the code block in which it was declared. In `MaxVariableDemo`, all of the variables declared within the main method are local variables. The scope of each variable in that program extends from the declaration of the variable to the end of the `Main` method --indicated by the first right curly bracket `}` in the program code.

Parameters are formal arguments to methods or constructors and are used to pass data to methods and constructors. The scope of a parameter is the entire method or constructor for which it is a parameter.

Exception-handler parameters are similar to parameters but are arguments to an exception handler rather than to a method or a constructor. The scope of an exception-handler parameter is the code block between { and } that follow a catch statement.

Consider the following code sample:

```
if {
    int i = 17;
```

```
System.out.println("The value of i = " +
i);
// error
```

The final line won't compile because the local variable `i` is out of scope. The scope of `i` is the block of code between the { and } . The `i` variable does not exist anymore after the closing `}` . Either the variable declaration needs to be moved outside of the `if` statement block, or the `println` method call needs to be moved into the `if` statement block.

3.1.4 Variable Initialization

Local variables and member variables can be initialized with an assignment statement when they're declared. The data type of the variable must match the data type of the value assigned to it. The `MaxVariableDemo` program provides initial values for all its local variables when they are declared. The local variable declarations from that program follow, with the initialization code set in **bold**:

```
// integers
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;

// real numbers
float largestFloat = Float.MAX_VALUE;
double largestDouble = Double.MAX_VALUE;

// other primitivditypes
```

```
char aChar = 'S'; boolean aBoolean = true;
```

Parameters and exception-handler parameters cannot be initialized in this way. The value for a parameter is set by the caller.

3.2 Operators

An operator performs a function on one, two, or three operands. An operator that requires one operand is called a unary operator. For example, ++ is a

- unary operator that increments the value of its operand by 1. An operator that requires two operands is a binary operator. For example, = is a binary operator that assigns the value from its righthand operand to its lefthand operand. And finally, a ternary operator is one that requires three operands. The Java programming language has one ternary operator, ? :, which is a
- short-hand if-else statement.

The unary operators support either prefix or postfix notation. Prefix notation means that the operator appears before its operand:

```
operator op          //prefix notation
```

Postfix notation means that the operator appears after its operand:

```
op operator          //postfix notation
```

All of the binary operators use infix notation, which means that the operator appears between its operands:

```
op1 operator op2     //infix notation
```

The ternary operator is also infix; each component of the operator appears between operands:

```
op1 ? op2 : op3       //infix notation
```

In addition to performing the operation, an operator returns a value. The return value and its type depend on the operator and the type of its operands. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and subtraction, return numbers-the result of the arithmetic operation. The data type returned by an arithmetic operator depends on the type of its operands: If you add two integers, you get an integer back. An operation is said to evaluate to its result.

Though there are other types of operators in Java, you will learn about three categories in this unit.

- Arithmetic Operators
- Relational and Conditional Operators
- Assignment Operators

3.2.1 Arithmetic Operators

The Java programming language supports various arithmetic operators for all floating point and integer numbers. These operators are + (addition), - (subtraction), * (multiplication), / (division), and % (modulo).

The following table summarizes the binary arithmetic operations in the Java programming language.

Operator	Use	Description
	Op1 + Op2	Adds op1 and op2
	Op1 - Op2	Subtract op2 from op1
	Op1 * Op2	Multiplies op1 by op2
	Op1 / Op2	Divides op1 by op2
•	Op1 % Op2	Computes the remainder of dividing op1 op2

Here's an example program, `ArithmeticDemo`, that defines two double-precision floating-point numbers and uses the five arithmetic operators to perform different arithmetic operations. This program also uses + to concatenate strings. The arithmetic operations are shown in **bold**:

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

//a few numbers
    int i = 37;

    int j = 42;
    double x = 27.475;
```

```

double y = 7.22; System.out.println("Variable values...");
System.out.println (" i = " + i);
System.out.println C j =
System.out.println (" x =
System.out.println C y =

// adding numbers
system. Out. Println (" Adding ...
system. Out Println (" +j =" + (I +j));
system. Out. Println (" x + y =" + ( x + y));

//subtracting number
system. out. Println (" Subtracting ...");
System,. Out println (" i -j="+(i—j ));
System. Out. Printin ("x —y =" + ( x — y));

//multiplying numbers
System.out.println("Multiplying...");
System.out.println(" i * j =" + (i * j));
System.out.println(" x * y =" + (x * Y));

//dividing numbers
System.out.println("Dividing...");
System.out.println(" i / j =" + (i/j));
ystem.out.println(" x / y =" + (x/y));

//computing the remainder resulting from dividing nos
System.out.println("Computing the remainder...");
System.out.println(" i o j =" + (i j));
System.out.println(" x o y =" + (x Y));

//mixing types
System.out.println("Mixing types...");
System.out.println(" j + y =" + (j + Y));
System.out.println(" i * x =" + (i * x));

```

The output from this program is:

```

Variable values... i = 37
          j = 42
          x = 27.475
          = 7.22

```

Adding...

$$i+j= 79$$

$$x+y= 34.695$$

Subtracting...

$$i - j = -5$$

$$x - y = 20.255$$

Multiplying...

$$i * j = 1554$$

$$x*y= 198.37$$

Dividing...

$$i/j= 0$$

$$x /y = 3.8054$$

Computing the remainder..

$$i \ \% \ j = 37$$

$$x \% y= 5.815$$

Mixing types...

$$j+y=49.22$$

$$i * x = 1016.58$$

Note that when an integer and a floating-point number are used as operands to a single arithmetic operation, the result is floating point. Tilt: integer is implicitly converted to a floating point number before the operation takes place. The following table summarizes the data type returned by the arithmetic operators, based on the data type of the operands. The necessary conversions take place before the operation is performed.

of data Type of Data Type of Operands

result

Long	Neither operand is a float or a double (integer arithmetic); at least one operand is a long.
Int	Neither operand is a float or a double (integer arithmetic): neither operand is a long.
Double	At least one operand is a double.
Float	At least one operand is a float; neither operand is a double

3.2.2 Relational and Conditional Operators

A relational operator compares two values and determines the relationship between them. For example, `!=` returns true if the two operands are unequal. This table summarizes the relational operators:

Operator	Use	Returns true if
	<code>Op1 > op2</code>	Op1 is greater than op2
	<code>Op1 >= op2</code>	Op1 is greater than or equal to op2
	<code>Op1 < op2</code>	Op1 is less than op2
	<code>Op1 <= op2</code>	Op1 is less than or equal to op2
<code>=</code>	<code>Op1 = Op2</code>	Op1 and op2 are equal
<code>!=</code>	<code>Op1 != Op2</code>	Op1 and op2 are not equal

Following is an example, `RelationalDemo`, that defines three integer numbers and uses the relational operators to compare them. The relational operations are shown in bold:

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

//a few Numbers
int i = 37; int j = 42; int k =
42;    System.out.println("Variable
values.
System.out.println(" i = " + i);
System.out.println(" j =
System.out.println(" k = " + k);
// greater than
system. Out. Println (" Greater than ...");
System . out println ("I > J = " + (I>J));
//false
system.. out println (" j>I="+ (j>I ));
// true
system. out. Println ( " k >j = " + ( k >j));
// false, they are equal

// greater than or equal to
```

```

system.. out. Println (" Greater than or equal to ... ");
system.out.println ("I . >=j = " + ( I >=j));
// false
system. out. Println ("j .= I = " + (j >= i));
// true
system. out. Println ( " k >= j = " + (k >= j));
// true

// less than
system. out. Println (" less than...");
system. out.println ("I < j = " + (I <j));
// true
system. out.println ( "j <I = " + ( j <i));
// false
system. .out. println ( "k < j = "+ (k < j));

// false
// Less than or equal to
system. out. Println (" less than or equal to ...");
system. out.println (" I <= j = " + (I <=j));
// true
system .out . println (" j <= I =" + ( j <=i));
// false
system.out . println (" k <= j = " + (k <= j));
//true

// equal to
system. out. Println ( " Equal to ...
system. out.println (" k = =j = " + (k = =j));
//true
system.out. println (" k != j = " + (k !=j));
// false

```

Here's the output from this program:

Variable values... i = 37

j =42 k =42

Greater	than	false
>I=		
> i =	true	
>j =	false	

Greater than or equal to.

i >=j = false

j >. i = true k >. j = true Less than...

```

i < j = true
j < i = false k < j = false
Less than or equal to.
true
j <= i = false k <= j = true
Equal to ...
i == j = false k == j = true
Not equal to ...
i != j = true k != j = false

```

Relational operators often are used with conditional operators to construct more complex decision-making expressions. The Java programming language supports six conditional operators-five binary and one unary--as shown in the following table.

Operator	Use	Returns true if
&&	Op1 && Op2	op1 and op2 are both true, conditionally evaluates op2
	Op1 Op2	either op1 or op2 is true, conditionally evaluates op2
	! op	Op is false
	Op1 & Op2	op1 and op2 are both true, always evaluates op1 and op2
	Op1 Op2	either op1 or op2 is true, always evaluates op1 and op
	Op1 ^ Op2	if op1 and op2 are different--that is if one or the other of the operands is true but not both

One
such
operator
is &&,
which

performs the conditional AND operation. You can use two different relational operators along with & & to determine whether both relationships are true. The following line of code uses this technique to determine whether an array index is between two boundaries. It determines whether the index is both greater than or equal to 0 and less than NUM_ENTRIES, which is a previously defined constant value.

```
0 <= index && index < NUM_ENTRIES
```

Note that in some instances, the second operand to a conditional operator may not be evaluated. Consider this code segment:
(numChars < LIMIT) && (...)

The `&&` operator will return true only if both operands are true. So, if `numChars` is greater than or equal to `LIMIT`, the left-hand operand for `&&` is false, and the return value of `&&` can be determined without evaluating the right-hand operand. In such a case, the interpreter will not evaluate the right-hand operand. This has important implications if the right-hand operand has side effects, such as reading from a stream, updating a value, or making a calculation.

When both operands are boolean, the operator `&` performs the same operation as `&&`. However, `&` always evaluates both of its operands and returns true if both are true. Likewise, when the operands are boolean, `|` performs the same operation as `||`. The `|` operator always evaluates both of its operands and returns true if at least one of its operands is true. When their operands are numbers, `&` and `|` perform bitwise manipulations.

3.2.3 Assignment Operators

You use the basic assignment operator, `=`, to assign one value to another. The `MaxvariableDemo`, program uses `=` to initialize all of its local variables:

```
// integers
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;
// real numbers
float largestFloat = Float.MAX_VALUE;
double largestDouble = Double.MAX_VALUE;
// other primitive types
char aChar = ' ';
boolean aBoolean = true;
```

The Java programming language also provides several shortcut assignment operators that allow you to perform an arithmetic, shift, or bitwise operation and an assignment operation all with one operator. Suppose you wanted to add a number to a variable and assign the result back into the variable, like this:

```
i = i + 2;
```

You can shorten this statement using the shortcut operator `+=`, like this:

```
i += 2;
```

The two previous lines of code are equivalent.

The following table lists the shortcut assignment operators and their lengthy equivalents:

	op1 = op2	op1 = op1 + op2
	op1 « = op2	op1 = op1 « op2
>>=	op1 »= op2	op1 = op1 » op2
>>>=	op1 >>= op2	op1 = op1 >>= op2
op2	op2	

4.0 Conclusion

Variables and operators are essential part of a language that cannot be overlooked. They are the bits and pieces that form the basis from which programs are developed. Understanding a programming language involves a good understanding of these building blocks. It has been clearly demonstrated that there is virtually nothing a programmer can do without using the legal variable name comparing values using the appropriate operator, and placing them where they ought to be.

5.1 Summary

When you declare a variable, you explicitly set the variable's name and data type. The Java programming language has two categories of data types: primitive and reference. A variable of primitive type contains a value.

The location of a variable declaration implicitly sets the variable's scope, which determines what section of code may refer to the variable by its simple name. There are four categories of scope: member variable scope, local variable scope, parameter scope, and exception-handler parameter scope.

You can provide an initial value for a variable within its declaration by using the assignment operator (=).

You can declare a variable as final. The value of a final variable cannot change after it's been initialized.

6.0 Tutor Marked Assignment

- 1 Which of the following are valid variable names?
 - a. Int

- b. AnInt
- c. I
- d. lie. 1
- f. thing1
- g. lthing
- h. ONE-HUNDRED
- i. ONE_ HUNDRED
- j. something2do

- 2 Answer the following questions about the Basics Demo program.
- a. What is the name of each variable declared in the program?
Remember that method parameters are also variables.
 - b. What is the data type of each variable?
 - c. What is the scope of each variable?

- 3 Consider the following code snippet:
- ```
int i = 10;
int n = i++o5;
```

- Ø What are the values of i and n after the code is executed?
- (ii) What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i ) ?
- (iii) What is the value of i after the following code snippet executes?  

```
int i = 8;
i »=2 ;
```
- (iv) What is the value of i after the following code snippet executes?  

```
int i = 17;
i »=1;
```
- (v) Write a program that calculates the number of US dollars equivalent to a given number of French francs. Assume an exchange rate of 6.85062 francs per dollar.

## 7.0 References and other Resources

<http://www.java.sun.com>

## UNIT 15: JAVA LANGUAGE BASICS 2 (EXPRESSIONS AND STATEMENTS)

### TABLE OF CONTENTS

- 1.0 Introduction
- 2.0 Objectives
- 3.1 Expressions
- 3.2 Statements
- 3.3 Blocks
- 3.4 Control Structures
- 4.0 Conclusion
- 5.0 Summary
- 6.0 Tutor Marked Assignment
- 7.0 References & Other Resources

#### 1.0 Introduction

Variables and operators, which you met in the previous unit, are basic building blocks of programs. You combine literals, variables, and operators to form expressions- segments of code that perform computations and return values. Certain expressions can be made into statements-complete units of execution. By grouping statements together with curly braces { and }, you create blocks of code.

#### 2.0 Objectives

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

- Define expressions and statements
- Describe how control structures are used in JAVA programming
- Use block statements
- Control programming sequence in loops.

#### 3.1 Expressions

Expressions perform the work of a program. Among other things, expressions are used to compute and to assign values to variables and to help control the execution flow of a program. The job of an expression is twofold: to perform the computation indicated by the elements of the expression and to return a value that is the result of the computation.

---

Definition: An expression is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value.

---

As discussed in the previous unit, operators return a value, so the use of an operator is an expression. This partial listing of the MaxVariablesDemo program shows some of the program's expressions in bold:

```
// other primitive types
char aChar =
boolean aBoolean = true;

// display them all
System.out.println ("The largest byte value is " +
largestByte);
...

if (Character.isUpperCase(aChar)) {
```

Each of these expressions performs an operation and returns a value.

| Expression                            | Action                                                                                                  | Value Returned                                      |
|---------------------------------------|---------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| aChar =                               | 'S' Assign the character 'S' to the character variable aChar                                            | the value of aChar after the assignment ('S')       |
| "The largest byte value + largestByte | Concatenate the string "The largest byte value is > and the value of largest Byte converted to a string | The resulting string: the largest byte value is 127 |
| Character.isUpperCase(aChar)          | Call the method isUpperCase                                                                             | "The return value of the method: true               |

The data type of the value returned by an expression depends on the elements used in the expression. The expression aChar = ' S' returns a character because the assignment operator returns a value of the same data type as its operands and aChar and ' S' are characters. As you see from the other expressions, an expression can return a boolean value, a string, and so on.

The Java programming language allows you to construct compound expressions and statements from various smaller expressions as long as the data types required by one part of the expression matches the data types of the other. Here's an example of a compound expression:

In this particular example, the order in which the expression is evaluated is unimportant because the results of multiplication is independent of order--the outcome is always the same no matter what order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results depending on whether you perform the addition or the division operation first:

```
x + y / 100 //ambiguous
```

You can specify exactly how you want an expression to be evaluated by using balanced parentheses ( and ) . For example to make the previous expression unambiguous, you could write:

```
(x + y) / 100 //unambiguous, recommended
```

If you don't explicitly indicate the order in which you want the operations in a compound expression to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators with a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Thus, the two following statements are equivalent:

```
x + y / 100
x + (y / 100) //unambiguous, recommended
```

When writing compound expressions, you should be explicit and indicate with parentheses which operators should be evaluated first. This will make your code easier to read and to maintain.

The following table shows the precedence assigned to the operators. The operators in this table are listed in precedence order: the higher in the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with a relatively lower precedence. Operators on the same line have equal precedence.

|                      |                                   |      |
|----------------------|-----------------------------------|------|
| Postfix operator     | } . ( params) expae++ expr--      |      |
| Unary operator       | ++expr - -expr =expr — expr-      |      |
| Creation or cast     | New ( type) expr                  |      |
| Multiplicative       | * / %                             |      |
| Additive             | + -                               |      |
| Shift                | c< >> >>>                         |      |
| Relational           | < > <= >= instance of             |      |
| Equality             |                                   |      |
| Bitwise AND          |                                   | When |
| Bitwise exclusion OR |                                   |      |
| Bitwise inclusive OR |                                   |      |
| Logical AND          | &&                                |      |
| Logical OR           | //                                |      |
| Conditional          |                                   |      |
| Assignment           | = += -= *= /= %= &= A= I= <<= >>= |      |
|                      | >>>=                              |      |

operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated in left-to-right order. Assignment operators are evaluated right to left.

### 3.2 Statements

Statement form a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;):

- Assignment expressions
- Any use of ++ or —
- Method calls
- Object creation expressions

These kinds of statements are called expression statements. Here are some examples of expression statements:

```
aValue = 8933.234;
//assignment statement

aValue++;
//increment statement
System.out.println(aValue);
//method call statement
Integer integerobject = new Integer(4);
```

```
//object creation statement
```

In addition to these kinds of expression statements, there are two other kinds of statements. A declaration statement declares a variable. You've seen many examples of declaration statements.

```
double aValue = 8933.234;
declaration statement
```

A control structure regulates the order in which statements get executed. The for loop and the if statement are both examples of control structures.

### 3.3 Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following listing shows two blocks from the MaxVariableSDemo program, each containing a single statement:

```
if (Character.isUpperCase(aChar))
 System.out.println("The character
" is upper case.");
else (
 System.out.println("The character" + aChar +
"is lower case.");
```

### 3.4 Control Structures

When you write a program, you type statements into a file. Without control structures, the interpreter executes these statements in the order they appear in the file from left to right, top to bottom. You can use control structures in your programs to conditionally execute statements, to repeatedly execute a block of statements, and to otherwise change the normal, sequential flow of control. For example, in the following code snippet, the `if` statement conditionally executes the `System.out.println` statement within the braces, based on the return value of `Character.isUpperCase(aChar)`

```
char c;
```

```
if (Character.isUpperCase(aChar))
 System.out.println("The character" + aChar +
"is uppercase.");
```

The Java programming language provides several control structures, which are listed in the following table.

| Statement Type     | Keyword                         |
|--------------------|---------------------------------|
| Looping            | While, do- while, for           |
| Decision           | If else, switch- case           |
| Exception handling | Try- catch — finally, throw     |
| Branching          | Break, continue, label:, return |

In this course material we will only discuss the first two: Looping and Decision making.

#### The while and do-while Statements

You use a *while* statement to continually execute a block of statements while a condition remains true. The general syntax of the while statement is:

```
while (expression) {
 statement
```

First , the while statement evaluates expression, which must return a boolean value, if the expression returns true, then the while statement executes the statement(s) associated with it. The while statement continues testing the expression and executing its block until the expression returns false.

The example program shown below, called While Demo, uses a while statement to step through the characters of a string, appending each character from the string to the end of a string buffer until it encounters the letter g.

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

String copyFromMe = "Copy this string until you " +
 "encounter the letter 'g'.";
StringBuffer copyToMe = new StringBuffer();

int i = 0;
char c = copyFromMe.charAt(i);
```

```
while (c != 'g')
 copyToMe.append(c);
 c = copyFromMe.charAt(++i);
```

```
System.out.println(copyToMe);
```

The value printed by the last line is: Copy this          strin.

The Java programming language provides another statement that is similar to the while statement--the do-while statement. The general syntax of the do-while is:

```
do {

statement (s)
} while (expression);
```

Instead of evaluating the expression at the top of the loop, do-while evaluates the expression at the bottom. Thus the statements associated with a do-while are executed at least *once*.

Here's the previous program rewritten to use do-while and renamed to DoWhileDemo:

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

String copyFromMe = "Copy this string until you" +
 "encounter the letter 'g'.";
StringBuffer copyToMe = new StringBuffer();

int i = 0;
char c = copyFromMe.charAt(i);

do {
 copyToMe.append(c);
 c = copyFromMe.charAt(++i);
} while (c != 'g'); System.out.println(copyToMe);
```

The value printed by the last line is: Copy this strain.

### **The for Statement**

The for statement provides a compact way to iterate over a range of values. The general form of the for statement can be expressed like this:



```
for (initialization; termination; increment) {
 statement
}
```

The initialization is an expression that initializes the loop-it's executed once at the beginning of the loop. The termination expression determines when to terminate the loop. This expression is evaluated at the top of each iteration of the loop. When the expression evaluates to false, the loop terminates. Finally, increment is an expression that gets invoked after each iteration through the loop. All these components are optional. In fact, to write an infinite loop, you omit all three expressions:

```
for (; ;) // infinite loop
```

Often for loops are used to iterate over the elements in an array, or the over the elements of an array and print them:

```
public class ForDemo
{
 public static void main(String[] args) {
 int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
 2000, 8, 622, 127 };

 for (int i = 0; i < arrayOfInts.length; i++)
 { System.out.println(arrayOfInts[i]); }

 System.out.println();
 }
}
```

The output of the program is: 32      87      3      589      12      1076  
2000 8      622      127.

Note that you can declare a local variable within the initialization expression of a for loop. The scope of this variable extends from its declaration to the end of the block governed by the for statement so it can be used in the termination and increment expressions as well. If the variable that controls a for loop is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring them within the for loop initialization expression limits their life-span and reduces errors.

#### The if else Statements

The if statement enables your program to selectively execute other statements, based on some criteria. For example, suppose that your program

prints debugging information, based on the value of a boolean variable named DEBUG. If DEBUG is true, your as x. might program prints debugging information, such as the value of a variable, such Otherwise, your program proceeds normally. A segment of code to implement this look like this:

```
if (DEBUG) {
 System.out.println("DEBUG: x = " + x);
}
```

This is the simplest version of the if statement: The executed if a condition is true. Generally, the simple form of if can be written like this:

```
If (expression) {
 statement (s)
}
```

What if you want to perform a different set of statements if the expression is false? You use the else statement for that. Consider another example. Suppose that your program needs to perform different actions depending on whether the user clicks the OK button or another button in an alert window. Your program could do this by using an i f statement along with an else statement:

```
// response is either OK or CANCEL depending
// on the button_ that the user pressed

if (response == OK) {
 // code to perform OK action} else {
 // code to perform Cancel action
}
```

The else block is executed if the i f part is false. Another form of the else statement, else if, executes a statement based on another expression. An if statement can have any number of companion else i f statements but only one else. Following is a program, IFELseDemo, that assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on:

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

 int testscore = 76;
 char grade;

 if (testscore >= 90) (
 grade = 'A';
```

```

Grade = 'B';
 I else if (testscore >. 70) 1
Grade = 'C';
 I else if (testscore >= 60) 1
grade = 'D'; } else {
 grade = 'F';
System.out.println("Grade+ grade) ;

```

The output from this program is:

Grade = C

You may have noticed that the value of test score can satisfy more than one of the expressions in the compound i f statement: 76 >= 70 and 76 >= 60. However, as the runtime system processes a compound i f statement such as this one, once a condition is satisfied, the appropriate statements are executed (grade = ' C' and control passes out of the i f statement without evaluating the remaining conditions.

The Java programming language supports an operator, ? :, that is a compact version of an if statement. Recall this statement from the Max Variables Demo program:

```

case.");
 } else {
System.out.println("The character + aChar ++ aChar +
"is lower case.");

```

Here's how you could rewrite that statement using the ? : operator:

```

System.out.println("The character " + aChar + " is " +
(Character.isUpperCase(aChar) ? "upper" "lower") + "case.");

```

The ? : operator returns the string "upper" if the isUpperCase method returns true. Otherwise, it returns the string "lower". The result is concatenated with other parts of a message to be displayed. Using ? : makes sense here because the i f statement is secondary to the call to the println method. Once you get used to this construct, it also makes the code easier to read.

### The Switch Statement

Use the switch statement to conditionally perform statements based on an integer expression. Following is a sample program, Switch Demo, that declares an integer named month whose value supposedly represents the month in a date. The program displays the name of the month, based on the value of month, using the switch statement:

```

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

int month = 8
switch (month)
case 1: System.out.println("January"); break;
case 2: System.out.println("February"); break;
case 3: System.out.println("March"); break;
case4: System.out.println("April"); break;
case 5: System.out.println("May"); break;
 case 6: System.out.println("June"); break;
case 7: System.out.println("July"); break;
case 8: System.out.println("August"); break; break;
case 9: System.out.println("September");
case 10: 10: System.out.println("October");
case 11.out.println("November"); break;
case12: ("December"); break
1
1
1

```

The switch statement evaluates its expression, in this case the value executes the appropriate case statement. Thus, the output of the program could be, you could implement this by using an if statement:

```

int month = 8;
if (month == 1)
{ System.out.println("January");
else if (month == 2) System.out.println("February");

// and so on

```

Deciding whether to use an if statement or a switch statement is a judgment call. You can decide which to use, based on readability and other factors. An if statement can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based only on a single integer value. Also, the value provided to each case statement must be unique.

Another point of interest in the switch statement is the break statement after each case. Each break statement terminates the enclosing switch statement, and the flow of control continues with the first statement following the switch block. The break statements are necessary because without them, the case statements fall through. That is, without an explicit break, control will flow sequentially through subsequent case statements. Following is an example,

Swim t.,-hDemo2, which illustrates why it might be useful to have case statements fall through:

```
public class SwitchDemo2 {
 public static void main(String[] args) {

int month = 2;
int year = 2000;
int numDays = 0;

switch (month)
 case 1:

case 3:
case 5:
case 7:
case 8:
case 10:
case 12:

 numDays = 31;
 break;
case 4: case 6:
case 9: case 11:
 numDays = 30;
 break;
case 2:
 if (((year % 4 == 0) && !(year % 100 == 0))
 11 (year % 400 == 0))
 numDays = 29;
 else
 numDays = 28; break;
System.out.println("Number of Days = " + numDays);
```

The output from this program is:

Number of Days = 29

Technically, the final break is not required because flow would fall out of the switch statement anyway. However, we recommend using a break for the last case statement just in case you need to add more case statements at a later date. This makes modifying the code easier and less error-prone.

Finally, you can use the default statement at the end of the switch to handle all values that aren't explicitly handled by one of the case statements.

```
int month = 8;
```

```
switch (month) (
```

```
case 1: System.out.println("January"); break;
case 2: System.out.println("February"); break;
case 3: System.out.println("March"); break;
case 4: System.out.println("April"); break;
case 5: System.out.println("May"); break;
case 6: System.out.println("June"); break;
case 7: System.out.println("July"); break;
case 8: System.out.println("August"); break;
case 9: System.out.println("September"); break;
case 10: System.out.println("October"); break;
case 11: System.out.println("November"); break;
case 12: System.out.println("December"); break;
```

```
default: System.out.println("Hey, that's not a valid month!"); break;
```

## Summary of Control Structures

constructs, a flexible if-else statement, a switch statement, exception-handling statements, and branching statements.

### 3.4.1 Loops

Use the while statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the top of the loop.

```
while (boolean expression)
 statement (s)
```

Use the do-while statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the bottom of the loop, so the statements within the do-while block execute at least once:

```
do 1
 statement (s)
1 while (expression);
```

The for statement loops over a block of statements and includes an initialization expression, a termination condition expression, and an increment expression.

```
for (initialization ; termination ; increment) { statement (s)
```

### 3.4.2 Decision-Making Statements

The Java programming language has two decision-making statements: if-else and switch. The more general-purpose statement is if; use switch to make multiple-choice decisions based on a single integer value. The following is the most basic if statement whose single statement block is executed if the boolean expression is true:

```
if (boolean expression)
{ statement (s)
```

Here's an if statement with a companion else statement. The if statement executes the first block if the boolean expression is true; otherwise, it executes the second block:

```
if (boolean expression) { statement (s)
} else { statement (s)
```

You can use else if to construct compound if statements:

```
if (boolean expression) {
statement (s)
} else if (boolean expression) {
statement (s)
} else if (boolean expression) {
statement (s)
} else { statement (s)
```

The switch statement evaluates an integer expression and executes the appropriate case statement.

```
switch (integer expression) {
case integer expression:
statement (s)
break;
```

default:

```
statement (s)
break; }
```

## 4.0 Conclusion

The parts that form the basics in JAVA have been dealt with on completion of this unit. With the knowledge gained from unit I to unit 4, it is expected that you are able to write a JAVA code that will be object-based and perform the task it is given effectively. The remaining units combine what you have learnt so far with the power of object-oriented programming.

## 5.0 Summary

An expression is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value. You can write compound expressions by combining expressions as long as the types required by all of the operators involved in the compound expression are correct. When writing compound expressions, you should be explicit and indicate with parentheses which operators should be evaluated first.

If you choose not to use parentheses, then the Java platform evaluates the compound expression in the order dictated by operator precedence. A statement forms a complete unit of execution and is terminated with a semicolon. There are three kinds of statements: expression statements, declaration statements, and control structures. You can group zero or more statements together into a block with curly brackets ( { and } ). Even though not required, we recommend using blocks with control structures even if there's only one statement in the block.

## 6.0 Tutor Marked Assignments

1 What are the data types of the following expressions, assuming that `i` type is `int`?

- a. `i > 0`
- b. `= 0` c. `i++` d. `(float) i`
- e. `i == 0`
- f. `"aString" + i`

2 Consider the following expression: `i--%5>0`

- a) What is the result of the expression, assuming that the value of `i` is initially 10?
- b) Modify the expression so that it has the same result but is easier for programmers to read.



3. Look at the Sort Demo program. What control structures does it contain?
4. What's wrong with the following code snippet:  

```
if 0 = 1) {
/* do something */
```
5. Look at the While demo program and the DoWhileDemo program. What would the output be from each program if you changed the value of each program's  
  
copyFromMe string to golly gee. This is fun. Explain why you think each program will have the predicted output.
6. Consider the following code snippet.  

```
if (aNumber >. 0)
if (aNumber == 0)
System.out.println("first string");
else System.out.println("second string");
System.out.println("third string");
```

  - a. What output do you think the code will produce if aNumber is 3?
  - b. Write a test program containing the code snippet; make aNumber 3. What is the output of the program? Is it what you predicted? Explain why the output is what it is. In other words, what is the control flow for the code snippet?
  - c. Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.
  - d. Use braces { and } to further clarify the code and reduce the possibility of errors in future.

**70****References and other Resources**

<http://www.java.sun.com>

## UNIT 16: OBJECT-ORIENTED PROGRAMMING IN C++

### TABLE OF CONTENTS

|     |                              |
|-----|------------------------------|
| 1.0 | Introduction                 |
| 20  | Objectives                   |
| 31  | History of C++               |
| 3.2 | Programming Using C++        |
| 4.0 | Conclusion                   |
| 5.0 | Summary                      |
| 6.0 | Tutor Marked Assignment      |
| 7.0 | References & Other Resources |

### 1.0 Introduction

More than any other programming language, C++ has become the most-widely used programming language. Its transitional ability from the more traditional concept to the object-oriented concept has helped to make it a language of choice for many.

### 20 Objectives

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

- know the origin of C++
- differentiate between procedural and object-oriented programming in C++.
- Compile a C++ program
- execute/run a C++ compiled file

### 31 History Of C++

#### 3.1.1 How C++ Evolved

As object-oriented analysis, design, and programming began to catch on, Bjarne Stroustrup took the most popular language for commercial software development, C, and extended it to provide the features needed to facilitate object-oriented programming. He created C++, and in less than a decade it has gone from being used by only a handful of developers at AT&T to being the programming language of choice for an estimated one million developers worldwide. It is expected that by the end of the decade, C++ will be the predominant language for commercial software development.

While it is true that C++ is a superset of C, and that virtually any legal C program is a legal C++ program, the leap from C to C++ is very significant. C++ benefited from its relationship to C for many years, as C programmers could ease into their use of C++. To really get the full benefit of C++, however, many programmers found they had to unlearn much of what they knew and learn a whole new way of conceptualizing and solving programming problems.

### **3.1.2 The ANSI Standard**

The idea of creating standards for C++ so that compiler developers will stick to a pattern of design evolved from the wide acceptance of the language and the rate at which different compilers took over the market. The Accredited Standards Committee, operating under the procedures of the American National Standards Institute (ANSI), is working to create an international standard for C++.

The ANSI standard is an attempt to ensure that C++ is portable--that code you write for Microsoft's compiler will compile without errors, using a compiler from any other vendor. For most students of C++, the ANSI standard will be invisible. The standard has been stable for a while, and all the major manufacturers support the ANSI standard.

## **3.2 Programming Using C++**

### **3.2.1 Preparing to Program**

C++, perhaps more than other languages, demands that the programmer design the program before writing it. Trivial problems don't require much design, but students are advised to make it a habit to first design the programs before writing it. Complex problems, however, such as the ones professional programmers are challenged with every day, do require design, and the more thorough the design, the more likely it is that the program will solve the problems it is designed to solve, on time and on budget. A good design also makes for a program that is relatively bug-free and easy to maintain. It has been estimated that fully 90 percent of the cost of software is the combined cost of debugging and maintenance. To the extent that good design can reduce those costs, it can have a significant impact on the bottom-line cost of the project.

The first question you need to ask when preparing to design any program is, "What is the problem I'm trying to solve?" Every program should have a clear, well-articulated goal.

The second question every good programmer asks is, "Can this be accomplished without resorting to writing custom software?" Reusing an old program, using pen and paper, or buying software off the shelf is often a better solution to a problem than writing something new. The programmer who can offer these alternatives will never suffer from lack of work; finding less-expensive solutions to today's problems will always generate new opportunities later.

Assuming you understand the problem, and it requires writing a new program, you are ready to begin your design.

### 3.2.2 Development Environment

This course material makes the assumption that your computer has a mode in which you can write directly to the screen, without worrying about a graphical environment, such as the ones in Windows or on the Macintosh.

Your compiler may have its own built-in text editor, or you may be using a commercial text editor or word processor that can produce text files. The important thing is that whatever you write your program in, it must save simple, plain-text files, with no word processing commands embedded in the text. Examples of safe editors include Windows Notepad, the DOS Edit command, Brief, Epsilon and EMACS. Many commercial word processors, such as WordPerfect, Microsoft Word, and dozens of others, also offer a method for saving simple text files.

The files you create with your editor are called source files, and for C++ they typically are named with the extension CPP, CP, or C. In this book, we'll name all the source code files with the CPP extension, but check your compiler for what it needs.

### 3.2.3 Procedural, Structured, and Object-Oriented Programming

Until recently, programs were thought of as a series of procedures that acted upon data. A procedure, or function, is a set of specific instructions executed one after the other. The data was quite separate from the procedures, and the trick in programming was to keep track of which functions called which other functions, and what data was

changed. To make sense of this potentially confusing situation, structured programming was created.

The idea behind structured programming is as simple as the idea of divide and conquer. A computer program can be thought of as consisting of a set of tasks. Any task that is too complex to be described

simply would be broken down into a set of smaller component tasks, until the tasks were sufficiently small and self-contained enough that they were easily understood.

As an example, computing the average salary of every employee of a company is a rather complex task. You can, however, break it down into these subtasks:

1. Find out what each person earns.
2. Count how many people you have.
3. Total all the salaries.
4. Divide the total by the number of people you have. Totaling the salaries can be broken down into
  1. Get each employee's record.
  2. Access the salary.
  3. Add the salary to the running total.
  4. Get the next employee's record.In turn, obtaining each employee's record can be broken down into
  1. Open the file of employees.
  2. Go to the correct record.
  3. Read the data from disk.

Structured programming remains an enormously successful approach for dealing with complex problems.

By the late 1980s, however, some of the deficiencies of structured programming had become all too clear.

First, it is natural to think of your data (employee records, for example) and what you can do with your data (sort, edit, and so on) as related ideas.

Second, programmers found themselves constantly reinventing new solutions to old problems. This is often called "reinventing the wheel," and is the opposite of reusability. The idea behind reusability is to build components that have known properties, and then be able to plug them into your program as you need them. Old-fashioned programs forced the user to proceed step-by-step through a series of

screens. Modern event-driven programs present all the choices at once and respond to the user's actions.

Object-oriented programming attempts to respond to these needs, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate that data.

The essence of object-oriented programming is to treat data and the procedures that act upon the data as a single "object"--a self-contained entity with an identity and certain characteristics of its own.

### 3.2.4 C++ and Object Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development: encapsulation, data hiding, inheritance, and polymorphism.

The property of being a self-contained unit is called encapsulation. With encapsulation, we can accomplish data hiding. Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally. Just as you can use a refrigerator without knowing how the compressor works, you can use a well-designed object without knowing about its internal data members.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. Once created, a well-defined class acts as a fully encapsulated entity--it is used as a whole unit. The actual inner workings of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it.

C++ supports the idea of reuse through inheritance. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type.

C++ supports the idea that different objects do "the right thing" through what is called function polymorphism and class polymorphism. "Poly" means "many," and "morph" means "form." Polymorphism refers to the same name taking many forms.

**Exercise:** What are the four pillars of object-oriented development  
Answer: encapsulation, inheritance, data hiding, polymorphism

### 3.2.5 Compiling the Source Code

Although the source code is somewhat cryptic, and anyone who doesn't know C++ will struggle to understand what it is for, it is still in what is called human-readable form. The source code file is not yet a program, and it can't be executed, or run, as a program can.

To turn source codes into programs, a compiler is used. How it is invoked differs from compiler to compiler.

After the source code is compiled, an object file is produced. This file is often named with the extension **OBJ**. This is still not an executable program, however. To turn this into an executable program, you must run the linker.

### 3.2.6 Creating an Executable File

C++ programs are typically created by linking together one or more **OBJ** files with one or more libraries. A library is a collection of linkable files that were supplied with your compiler, that you purchased separately, or that you created and compiled. All C++ compilers come with a library of useful functions (or procedures) and classes that you can include in your program. A function is a block of code that performs a service, such as adding two numbers or printing to the screen. A class is a collection of data and related functions.

The steps to create an executable file are

1. Create a source code file, with a **CPP** extension.
2. Compile the source code into a file with the **OBJ** extension.
3. Link your **OBJ** file with any needed libraries to produce an executable program

### 3.2.7 The Development Cycle

If every program worked the first time they are tried, that would be the complete development cycle: Write the program, compile the source code, link the program, and run it. Unfortunately, almost every program, no matter how trivial, can and will have errors, or bugs, in the program. Some bugs will cause the compile to fail, some will cause the link to fail, and some **will** only show up when you run the program.

Whatever type of bug you find, you must fix it, and that involves editing your source code, recompiling and relining, and then rerunning the program.

## **40 Conclusion**

C++ is a landmark language in the programming world. It bridges the gap between its antecedents and successors in that it is procedural, structured and also object-oriented in nature. It serves as a good development tool to use for a wide range of applications from highly scientific to Marjory-database applications (in conjunction with databases). The advent of C++ has helped subsequent language (compiler) developers to focus more on the object-oriented technology and make a better world for programmers.

## **5.0 Summary**

In this unit, you have a good understanding of how C++ evolved and what problems it was designed to solve. You should feel confident that C++ provides the tools of object-oriented programming and the performance of a systems-level language, which makes C++ the development language of choice for many.

## **6.0 Tutor Marked Assignments**

1. Define Procedural Programming
2. What is a Structured Program
3. Differentiate between Procedural Programming and Object-Oriented Programming in C++
4. What are the steps necessary to make a C++ program executable

## **7.0 References And Other Resources**

Teach yourself C++ in 21 days, Greg Wiegand <http://www.mcp.com>  
Borland C++ Object-Oriented Programming.



## UNIT 17: C++ LANGUAGE BASICS

### TABLE OF CONTENTS

|     |                                |
|-----|--------------------------------|
| 1.0 | Introduction                   |
| 2.0 | Objectives                     |
| 3.1 | VARIABLES                      |
| 3.2 | DEFINING A VARIABLE            |
| 3.3 | CONSTANTS                      |
| 4.0 | Conclusion                     |
| 5.0 | Summary                        |
| 6.0 | Tutor Marked Assignment        |
| 7.0 | References And Other Resources |

#### 1.0 Introduction

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate that data.

In C++ a variable is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

#### 2.0 Objectives

In this unit, you will learn

- How to declare and define variables and constants.
- How to assign values to variables and manipulate those values.
- How to write the value of a variable to the screen.

#### 3.1 Variables

Your computer's memory can be viewed as a series of cubbyholes. Each cubbyhole is one of many, such holes, all lined up. Each cubbyhole--or memory location--is numbered sequentially. These numbers are known as memory addresses. A variable reserves one or more cubbyholes in which you may store a value.

Your variable's name (for example, my Variable) is a label on one of these cubbyholes, so that you can find it easily without knowing its actual memory address. Depending on the size of my Variable, it can take up one or more memory addresses.

##### 3.1.1 Setting Aside Memory

When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth. This

information tells the compiler how much room to set aside and what kind of value you want to store in your variable.

it needs two bytes of memory, or two cubbyholes. The type of the variable (for example, integer) tells the compiler how much memory (how many cubbyholes) to set aside for the variable.

Because computers use bits and bytes to represent values, and because memory is measured in bytes, it is important that you understand and are comfortable with these concepts.

### 3.1.2 Size of Integers

On any one computer, each variable type takes up a single, unchanging amount of room. That is, an integer might be two bytes on one machine, and four on another, but on either computer it is always the same, day in and day out.

A char variable (used to hold characters) is most often one byte long. A short integer is two bytes on most computers, a long integer is usually four bytes, and an integer (without the keyword short or long) can be two or four bytes. Listing 2.1 should help you determine the exact size of these types on your computer.

New Term: A character is a single letter, number, or symbol that takes up one byte of memory.

Listing 2.1. Determining the size of variable types on your computer.

```

1: #include <iostream.h>
2:
3: int main()
4: (
5: Cout « "The size of an int is:VW « sizeof(int) "
6: Cout « "The size of a short int isAt" sizeof(short) "
7: Cout « "The size of a long int isAt" sizeof(long) "
8: Cout « "The size of a char isAt\t" sizeof(char) "
9: Cout « "The size of a float isAt\t" « sizeof(float) "
10: Cout « "The size of a double is:\t" sizeof(double) cc"
11:
12: return 0;
13:)

```

Output: The size of an int is: 2 bytes.

The size of a Short int is: 2 bytes.

The size of a long int is: 4 bytes.

The size of a char is k bytes.

The size of a float is: 4 bytes.

The size of a double is: 8 bytes.

void :

NOTE: On your computer, the number of bytes presented might be different.

Example: Most of the time, the use of the sizeof function is pretty familiar. The one new feature is the use of the sizeof function in lines 5 through 10. size\_of() is provided by the compiler, and it tells you the size of the object you pass in as a parameter. For example, on line 5 the keyword int is passed into size\_of(). Using size\_of(), I was able to determine that on my computer an int is equal to a short int, which is 2 bytes.

### 3.1.1 Signed and Unsigned

- In addition, all integer types come in two varieties: signed and unsigned. The idea here is that sometimes you need negative numbers, and sometimes you don't. Integers (short, and long), without the word "unsigned" are assumed to be signed. Signed integers are either negative or positive. Unsigned integers are always positive.

Because you have the same number of bytes for both signed and unsigned integers, the largest number you can store in an unsigned integer is twice as big as the largest positive number you can store in a signed integer. An unsigned integer can handle numbers from 0 to 65,535. Half the numbers represented by a signed short are negative, thus a signed short can only represent numbers from -32,768 to 32,767. If this is confusing, be sure to read Appendix A, "Operator Precedence."

### 3.1.4 Fundamental Variable Types

Several other variable types, are built into C++. They can be conveniently divided into integer variables (the type discussed so far), floating-point variables, and character variables.

Floating-point variables have values that can be expressed as fractions—that is, they are real numbers. Character variables hold a

single byte and are used for holding the 256 characters and symbols of the ASCII and extended ASCII character sets:

New Term: The ASCII character set is the set of characters standardized for use, the pre-ASCII is an acronym for American Standard Code for Information Interchange. Nearly every computer operating system supports ASCII, though many support other international character sets, as well, as well.

The types of variables used in C++ programs are described in Table 2.1. This table shows the variable type, how much room this book assumes it takes in memory, and what kinds of values it can hold. The values that can be stored in the variable are given in the table. The size of the variable is given in the table.

Table 2.1. Variable types, C++.

| Type                  | Size    | Values                          |
|-----------------------|---------|---------------------------------|
| signed short int      | 2 bytes | -32,768 to 32,767               |
| unsigned short int    | 2 bytes | 0 to 65,535                     |
| signed long int       | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long int     | 4 bytes | 0 to 4,294,967,295              |
| signed int (16 bit)   | 2 bytes | -32,768 to 32,767               |
| unsigned int (16 bit) | 2 bytes | 0 to 65,535                     |
| signed char           | 1 byte  | -128 to 127                     |
| unsigned char         | 1 byte  | 0 to 255                        |
| float                 | 4 bytes | -3.4e+38 to 3.4e+38             |
| double                | 8 bytes | -1.7e+308 to 1.7e+308           |

The size of variables is given in the table. The values that can be stored in the variable are given in the table. The size of the variable is given in the table.

The size of variables is given in the table. The values that can be stored in the variable are given in the table. The size of the variable is given in the table.

### 3.2 Defining A Variable

You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces. Legal variable names include `x`, `J22grsnf`, and `myAge`. Good variable names tell you what the variables are for, using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called `myAge`:

```
int myAge;
```

As a general programming practice, avoid such horrific names as `J22grsnf`, and restrict single-letter variable names (such as `x` or `i`) to variables that are used only very briefly. Try to use expressive names such as `myAge` or `howMany`. Such names are easier to understand three weeks later when you are scratching your head trying to figure out what you meant when you wrote that line of code.

Try this experiment: Guess what these pieces of programs do, based on the first few lines of code:

Example 1 `main()`

```
unsigned short x;
unsigned short y;
ULONG z;
z = x * Y.
```

Example 2  
`main ( )`

```
unsigned short Width; unsigned short Length; unsigned short Area; Area =
Width *Length;
```

Clearly, the second program is easier to understand, and the inconvenience of having to type the longer variable names is more than made up for by how much easier it is to maintain the second program.

#### 3.2.1 Case Sensitivity

11.5.1  
1985/01  
219000 00  
11/00/00  
1985/01/01

C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named "age" is different from "Age," which is different from AGE.

NOTE: Some compilers allow you to turn case sensitivity off. Don't be tempted to do this; your programs won't work with other compilers, and other C++ programmers will be very confused by your code.

There are various conventions for how to name variables, and although it doesn't much matter which method you adopt, it is important to be consistent throughout your program.

Many programmers prefer to use all lowercase letters for their variable names. If the name requires two words (for example, my car), there are two popular conventions: mycar or myCar. The latter form is called camel-notation, because the capitalization looks something like a camel's hump.

Some people find the underscore character (mycar) to be easier to read, while others prefer to avoid the underscore, because it is more difficult to type. This book uses camel-notation, in which the second and all subsequent words are capitalized: myCar, the QuicicBrownFox, and so forth.

NOTE: Many advanced programmers employ a notation style that is often referred to as Hungarian notation. The idea behind Hungarian notation is to prefix every variable with a set of characters that describes its type. Integer variables might begin with a lowercase letter i, longs might begin with a lowercase l. Other notations indicate constants, global, pointers, and so forth. Most of this is much more important in C programming, because C++ supports the creation of user-defined types (see Day 6, "Basic Classes") and because C++ is strongly typed.

### 3.2.2 Keywords

Some words are reserved by C++, and you may not use them as variable names. These are keywords used by the compiler to control your program. Keywords include generally, any reasonable name for a variable is almost certainly not a keyword.

**DO** define a variable by writing the type, then the variable name. **DO** use meaningful variable names. **DO remember that C++ is case sensitive.** **DON'T use C++ keywords as variable names.** **DO understand the number of bytes each variable type consumes in memory, and what values can be stored in variables of that type.** **DON'T use unsigned variables for negative numbers.**



6). typedef K Heti H \Aciting tic kc\\nd typecici tt\*th'. cd H. the  
 .xistinct t and 41. ten the.nc1/4, ft; e.s'trn i lc  
 ty ge is us by ped !writing the keyword 'typedef, followed by the  
 " i f 1 A i R I W Y P W A - P W r t i k e r . W I Y e n e

For example

Vilref Wittig

nu can use an) [\\11](#),2.1(

have written unsigned short int

creates the new name USHORT that you can use anywhere you might

~~USHORT~~ rather than litsiLmeci shoo ini

A°dg'nfPROXltgi 41',1,1gclgf

using the type definition

USHOter 'ober than unsigned short int.

Listing/? .6c44ge49cirtiplirgcl4d

ifinclucle costicam 1 1.>

2t // Demonstrates typedef keyword

44:anti& aciostneabsibart irfi I St IOR

4:

5. typetleiLunitigned short int USHORT;

6:

7: mold Kafiri()'

8.: f . '

g: IffSliellitT Width = 5;

10: USHORill Length;

**11: .Length =10;**

12: USHORT Area = Width \*-

13: cout << "Width:" << Width-<c "\n";

14:. cout << "Length: " << I engh

15: cout << "Area: " << Area << endl;

16: 1

Output liVidth:5

Length: •10

Area: 50

Analysis: On line 5, USHORT is type defined as a synonym for unsigned short int. The program is very much like Listing 2.2, and the output is the same.

### 3.2.4 Characters

Character variables (type.char) arn typically byte, enoagh to h Ad 256 values (see Appendix C). A char can be interpreted as a small number (0-255) or as a member of the ASCII set. ASCII stands for the

ISO (International Standards Organization) equ;valent are a way to encode all the letters, numerals, and punctuation marks.



are numbers. In fact, all they really know about is whether or not a sufficient amount of electricity is at a particular junction of wires. If so, it is represented internally as a 1; if not, it is represented as a 0. By grouping ones and zeros, the computer is able to generate patterns that can be interpreted as numbers, and these in turn can be assigned to letters and punctuation.

In the ASCII code, the lowercase letter "a" is assigned the value 97. All the lower- and uppercase letters, all the numerals, and all the punctuation marks are assigned values between 1 and 128. Another 128 marks and symbols are reserved for use by the computer maker, although the IBM extended character set has become something of a standard.

### 3.2.5 Characters and Numbers

When you put a character, for example, 'a', into a char variable, what is really there is just a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters (represented by a single quotation mark and then a letter, numeral, or punctuation mark, followed by a closing single quotation mark) and one of the ASCII values.

The value/letter relationship is arbitrary; there is no particular reason that the lowercase "a" is assigned the value 97. As long as everyone (your keyboard, compiler, and screen) agrees, there is no problem. It is important to realize, however, that there is a big difference between the value 5 and the character '5'. The latter is actually valued at 53, much as the letter 'a' is valued at 97.

Listing 2.4. Printing characters based on numbers

```
1: #include <iostream.h>
2: int main() {
4: for (int i = 32; i<128; i++)
5: cout << (char) i;
6: return 0;
7: }
```

Output:!"#\$%&'()\*+,-./0123456789;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ^\_`abcdefghijklmnopqrstuvwxyz{|}~

This simple program prints the character values for the integers 32 through 127.

### 3.2.6 Special Printing Characters

The C++ compiler recognizes some special characters for formatting. Table 2.2 shows the most common ones. You put these into your code by typing the backslash (called the escape character), followed by the character. Thus, to put a tab character into your code, you would enter a single quotation mark, the slash, the letter t, and then a closing single quotation mark:

```
char tabCharacter = 1e;
```

This example declares a char variable (tabCharacter) and initializes it with the character value \t, which is recognized as a tab. The special printing characters are used when printing either to the screen or to a file or other output device.

**New Term:** An escape character changes the meaning of the character that follows it. For example, normally the character n means the letter n, but when it is preceded by the escape character (\) it means new line.

**Table 2.2. The Escape Characters.**

| Character | What it means |
|-----------|---------------|
| \n        | new line      |
| \t        | Tab           |
| \b        | backspace     |
| \",       | double quote  |
| \'        | single quote  |
| \?        | question mark |
| \\        | backslash     |

## 33 Constants

Like variables, constants are data storage locations. Unlike variables, and as the and you cannot assign a new value later.

### 3.3.1 Literal Constants

C++ has two types of constants: literal and symbolic.

A literal constant is a value typed directly into your program wherever it is needed. For example

```
int myAge = 39;
```

myAge is a variable of type `int`; 39 is a literal constant. You can't assign a value to 39, and its value can't be changed.

### 3.3.2 Symbolic Constants

variable is represented. Unlike a variable, however, after a constant is initialized, its value can't be changed.

If your program has one integer variable named `students` and another named `classes`, you could compute how many students you have, given a known number of classes, if you knew there were 15 students per class:

```
students = classes * 15;
```

In this example, 15 is a literal constant. Your code would be easier to read, and easier to maintain, if you substituted a symbolic constant for this value:

```
students = classes * studentsPerClass
```

If you later decided to change the number of students in each class, you could every place you used that value.

There are two ways to declare a symbolic constant in C++. The old, traditional, and traditional way, you would enter this:

```
#define studentsPerClass 15
```

Note that `studentsPerClass` is of no particular type (`int`, `char`, and so on). `#define` does a simple text substitution. Every time the preprocessor sees the word `studentsPerClass`, it puts in the text 15.

Because the preprocessor runs before the compiler, your compiler never sees your constant; it sees the number 15.

Although `#define` works, there is a new, much better way to define constants in C++:

```
const unsigned short int studentsPerClass = 15;
```

This example also declares a symbolic constant named `studentsPerClass`, but this time `studentsPerClass` is typed as an unsigned short int. This method has several advantages in making your code easier to maintain and in preventing bugs. The biggest difference is that this constant has a type, and the compiler can enforce that it is used according to its type.

**NOTE:** Constants cannot be changed while the program is running. If you need to change `studentsPerClass`, for example, you need to change the code and recompile.

intended. **DO** watch for numbers overrunning the size of the integer and wrapping around incorrect values. **DO** give your variables meaningful names that reflect their use. **DON'T** use keywords as variable names.

### 3.3.3 Enumerated Constants

Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values. For example, you can declare `COLOR` to be an enumeration, and you can define that there are five values for `COLOR`: `RED`, `BLUE`, `GREEN`, `WHITE`, and `BLACK`.

The syntax for enumerated constants is to write the keyword `enum`, followed by the type name, an open brace, each of the legal values separated by a comma, and finally a closing brace and a semicolon. Here's an example:

```
enum COLOR (RED, BLUE, GREEN, WHITE, BLACK.);
```

This statement performs two tasks:

1. It makes **COLOR** the name of an enumeration, that is, a new type.
2. It makes **RED** a symbolic constant with the value 0, **BLUE** a symbolic constant with the value 1, **GREEN** a symbolic constant with the value 2, and so forth.

otherwise, the first constant will have the value 0, and the rest will count up from there. Any one of the constants can be initialized with a

particular value, however, and those that are not initialized will count upward from the ones before them. Thus, if you write

```
enum Color { RED=100, BLUE, GREEN=500, WHITE,
BLACK=700};
```

then RED will have the value 100; BLUE, the value 101; GREEN, the value 500; WHITE, the value 501; and BLACK, the value 700.

You can define variables of type COLOR, but they can be assigned only one of the enumerated values (in this case, RED, BLUE, GREEN, WHITE, or BLACK, or else 100, 101, 500, 501, or 700). You can assign any color value to your COLOR variable. In fact, you can assign any integer value, even if it is not a legal color, although a good compiler will issue a warning if you do. It is important to realize that enumerator variables actually are of type unsigned int, and that the enumerated constants equate to integer variables. It is, however, very convenient to be able to name these values when working with colors, days of the week, or similar sets of values. Listing 2.5 presents a program that uses an enumerated type.

**Listing 2.5. A demonstration of enumerated constants.**

```
1:#include <iostream.h>
2:int main()
3.(
4: enum Days { Sunday, Monday, Tuesday, Wednesday,
 Thursday,
 Friday, Saturday };
5:
6: Days DayOff;
7: int x; 8:
9: Gout << "What day would you like off (0-6)? ";
10: cin x;
11: DayOff = Days (x)
12:
13: if (DayOff == Sunday || DayOff == Saturday)
14: Gout << " \nYou're already off on weekends\n";
15: e l s e
16: cout << " \nOkay, I'll put in the vacation day\n";
17: return 0;
18.1
```

Output: What day would you like off (0-6)? 1

Okay, I'll put in the vacation day.

What day would you like off (0-6)? 0

You're already off on weekends!

Analysis: On line 4, the enumerated constant DAYS is defined, with seven values counting upward from 0. The user is prompted for a day on line 9. The chosen value, a number between 0 and 6, is compared on line 13 to the enumerated values for Sunday and Saturday, and action is taken accordingly.

**NOTE:** For this and all the small programs in this book, I've left out all the code you would normally write to deal with what happens when the user types inappropriate data. For example, this program doesn't check, as it would in a real program, to make sure that the user types a number between 0 and 6. This detail has been left out to keep these programs small and simple, and to focus on the issue at hand.

#### 4.0 Conclusion

Writing a C++ program needs a careful understudy of the rudiments of the project which should be designed beforehand to know the format the coding will take and the different variables that may be used. As a programmer, care should be taken to ensure the right variable type is used for efficient use of memory space.

#### 5.0 Summary

This unit has discussed numeric and character variables and constants, which are either integral (char, short, and long int) or they are floating point (float and double). Numeric variables can also be signed or unsigned. Although all the types can be of various sizes among different computers, the type specifies an exact size on any given computer.

You must declare a variable before it can be used, and then you must store the type of data that you've declared as correct for that variable. If you put too large a number into an integral variable, it wraps around and produces an incorrect result.

This unit also reviewed literal and symbolic constants, as well as remunerated constants, and showed two ways to declare a symbolic constant: using #define and using the keyword const.

## 6.0 Tutor Marked Assignment

- 1 What is the difference between an integral variable and a floating-point variable?
- 2 What are the differences between an unsigned short int and a long int?
- 3 What are the advantages of using a symbolic constant rather than a literal constant'?
- 4 What are the advantages of using the const keyword rather than
- 5 Given this enum, what is the value of BLUE?  

```
enum COLOR { WHITE, BLACK = 100, RED, BLUE, GREEN = 300 } ;
```
- 6 Which of the following variable names are good, which are bad, and which are invalid?
  - a. Age
  - b. !ex
  - c. R791
  - d. TotalIncome
  - e. Invalid
- 7 What would be the correct variable type in which to store the following information?
  - a. Your age.
  - b. The area of your backyard.
  - c. The number of stars in the galaxy.
  - d. The average rainfall for the month of January.
- 8 Create good variable names for this information.
- 9 Declare a constant for pi as 3.14159.
- 10 Declare a float variable and initialize it using your pi constant.

## 7.0 References And Other Resources

Teach yourself C++ in 21 days, Greg Wiegand, <http://www.mcp.com>  
Borland C++ Object-Oriented Programming.

## UNIT 18: C++ EXPRESSIONS AND STATEMENTS

### TABLE OF CONTENTS

|     |                              |
|-----|------------------------------|
| 1.0 | Introduction                 |
| 2.0 | Objectives                   |
| 3.1 | Statements                   |
| 3.2 | Operators                    |
| 3.3 | The If Statement             |
| 3.4 | Logical Operators            |
| 4.0 | Conclusion                   |
| 5.0 | Summary                      |
| 6.0 | Tutor Marked Assignment      |
| 7.0 | References & Other Resources |

### 1.0 Introduction

At its heart, a program is a set of commands executed in sequence. The power in a program comes from its capability to execute one or another set of commands, based on whether a particular condition is true or false.

### 2.0 Objectives

At the end of this unit, you will be able to

- define statements.
- define what blocks are.
- describe what expressions are.
- demonstrate how to branch your code based on conditions.

### 3.1 Statements

In C++ a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement). All C++ statements end with a semicolon, even the null statement, which is just the semicolon and nothing else. One of the most common statements is the following assignment statement:

```
X=a+b;
```

Unlike in algebra, this statement does not mean that x equals a+b. This is read, "Assign the value of the sum of a and b to x," or "Assign to x, a+b." Even though this statement is doing two things, it is one statement and thus has one semicolon. The assignment operator assigns whatever is on the right side of the equal sign to whatever is on the left side.



### 3.1.1 Blocks and Compound Statements

Any place you can put a single statement, you can put a compound statement, also called a block. A block begins with an opening brace ( { ) and ends with a closing brace ( } ). Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon. For example

```
temp = a;
a = b;
b = temp;
```

This block of code acts as one statement and swaps the values in the variables a and b.

### 3.1.2 Expressions

Anything that evaluates to a value is an expression in C++. An expression is expressions are statements.

Examples of valid expressions are given below:

```
3.2 // returns the value 3.2
PI // float const that returns the value 3.14
```

```
SecondsPerMinute // int const that returns 60
constant equal to 60, all three of these statements are expressions.
```

A little complicated expression

```
x = a + b;
```

not only adds a and b and assigns the result to x, but returns the value of that assignment (the value of x) as well. Thus, this statement is also an expression. Because it is an expression, it can be on the right side of an assignment operator:

Listing 4.1. Evaluating complex expressions.

```
1: #include <iostream.h>
2: int main() 3. {
4: int a=0, b=0, x=0; y=35;
5: cout << "a: " << a << " b : " << b;
6: cout << " x: " << x << " y: " << y << endl; 7: a=9;
8: b = 7;
9: y = x = a + b ;
```

```

10: cote .1& *a: « a cl.< *b: " -t<b;-1/4
11: cout<ex w< Cc 4' << yitend;12: return 0;

```

Copy and run the lines of code above.

## 3.2 Operators

1.

An operator is a symbol that causes the compiler to take an action. Operators act on operands, and in C++ all **operands are expressions**. In C++ there are several different categories of operators. Two of these categories are

- 0 Assignment operators.
- Mathematical operators.

### 3.2.1 Assignment Operators

The assignment operator (=) causes the operand on the left side of the assignment operator to have its value changed to the value on the right side of the assignment operator. The expression

```
x = a + b;
```

assigns the value that is the result of adding a and b to the operand A.

#### 3.2.2 Mathematical Operators

There are five mathematical operators: addition (+), subtraction (-), multiplication (\*), division (/), and modulus (%).

Addition and subtraction work as you would expect, although subtraction with unsigned integers can lead to surprising results, if the result is a negative number. You saw something much like this yesterday, when variable overflow was described. Listing 4.2 shows what happens when you subtract a large unsigned number from a small unsigned number.

**Listing 4.2.** A demonstration of subtraction and integer overflow.

```

1: // Listing 4.2 - demonstrates subtraction and
2: // integer overflow
3: #include <fstream.h>
4: '
5: int main()
6: {
7: unsigned int difference;
8: unsigned int bigNumber = 100;
9: unsigned int smallNumber = 50;
10: difference = bigNumber - smallNumber,

```

```

11: cout << "Difference is: " << difference;
12: difference = smallNumber - bigNumber;
13: cout << "Now difference is: " << difference << endl;
14: return 0;
15: }

```

Copy and run the lines of code above.

### MS Integer Division and Modulus

Integer division is somewhat different from everyday division. When you divide 21 by 4, the result is a real number (a number with a fraction). Integers don't have fractions, and so the "remainder" is truncated. The answer is therefore 5. To get the remainder, you take 21 modulus 4 (21 % 4) and the result is 1, for the modulus operator tells you the remainder after an integer division,

### 3.2.4 Increment and Decrement Operators

The most common value to add (or subtract) and then reassign into a variable is 1. In C++, increasing a value by 1 is called incrementing, and decreasing by 1 is called decrementing. There are special operators to perform these actions.

The increment operator (++) increases the value of the variable by 1, and the decrement operator (--) decreases it by 1. Thus, if you have a variable, C, and you want to increment it, you would use this statement:

```

C++; // Start with C and increment it. This statement is equivalent to
the more verbose statement
C=C+ 1;

```

which you learned is also equivalent to the moderately verbose statement

### 3.13 Prefix and Postfix

Both the increment operator (++) and the decrement operator (--) come in two varieties: prefix and postfix. The prefix variety is written before the variable name (++sayAge); the postfix variety is written after (myAge--).

- when you have to assign the result to another variable, it matters very much. The prefix operator is evaluated before the assignment, the postfix is evaluated after.

The semantics of prefix is this: Increment the value and then fetch it. The semantics of postfix is different: Fetch the value and then increment the original.

This can be confusing at first, but if  $x$  is an integer whose value is 5 and you write `int a = ++x;`

you have told the compiler to increment  $x$  (making it 6) and then fetch that value and assign it to  $a$ . Thus,  $a$  is now 6 and  $x$  is now 6.

If, after doing this, you write  
`int b = x++;`

you have now told the compiler to fetch the value in  $x$  (6) and assign it to  $b$ , and then go back and increment  $x$ . Thus,  $b$  is now 6, but  $x$  is now 7. Listing 4.3 shows the use and implications of both types.

Listing 4.3. A demonstration of prefix and postfix operators.

```

1: // Listing 4.3 - demonstrates use of
2: // prefix and postfix increment and
3: // decrement operators
4: #include <iostream.h> 5: int main()
6: {
7: int myAge = 39; // initialize two integers

8: int yourAge = 39;
9: cout << "I am: " << myAge << " years old\n";
10: cout << "You are: " << ++yourAge << yourAge << " years old\n";
11: myAge++; // postfix increment
12: ++yourAge; // prefix increment
13: cout << "One year passes\n";
14: cout << "I am: " << myAge << " years old\n";
15: cout << "You are: " << yourAge << " years old\n";
16: cout << "Another year passes\n";
17: cout << "I am " << myAge++ << " years old\n";
18: cout << "You are: " << yourAge << " years old\n";
19: cout << "Let's print it again\n";
20: cout << "I am: " << myAge << " years old\n";

```

```

21: cout << "You are: " << yourAge << " years olchn";
22: return 0;
23:

```

**Copy and run the lines of code above.**

### 3.2.6 Operator Precedence

In the complex statement

```
x = 5 + 3 * 8;
```

which is performed first, the addition or the multiplication? If the addition is performed first, the answer is  $8 * 8$ , or 64. If the multiplication is performed first, the answer is  $5 + 24$ , or 29.

Every operator has a precedence value. Multiplication has higher *precedence* than addition, and thus the value of the expression is 29. When two mathematical operators have the same precedence, they are performed in left-to-right order. Thus

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

is evaluated multiplication first, left to right. Thus,  $8 * 9 = 72$ , and  $6 * 4 = 24$ . Now the expression is essentially

```
= 5 + 3 + 72 + 24;
```

Now the addition, left to right, is  $5 + 3 = 8$ ;  $8 + 72 = 80$ ;  $80 + 24 = 104$ .

Be careful with this. Some operators, such as assignment, are evaluated in right-to-left order! In any case, what if the *precedence* order doesn't meet your needs? Consider the expression

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

In this expression, you do not want to multiply the NumMinutesToType variable by 60 and then add it to NumMinutesToThink. You want to add the two variables to get the total number of minutes, and then you want to multiply that number by 60 to get the total seconds,

In this case, you use parentheses to change the precedence order. Items in parentheses are evaluated at a higher precedence than any of the mathematical operators. Thus

```
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60
```

will accomplish what you want..

### 1.2.7 Relational Operators

The relational operators are used to determine whether two numbers are equal, or if one is greater or less than the other. Every relational statement evaluates to either 1 (TRUE) or 0 (FALSE). The relational operators are presented later.

If the integer variable `myAge` has the value 39, and the integer variable `yourAge` has the value 40, you can determine whether they are equal by using the relational "equals" operator:

```
myAge == yourAge; // is the value in myAge the same as in/
yourAge?
```

This expression evaluates to 0, or false, because the variables are not equal. The expression

```
myAge > yourAge; // is myAge greater than yourAge?
evaluates to 0 or false.
```

There are six relational operators: equals (`==`), less than (`<`), greater than (`>`), less than or equal to (`<=`), greater than or equal to (`>=`), and not equals (`!=`).

## 3.3 The if Statement

Normally, your program flows along line by line in the order in which it appears in your source code. The `if` statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.

The simplest form of an `if` statement is this:

```
if (expression)
 statement;
```

The expression in the parentheses can be any expression at all, but it usually contains one of the relational expressions. If the expression has the value 0, it is considered false, and the statement is skipped. If it has any nonzero value, it is considered true, and the statement is executed. Consider the following example:

```
if (bigNumber > smallNumber)
 bigNumber = smallNumber;
```

This code compares `bigNumber` and `smallNumber`. If `bigNumber` is larger, the second line sets its value to the value of `smallNumber`.

Because a block of statements surrounded by braces is exactly equivalent to a single statement, the following type of branch can be quite large and powerful:

```
if (expression)
statement 1;
statement 2;
statement 3;
```

Here's a simple example of this usage:

```
if (bigNumber > smallNumber)

bigNumber = smallNumber;
Gout << "bigNumber: " << bigNumber << "VC;

cout << "smallNumber: " << smallNumber << "An";
```

This time, if `bigNumber` is larger than `smallNumber`, not only is it set to the value of `smallNumber`, but an informational message is printed. Listing 4.4 shows a more detailed example of branching based on relational operators.

Listing 4.4. A demonstration of branching based on relational operators.

```
1: //Listing 4.4 - demonstrates if statement
2: // used with relational operators
3: #include <iostream.h>
4: int main()
5: {
6: int RedSoxScore, YankeesScore;
7: cout << "Enter the score for the Red Sox: ";
8: cin >> RedSoxScore; 9:
10: Gout << "Enter the score for the Yankees: ";
11: cin >> YankeesScore; 12:
13: cout << "\n"; 14:
15: if (RedSoxScore > YankeesScore)
16: Gout << "Go Sox An";
17:
18: if (RedSoxScore < YankeesScore)
20:
21: Gout << "Go Yankees An";
```

```

21: cout << "Happy days in NeW York! \n"; 22:
23:
24: if (RedSoxScore == YankeesScore) 25:
26: Gout << "ktie? Naah, can't beAn";
27: cout << "Give me the real score for the Yanks: ";
28: cm n >> YanlceesScore;
29:
30: if (RedSoxScore > YankeesScore)
31: cout << "Knew it! Go Sox!"; 32:
33: if (YankeesScore > RedSoxScore)
34: Gout << "Knew it! Go Yanks!"; 35:
36: if (YankeesScore == RedSoxScore)
37: cout << "Wow, it really was a tie!"; 38: 1
39:
40: cout << "knThanks for telling meAn"; 41: trn()
42:)

```

In this example, getting a true result in one `if` statement does not stop other `if` statements from being tested.

Copy and run the lines of code above.

### 3.3.1 If else

Often your program will want to take one branch if your condition is true, another if it is false. In Listing 4.3, you wanted to print one message (Go Sox!) if the first test (`RedSoxScore > Yankees`) evaluated `TRUE`, and another message (Go Yanks ! ) if it evaluated `FALSE`.

The method shown so far, testing first one condition and then the other, works fine but is a bit cumbersome. The keyword `else` can make for far more readable code:

```

if (expression)
 statement;
else
 statement;

```

**Listing 4.5 demonstrates the use of the keyword `else`.**

```

1: // Listing 4.5 - demoitstates if statement
2: // with else clause
3: #include <iostream.h>
4: int main()
5:
6: int firstNumber, seconOlumber;
7: cout << "Please enter a big number:";

```



```

8: cm n >> firstNumber:
9: Gout « "\nPlease enter a smaller number: ";
10: cin » secondNumber:
11: if (firstNumber > secondNumber)
12: Gout « "\nThanksRn";
13: e l s e
14: cout « " \nOops. The second is bigger!";
15:
16: return 0;
17: }

```

Listing 4.5. Demonstrating the else keyword.

Copy and run the lines of code above.

### 3.3.2 Advanced/Nested if Statements

it is worth noting that any statement can be used in an if or else clause, even following form:

```

if (expression 1)
if (expression2) statement 1; else

if (expression3)

 statement2;
else
 statement3;
else
 statement4;

```

This cumbersome if statement says, "If expression I is true and expression2 is true, execute statement I. If expression] is true but expression2 is not true, then if expression3 is true execute statement2. If expression) is true but expression2 and expression3 are false, execute statement3. Finally, if expression) is not true, execute statement4." As you can see, complex if statements can be confusing!

**Listing 4.6 gives an example of such a complex I f statement.**

Listing 4.6. A complex, nested if statement.

```

1: // Listing 4.5 - a complex
2: // if statement
3: #include <iostream.h>
4: int main()

```

```

5. {
6: // Ask for two numbers
7: // Assign the numbers to bigNumber and 1
8: // If bigNumber is bigger than littleIsTutnbe
9: // see if they are evenly divisible
10: // If they are, see if they are the same
11:
12: int firstIsLumber >= secondNumber)
13: Cout <, " Enter two numbes . /nFirst : ";
14: cm n » firstNumber;
15: Cout « InSecond: ";
16: cm n » secondNumber;
17: cout « "/n/n";
18:
19: if (firstNumber a secondNumber)
20: { (21: if (first Number % secondNuber) == 0) // evenly
divisible?
22: }
23: if (firstNumber == secondNumber)
24: cout « "They are the same Rn";
25: e l s e
26: cout « "They are not evently divisibleRn";
27: }
28:
29 else cout « " They second one is largerRn";

30: }
31: e l s e
32: cout << " Hey! The second one is larger RC;
33: return)'
34: }

```

copy and **run the lines of code above.**

### 3.3.3 Using Braces in Nested if Statements

**Although it** is legal to teem out the braces on i f statements that are only a single statement, and it is legal to nest if statements, such as

```

if (x > y) // if x is bigger than y
 if (x < z) // and itx is smaller than z
 x = y;// then set x to the value in z

```

when writing large nested statements, this can cause enormous confusion. Remember, white space and indentation are a convenience for the programmer; they make no difference to the compiler. It is easy to confuse the logic and inadvertently assign an else statement to the wrong if statement. Listing 4.7 illustrates this problem.

**Listing 4.7. A demonstration of why braces help clarify which else statement goes with which if statement.**

```

1: // Listing 4.7 - demonstrates why braces
2: // are important in nested if statements
3: #include <iostream.h>
4: int main()
5:
6: int x;
7: cout << "Enter a number less than 10 or
8: cin >> x;
9: cout << "VI";
10:
11: if (x > 10)
12: if (x > 100)
13: cout << "More than 100, Thanks!\n";
14: else // not the else intended!
15: cout << "Less than 10. Thanks!\n";
16:
17: return 0;
18: 1

```

**Copy and run the lines of code above.**

**Listing 4.8 fixes the problem by putting in the necessary braces.**

**Listing 4.8. A demonstration of the proper use of braces with an if statement**

```

1: // Listing 4.8 - demonstrates proper use of braces
2: // in nested if statements
3: #include <iostream.h>
4: int main() 5. (
5: int x;
6: cout << "Enter a number less than 10 or greater than 100:
7: cin >> x;
8: cout <<
9:
10:
11: if (x > 10) 12: {

```

```

15: 1
(x 16: Else II not the else intended! > 100)
17: coin « "Less than 10.
18: return 0:
19: 1
13: if
14:
cout << "More than 100, Thankshn";
31

```

Copy and run the lines of code above.

### 3.4 Logical Operators

Often you want to ask more than one relational question at a time. "Is it true that x is greater than y, and also true that y is greater than z?" A program might need to determine that both of these conditions are true, or that some other condition is true, in order to take an action.

Imagine a sophisticated alarm system that has this logic: "If the door alarm sounds AND it is after six p.m. AND it is NOT a holiday, OR if it is a weekend, then call the police." C++'s three logical operators are used to make this kind of evaluation. These operators are listed in Table 3.1.

Table 3.1. The Logical Operators.

| Operator | Symbol | Example                    |
|----------|--------|----------------------------|
| AND      | &&     | expression1 && expression2 |
| OR       |        | expression1    expression2 |
| NOT      | !      | ! expression               |

#### 3.4.1 Logical AND

A logical AND statement evaluates two expressions, and if both expressions are true, the logical AND statement is true as well. If it is true that you are hungry, AND it is true that you have money, THEN it is true that you can buy lunch. Thus,

```
if ((x == 5) && (y == 5))
```

would evaluate TRUE if both x and y are equal to 5, and it would evaluate FALSE if either one is not equal to 5. Note that both sides must be true for the entire expression to be true.

Note that the logical AND is two & & symbols. A single & symbol is a different operator.

### 3.4.2 Logical OR

A logical OR statement evaluates two expressions. If either one is true, the expression is true. If you have money OR you have a credit card, you can pay the bill. You don't need both money and a credit card; you need only one, although having both would be fine as well.. Thus,

```
if ((x == 5) || (y == 5))
```

evaluates TRUE if either x or y is equal to 5, or if both are.

Note that the logical OR is two || symbols. A single | symbol is a different operator

### 3.4.3 Logical NOT

A logical NOT statement evaluates true if the expression being tested is false. Again, if the expression being tested is false, the value of the test is TRUE! Thus

```
if (!(x == 5))
```

is true only if x is not equal to 5. This is exactly the same as writing

```
if (x != 5)
```

### 3.4.4 Relational Precedence

Relational operators and logical operators, being C++ expressions, each precedence order (see Appendix A) that determines which relations are evaluated first. This fact is important when determining the value of the statement

```
if (x > 5 && y > 5 || z > 5)
```

It might be that the programmer wanted this expression to evaluate TRUE if both x and y are greater than 5 or if z is greater than 5. On the other hand, the programmer might have wanted this expression to evaluate TRUE only if x is greater than 5 and if it is also true that either y is greater than 5 or z is greater than 5.

If x is 3, and y and z are both 10, the first interpretation will be true (z is greater than 5, so ignore x and y), but the second will be false (it isn't true that both x and y are greater than 5 nor is it true that z is greater than 5).

Although precedence will determine which relation is evaluated first, parentheses can both change the order and make the statement clearer:

```
if((x > 5) && (y > 5 || z > 5))
```

Using the values from earlier, this statement is false. Because it is not true that x is greater than 5, the left side of the AND statement fails, and thus the entire statement is false. Remember that an AND statement requires that both sides be true--something isn't both "good tasting" AND "good for you" if it isn't good tasting.

**NOTE:** It is often a good idea to use extra parentheses to clarify what you want to group. Remember, the goal is to write programs that work and that are easy to read and understand.

### 3.4.5 Conditional (Ternary) Operator

The conditional operator (? :) is C++'s only ternary operator; that is, it is the only operator to take three terms.

The conditional operator takes three expressions and returns a value:

```
(expression 1) ? (expression2) : (expression3)
```

This line is read as "If expression) is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable.

Listing 4.9 shows an if statement rewritten using **the** conditional operator.

Listing 4.9. A demonstration of the conditional operator. •

```
1: // Listing 4.9- demonstrates the conditional operator
2: / /
3: #include <iostream.h>
4: int main()
5: {
6: int x, y, z;
7: cout << "Enter two numbers\n":
8: cout << "First: ";
9: cin >> x;
10: cout << "\nSecond:
11: cin >> y;
```

```

12: cout << "\n":
13:
14: if (x > y)
15: z = .
16: Else
17: z=
18:
19: cout << "z: "
20: cout << "\n":
21:
22: z = -(x > y) ? x.y;
23:
24: cout << "z:"
25: cout << "\n":
26: return 0;
27: }

```

Copy and run the lines of code above.

#### 4.0 Conclusion

The composition of any program is made of the small bits of expressions and statements. Learning and understanding a programming language involve knowing the combination of these bits and pieces. Without them a programmer has not even started. They are the blood through which the other components of a program runs.

#### 5.0 Summary

**This** unit has covered a lot of material. You have learned what C++ statements and expressions are, what C++ operators do, and how C++ if statements work.

You have seen that a block of statements enclosed by a pair of braces can be used anywhere a single statement can be used.

You have learned that every expression evaluates to a value, and that value can be tested in an if statement or by using the conditional operator. You've also seen how to evaluate multiple statements using the logical operator, how to compare values using the relational operators, and how to assign values using the assignment operator.

You have explored operator precedence. And you have seen how parentheses to manage.

## 6.0 Tutor Marked Assignments

- I Why use unnecessary parentheses when precedence will determine which first?
2. If the relational operators always return 1 or 0, why are other values considered
3. What effect do tabs, spaces, and new lines have on the program?  
Are negative numbers true or false?

## 740 References And Other Resources

Teach yourself C++ in 21 days, Greg Wiegand, <http://www.mcp.com>

Borland C++ Object-Oriented Programming.



## UNIT 19: CLASSES I

### TABLE OF CONTENTS

|     |                                |
|-----|--------------------------------|
| 1.0 | Introduction                   |
| 2.0 | Objectives                     |
| 3.1 | Classes and their Members      |
| 3.2 | Defining An Object             |
| 3.3 | Classes Versus Objects         |
| 3.4 | Accessing Class Members        |
| 3.5 | Access Modifiers               |
| 3.6 | Implementing Class Methods     |
| 4.0 | Conclusion                     |
| 5.0 | Summary                        |
| 6.0 | Tutor Marked Assignments       |
| 7.0 | References And Other Resources |

### 1.0 introduction

Classes extend the built-in capabilities of C++ to assist you in representing and solving complex, real world problems. It is the beginning of the representation of the object-oriented programming in C++.

Objects are simply instances of classes as variables are instances of primitive types.

### 2.0 Objectives

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

- Identify classes and objects.
- define a new class and create objects of that class.
- describe member functions and data members.
- define constructors and how to use them.

### 3.1 Classes And Their Members

You make a new type by declaring, a class. A class is just a collection of variables--often of different types--combined with a set of related functions.

Often: way to think about a car is as a collection of wheels, doors, seats, windows, and so forth. Another way is to think about what a car can do: It can move, speed up, slow down, stop, park, and so on. A class enables you to encapsulate, or bundle, these various parts and various functions into one collection, which is called an object.

Encapsulating everything you know about a car into one class has a number of advantages for a programmer. Everything is in one place, which makes it easy to refer to, copy, and manipulate the data. Likewise, clients of your class--that is, the parts of the program that use your class--can use your object without worry about what is in it or how it works.

A class can consist of any combination of the variable types and also other class types. The variables in the class are referred to as the member variables or data members. A Car class might have member variables representing the • seats, radio type, tires, and so forth.

New Term: Member variables , also known as data members , are the variables in your class, Member variables are part of your class, just like the wheels and engine are part of your car.

The functions in the class typically manipulate the member variables. They are referred to as member functions or methods of the class. Methods of the Car class might include Start() and Brake(). A Cat class

might have data members that represent age and weight; its methods might include Sleep(), Meow(), and Chase Mice().

New Term: Member functions , also known as methods , are the functions in your class. Member functions are as much a part of your class as the member variables. They determine what the objects of your class can do.

### Declaring A Class

To declare a class, use the class keyword followed by an opening brace, and then list the data members and methods of that class. End the declaration with a closing brace and a semicolon. Here's the declaration of a class called Cat:

```
class Cat
```

```
• unsigned int itsAge;
•, unsigned int its Weight;
 Meow();
```

Declaring this class doesn't allocate memory for a Cat. It just tells the compiler what a Cat is, what data it contains (itsAge and its Weight), and what it can do (Meow()). It also tells the compiler how big a Cat is--that is, how much room the compiler must set aside for each Cat that you create. In this example, if an integer is two bytes, a Cat is only four bytes big: itsAge is two bytes; and its Weight is another two bytes. Meow() takes up no room, because no storage space is set aside for member functions (methods).

### 32 Defining An Object

You define an object of your new type just as you define an integer variable:

```
unsigned int Gross Weight; // define an unsigned integer
```

```
Cat Frisky; // define a Cat
```

This code defines a variable called Gross Weight whose type is an unsigned integer. It also defines Frisky, which is an object whose class (or type) is Cat.

### 3.3' Classes Versus Objects

You never pet the definition of a cat; you pet individual cats. You draw a distinction between the idea of a cat, and the particular cat that right now is shedding all over your living room. In the same way, C++ differentiates between the class Cat, which is the idea of a cat, and each individual Cat object. Thus, Frisky is an object of type Cat in the same way in which Gross Weight is a variable of type unsigned int.

New Term: An object is an individual instance of a class.

### 34 Accessing Class Members

Quite you define an actual Cat object--for example, Frisky--you use the dot operator (.) to access the members of that object. Therefore, to assign 50 to Frisky's Weight member variable, you would write Frisky.Weight = 50;

In the same way, to call the Meow() function, you would write Frisky.

```
Meow();
```

When you use a class method, you call the method. In this example, you are calling Meow() on Frisky.

#### 3.4.1 Assign to Objects, Not to Classes

In C++ you don't assign values to types; you assign values to variables. For example, you would never write

```
int = 5; // wrong
```

The compiler would flag this as an error, because you can't assign 5 to an integer. Rather, you must define an integer variable and assign 5 to that variable. For example,

```
int x; // define x to be an int
```

```
x = 5; // set x's value to 5
```

This is a shorthand way of saying, "Assign 5 to the variable x, which is of type int." In the same way, you wouldn't write

```
Cat.age=5; // wrong
```

The compiler would flag this as an error, because you can't assign 5 to the age part of a Cat. Rather, you must define a Cat object and assign 5 to that object. For example,

```
Cat Frisky; // just like int x;
```

```
Frisky.age = 5; // just like x = 5;
```

### 3.4.2 If You Don't Declare It, Your Class Won't Have It

Try this experiment: Walk up to a three-year-old and show her a cat. Then say, "This is Frisky. Frisky knows a trick. Frisky, bark." The child will giggle and say, "No, silly, cats can't bark."

If you wrote

```
Cat Frisky; // make a Cat named Frisky
Frisky.Bark() // tell Frisky to bark
```

the compiler would say, No, silly, Cats can't bark. (Your compiler's wording may vary). The compiler knows that Frisky can't bark because the Cat class doesn't have a Bark() function. The compiler wouldn't even let Frisky meow if you didn't define a Meow() function.

## 3.5 Access Modifiers

There are three keywords in C++ that determine the level of access the users will have on objects. The declaration of these must be done at class declaration level. They are private, public and protected. We shall discuss private and public in this unit and leave protected for the next.

### 3.5.1 Private Versus Public

Other keywords are used in the declaration of a class. Two of the most important are public and private.

All members of a class--data and methods--are private by default. Private members can be accessed only within methods of the class itself. Public members can be accessed through any object of the class. This distinction is both important and confusing. To make it a bit clearer, consider an example from earlier in this unit:

```
class Cat
{
 unsigned int itsAge;
 unsigned int itsWeight;
 Meow 0;
 I;
```

In this declaration, `itsAge`, `its Weight`, and `Meow()` are all private, because all members of a class are private by default. This means that unless you specify otherwise, they are private

However, if you write

```
Cat Boots;
Boots.itsAge=5; // error! can't access private data!
```

the compiler flags this as an error. In effect, you've said to the compiler, "I'll access `itsAge`, `its Weight`, and `Meow()` only from within member functions of the `Cat` class." Yet here you've accessed the `itsAge` member variable of the `Boots` object from outside a `Cat` method. Just because `Boots` is an object of class `Cat`, that doesn't mean that you can access the parts of `Boots` that are private.

This is a source of endless confusion to new C++ programmers. Why can't `Boots` access his own age?" The answer is that `Boots` can, but you can't. `Boots`, in his own methods, can access all his parts--public and private. Even though you've created a `Cat`, that doesn't mean that you can see or change the parts of it that are private.

The way to use `Cat` so that you can access the data members is

```
class Cat
```

```
public:
unsigned int itsAge; unsigned int its Weight; Meow 0;
```

Now `itsAge`, `its Weight`, and `Meow()` are all public. `Boots.itsAge=5` compiles without problems.

Listing 6.1 shows the declaration of a `Cat` class with public member variables.

Listing 6.1. Accessing the public members of a simple class.

```
1: // Demonstrates declaration of a class and
2: // definition of an object of the class, 3:
4: #include <ostream.h> // for cout 5:
6: class Cat // declare the class object 7.
8: public: // members which follow are public 9: int itsAge;
10: int its Weight; 11.);
12:
13:
14: void main()
```

```
15:
16: Cat Frisky;
17: Frisky.itsAge = 5; //assign to the member variable
18: cout "Frisky is a cat who is " ;
19: cout « Frisky.itsAge « " years oldAn";
20:
```

Output: Frisky is a cat who is 5 years old.

**Analysis:** Line 6 contains the keyword `class`. This tells the compiler that what follows is a declaration. The name of the new class comes after the keyword `class`. In this case, it is `Cat`. The body of the declaration begins with the opening brace in line 7 and ends with a closing brace and a semicolon in line 11. Line 8 contains the keyword `public`, which indicates that everything that follows is public until the keyword `private` or the end of the class declaration.

Lines 9 and 10 contain the declarations of the class members `itsAge` and `its Weight`

Line 14 begins the main function of the program. `Frisky` is defined in line 16 as an instance of a `Cat`--that is, as a `Cat` object. `Frisky`'s age is set in line 17 to 5. In lines 18 and 19, the `itsAge` member variable is used to print out a message about `Frisky`.

**NOTE:** Try commenting out line 8 and try to recompile. You will receive an error on line 17 because `itsAge` will no longer have public access. The default for classes is private access.

### 3.5.2 make data members private

As a general rule of design, you should keep the data members of a class private. Therefore, you must create public functions known as accessor /mutator methods to set and get the private member variables.

These access or /mutator methods are the member functions that other parts of your program call to get and set your private member variables.

**New Term:** A public accessor/getter method is a class member function used either to read the value of a private class member variable while a mutator/setter method is used to set its value.

use the data, instead of working through accessor/mutator functions. Accessor/mutator functions enable you to separate the details of how the data is stored from how it is used. This enables you to change how the data is stored without having to rewrite functions that use the data.

If a function that needs to know a Cat's age accesses `itsAge` directly, that function would need to be rewritten if you, as the author of the Cat class, decided to change how that data is stored. By having the function call `GetAge()`, your Cat class can easily return the right value no matter how you arrive at the age. The calling function doesn't need to know whether you are storing it as an unsigned integer or a long, or whether you are computing it as needed.

This technique makes your program easier to maintain. It gives your code a longer life because design changes don't make your program obsolete.

accessor/mutator methods. Note that this is not an executable listing.

Listing 6.2. A class with accessor/mutator methods.

```

1: // Cat class declaration
2: // Data members are private, public accessor/mutator methods
3: // mediate setting and getting the values of the private data
4:
5: class Cat
6:
7: public:
8: // public accessor/mutators
9: unsigned int GetAge();
10: void SetAge(unsigned int Age); 11:
12: unsigned int GetWeight();
13: void SetWeight(unsigned int Weight); 14:
15: // public member functions 16. Meow();
17:
18: // private data members 19: private:
20: unsigned int itsAge;
21: unsigned int its Weight;
22:
23:);
```

Analysis: This class has five public methods. Lines 9 and 10 contain the accessor/mutator methods for `itsAge`. Lines 12 and 13 contain the accessor/mutator methods for `its Weight`. These accessor/mutator functions set the member variables and return their values.

The public member function Meow() is declared in line 16. Meow() is not an accessor/mutator function. It doesn't get or set a member variable; it performs another service for the class, printing the word Meow.

The member variables themselves are declared in lines 20 and 21.

To set Frisky's age, you would pass the value to the SetAge() method, as in

```
Cat Frisky;
Frisky.SetAge(5); // set Frisky's age using the public accessor/mutator
```

### 3.5.3 Privacy Versus Security

Declaring methods or data private enables the compiler to find programming mistakes before they become bugs. Any programmer worth his consulting fees can find a way around privacy if he wants to. Structure, the inventor of C++, said, "The C++ access control mechanisms provide protection against accident--not against fraud." (ARM, 1990.)

**DO** declare member variables private. **DO** use public accessor/mutator methods. **DON'T** try to use private member variables from outside the class. **DO** access private member variables from within class member functions.

## 3.6 Implementing Class Methods

As you've seen, an accessor/mutator function provides a public interface to tilt; private data members of the class. Each accessor/mutator function, along with any other class methods that you declare, must have an implementation. III: implementation is called the function definition.

A member function definition begins with the name of the class, followed by two colons, the name of the function, and its parameters. Listing 6.3 shows the complete declaration of a simple Cat class and the implementation of its accessor/mutator function and one general class member function.

Listing 6.3. Implementing the methods of a simple class.

```
1: // Demonstrates declaration of a Class and
2: // definition of class methods,
3:
4: #include <ostream.h> // for cout
5:
```



```

6: class Cat // begin declaration of the class
7:
8: public: // begin public section
9: int GetAge(); // accessor/mutator function
10: void SetAge (int age); // accessor/mutator function
11: void Meow(); // general function
12: private: // begin private section
13: int itsAge; // member variable
14: 1 ;
15:
16: // GetAge, Public accessor/mutator function
17: // returns value of itsAge member
18: int Cat::GetAge()
19: 1
20: return itsAge;
21: 1
22:
23: // definition of SetAge, public
24: // accessor/mutator function
25: // returns sets itsAge member
26: void Cat::SetAge(int age)
27:
28: II set member variable its age to
29: // value passed in by parameter age
30: itsAge =
31: 1
32:
33: // definition of Meow method
34: // returns: void
35: // parameters: None
36: // action: Prints "meow" to screen
37: void Cat::Meow()
38:
39: cout << "MeowAn";
40: 1
41:
42: // create a cat, set its age, have it
43: // meow, tell us its age, then meow again.
44: int main()
45:
46: Cat Frisky;
47: Frisky.SetAge(5);
48: Frisky.Meow();
49: cout << "Frisky is a cat who is
50: cout << Frisky.GetAge() << "
51: Friskv.Meown :
52: return 0;
53: 1
 Olt put: Meow.

```

Frisky is a cat who is 5 years old.

Meow.

Analysis: Lines 6-14 contain the definition of the Cat class. Line 8 contains the keyword `public`, which tells the compiler that what follows is a set of public members. Line 9 has the declaration of the public accessor/mutator method `GetAge()`. `GetAge()` provides access to the private member variable `itsAge`, which is declared in line 13. Line 10 has the public accessor/mutator function `SetAgeQ`. `SetAge()` takes an integer as an argument and sets `itsAge` to the value of that argument. Line 11 has the declaration of the class method `Meow()`. `Meow()` is not an accessor/mutator function. Here it is a general method that prints the word Meow to the screen.

private member variable `itsAge`. The class declaration ends with a closing brace and semicolon in line 14.

Lines 18-21 contain the definition of the member function `GetAge()`. This, method takes no parameters; it returns an integer. Note that class methods include the class name followed by two colons and the function name (Line 18). This syntax tells the compiler that the `GetAge()` function that you are defining here is the one that you declared in the Cat class. With the exception of this header line, the Retagged function is created like any other function.

The `GetAge()` function takes only One line; it returns the value in `itsAge`. Note that the `main()` function cannot access `itsAge` because `itsAge` is private to the Cat class. The `main()` function has access to the public method `GetAge()`. Because `GetAge()` is a member function of the Cat class, it has full access to the `itsAge` variable. This access enables `GetAge()` to return the value of `itsAge` to `main()`.

Line 26 contains the definition of the `SetAgeQ` member function. It takes an integer parameter and sets the value of `itsAge` to the value of that parameter in line 30. Because it is a member of the Cat class, `SetAge()` has direct access to the member variable `itsAge`.

Line 37 begins the definition, or implementation, of the `Meow()` method of the Cat class. It is a one-line function that prints the word Meow to the screen, followed by a new line. Remember that the `\n` character prints a new , line to the screen.

Line 44 begins the body of the program with the familiar `main()` function. In this case, it takes no arguments and returns void. In line 46, `main()` declares a Cat named Frisky. In line 47, the value 5 is assigned to the `itsAge` member variable by way of the `SetAge()` accessor/mutator method. Note that the

method is called by using the class name (Frisky) followed by the member operator (.) and the method name (SetAge()). In this same way, you can call any of the other methods in a class.

Line 48 calls the Meow() member function, and line 49 prints a message using the GetAge() accessor/mutator. Line 51 calls Meow() again.

#### 4.0 Conclusion

Classes act as a gateway to object-oriented programming in C++. The much desired technology is made available through the use of classes. It extends the privilege Of creating own types, as the primitive types are created by compiler developers, that suit the development by programmers, make them more in control and make their jobs much easier to accomplish.

#### 5.0 Summary

In this unit, you learned how to create new data types called classes. You learned how to define variables of these new types, which are called objects. A class has data members, which are variables of various types, including other classes. A class also includes member functions--also known as methods. You use these member functions to manipulate the data members an( to perform other services.

Class members, both data and functions, can be public or private. Public members are accessible to any part of your program. Private members are accessible only to the member functions of the class.

#### 6.0 Tutor Marked Assignments

1. How big is a class object? •
2. If I declare a class Cat with a private member itsAge and then define two Cat objects, Frisky and Bcsots, can Boots access Frisky's itsAge member variable?
3. Why shouldn't I make all the data members public?
4. If using a const function to change the class causes a compiler error, why shouldn't I just leave out the word const and be sure to avoid errors?
5. Is there ever a reason to use a structure in a C++ program?
6. What is the dot operator, and what is it used for?
7. Which sets aside memory - class declaration or definition?
8. Is the declaration of a class its interface or its implementation?
9. What is the difference between public and private data members'?
10. Can member functions be private?
11. Can data members be public?

12. Can two objects of the same class have different values in their data members?
13. Do class declarations end with a semicolon? Do class method definitions?

## 7.0 References And Other Resources

1. **Teach** yourself C++ in 21 days, Greg Wiegand, <http://www.mcp.com>
2. Borland C++ Object-Oriented Programming.

**UNIT 20: CLASSES II****TABLE OF CONTENTS**

|     |                              |
|-----|------------------------------|
| 1.0 | Introduction                 |
| 2.0 | Objectives                   |
| 3.1 | Constructors And Destructors |
| 3.2 | Structures                   |
| 4.0 | Conclusion                   |
| 5.0 | Summary                      |
| 6.0 | Tuitor Marked Assignment     |
| 7.0 | References & Other Resources |

**1.0 Introduction**

The idea of using classes is good, but if it does not give the flexibility and ease of use of the primitive types, the aim of its object-oriented nature will be defeated. Programs will still be as complex as ever and compiler developers will have created another problem while erasing one. The use of constructors and destructors helps a lot in doing away with this development. This unit tells about how the C++ programmer can use the constructor and destructor technology and achieve fantastic result.

**2.0 Objectives**

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

- describe constructors and destructors
- demonstrate how to declare constructors and destructors
- explain how to access object variables without changing them
- define inline

implementation

**3.1 Constructors And Destructors**

Constructors and destructors are simply functions, although special ones, in C++. A constructor is used in classes to initialize variables and create memory spaces. A destructor, on the other hand, is used to free memory spaces. We shall use the knowledge of primitive types to explain them. There are two ways to define an integer variable. You can define the variable

Or you can define the integer and immediately initialize it. For example,

```
int Weight = 7; // define and initialize to 7
```

Initialization combines the definition of the variable with its initial assignment. Nothing stops you from changing that value later. Initialization ensures that your variable is never without a meaningful value.

How do you initialize the member data of a class? Classes have a special member function called a constructor. The constructor can take parameters as needed, but it cannot have a return value--not even void. The constructor is a class method with the same name as the class itself.

Whenever you declare a constructor, you'll also want to declare a destructor. Just as constructors create and initialize objects of your class, destructors clean up after your object and free any memory you might have allocated. A destructor always has the name of the class, preceded by a tilde (~). Destructors take no arguments and have no return value. Therefore, the Cat declaration includes

```
~Cat ();
```

### 3.1.1 Default Constructors and Destructors

If you don't declare a constructor or a destructor, the compiler makes one for you. The default constructor and destructor take no arguments and do nothing.

What good is a constructor that does nothing? In part, it is a matter of form. All objects must be constructed and destructed, and these do-nothing functions are called at the right time. However, to declare an object without passing in parameters, such as

```
Cat Rags; // Rags gets no parameters you must have a constructor
in the form
Cat ();
```

When you define an object of a class, the constructor is called. If the Cat constructor took two parameters, you might define a Cat object by writing

```
Cat Frisky (5,7);
```

If the constructor took one parameter, you would write Cat Frisky (3); In the event that the constructor takes no parameters at all, you leave off the parentheses and write Cat Frisky ;

This is an exception to the rule that states all functions require parentheses, even if they take no parameters. This is why you are able to write

Cat Frisky;

which is a call to the default constructor. It provides no parameters, and it leaves off the parentheses. You don't have to use the compiler-provided default constructor. You are always free to write your own constructor with no parameters. Even constructors with no parameters can have a function body in which they initialize their objects or do other work.

As a matter of form, if you declare a constructor, be sure to declare a destructor, even if your destructor does nothing. Although it is true that the default destructor would work correctly, it doesn't hurt to declare your own. It makes your code clearer.

Listing 5.1 rewrites the Cat class to use a constructor to initialize the Cat object, setting its age to whatever initial age you provide, and it demonstrates where the destructor is called.

Listing 5.1 Using constructors and destructors.

```

1: // Demonstrates declaration of a constructors and
2: // destructor for the Cat class
3:
4: #include <iostream.h> //for cout
5:
6: class Cat //begin declaration of the class
7. I
8: public: // begin public section
9: Cat(int initialAge); // constructor
10: -Cat(); // destructor
11: int GetAge(); // accessor/mutator function
12: void SetAge(int age); // accessor/mutator function
13: void Meow();
14: p r i v a t e : // begin private section
15: i n t i t s A g e ; // member
variable
16: };
17:
18:// constructor of Cat,
19:Cat::Cat(int initialAge)
20:1
21: itsAge = initialAge;
22:j
23:
24:Cat::~-Cat()// destructor, takes no action
25:} // destructor, takes no action
26:)
```

```

27:
28:// GetAge, Public accessor/mutator function
29:// returns value of itsAge member
30: int Cat::GetAge()
31: {
32: return itsAge;
33: }
34:
35: // Definition of SetAge, public
36: // accessor/mutator function
37:
38: void Cat::SetAge(int age)
39: {
40: // set member variable its age to
41: // value passed in by parameter age
42: itsAge = age;
43: }
44:
'215: // definitiOn'of Meow method
46: // returns: void
47: // parameters: None
48: // action: Prints "meow" to screen
49: void Cat::Meow()
50: {
51: cout << "Meow.\n";
52: }
53:
54: // create a cat, set its age, have it
55: // meow, tell us its age., then meow again.
56: int main()
57: {
58: Cat Frisky(5);
59: Frisky.Meow();
60: cout << "Frisky is a cat who is " ;
61: cout << Frisky.GetAge() << "years old.\n";
62: Frisky.Meow();
63: Frisky.SetAge(7);
64: cout << "Now Frisky is " ;
65: cout << Frisky.GetAge() << " years old.\n";
66: }
67:

```

Output: Meow.

Frisky is a cat who is 5 years old. Meow.

Now Frisky is 7 years old.

Analysis: Line 9 adds a constructor that takes an integer. Line 10 declares the destructor, which takes no parameters. Destructors never



take parameters, and neither constructors nor destructors return a value--not even void.

Lines 19-22 show the implementation of the constructor. It is similar to the implementation of the SetAge() accessor/mutator function. There is no return value.

Lines 24-26 show the implementation of the destructor -Cato. This function does nothing, but you must include the definition of the function if you declare it in the class declaration.

Line 58 contains the definition of a Cat object, Frisky. The value 5 is passed in to Frisky's constructor. There is no need to call SetAge() because Frisky was created with the value 5 in its member variable itsAge, as shown in line 61. In line 63, Frisky's itsAge variable is reassigned to 7. Line 65 prints the new value.

DO use constructors to initialize your objects. DON'T give constructors or destructors a return value. DON'T give destructors parameters.

### 3.12 Constant Member Functions

If you declare a class method `const`, you are promising that the method won't change the value of any of the members of the class. To declare a class method constant, put the keyword `const` after the parentheses but before the semicolon. The declaration of the constant member function `SomeFunction()` takes no arguments and returns void. It looks like this:

```
void SomeFunction() const;
```

Accessor/mutator functions are often declared as constant functions by using the `const` modifier. The Cat class has two accessor/mutator functions:

```
void SetAge(int anAge); int GetAge() ;
```

`SetAge()` cannot be `const` because it changes the member variable `itsAge`. `GetAge()`, on the other hand, can and should be `const` because it doesn't change the class at all. `GetAge()` simply returns the current value of the member variable `itsAge`. Therefore, the declaration of these functions should be written like this:

```
void SetAge(int anAge); int GetAge() const;
```

If you declare a function to be const, and the implementation of that function changes the object by changing the value of any of its members, the compiler flags it as an error. For example, if you wrote `GetAge()` in such a way that it kept count of the number of times that the Cat was asked its age, it would generate a compiler error. This is because you would be changing the Cat object by calling this method.

NOTE: Use `const` whenever possible. Declare member functions to be `const` whenever they should not change the object. This lets the compiler help you find errors; it's faster and less expensive than doing it yourself.

Each time you do, you enable the compiler to catch your errors, instead of letting your errors become bugs that will show up when your program is running.

### 3.1.3 Interface Versus Implementation

As you've learned, clients are the parts of the program that create and use objects of your class. You can think of the interface to your class--the class declaration--as a contract with these clients. The contract tells what data your class has available and how your class will behave.

For example, in the Cat class declaration, you create a contract that every Cat will have a member variable `itsAge` that can be initialized in its constructor, assigned to by its `SetAge()` accessor/mutator function, and read by its `GetAge()` accessor/mutator. You also promise that every Cat will know how to `Meow()`.

If you make `GetAge()` a `const` function--as you should--the contract also promises that `GetAge()` won't change the Cat on which it is called.

C++ is strongly typed, which means that the compiler enforces these contracts by giving you a compiler error when you violate them. Listing 5.2 demonstrates a program that doesn't compile because of violations of these contracts.

WARNING: Listing 5.2 does not compile!

Listing 5.2. A demonstration of violations of the interface.

```
// Demonstrates compiler errors
```

```

3:
4: #include <over stream's> //Cat
5:
6: class Cat
7:
8: public:
9: Cat(int initialAge);
10: ~Cat();
11: int GetAge() const; // const accessor/mutator function
12: void SetAge (int age);
13: void Meow();
52
14: private:
15: int itsAge; 16:);
17:
18: // constructor of Cat,
19: Cat::Cat(int initialAge)
20: {
21: itsAge = initialAge;
21: cout << "Cat Constructor\n";
22:
23:
24: Cat::~Cat() // destructor, takes no action
25: {
26:
27:
28:
29:
30:
31:
32: return (itsAge++); // violates const!
33: 1
34:
35: // definition of SetAge, public
36: // accessor/mutator function
37:
38: void Cat:SetAge(int age)
39:
40: // set member variable its age to
41: // value passed in by parameter age
 itsAge = age;
43: 1 44:
45: // definition of Meow method
46: // returns: void

```

```

47: //
 cout << "Cat Destructor\n"; parameters: None
48: 1 GetAge, const function // action:
Prints "meow" to screen
49:
 1 but we violate const!
 int Cat::GetAge() const

 void Cat::Meow()
50: 1
51: cout << "Meow. n";
52: 1
53:
54: // demonstrate various violations of the
55 // interface, and resulting compiler errors
56 56: int main()
57:
58: Cat Frisky; // doesn't match declaration
59: Frisky.Meow();
60: Frisky.Bark(); // No, silly, cat's can't bark.
61: Frisky.itsAge = 7; // itsAge is private
62: return 0;
63:

```

Analysis: As it is written, this program doesn't compile. Therefore, there is no output. This program was fun to write because there are so many errors in it.

Line 11 declares GetAge() to be a const accessor/mutator function--as it should be. In the body of GetAge(), however, in line 32, the member variable itsAge is incremented. Because this method is declared to be const, it must not change the value of itsAge. Therefore, it is flagged as an error when the program is compiled..

In line 13, Meow() is not declared const. Although this is not an error, it is bad programming practice. A better design takes into account that this method doesn't change the member variables of Cat. Therefore, Meow() should be const.

Line 58 shows the definition of a Cat object, Frisky. Cats now have a constructor, which takes an integer as a parameter. This means that you must pass in a parameter. Because there is no parameter in line 58, it is flagged as an error.

Line 60 shows a call to a class method, Bark(). Bark() was never declared. Therefore, it is illegal.

Line 61 shows itsAge being assigned the value 7. Because itsAge is a private data member, it is flagged as an error when the program is compiled.

### 3.1.4 Where to Put Class Declarations and Method Definitions

Each function that you declare for your class must have a definition. The definition is also called the function implementation. Like other functions, the definition of a class method has a function header and a function body.

The definition must be in a file that the compiler can find. Most C++ compilers want that file to end with .0 or.CPP.

NOTE: Many compilers assume that files ending with C are C programs, and that C++ program files end with CPP. You can use any extension, but CPP will minimize confusion.

You are free to put the declaration in this file as well, but that is not good programming practice. The convention that most programmers adopt is to put the declaration into what is called a header file, usually with the same name but ending in H, HP, or HPP. This book names the header files with HPP, but check your compiler to see what it prefers.

For example, you put the declaration of the Cat class into a file named CATAPP, and you put the definition of the class methods into a file called CAT.CPP. You then attach the header file to the CPP file by putting the following code at the top of CAT.CPP:

```
#include Cat.hpp
```

This tells the compiler to read CAT.HPP into the file, just as if you had typed in its contents at this point. Why bother separating them if you're just going to read them back in? Most of the time, clients of your class don't care about the implementation specifics. Reading the header file tells them everything they need to know; they can ignore the implementation files.

**NOTE:** The declaration of a class tells the compiler what the class is, what data it holds, and what functions it has. The declaration of the class is called its interface because it tells the user how to interact with the class. The interface is usually stored in an HPP file, which is referred to as a header file. The function definition tells the compiler how the function works. The function definition is called the implementation of the class method, and it is kept in a CPP file. The implementation details of the class are of concern only to the author of the class. Clients of the class--that is, the parts of the program that use the class--don't need to know, and don't care, how the functions are implemented.

### 3.1.5 Milne Implementation

Just as you can ask the compiler to make a regular function inline, you can make class methods inline. The keyword `Inline` appears before the return value. The inline implementation of the `Get Weight()` function, for example, looks like this:

```
inline int Cat::GetWeight()
-
return its Weight; // return the Weight data member
```

You can also put the definition of a function into the declaration of the class, which automatically makes that function inline. For example,

```
class Cat

public:
int Get Weight() { return its Weight; // Inline
void Set Weight(int aWeight);
I;
```

Note the syntax of the `Get Weight()` definition. The body of the inline function begins immediately after the declaration of the class method; there is no semicolon after the parent-theses. Like any function, the definition begins with an opening brace and ends with a closing brace. As usual, white space doesn't matter; you could have written the declaration as

```
class Cat

public:
```

```

int Get Weight()
return its Weight; // inline
void Set Weight(int aWeight);
);

```

Listings 5.3 and 5.4 re-create the Cat class, but they put the declaration in CAT.HPP and the implementation of the functions in CAT.CPP. Listing 5.4 also changes the accessor/mutator functions and the Meow() function to inline.

Listing 5.3. Cat class declaration in CAT.HPP

```

1: #include <ostream.h>
2: class Cat
3: I
4: public:
5: Cat (int initial Age);
6: ~Cat();
7: int GetAge() I return itsAge;) // inline!
8: void SetAge (int age) (itsAge = age;) // inline!
9: void Meow() (cout << "MeowAn":1 inline!
10: private:
11: int itsAge;
12: I;

```

Listing 5.4. Cat implementation in CAT.CPP.

```

If Demonstrates inline functions
// and inclusion of header files

3:
4: finclude "cat.hpp" I/be sure to include the header files!
5:
6:
7: Cat::Cat(int initialAge) //constructor
8:
9: itsAge = initialAge;
10:
11:
12: Cat::~Cat() //destructor, takes no action
13: 1
14:)
15:
16: // Create a cat, set its age, have it
17: // meow, tell us its age. then meow again.
18: int main()
19:

```

```

20: Cat Frisky(5);
21: Frisky.Meow();
22: cout << "Frisky is a cat who is "
23: cout << Frisky.GetAge() << "
24: Frisky.Meow();
25: Frisky.SetAge(7);
26: cout << "Now Frisky. is "
27: cout << Frisky.GetAge() << "
28: return 0;
29: 1

```

Output: Meow.

Frisky is a cat who is 5 years old. Meow.

Now Frisky is 7 years old.

Analysis: The code presented in Listing 5.3 and Listing 5.4 is similar to the code in Listing 5.1, except that three of the methods are written inline in the declaration file and the declaration has been separated into CAT.HPP.

GetAge() is declared in line 7, and its inline implementation is provided. Lines 8 and 9 provide more inline functions, but the functionality of these functions is unchanged from the previous "outline" implementations.

Line 4 of Listing 5.4 shows #include "cat.hpp", which brings in the listings from CATI-IPP. By including cat.hpp, you have told the precompiler to read cat.hpp into the file as if it had been typed there, starting on line 5.

This technique allows you to put your declarations into a different file from your implementation, yet have that declaration available when the compiler needs it. This is a very common technique in C++ programming. Typically, class declarations are in an HPP file that is then #included into the associated CPP file.

Lines 18-29 repeat the main function from Listing 5.1. This shows that making these functions inline doesn't change their performance.

```

35: void SetUpperRight(Point Location) (itsUpperRight
 Location;
36: void SetLowerRight(Point Location) (itsLowerRight
 Location; }
37:
38: void SetTop(int top) (itsTop = top; 1

```



```

31: 1
32:
33. int main()
34. {
35. //initialize a local Rectangle variable
36. Rectangle MyRectangle (100, 20, 50, 80);
37.
38. int Area = MyRectangle.GetArea();
39.
40. cout << "Area:"
41. cout << "Upper Left X Coordinate: ";
42. cout << MyRectangle.GetUpperLeft().GetX()
43. return 0;
44. }

```

Output: Area: 3000  
Upper Left X Coordinate: 20

Analysis: Lines 3-14 in Listing 5.5 declare the class Point, which is used to hold a specific x,y coordinate on a graph. As written, this program doesn't use Points much. However, other drawing methods require Points.

Within the declaration of the class Point, you declare two member variables (itsX and itsY) on lines 12 and 13. These variables hold the values of the coordinates. As the x-coordinate increases, you move to the right on the graph. As the y-coordinate increases, you move upward on the graph. Other graphs use different systems. Some windowing programs, for example, increase the y-coordinate as you move down in the window.

The Point class uses inline accessor/mutator functions to get and set the X and Y points declared on lines 7-10. Points use the default constructor and destructor. Therefore, you must set their coordinates explicitly.

Line 17 begins the declaration of a Rectangle class. A Rectangle consists of four points that represent the corners of the Rectangle.

The constructor for the Rectangle (line 20) takes four integers, known as top, left, bottom, and right. The four parameters to the constructor are copied into four member variables (Listing 5.6) and then the four Points are established.

In addition to the usual accessor/mutator functions, `Rectangle` has a function `GetArea()` declared in line 43. Instead of storing the area as a variable, the `GetArea()` function computes the area on lines 28-29 of Listing 5.6. To do this, it computes the width and the height of `Ole` rectangle, and then it multiplies these two values.

Getting the x-coordinate of the upper-left corner of the rectangle requires that you access the Upper Left point, and ask that point for its X value. Because `GetUpperLeft()` is a method of `Rectangle`, it can directly access the private data of `Rectangle`, including `itsUpperLeft`. Because `itsUpperLeft` is a `Point` and

`Point`'s `itsX` value is private, `GetUpperLeft()` cannot directly access this data. Rather, it must use the public accessor/mutator function `GetX0` to obtain that value.

Line 33 of Listing 5.6 is the beginning of the body of the actual program. Until line 36, no memory has been allocated, and nothing has really happened. The only thing you've done is tell the compiler how to make a point and how to make a rectangle, in case one is *ever* needed.

In line 36, you define a `Rectangle` by passing in values for Top, Left, Bottom, and Right.

In line 38, you make a local variable, `Area`, of type `int`. This variable holds the area of the `Rectangle` that you've created. You initialize `Area` with the value returned by `Rectangle`'s `GetArea0` function.

A client of `Rectangle` could create a `Rectangle` object and get its area without ever looking at the implementation of `GetArea0`.

`RECT.HPP` is shown in Listing 5.5. Just by looking at the header file, which contains the declaration of the `Rectangle` class, the programmer knows that `GetArea0` returns an `int`. How `GetArea0` does its magic is not of concern to the user of class `Rectangle`. In fact, the author of `Rectangle` could change `GetArea0` without affecting the programs that use the `Rectangle` class.

### 3.2 Structures

A very close cousin to the `class` keyword is the keyword `struct`, which is used to declare a structure. In C++, a structure is exactly like a class, except that its members are public by default. You can declare a structure exactly as you declare a class, and you can give it exactly the same data members and functions. In fact, if you follow the good programming practice of always explicitly declaring the private and public sections of your class, there will be no difference whatsoever.

Try re-entering Listing 5.5 with these changes:

- In line 3, change class Point to struct Point.
- In line 17, change class Rectangle to struct Rectangle.

Now run the program again and compare the output. There should be no change.

### 3.3.1 Why Two Keywords Do the Same Thing

You're probably wondering why two keywords do the same thing. This is an accident of history. When C++ was developed, it was built as an extension of the C language. C has structures, although C

structures don't have class methods. Bjarne Stroustrup, the creator of C++, built upon structs, but he changed the name to class to represent the new, expanded functionality.

- **DO** put your class declaration in a .h file and your implementation functions in a .cpp file. **DO** use const whenever you can. **DO** understand classes before you move on.

## 4.0 Conclusion

Constructors and destructors give classes the power it wields. This unit has introduced you to some concepts that enabled **you to extend your C++** compiler and work in an environment that is suitable for the nature of your software development.

## 5.0 Summary

Class constructors initialize objects. Class destructors destroy objects and are often used to free memory allocated by methods of the class.

It is good programming practice to isolate **the interface, or declaration**, of the implementation of the class methods is written in a file **with a CPP extension**.

## 6.0 Tutor Marked Assignment

1. If using a const function to change the class causes a compiler error, why shouldn't I just leave out the word const and be sure to avoid errors?
2. What do you understand as class composition?
3. Explain the importance of the keyword: inline
4. Is there ever a reason to use a structure in a C++ program?
5. Describe the functions of constructors and destructors.
6. What are access modifiers?
7. Write the code that declares a class called Employee with these data members: age, years Of **Service**, and **Salary**.
  - Write your own constructor and destructor for the class

- Use the right access modifiers for the members
- Use the appropriate accessor and mutator methods

## 7.0 References And Other Resources

Teach Yourself C++ in 21 days, Greg Wiegand, <http://www.mcmcom>  
 Borland C++ Object-Oriented Programming

Dcyvtivb tbikg 1'4

Adlego — (94)

••=-• lute • Com '

Imvdi

Lam

\* (S) for law

4h.31Lcæ

.....

Th



# NATIONAL OPEN UNIVERSITY OF NIGERIA

## Form QST1 Questionnaire

Dear Student,

While studying the units of this course, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill out and send us the following questionnaire, which pertains to this course. If you find the space provided insufficient, kindly use additional sheet.

Course Code: \_\_\_\_\_ Course Title: \_\_\_\_\_

1 How many hours did you need for studying each of these units?

|              |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|--------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Unit         | 1  | 2  | 3  | 4  |    |    | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| No. of hours |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Unit         | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |    |    |
| No. of hours |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

2. Which of these units do you find most difficult to understand?

3. Please give specific problem you find difficult with the unit.

---

---

---

---

5. How would you like the unit improved?

---

---

---

Please Mail to  
The Course Coordinator ..... THROUGH the Study Centre Manager  
National Open University of Nigeria  
Victoria Island,  
Lagos.



# NATIONAL OPEN UNIVERSITY OF NIGERIA

## Form QST2 Questionnaire

In the questions below, we ask you to reflect on your experience of the course as a whole.

|    |                                                                                                                              |                       |                       |                       |                                   |       |
|----|------------------------------------------------------------------------------------------------------------------------------|-----------------------|-----------------------|-----------------------|-----------------------------------|-------|
| 1  | Course Code and Title                                                                                                        |                       |                       |                       |                                   |       |
| 2  | Mother tongue                                                                                                                |                       |                       |                       |                                   |       |
| 3  | I am registered for a                                                                                                        |                       |                       | Degree/Programme      |                                   |       |
| 4  | Why did I choose to take this course?.....                                                                                   |                       |                       |                       |                                   |       |
| 5  | Which study unit did I enjoy the most and why <sup>9</sup>                                                                   |                       |                       |                       |                                   |       |
| 6  | Which study unit did I enjoy the least and why <sup>9</sup>                                                                  |                       |                       |                       |                                   |       |
| 7  | Was the course material easy to understand or difficult'?                                                                    |                       |                       |                       |                                   |       |
| 8  | Which particular topic do I understand better than before and how <sup>9</sup>                                               |                       |                       |                       |                                   |       |
| 9  | Does the course have any practical applications in the real world, e.g. for the work I currently do?...YES/NO? EXPLAIN ..... |                       |                       |                       |                                   |       |
| 10 | What aspects would I like to know more about or study further'?                                                              |                       |                       |                       |                                   |       |
|    | How could the course be improved'?                                                                                           |                       |                       |                       |                                   |       |
| 12 | <u>Other comments about the course (Please Tick).....</u>                                                                    |                       |                       |                       |                                   |       |
|    | <b>Items</b>                                                                                                                 | <b>Excellent</b>      | <b>Very Good</b>      | <b>Poor</b>           | <b>Give specific examples, if</b> |       |
|    |                                                                                                                              |                       | <b>Good</b>           |                       | <b>p o o r</b>                    |       |
|    | Presentation Quality                                                                                                         | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/>             | _____ |
|    | Language and Style                                                                                                           | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/>             | _____ |
|    | Illustrations Used<br>(diagrams, tables, etc.)                                                                               | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/>             | _____ |
|    | Conceptual Clarity                                                                                                           | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/>             | _____ |
|    | Self Assessment<br>Questions                                                                                                 | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/>             | _____ |
|    | Facilitators response to<br><b>TIN/A, Questions</b>                                                                          | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> | <input type="radio"/>             | _____ |