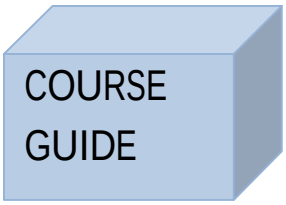
NATIONAL OPEN UNIVERSITY OF NIGERIA

COURSE CODE: CIT 467

COURSE TITLE: VISUAL PROGRAMMING & APPLICATIONS



CSC 467

VISUAL PROGRAMMING & APPLICATIONS

Course Developer/Writer — Dr J.N. Ndunagu

National Open University of Nigeria

Course Editor — Prof. H. C. Inyiama

Course Co-ordinator — Dr J.N. Ndunagu

NATIONAL OPEN UNIVERSITY OF NIGERIA

COURSE GUIDE CIT 467 - VISUAL PROGRAMMING & APPLICATIONS

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

Abuja Annex 245 Samuel Adesujo Ademulegun Street
Central Business District
Opposite Arewa Suites
Abuja

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

National Open University of Nigeria 2011

First Printed -------- -

ISBN:

Printed by ……..

For National Open University of Nigeria

TABLE OF CONTENTS

Introduction

Welcome to CIT 467: Visual Programming &Applications, which is a three credit unit course offered in the fourth year to students of the undergraduate degree programme in Computer Science. There are fifteen study Units in this course. There are no perquisites for studying this course. It has been developed with appropriate local and foreign examples suitable for the audience.

This course guide is for distance learners enrolled in the B.Sc. Computer Science programme of the National Open University of Nigeria. This guide is one of the several resource tools available to you to help you successfully complete this course and ultimately your programme.

In this guide you will find very useful information about this course: aims, objectives, what the course is about, what course materials you will be using; available services to support your learning; information on assignments and examination. It also offers you guidelines on how to plan your time for study; the amount of time you are likely to spend on each study unit; your tutor –marked assignments.

I strongly recommend that you go through this course guide and complete the feedback form at the end before you begin your study of the course. The feedback form must be submitted to your tutorial facilitator along with your first assignment.

I wish you all the best in your learning experience and successful completion of this course.

What you will learn in this Course

The overall aim of this course, CIT 467, is to create web pages using HTML. It starts with the basics and then moves on to the more advanced concepts. The Visual programming Languages and examples are also treated. You will also learn about creating lists, getting feedback with forms, creating tables and frames. Finally, you will be introduced to Java, dealing with Java variables and Operators, Statements and Control flow Statements.

Course Aim

This course aims at designing web pages using HTML and Java. You are not expected to have experience in the languages before using this course material. It is hoped that the knowledge would help you become proficient in HTML, fully versed in the language's syntax, semantics and elements of style.

Course Objectives

In order to achieve this aim, the course has a set of objectives. Each unit has specific objectives which are included at the beginning of the unit. You are expected to read these objectives before you study the unit. You may wish to refer to them during your study to check on your progress. You should always look at the unit objectives after completion of each unit. By doing so, you would have followed the instructions in the unit.

Below are the comprehensive objectives of the course as a whole. By meeting these objectives, you should have achieved the aim of the course. Therefore, after going through this course you should be able to:

- Define Visual Programming Language (VPL) in your own words
- Recount a brief history of Visual Programming Language
- Compare VPL and other types of programming Languages
- List 4 (four) examples of visual programming languages
- List four (4) examples of programming languages that are not VPL.

- Define an Icon in your own words
- Briefly describe formal specification of VPLs.

- Compare and Contrast Picture-processing grammars and Graph grammars
- Give two (2) examples of standard methods of parsing where context-free and context-dependent grammars are applicable.
- Describe the program -Alternate Reality Kit
- Compare and Contrast Visual Imperative Programming (VIPR) and C++
- Enumerate the benefits of Cube over non visual programming Languages
- List the uses of Form/3 programming Language
- Define HTML
- Describe the History of HTML
- Explain how HTML works
- Describe how to open a Notepad
- Describe markup tag in HTML
- Write a simple HTML Document using Notepad editor
- Describe the head and body section of HTML
- Create a headline for your document using head section
- Describe the paragraph marker.
- Describe the physical markup tags
- Describe the logical markup tags
- State the difference between formatting tags and preformatted text
- Define hypertext link
- List types and uses of hypertext link
- **S**tate the steps of creating hyperlinks
- Create hyperlinks
- Define unordered lists, ordered list, definition list and list within list
- State the steps in creating unordered lists, ordered list, definition list and list within list
- Distinguish between Ordered list and list within list

- Define an Object Oriented Programming in your own words.
- Differentiate between a Class and an Object
- Enumerate five (5) features of an Object Oriented Programming Language

- Define Java in your own words
- Enumerate the uses of Java
- Describe the History of Java
- Describe Standalone type of Java
- Create new classes for each real-world object that you observed.
- Display Java operators in order of precedence
- List variable types in Java

- Describe an array

- Declaring a Variable to refer to an array

- Create, Initialize and access an Array
- Describe the Switch Statement

- List the types of Branching Statements
- Differentiate between if- then and if-then –else Statements
- Write simple Java program using flow of control

Working through this Course

To complete this course, you are required to read each study unit, read the textbooks and read other materials which may be provided by the National Open University of Nigeria.

Each unit contains tutor marked assignments and at certain points in the course you would be required to submit assignment for assessment purposes. At the end of the course there is a final examination. The course should take you about a total of 14 weeks to complete. Below is the list of all the components of the course, what you have to do and how you should allocate your time to each unit in order to complete the course on time and successfully.

This course entails that you spend a lot of time to read and practice. For easy understanding of this course, I will advise that you avail yourself the opportunity of attending the tutorials sessions where you would have the opportunity to compare your knowledge with that of other people, and also have your questions answered.

The Course Material

The main components of this course are:

1. The Course Guide
2. Study Units
3. Further Reading/References
4. Assignments
5. Presentation Schedule

Study Units

There are 15 study units and 4 modules in this course. They are:

MODULE 1 –COURSE INTRODUCTION

Unit 1 Visual Programming Language

Unit 2 Theory of Visual Programming Languages

Unit 3 Examples of Visual Programming Languages

MODULE 2 – WEBSITE DESIGN

UNIT 1 –What is HTML?

UNIT 2 - Getting Started with HTML

UNIT 3 - Understanding the Basics of Html

MODULE 3 PHYSICAL MARKUP TAGS, HYPERTEXT LINKS, CREATING LISTS IN HTML

Unit 1 Formatting Text

Unit 2 Using Hypertext Links

Unit 3 Creating Lists In Html

MODULE 4: JAVA PROGRAMMING LANGUAGE

UNIT 1: Object Programming Languages

Unit 2: What is Java?

Unit 3: Variables and Operators in Java

Unit 4: Arrays and Expressions in Java
Unit 5: Control Flow Statement

Recommended Texts

These texts and references will be of enormous benefit to you in learning this course:

- Abbate, Janet. *Inventing the Internet*. Cambridge: MIT Press, 1999.

- About Java applets in Ruby
- An example of the 2005 year performance benchmarking
- Bemer, Bob, "A History of Source Concepts for the Internet/Web"

- Borning, A. H. (1981). The programming language aspects of thinglab, a constraint oriented simulation laboratory. *ACM Trans. Programming Languages and Systems*, 3(4):353–387, October 1981.
- Brown, M. and Sedgewick, R. (1984). A system for algorithm animation. In *Proc. Of SIGGRAPH '84*, pp. 177–186.
- Burnett, M. M. and Ambler, A. L. (1992). A declarative approach to event-handling in visual programming languages. In *Proc. 1993 IEEE Symposium Visual Languages*, pp. 34–40, Seattle, Washington.
- Burnett, M. M. and Baker, M. J. (1994). A classification system for visual programming languages. *J. Visual Languages and Computing*, pp. 287–300.
- Burnett, M., Baker, M., Bohus, C., Carlson, P., Yang, S. and Zee, P. (1995). "Scaling Up Visual Programming Languages", Computer 28(3), IEEE CS Press, pp. 45-54.

- Campbell-Kelly, Martin; Aspray, William. *Computer: A History of the Information Machine*. New York: BasicBooks, 1996.

- Chang, S. (1987). Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39.
- Chang, S. K., "A Visual Language Compiler for Information Retrieval by Visual Reasoning," IEEE Transactions on Software Engineering, pp. 1136-1149, 1990.
- Chang, S.-K., editor. *Principles of Visual Programming Systems*. Prentice Hall, New York, 1990.
- Citrin, W., Doherty, M., and Zorn, B. 1994. Design of a completely visual object-oriented programming language. In Burnett, M., Goldberg, A., and Lewis, T., editors, *Visual Object- Oriented Programming*. Prentice-Hall, New York..
- Citrin, W., Hall, R., and Zorn, B. 1995 Programming with visual expressions. In *Proc. 1995 IEEE Symposium Visual Languages*, pp. 294–301.

- Clark, David D., "The Design Philosophy of the DARPA Internet Protocols", Computer Communications Review 18:4, August 1988, pp. 106–114
- Cortado applet to play ogg format
- Cox P.T and Pietrzykowski T. 1988. "Using a Pictorial Representation to combine DataFlow and Object-orientation in a language independent programming mechanism", Proceedings of the International Computer Science Conference, pp. 695-704
- Cox, P. T. and Pietryzkowsky, T. 1990. Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism. In Glinert, E. P., editor, *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, CA.
- Erwig, M. and Meyer, B. (1995). Heterogeneous visual languages: Integrating visual and textual programming. In *Proc. 1995 IEEE Symposium Visual Languages*, pp. 318–325.
- Finzer, W. and Gould, L. (1984). Programming by Rehearsal. *BYTE*, 9(6):187–210,

- ***Generation of Source Code*** .NET, Java, C++, XSD, DDL, PHP, CORBA, Python & more. Free Trial! www.sparxsystems.com
- Golin, E. J. *A method for the specification and parsing of visual languages*. PhD dissertation, Brown University, 1990.
- Gosling, James; Joy Bill; Steele, Guy; and Bracha, Gillad (2005). *Java Language Specification* (3rd ed.). Addison-Wesley Professional. http://java.sun.com/docs/books/jls/index.html.

- Graham, Ian S. *The HTML Sourcebook: The Complete Guide to HTML*. New York: John Wiley and Sons, 1995.
- Java 2 SDK download http://java.sun.com/jdk
- ***Java Persistence Tools*** OpenJPA, Toplink, Hibernate Suppt No Lock-in, Eclipse-Based www.myeclipseide.com
- Java.Sun.com
- Java/C++ performance test http://www.geko.net.au/~sprack/perform/index.html
- JavaChannel.net
- JavaWorld.com
- Jraft.com
- Jython applet page
- Kawa, a basic Java development environment http://www.tek-tools.com/kawa
- Krol, Ed. *Hitchhiker's Guide to the Internet*, 1987.
- Krol, Ed. *Whole Internet User's Guide and Catalog*. O'Reilly & Associates, 1992.

- Lakin, F. Spatial parsing for visual languages. In Chang, S.-K., Ichikawa, T., and Ligomenides, P., editors, *Visual Languages*, pp. 35–85. Plenum Press, New York, 1986.
- ***Learn Ethical Hacking*** Ethical Hacking Training Bootcamp now in Lagos by Innobuzz! www.innobuzz.in/Hacking
- ***Multicore Programming*** Join Cavium University Program Free Teaching Material Lecture Code Lab University.Cavium.com
- Najork, M. 1995. Visual programming in 3-d. *Dr. Dobb's Journal*, 20(12):18–31.
- Najork, M. and Kaplan, S.1991. The cube language. In *Proc. 1991 IEEE Workshop Visual Languages*, pp. 218–224, Kobe, Japan, 1991
- ObjectPlanet.com, an applet that works as news ticker
- Parker, Richard O. 1993. Easy Object Programming for the Macintosh using AppMaker and THINK Pascal, Prentice-Hall, 1993
- Patrick Naughton , Herbert Schildt (1999). *Java 2: The Complete Reference*, third edition. The McGraw-Hill Companies, . ISBN 0-07-211976-4
- Paul Falstad online applet portal
- Performance tests show Java as fast as C++," Carmine Mangione (*JavaWorld*, February 1998)
- Rekers, J. and Sch¨urr, A. A graph grammar approach to graphical parsing. In *Proc. 1995 IEEE Symposium Visual Languages*, Darmstadt, Germany, 1995.
- Schmucker, 1994 "DemoDialogs in Prograph CPX", FrameWorks, Volume 8, Number 2, (March/April 1994), pp. 8-13.
- Schmucker, Kurt, 1988. Object-Oriented Programming for the Macintosh, Hayden,

- *Scientific American Special Issue on Communications, Computers, and Networks*, September, 1991
- Sferyx.com, a company that produces applets acting as WYSWYG editor.

- Smith, D. C .(1975). *PYGMALION: A Creative Programming Environment.* PhD dissertation, Stanford University.
- Smith, R. 1996. The alternate reality kit : An animated environment for creating interactive simulations.In *Proc. 1986 IEEE Workshop Visual Languages*, pp. 99–106, 1986.
- Smith, R. B. 1987. Experiences with the alternate reality kit: An example of the tension between literalism and magic. *IEEE CG & A*, 7(9):42–50, September 1987.
- Sutherland, I. B. (1963). SKETCHPAD, a man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pp. 329–346.
- TGS, 1989 - TGS Systems, "Prograph Syntax and Semantics", Prograph 2.5 manuals, Appendix IV, Sept. 1989 (first printing), July 1990 (second printing)
- The home site of the 3D protein viewer (Openastexviewer) under LGPL
- The home site of the chess applet under BSD
- The home site of the Mandelbrot set applet under GPL
- The Java Virtual Machine specification http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
- Top 13 Things Not to Do When Designing a Website
- Tortora, G.(1990) Structure and interpretation of visual languages. In Chang, S.-K., editor, *Visual Languages and Visual Programming*, pp. 3–30. Plenum Press, New York.
- Vermeulen, Ambler, Bumgardner, Metz, Misfeldt, Shur, Thompson (2000). *The Elements of Java Style*. Cambridge University Press, . ISBN 0-521-77768-2

Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are 15 tutor marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor marked assignment. You should therefore endeavor to meet the deadlines.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination. Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark. At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

Tutor Marked Assignments (TMAs)

There are 15 TMAs in this course. You need to submit all the TMAs. The best 4 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless on extraordinary cases.

Final Examination and Grading

The final examination on CIT 467 will last for a period of 3 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the tutor marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

Course marking Scheme

The following table includes the course marking scheme

Table 1 Course Marking Scheme

Assessment	Marks
Assignments 1-15	15 assignments, 40% for the best 4 Total = 10% X 4 = 40%
Final Examination	60% of overall course marks
Total	100% of Course Marks

Course Overview

This table indicates the units, the number of weeks required to complete them and the assignments.

Table 2: Course Organizer

Unit	**Title of the work**	**Weeks Activity**	**Assessment (End of Unit)**
	Course Guide	**Week**	
Module 1	**MODULE 1 –COURSE INTRODUCTION**		
Unit 1	Visual Programming Language	Week 1	Assessment 1
Unit 2	Theory of Visual Programming Languages	Week 2	Assessment 2

Unit 3	Examples of Visual Programming Languages	Week3	Assessment 3
Module 2	MODULE 2 – WEBSITE DESIGN		
Unit 1	Understanding the WWW	Week 4	Assessment 4
Unit 2	HTML and the Web	Week 5	Assessment 5
Unit 3	Getting Started with HTML	Week 6	Assessment 6
Unit 4	Understanding the Basics of HTML	Week 7	Assessment 7
Module 3	MODULE 3 PHYSICAL MARKUP TAGS, HYPERTEXT LINKS, CREATING LISTS IN HTML		
Unit 1	Formatting Text	Week 8	Assessment 8
Unit 2	Using Hypertext Links	Week 9	Assessment 9
Unit 3	Creating Lists In Html	Week 10	Assessment 10
Module 4	**MODULE 4: JAVA PROGRAMMING LANGUAGE**		
Unit 1	Object Programming Languages	Week 11	Assessment 11
Unit 2	What is Java?	Week 12	Assessment 12
Unit 3	Variables and Operators in Java	Week 13	Assessment 13
Unit 4	Arrays and Expressions in Java	Week 14	Assessment 14
Unit 5	Control Flow Statement	Week 15	Assessment 15

How to get the best from this course

In distance learning, the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study

materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

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

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

- Read this *Course Guide* thoroughly.
- Organize a study schedule. Refer to the 'Course Overview' for more details.

Note the time you are expected to spend on each unit and how the assignments relate to the units. Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.

- Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they lag behind in their course work.
- Turn to *Unit 1* and read the introduction and the objectives for the unit.
- Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.
- Work through the unit. The content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit you will be instructed to read sections from your set books or other articles. Use the unit to guide your reading.
- Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.
- When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
- When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments on the tutor-marked assignment form. Consult your tutor as soon as possible if you have any questions or problems.

- After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this *Course Guide*).

- The following might be circumstances in which you would find help necessary. Contact your tutor if:

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

You should try your best to attend the tutorials. This is the only chance to have face to face contact with your tutor and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussions actively. GOODLUCK!

MODULE 1 –COURSE INTRODUCTION

UNIT 1- VISUAL PROGRAMMING LANGUAGE?

CONTENTS

UNIT 2 – THEORY OF VISUAL PROGRAMMING LANGUAGES

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content

- 3.1 Generalized Icon
- 3.2 Formal Specification of Visual Programming Languages
- 3.3 Analysis of Visual Programming Languages
  - 3.3.1 Picture-Processing Grammars
  - 3.3.2 Precedence Grammars
  - 3.3.3 Context-Free and Context-Dependent Grammars
  - 3.3.4 Graph Grammars

4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment

7.0 References/Further Readings

UNIT 3 EXAMPLES OF VISUAL PROGRAMMING LANGUAGES

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content

- 3.1 ARK
- 3.2 VIPR
- 3.3 Prograph
- 3.4 Forms/3
- 3.5 Cube

4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment

7.0 References/Further Readings

MODULE 2 – WEBSITE DESIGN

U NIT 1 – Understanding the WWW

UNIT 3 - Getting Started with HTML

1.0 Introduction

2.0 Objectives

3.0 Main Content

- 3.1– How to Use Notepad
- 3.2– How to Use Markup Tags
- 3.3 - How to Write a Simple HTML Document
- 3.4 - How to Use Special HTML Editing Software

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignment

7.0 Further Reading and Other Resources

UNIT 4 - UNDERSTANDING THE BASICS OF HTML

1.0 Introduction

2.0 Objectives

3.0 Main Content

- 3.1– How to Use the Head Section
- 3.2- How to Use the Body Section
- 3.3 - How to Use Headings
- 3.4 - How to Use the Paragraph Tag
- 3.5 - How to Use Special Characters

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignment

7.0 Further Reading and Other Resources

MODULE 3 PHYSICAL MARKUP TAGS, HYPERTEXT LINKS, CREATING LISTS IN HTML

UNIT 1 FORMATTING TEXT

CONTENT

UNIT 2 USING HYPERTEXT LINKS

6.0 Tutor Marked Assignment

7.0 Further Reading and Other Resources

UNIT 3– CREATING LISTS IN HTML

1. 0. Introduction

2.0. Objectives

3.0 Main Content

3.1 How to Create Unordered Lists

3.2 How to Create Ordered Lists

3.3. How to Create Definition Lists

3.4.How to Create Lists within Lists

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignment

7.0 Further Reading and Other Resources

MODULE 4: JAVA PROGRAMMING LANGUAGE

UNIT 1: OBJECT PROGRAMMING LANGUAGES

2.0 Introduction

2.0 Objectives

3.0 Main Content

3.1 Evolution of A New Paradigm
3.2 What is Object Oriented Programming
3.3 History of Object Oriented Programming
3.4 Features of Object-Oriented Programming
3.4.1 Classes and Objects
3.4.2 Encapsulation

UNIT 2: JAVA LANGUAGE

UNIT 4 ARRAYS AND EXPRESSIONS IN JAVA

CONTENT

MODULE 1 VISUAL PROGRAMMING LANGUAGE

Unit 1 What is a Visual Programming Languages?

Unit 2 Theory of Visual Programming Languages

Unit 3 Examples of Visual Programming Languages

UNIT 1 WHAT IS A VISUAL PROGRAMMING LANGUAGE?

CONTENTS

1.0 INTRODUCTION

From cave paintings to hieroglyphics to paintings of Campbell's soup cans, humans have long communicated with each other using images. Many people think and remember things in terms of pictures. They relate to the world in an inherently graphical way and use imagery as a primary component of creative thought -Smith (1975). In addition, textual programming languages have proven to be rather difficult for many creative and intelligent people to learn to use effectively. Reducing or removing entirely the necessity of translating visual ideas into somewhat artificial textual representations can help to mitigate this steep learning curve problem. Furthermore, a variety of applications, including scientific visualization and interactive simulation authoring, lend themselves particularly well to visual development methods. This unit defines Visual Programming Language(VPL), describes the history and classifies it.

2.0 OBJECTIVES

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

- Define VPL in your own words
- Recount a brief history of Visual Programming Language
- Compare VPL and other types of programming Languages
- List 4 (four) examples of visual programming languages you know
- List four (4) examples of programming languages that are not VPL.

3.0 MAIN CONTENT

3.1 – What is a Visual Programming Language?

In computing, VPL is any programming language that allows the user to specify a program in a two-(or more) - dimensionsional way. Conventional textual languages are not considered two-dimensional since the compiler or interpreter processes them as one-dimensional streams of characters. A VPL allows programming with visual expressions - spatial arrangements of textual and graphical symbols. It is also known as any programming language that lets users create programs by manipulating program elements graphically rather than by specifying them textually. A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols used either as elements of syntax or secondary notation. Many VPLs are based on the idea of "boxes and arrows," where boxes or other screen objects are treated as entities, connected by arrows, lines or arcs which represent relations.

VPLs may be further classified, according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages. Visual programming environments provide graphical or iconic elements which can be manipulated by the user in an interactive way according to some specific spatial grammar for program construction.

A visually transformed language is a non-visual language with a superimposed visual representation. Naturally visual languages have an inherent visual expression for which there is no obvious textual equivalent. Visual basic, visual c++ and the entire microsoft Visual family are not, despite their names, visual programming languages. They are textual languages which use a graphical user interface to make programming interfaces easier. The user interface portion of the programming environment is visual, the languages are not. Because of the confusion caused by the multiple meanings of the term "visual programming", Fred Lakin has proposed the term "executable graphics" as an alternative to VPL.

Some examples of visual programming languages are prograph, pict, tinkertoy, fabrik, code 2.0 and hyperpascal.

The field of visual programming has grown from a marriage of work in computer graphics, programming languages, and human-computer interaction. It should come as no surprise, then, that much of the seminal work in the field is also viewed as pioneering work in one of the other disciplines. Ivan Sutherland's groundbreaking Sketchpad system stands out as the best example of this trend - Sutherland (1963). Sketchpad, designed in 1963 on the TX-2 computer at MIT, has been called the first computer graphics application. The system allowed users to work with a lightpen to create 2D graphics by creating simple primitives, like lines and circles, and then applying operations, such as copy, and constraints on the geometry of the shapes. Its graphical interface and support for user-specifiable constraints stand out as Sketchpad's most important contributions to visual programming languages. By defining appropriate constraints, users could develop structures such as complicated mechanical linkages and then move them about in real time. We will see the idea of visually specified constraints and constraint-oriented programming resurface in a number of later VPLs. Ivan Sutherland's brother, William, also made an important early contribution to visual programming in 1965, when he used the TX-2 to develop a simple visual dataflow language. The system allowed users to create, debug, and execute dataflow diagrams in a unified visual environment -Najork (1995).

The next major milestone in the genesis of VPLs came in 1975 with the publication of David Canfield Smith's PhD dissertation entitled "Pygmalion: A Creative Programming Environment" -Smith (1975). Smith's work marks the starting point for a number of threads of research in the field which continue to this day. For example, Pygmalion embodied an icon-based programming paradigm in which the user created, modified, and linked together small pictorial objects, called icons, with defined properties to perform computations. Much work has since gone into formalizing icon theory, as will be discussed below, and many modern VPLs employ an icon-based approach. Pygmalion also made use of the concept of programmingby- example wherein the user shows the system how to perform a task in a specific case and the system uses this information to generate a program which performs the task in general cases. In Smith's system, the user sets the environment to "remember" mode, performs the computation of interest, turns off "remember" mode, and receives as output a program, in a simple assembly-like subset of Smalltalk, which performs the computation on an arbitrary input.

The following presents a summary of the classification scheme discussed below:

1. Purely visual languages
2. Hybrid text and visual systems
3. Programming-by-example systems
4. Constraint-oriented systems
5. Form-based systems

The single most important category has to be purely visual languages. Such languages are characterized by their reliance on visual techniques throughout the programming process. The programmer manipulates icons or other graphical representations to create a program which is subsequently debugged and executed in the same visual environment. The program is

compiled directly from its visual representation and is never translated into an interim text-based language. Examples of such completely visual systems include VIPR, Prograph, and PICT. In much of the literature in the field, this category is further subdivided into sections like iconic and non-iconic languages, object-oriented, functional, and imperative languages. However, for our purposes a slightly larger granularity helps to emphasize the major visually-oriented differences between various VPLs.

One important subset of VPLs attempts to combine both visual and textual elements. These hybrid systems include both those in which programs are created visually and then translated into an underlying high-level textual language and systems which involve the use of graphical elements in an otherwise textual language. Examples in this category include Rehearsal World and work by Erwig et. Al (1995). In the former, the user trains the system to solve a particular problem by manipulating graphical "actors," and then the systems generates a Smalltalk program to implement the solution. The latter involves work on developing extensions to languages like C and C++which allow programmers to intersperse their text code with diagrams. For instance, one can define a linked list data structure textually and then perform an operation like deletion of a node by drawing the steps in the process.

In addition to these two major categories, many VPLs fall into a variety of smaller classifications. For example, a number of VPLs follow in the footsteps of Pygmalion by allowing the user to create and manipulate graphical objects with which to "teach" the system how to perform a particular task. Rehearsal World, described above, fits into this category of programming by example. Some VPLs can trace their lineage back, in part, to Sutherland's constraint manipulations in Sketchpad. These constraint-oriented systems are especially popular for simulation design, in which a programmer models physical objects as objects in the visual environment which are subject to constraints designed to mimic the behavior of natural laws, like gravity. Constraint-oriented systems have also found application in the development of graphical user interfaces. Thinglab and ARK, both primarily simulation VPLs, stand out as quintessential examples of constraint-based languages. A few VPLs have borrowed their visualization and programming metaphors from spreadsheets. These languages can be classified as form-based VPLs. They represent programming as altering a group of interconnected cells over time and often allow the programmer to visualize the execution of a program as a sequence of different cell states which progress through time -- Burnett & Ambler (1992). Forms/3 is the current incarnation of the progenitor of this type of VPL, and it will be covered in detail below. It is important to note that in each of the categories mentioned above, we can find examples of both general-purpose VPLs and languages designed for domain-specific applications.

The field of visual programming has evolved greatly over the last ten years. Continual development and refinement of languages in the categories discussed above have led to some work which was initially considered to be part of the field being reclassified as related to but not actually exemplifying visual programming. These VPL orphans, so to speak, include algorithm animation systems, such as BALSA-Brown & Sedgewick (1984), which provide interactive graphical displays of executing programs and graphical user interface development tools, like those provided with many modern compilers including Microsoft Visual C++. Both types of systems certainly include highly visual components, but they are more graphics applications and template generators than actual programming languages.

4.0 CONCLUSION

Visual basic, visual c++ and the entire microsoft Visual family are not, despite their names, visual programming languages. They are textual languages which use a graphical User Interface (GUI) to make programming interfaces easier. The user interface portion of the programming environment is visual, the languages are not.

5.0 SUMMARY

In this Unit you learnt:

- Definition of Visual Programming Language?
- History of Visual Programming Languages
- Classification of Visual Programming Languages

6.0 TUTOR-MARKED ASSIGNMENT

- Describe the history of Visual Programming
- Distinguish between Visual Programming language and any other type of language you know
- Define VPL in your own words

7.0 REFERENCES

- Burnett, M., Baker, M., Bohus, C., Carlson, P., Yang, S. and Zee, P. (1995). "Scaling Up Visual Programming Languages", Computer 28(3), IEEE CS Press, pp. 45-54.
- Borning, A. H. (1981). The programming language aspects of thinglab, a constraint oriented simulation laboratory. *ACM Trans. Programming Languages and Systems*, 3(4):353–387, October 1981.
- Brown, M. and Sedgewick, R. (1984). A system for algorithm animation. In *Proc. Of SIGGRAPH '84*, pp. 177–186.
- Burnett, M. M. and Ambler, A. L. (1992). A declarative approach to event-handling in visual programming languages. In *Proc. 1993 IEEE Symposium Visual Languages*, pp. 34–40, Seattle, Washington.
- Burnett, M. M. and Baker, M. J. (1994). A classification system for visual programming languages. *J. Visual Languages and Computing*, pp. 287–300.
- Chang, S. (1987). Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39.
- Erwig, M. and Meyer, B. (1995). Heterogeneous visual languages: Integrating visual and textual programming. In *Proc. 1995 IEEE Symposium Visual Languages*, pp. 318–325.
- Finzer, W. and Gould, L. (1984). Programming by Rehearsal. *BYTE*, 9(6):187–210,
- Najork, M. 1995. Visual programming in 3-d. *Dr. Dobb's Journal*, 20(12):18–31.
- Smith, D. C .(1975). *PYGMALION: A Creative Programming Environment.* PhD dissertation, Stanford University.

- Sutherland, I. B. (1963). SKETCHPAD, a man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pp. 329–346.

UNIT 2 THEORY OF VISUAL PROGRAMMING LANGUAGES

CONTENTS

1.0 INTRODUCTION

In this unit, we survey the theoretical advances in the field of Visual Programming Languages, mostly derived from early work by S.-K. Chang on generalized icon theory. To set up the framework for the discussion which follows, we put forth some definitions from Chang 1990.

2.0 OBJECTIVES

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

- Define an Icon in your own words
- Briefly describe formal specification of VPLs.
- Compare and Contrast Picture-processing grammars and Graph grammars
- Give two (2) examples of standard methods of parsing where context-free and context-dependent grammars are applicable.

3.0 MAIN CONTENT

3.1 – **Generalized Icon**

- An Icon is an object with the dual representation of a logical part (the meaning) and a physical part (the image).

- **Iconic system -**A structured set of related icons.

- **Iconic sentence (visual sentence) -** A spatial arrangement of icons from iconic system.
- **Visual language -** A set of iconic sentences constructed with given syntax and semantics.
- **Syntactic analysis (spatial parsing) -** An analysis of an iconic sentence to determine the underlying structure.
- **Semantic analysis (spatial interpretation) -**An analysis of an iconic sentence to determine the underlying meaning.

3.2 Formal Specification of Visual Programming Languages

A spatial arrangement of icons that constitutes a visual sentence is a two-dimensional counterpart of a one dimensional arrangement of tokens in conventional (textual) programming languages. In those languages, a program is expressed as a string in which terminal tokens are concatenated to form a sentence whose structure and meaning are discovered by syntactic and semantic analysis, respectively. Thus, the construction rule is implicit in the language and need not be spelled-out as part of the language specification. Conversely, in visual programming languages we distinguish three construction rules that are used to arrange icons: horizontal concatenation (denoted by &), vertical concatenation (denoted by ^), and spatial overlay (denoted by +).

In formalizing visual programming languages, it is customary to distinguish *process icons* from *object icons*. The former express computations; the latter can be further subdivided into *elementary object icons* and *composite object icons*. The elementary object icons identify primitive objects in the language, whereas the composite object icons identify objects formed by a spatial arrangement of the elementary object icons.

Finally, the term *elementary icons* are used to refer to both process icons and elementary object icons and denote those icons that are primitives in the language. Since a picture (or, icon, in our case) is worth a thousand words, we attempt to illustrate all of the above concepts in Figure 1 which demonstrates a few icons from the Heidelber icons set -Rhor (1986) and a complete visual sentence.

A visual programming language is specified by a triple (ID,G_0,B), where ID is the icon dictionary, G_0 is a grammar, and B is a domain-specific knowledge base -Tortora (1990). The icon dictionary is the set of generalized icons each of which is represented by a pair (Xm,Xi), with a logical art Xm (the meaning) and a physical part Xi (the image). The grammar G_0 specifies how composite object icons may be constructed from elementary icons by using spatial arrangement operators. Note that we need to specify spatial composition operators as terminals in the grammar precisely because they are no longer implicit in the language definition.
The knowledge base B contains domain-specific information necessary for constructing the meaning of a given visual sentence. It contains information regarding event names, conceptual relations, names of resulting objects, and references to the resulting objects.

3.3 Analysis of Visual Programming Languages

The syntactic analysis of visual sentences (also known as spatial parsing -Lakin (1986) is based upon a number of approaches- Chang (1990). Here, we present a partial listing of such approaches.

3.3.1 Picture-processing grammars

Originally designed to parse digital pictures on a square grid, these grammars are based on the fact that digital pictures are composed of pixels. These grammars discover the structure of visual sentence by composing individual pixels into recognizable visual elements (lines, arks, etc.) [Golin 1990]. This approach is useful when an iconic system needs to be able to recognize icons with a certain level of error tolerance (e.g. handwritten digits).

3.3.2 Precedence grammars

This spatial parsing grammar can be used for two-dimensional mathematical expression analysis and printed-page analysis. Precedence grammars are more suitable for syntactic analysis of visual sentences constructed from elementary icons and iconic operators. The parse tree is constructed by comparing precedences of operators in a pattern and subdiving the pattern into one or more subpatterns.

3.3.3 Context-free and context-dependent grammars

These grammars are used to specify composition of visual sentences using familiar formalisms, and so many standard methods of parsing such grammars are applicable.

3.3.4 Graph grammars

These are by far the most powerful (albeit least efficient) specifications of visual languages. These formalisms provide for the most means for establishing context relationships and much recent work has been devoted to making parsing with graph grammars computationally feasible -Rekers & Sch¨urr (1995).

A parse tree produced by one of the above parsing methods is subsequently analyzed using traditional approaches to semantic analysis (e.g. attribute grammars, ad-hoc tree computations, etc.).
Because the field of visual programming languages has only recently entered a more or less mature stage, much of the work on formalization of visual languages is still in its infancy

3.4 Visual Language Issues

Some common language issues in light of which the following presentation of visual languages is cast -Burnett . These issues are mostly applicable to *general-purpose visual languages* (suitable for producing executable programs of reasonable size), although certain issues will also be relevant to *domainspecific languages* (designed to accommodate a particular domain such as software engineering or scientific visualization).

3.4.1 Control Flow

Similarly to conventional programming languages, visual languages embrace two notions of flow of control in programs: imperative and declarative.

With the imperative approach, a visual program constitutes one or more control-flow or dataflow diagrams which indicate how the thread of control flows through the program. A particular advantage of such approach is that it provides an effective visual representation of parallelism. A disadvantage of this method is that a programmer is required to keep track of how sequencing of operations modifies the state of the program, which is not always an intended feature of the system (especially if it is designed to accommodate novices).

An alternative to imperative semantics of flow control is to use a declarative style of programming. With this approach, one only needs to worry what computations are performed, and not how the actual operations are carried out. Explicit state modification is avoided by using single assignment: a programmer creates a new object by copying an existing one and specifying the desired differences, rather than modifying the existent object's state. Also, instead of specifying a sequence of state changes, the programmer defines operations by specifying object dependencies. For example, if the programmer defines Y to be $X + 1$, this explicitly states that Y is to be computed using object in X, allowing the system to infer that X's value needs to be computed first. Thus, the sequencing of operations is still present, but must be inferred by the system rather than defined by the programmer. Off course, special care must be taken by the system that circular dependencies are detected and signaled as errors.

3.4.2 Procedural Abstraction

We distinguish two levels of procedural abstraction. High-level visual programming languages are not complete programming languages, i.e. it is not possible to write and maintain an entire program in such a language and inevitably there are some underlying non-visual modules that are combined using a visual language. This approach to visual programming is found in various domain-specific systems such as software maintenance tools and scientific visualization environments. At the opposite end of the scale, are low-level visual languages which do not allow the programmer to combine fine-grained logic into procedural modules.

This methodology is also useful in various domain-specific languages such as logic simulators. General purpose visual programming languages normally cover the entire spectrum of programming facilities ranging from low-level features, including conditionals, recursion, and iteration, to high-level facilities that allow one to combine low-level logic into abstract modules (procedures, classes, libraries, etc.).

3.4.3 Data Abstraction

Data abstraction facilities are only found in general-purpose programming languages. The notion of data abstraction in visual programming is very similar to the notion of data abstraction in conventional programming languages, with the only requirements being that abstract data types be defined visually (as opposed to textually), have a visual (iconic) representation, and provide for interactive behavior.

4.0 CONCLUSION

In this unit, we restrict our discussion to two-dimensional visual languages, although everything that follows can be generalized to three (and more) dimensions.

5.0 SUMMARY:

In this Unit you learnt:

- Generalized Icon
- Formal Specification of Visual Programming Languages
- Visual Language Issues
- Analysis of Visual Programming Languages

6.0 TUTOR-MARKED ASSIGNMENT

- Define Icon in your own words
- Briefly describe formal specification of VPLs.
- Compare and Contrast Picture-processing grammars and Graph grammars
- Give two (2) examples of standard methods of parsing where context-free and context-dependent grammars are applicable.

7.0 REFERENCES/FURTHER READINGS

- Chang, S. K., "A Visual Language Compiler for Information Retrieval by Visual Reasoning," IEEE Transactions on Software Engineering, pp. 1136-1149, 1990.
- Chang, S.-K., editor. *Principles of Visual Programming Systems.* Prentice Hall, New York, 1990.
- Golin, E. J. *A method for the specification and parsing of visual languages.* PhD dissertation, Brown University, 1990.
- Lakin, F. Spatial parsing for visual languages. In Chang, S.-K., Ichikawa, T., and Ligomenides, P., editors, *Visual Languages*, pp. 35–85. Plenum Press, New York, 1986.
- Rekers, J. and Sch¨urr, A. A graph grammar approach to graphical parsing. In *Proc. 1995 IEEE Symposium Visual Languages*, Darmstadt, Germany, 1995.
- Tortora, G. Structure and interpretation of visual languages. In Chang, S.-K., editor, *Visual Languages and Visual Programming*, pp. 3–30. Plenum Press, New York, 1990.

UNIT 3 EXAMPLES OF VISUAL PROGRAMMING LANGUAGES

CONTENTS

1.0 INTRODUCTION

This unit presents five (5) visual programming languages namely: ARK, VIPR, Prograph, Forms/3 and Cube with their samples.

2.0 OBJECTIVES

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

- Describe the program -Alternate Reality Kit
- Compare and Contrast VIPR and C++
- Enumerate the benefits of Cube over non visual programming Languages
- List the uses of Form/3 programming Language

3.0 MAIN CONTENT

3.1 ARK

More than 10 years after its inception, the Alternate Reality Kit (ARK), designed by R. Smith at Xerox PARC, remains one of the more unique and visionary domain-specific VPLs. ARK, implemented in Smalltalk- 80, provides users with a 2D animated environment for creating interactive simulations. The system is intended to be used by non-expert programmers to create simulations and by an even wider audience to interact with the simulations. In order to help users to understand the fundamental laws of nature, ARK uses a highly literal metaphor in which the user controls an on-screen hand which can interact with physical objects, like balls and blocks, which possess masses and velocities and with objects, called interactors,

representing physical laws, like gravity [Smith 1986]. By giving a kind of physical reality to abstract laws, the system attempts to remove some of the mystery surrounding the ways in which such laws interact with objects and each other. Users can modify any objects in the environment using constructs called message boxes and buttons, viewing the results of their changes in real time. The simulation runs in an "alternate reality" contained within a window inside an all-encompassing "meta-reality." The structure is very much like a modern windows-and-desktop GUI. The programmer can move the hand between alternate realities and pull objects out of the simulation and into meta-reality at anytime. An object which has been lifted out of its alternate reality does not participate in the simulation until it is dropped out of meta-reality.

The example of a user reaching into and removing an object from an alternate reality highlights one of the more interesting design issues in ARK, namely the necessity to occasionally break with the highly literal physical world metaphor in order to provide useful functionality. Smith refers to this issue as the tension between magic and literalism in ARK [Smith 1987]. While using a hand which can grab physical objects is a highly literal component of the system, allowing a user to reach into a simulation and alter or remove objects with no regard to the physical laws currently at work in the environment clearly provides the user with what could be considered "magical" powers. The question of when to allow a magical event or action in ARK to conflict with the physical metaphor parallels a similar concern in the design of more traditional VPLs. In developing most VPLs, researchers have had to decide on the appropriate uses of text in their system. While it is possible to design a system which uses no text whatsoever, and such systems have been created, the resulting programs are often very difficult to read and understand. Most VPLs, even those which are completely visual, use text, at the very least, to label variables and functions in programs. Thus designers must face the same problem as was addressed in ARK. They must attempt to balance consistency of visual representation with usability.

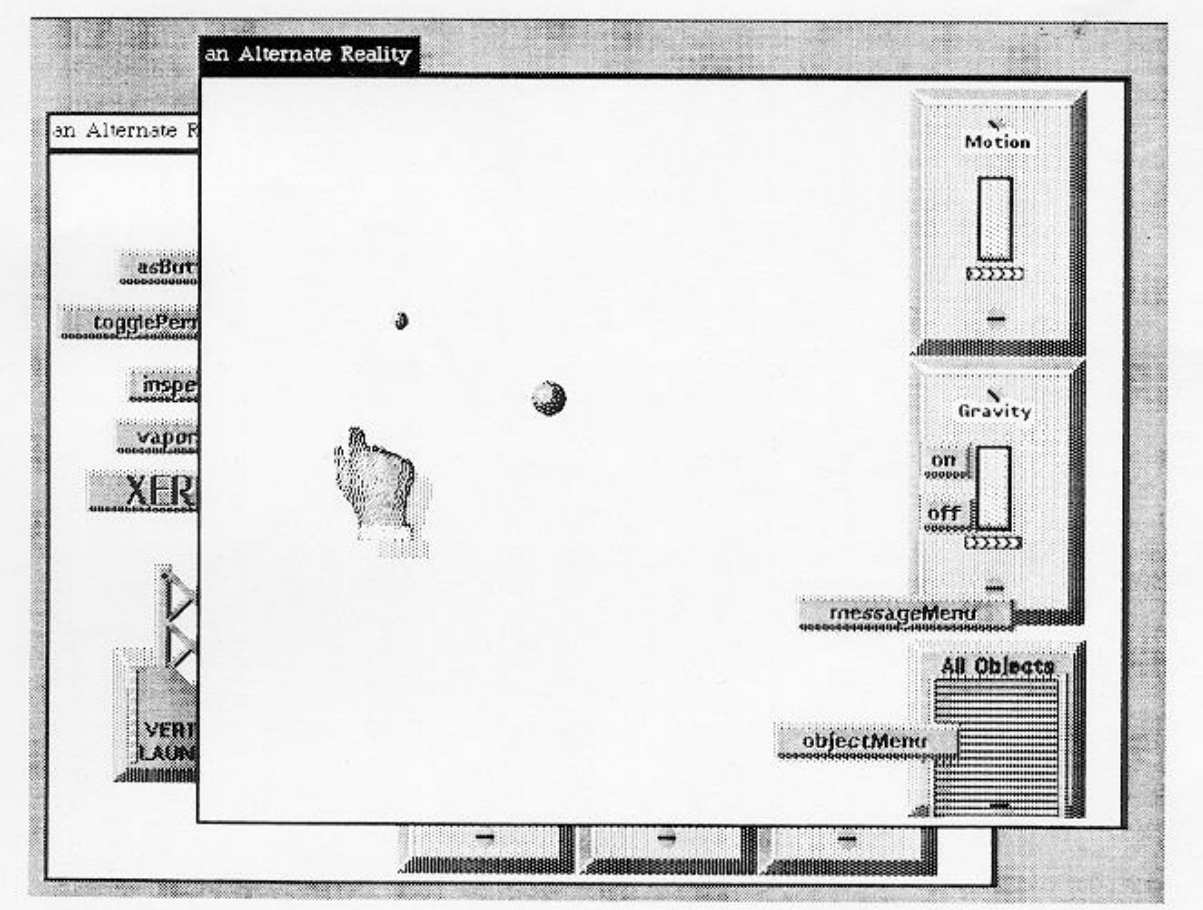


Figure 1: Planetary Orbit Simulation in ARK

Although ARK targets a fairly specific application domain, it supports a powerful programming model. Programmers can not only create simulations by linking together various pre-built objects and interactors, but they can also develop new interactors. Programming a simulation, such as the planetary orbit simulation shown in Figure 1, involves first generating physical objects, like balls, from the object warehouse in the lower right corner of the display. By clicking on the objectMenu button, the programmer can choose to instantiate any object available in the environment. After creating some physical objects, the programmer follows the same procedure to place interactors, like the Motion and Gravity objects in Figure1, in the alternate reality. The programmer now uses the messageMenu button to find out what sorts of messages to which the various interactors respond by placing the button on an interactor and pressing it with the hand. This generates a list of all messages appropriate to that interactor. The programmer chooses one, such as “off” for the gravity interactor in the orbit simulation, and the system generates a message box. Message boxes are objects which can send and receive Smalltalk messages. The programmer links the new message box, in our case one which generates the message “off”, to the appropriate interactor by joining them with a dotted line. The message box can then be collapsed to a single button as in Figure1. Interactors affect all the objects in the same alternate reality, so after specifying all the necessary controls, a programmer can begin the simulation.

It is important to note that all of the objects in the underlying Smalltalk environment are available to the ARK programmer. Objects which are not ARK-specific appear as

representative objects, like the TwoVector object in Figure 3. As shown in the figure, such Smalltalk objects can be linked with ARK objects in the same way as native objects. The example shown involves using a TwoVector object as the input to a button which sets the velocity of a disk.

Clearly, ARK interactors behave much like constraints on the physical objects in the alternate reality. Thus, creating and modifying interactors exemplify ARK's constraint-oriented features. A programmer can generate new interactors by creating networks of message boxes. As a simple example, consider developing a frictional force interactor by creating a message box which adds a force to an object proportional to the negative of its velocity [Smith 1987]. The message box can be set to continuously send its message, and when its behavior has been verified, the programmer can convert it to an interactor.

3.2 VIPR

VIPR, or Visual Imperative Programming, developed by Citrin et. al at the University of Colorado represents a unique approach to completely visual general purpose programming. Rather than relying on icons, forms, or other traditional graphical representations, VIPR uses nested series of concentric rings to visualize programs, as shown in Figure 3. Each step in a computation involves merging two rings in the presence of a state object which is connected to the outermost ring. One can visualize program execution as walking down a network of pipes which branches off in different directions while changing the state based on actions written on the inside of the pipes [Citrin et al. 1994].

The ongoing development of VIPR has been motivated, in part by a desire to create an object-oriented language which is relatively easy to learn and use. As a result, VIPR includes most of the common attributes of object-oriented languages, including inheritance, polymorphism, and dynamic dispatch. The language's semantics have been defined to be similar to C++in order to make it easier for experienced programmers to read and understand VIPR programs. However, the language is entirely visual, so the semantics of a VIPR program can be understood by applying a small number of graphical rewrite rules. This relationship to C++also means that VIPR provides support for both low- and high-level programming.

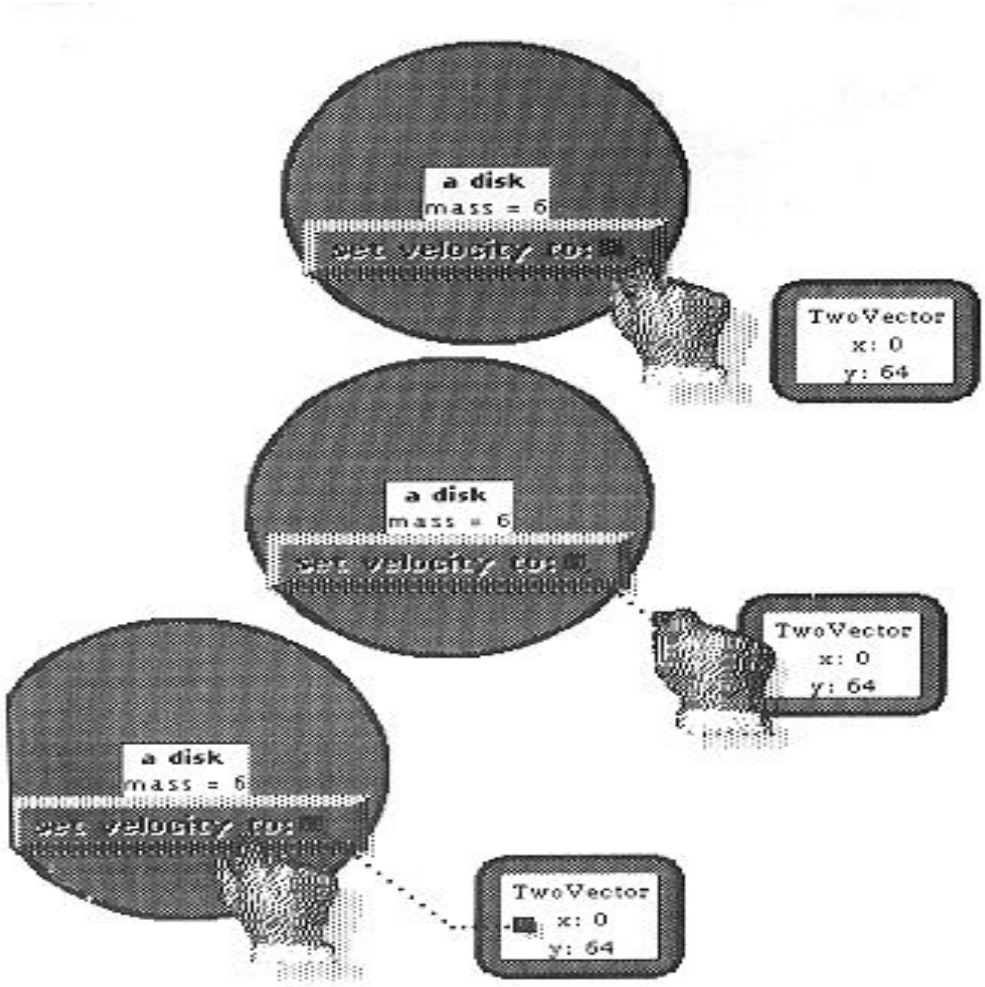


Figure 2: Accessing a Smalltalk object in ARK

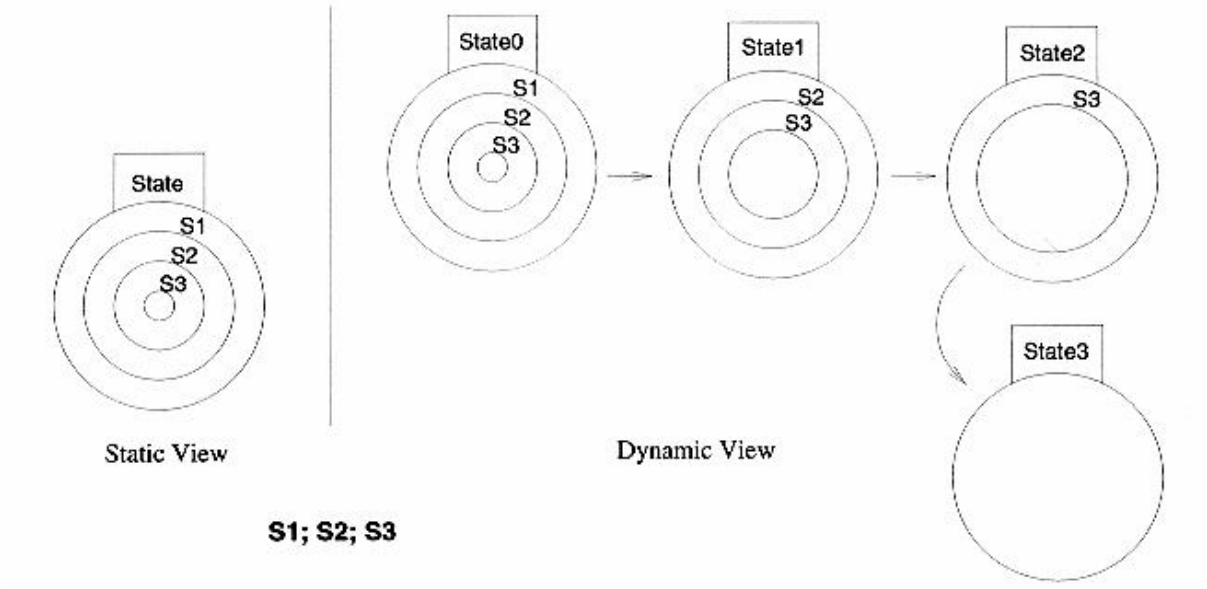


Figure 3: Visualization of program execution in VIPR

Figure 4 presents examples of if/then/else and while/do statements in VIPR. Arrows in the figures represent substitution. If execution reaches a ring from which an arrow emanates, then this ring can be replaced with the ring structure to which the arrow points. This idea of substitution becomes more important in function calls, as shown in Figure 5. The left side of the figure represents a call to the function defined on the right. Small circles internal and tangent to rings indicate parameters, so *m* is a parameter in the call to *fun*.

Every function must have at least one parameter, called the return address or continuation parameter. This parameter indicates the next statement to execute following the function call. The continuation parameter is always located in the lower right corner of the function definition and function call rings. All other parameters can be matched from function call to function definition by either their location with respect to the continuation parameter or by an optional label. For example, in Figure 5 parameter *m* in the call matches to *x* in the definition, because they are in the same location relative to the continuation parameter.

In the example shown, the small circle inside the continuation parameter ring indicates that the function returns a value. By performing the substitutions called for by the arrows in the figure, we can see that the variable result is passed through to *n* in the function call.

Figure 6 shows an example of defining a simple geometric point class in VIPR along with the equivalent C++class definition. The entire class is surrounded by a dotted ring. Private fields or methods are surrounded by double boxes and are not visible in an instance of a class until execution enters a method of the class.

Method definitions follow the scheme of function definitions discussed above. Figure 7, Figure 8 and Figure 9 show a small example program which makes use of the point class. Note that variables which have been declared to be pointers to a particular class but have not yet been initialized point to shaded instances of the class. These are referred to as pseudo-instances [Citrin et al. 1994]. The instances become unshaded after the variables are initialized. The only other important aspect of the example to note is the appearance of the self variable in the state upon entry into the “xDistance” method. Clearly, in a program with a fairly large number of classes and subroutines, the display could become rather crowded. In order to improve program visualization for large-scale projects, Citrin et. al have begun work on a variety of new display methods for the VIPR environment, including zooming and fisheyeing [Citrin et al. 1996].

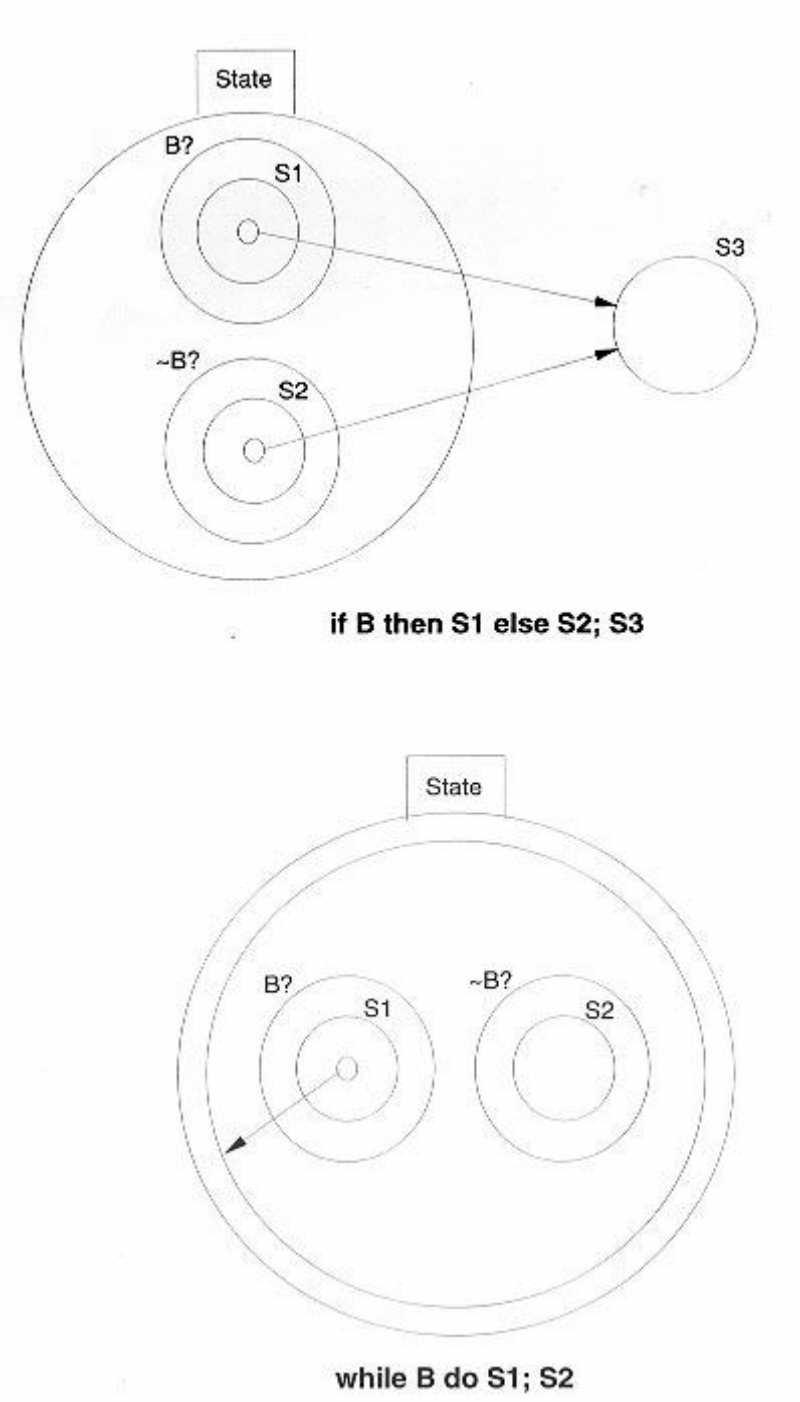


Figure 4: Example Control Constructs in VIPR

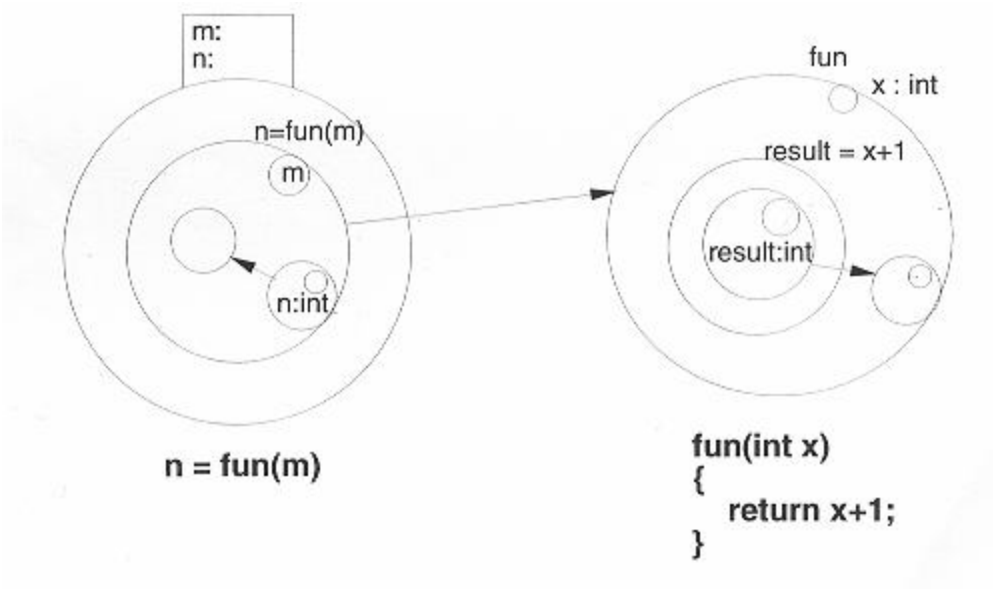


Figure 5: Function definition and call in VIPR

The VIPR group has also developed a visual representation for the lambda calculus which they refer to as VEX for Visual EXpressions. VEX is intended to become an expression-oriented component of VIPR [Citrin et al. 1995]. We will only take a brief look at its major features. Figure 10 shows the textual and visual representations for the Y combinator. As in VIPR, parameters are represented by small circles inside and tangent to main rings, so f and x are parameters in the example. Function application is represented by adjacent closed figures, and arrows point from the applied functions to their argument. In VEX, free and bound identifiers are easily recognized. Each identifier is connected by an undirected edge to a labeled root node. Free identifiers are connected to roots which are not inside and tangent to any rings, while bound identifiers are connected to internally tangent roots. Thus, in Figure 11 identifier 2 is free in the overall expression while identifier 5 is bound inside the expression represented by ring 3. Graphical equivalents have been devised for _-conversion, _-reduction, and _-reduction, but a detailed discussion of these is beyond the scope of this report.

3.3 Prograph

Prograph language is considered to be the most (commercially) successful of the general-purpose visual languages - Cox & Pietryzkowsky (1990).

The research on Prograph originated in 1982 at the Technical University of Nova Scotia. Since then, several versions of the language have been released, with the most recent (Prograph/CPX) being commercialized by Pictorius, Inc.

Prograph is a visual object-oriented language. It combines the familiar notions of classes and objects with a powerful visual dataflow specification mechanism. Prograph is an imperative language, providing explicit control over evaluation order. Of particular interest are Prograph's cases and multiplexes, the special control structures which are intended to replace explicit iteration and provide sophisticated flow control. We will discuss these as part of an example below.

Prograph allows the programmer to work on both high and low levels, allowing him or her to design and maintain more or less complicated software. Primitive computations such as arithmetic operators, method calls, etc. are combined to form method bodies by dataflow diagrams. The methods are then organized into classes, which are, in turn, organized into class hierarchies. In addition, Prograph provides the programmer

```
class point {
    int x, y;
 public:
    void moveTo(int newx, int newy)
      { x = newx; y = newy; }

    int xDistance(point* p)
      {
        // Statement 3
        return p->x - x;
      }
};
```

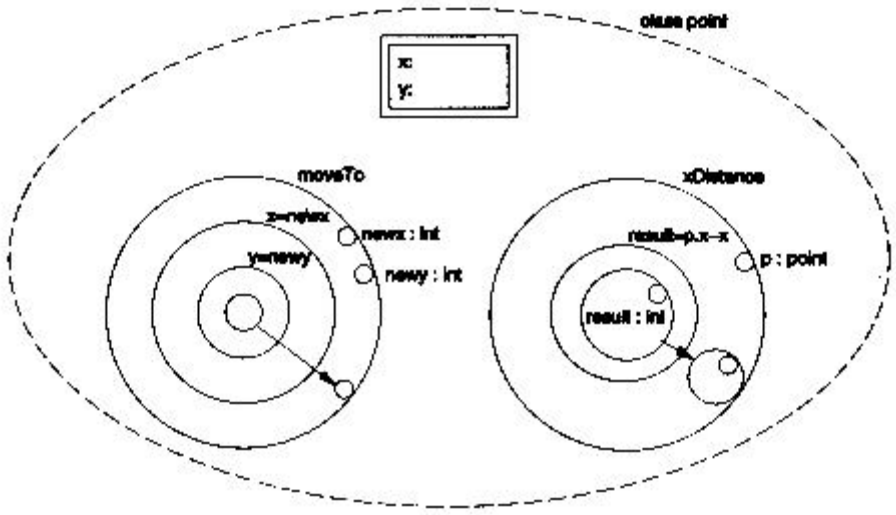
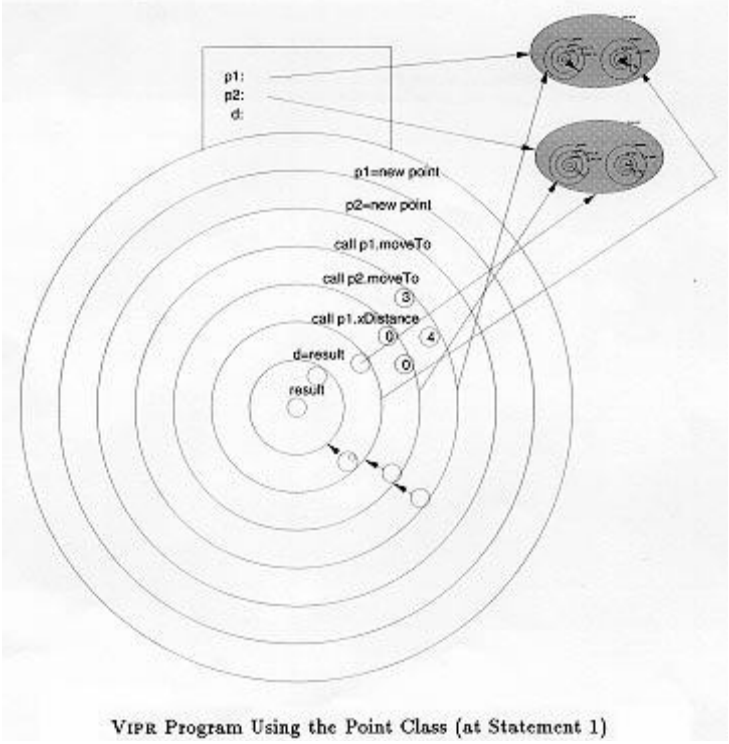


Figure 6: Definition of a point class in VIPR and the equivalent class

```
main()
{
    point *p1, *p2;
    int d;
                                // Statement 1
    p1 = new point;
    p2 = new point;
    p1->moveTo(3,4);
    p2->moveTo(0,0);
                                // Statement 2
    d = p1->xDistance(p2);
                                // Statement 4
}
```



VIPR Program Using the Point Class (at Statement 1)

Figure 7: Example program using point class

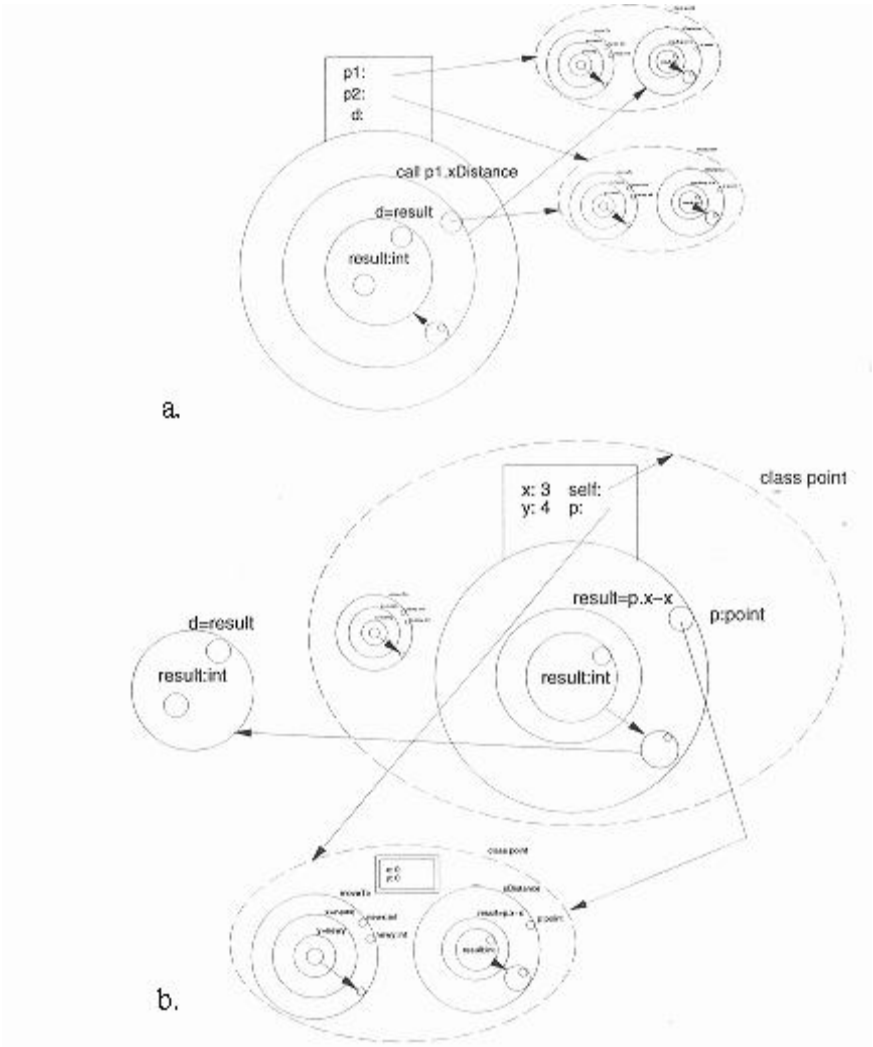


Figure 8: Example program using point class (Cont.) a. at Statement 2 b. inside xDistance

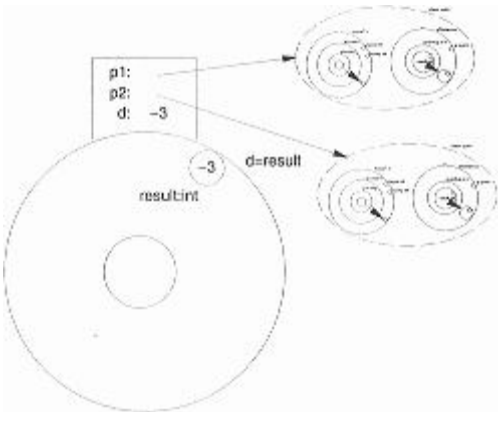


Figure 9: Example program using point class (Cont.) at Statement 4

with so-called persistent objects that can be stored in a Prograph database between different invocations of the program.

We will now introduce some of the more interesting Prograph features by means of an example of topological sort algorithm on directed graphs. Graphs will be represented by

adjacency lists, i.e. a graph is a list of lists, each of which corresponds to a node. A list representing a node consists of the name of the node, followed by names of all nodes at the heads of outer edges of this node.
Figure 12 depicts a Prograph program to perform topological sort. The “Methods” window contains the names of all functions in this program, with “CALL” being the top-level routine.

Methods consist of one or mores *cases*. Each case of a method is a dataflow diagram that specifies how the case is to be executed. Each dataflow primitive designates an operation to be performed on data which enters through one or more *terminals* (at the top of the operation icon). The output of an operation is produced at one or more *roots* (at the bottom of the operation icon). The edges of a flow diagram indicate how data propagates from the root of one operation to the terminals of the next, i.e. the order of execution is data-driven.

Method “CALL” (also shown on Figure 12) consists of two *cases*. The case number, as well as the total number of cases, is indicated in the title bar of the window containing the method’s dataflow diagram. It contains calls to two system methods “ask” and “show” (underlined method names designate system methods) and a call to a user-defined method “sort” whose definition is presented on Figure 13.

The execution of “CALL” begins by evaluating the result of calling method “ask” and the constant “()” – the empty list. “ask” obtains input from the user which becomes the value of its root. After the evaluation of “ask” and the constant is competed, their results are passed to “sort” which is the next operation to be executed. Note that the call to “sort” is accompanied by a control called *next-on-failure* represented by an icon to the left of the operation icon. This indicates that if call to “sort” *fails* (which implies that the graph contains a cycle; see below for the explanation on how this *failure* condition is generated), execution of case 1 of “CALL” should halt and case 2 should be executed.
In case 1 of “CALL”, the operation sort is a *multiplex* (which is pictorially distinguished a simple operation by being drawn in a three-dimensional fashion to emphasize the fact that it is executed many times). At least one of the roots or the terminals of the multiplex is annotated with an icon that indicates the type of the multiplex. The multiplex in Figure 13, is called an *iterative multiplex* (a different kind of multiplex is presented when we discuss the method “sort”). The annotations on this multiplex indicate that output values on the loop roots will be passed to the loop terminals as input values for the next iteration. The execution of the loop continues until a special control called *terminate-on-success* is not executed inside method “sort”.

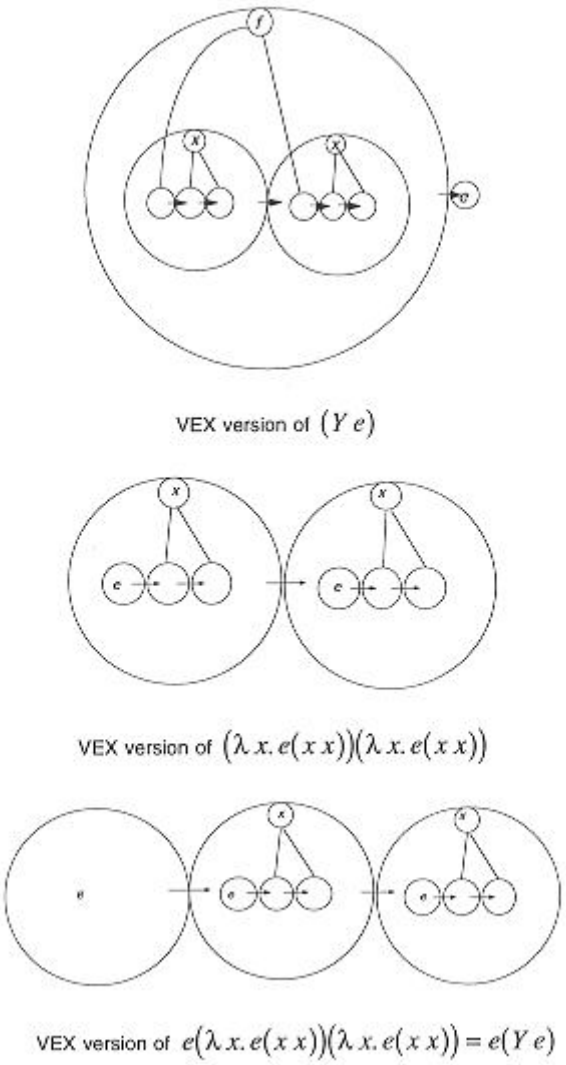


Figure 10: Y combinator (($Y\,e$) = (_x.e(xx))(_x.e(xx)) = e((_x.e(xx))(_x.e(xx))) = e($Y\,e$)) expressed
textually in VEX

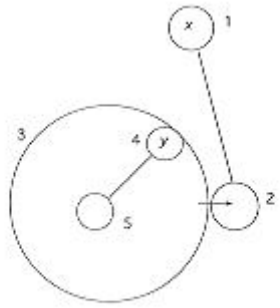


Figure 11: VEX syntax

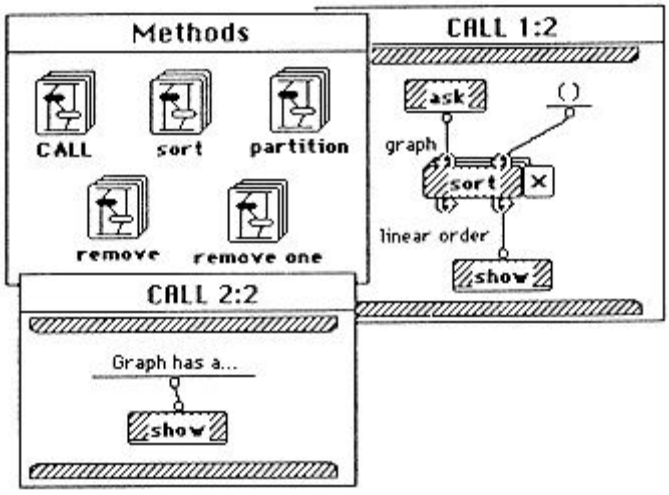


Figure 12: Prograph example – A topological sort algorithm

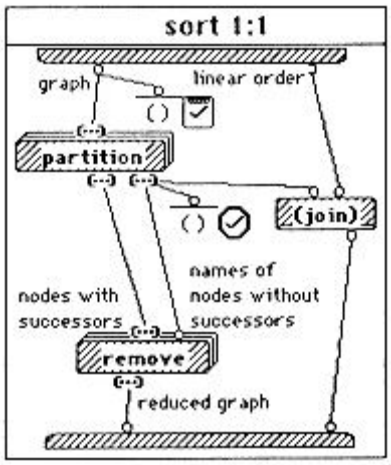


Figure 13: Prograph example – Method “sort” for topological sort algorithm

Method “sort” (Figure 13) contains two controls that terminate the execution of the method. The first such control (comparing “sort” input to “()”), called terminate-on-success, indicates to the callee of “sort” that multiplex is to be terminated with “success” status. The second control (comparing output of partition operation to “()”), called terminate-and-fail-on-success, indicates to the callee that the multiplex is to be terminated with “fail” status. As discussed above, this has the effect of failing the execution of the callee and passing control to its next case (case 2 of “CALL” in our example).

Of particular interest is a different kind of multiplex contained in the method “sort”, called a *parallel multiplex*. These include calls to “partition” and “remove”. This multiplex means that an operation should be applied to each element of an input list (similar to “map” in Lisp) *in parallel*.

Execution of any operation, including method call, can result in any of the following outcomes: execution succeeds, execution fails, execution results in an error. The

differentiation of “failures” from “errors” distinguishes Prograph from other case-based languages (such as Prolog) by providing a finer control mechanism.
In the case of an error, execution of the program is halted. In other cases, execution continues, according to a control primitive attached to the operation in the body of the callee. This control primitive, marked either by a check or by a cross to indicate whether it is executed on success or on failure respectively, always halts execution of current case and diverts thread of control in one of the following ways:

- Start execution of the next case.
- Indicate failure of the method within which it is contained.
- End execution of calling multiplex.
- Indicate failure in the execution of calling multiplex.

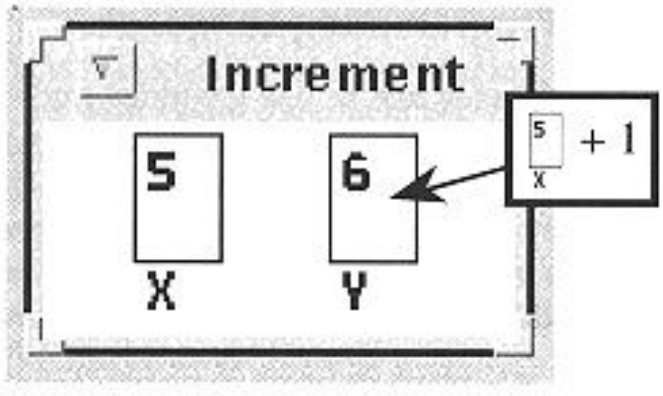


Figure 14: Forms/3: Sample Form

3.4 Forms/3

Forms/3 - Burnett (1994) is another general-purpose object-oriented visual programming language whose features emphasize data abstraction. However, unlike Prograph and VIPR), no inheritance or explicit message-passing is supported.

Forms/3 borrows the spreadsheet metaphor of cells and formulas to represent data and computation respectively. A particular feature of Forms/3 is that cells may be organized into a group called *form*, a basic data abstraction mechanism. A form may be given pictorial representation (an icon) and it may be instantiated into an object. In a sense, a form corresponds to a prototype object in prototype-based object oriented languages.

In Forms/3 data (values) and computation (formulas) are tightly coupled. Every object resides in a cell and is defined declaratively using a formula. Objects can only be created via formulas and each formula produces an object as a result of its evaluation. Formulas provide a facility to request results from other objects and create new objects: there’s no explicit message passing.

The programmer creates a new Forms/3 program by creating a new form, adding cells to it, and specifying the formulas. A sample form is depicted in Figure 14. The formulas for this

form were specified by first selecting cell “X”, typing “5”, selecting cell “Y”, clicking on “X” and typing “+ 1”. The programmer could have also referred to cell “X” by typing its name, rather than selecting it on the screen.

Forms/3 implements a declarative approach to flow control combined with the time dimension in an approach that the authors call “vectors in time”. With this approach, each vector defines a sequence of objects that represent the value of that cell at different points in time. Returning to the sample form in Figure 14, if *X* defines a time vector of numeric objects such as <1 2 3 4 5>, then *Y* defines a time vector <2 3 4 5>. Forms/3 provides the programmer with explicit access to the time dimension, and so iteration can be implemented very elegantly even with this declarative approach. Consider an example of Figure 15, a form designed to compute *nth* Fibonacci number. Here, “earlier” is one of the time-based operations in Forms/3.

Just as in other general-purpose visual programming languages, Forms/3 allows the programmer to work on both low and high levels. Low-level programming in Forms/3 is performed via formulas, while higher-level abstraction is realized by collecting cells into forms.

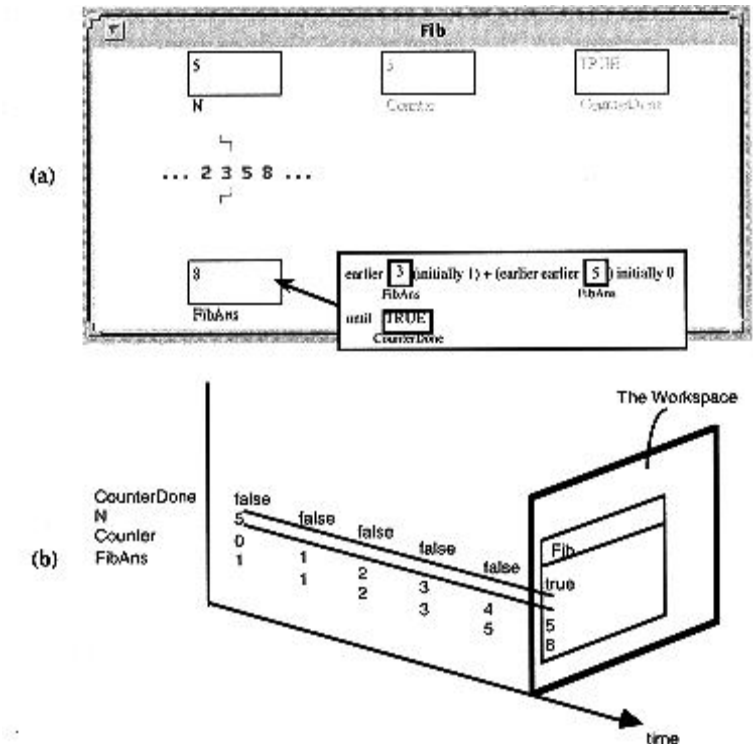


Figure 15: Forms/3 – computing Fibonacci numbers. (a) A form and a formula for one of the cells, and (b) a conceptual sketch of workspace in time

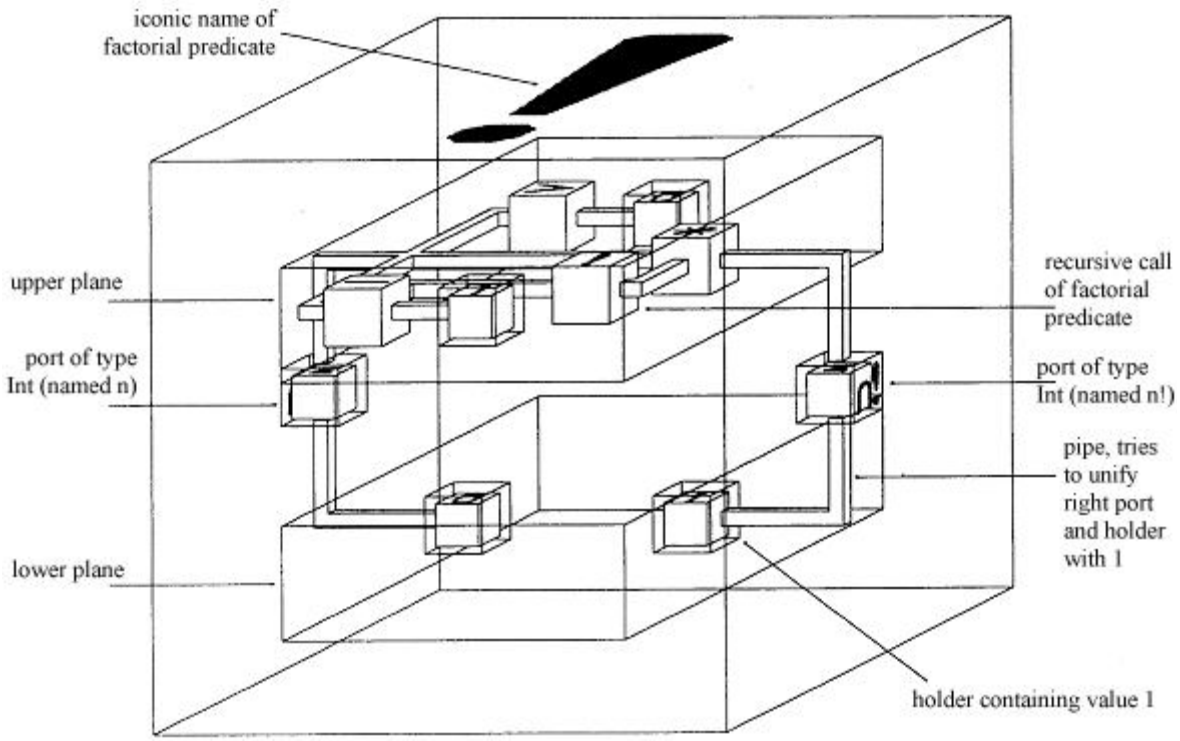


Figure 16: Function to compute the factorial of a number in Cube

In conclusion of our discussion of Forms/3 it is worth mentioning that unlike some visual programming language, Forms/3 does not aim to eliminate text completely: the presence of text in formulas is a feature of the language. The objective of the language is to use visual techniques such as direct manipulation and continuous visual feedback to enhance the process of programming

3.5 Cube

Cube, by M. Najork, represents an important advance in the design of visual programming languages in that it is the first three dimensional VPL. Since Cube programs are translated into simpler internal representations for type checking and interpreting, the language would fall into the category of a hybrid according to our taxonomy. However, the user is never exposed to any textual representation, so the argument could be made that Cube comes very close to being a completely visual language. The language uses a dataflow metaphor for program construction. Working in 3D provides a number of important benefits over more traditional 2D VPLs. For example, working in three dimensions allows the system to display more information in an environment which is easier to interact with than a 2D representation which uses the same screen size - Najork & Kaplan (1991). In the 3D display, the programmer is free to move his or her viewpoint anywhere inside the virtual world in order to look at any particular section of a program from any viewpoint. This sort of flexibility is not available in most 2D VPLs.
Figure 16 shows the main components of a Cube program as they appear in a recursive function to compute the factorial of a given number - Najork (1995). Cube programs are composed primarily of *holder cubes*, *predicate cubes*, *definition cubes*, *ports*, *pipes*, and *planes*. The entire structure in Figure 16 is surrounded by a definition cube which associates

the icon "!" with the function defined within the cube. The definition cube has two ports connected to it, one on the left and on the right. The left-hand port serves as the input while the right-hand port provides output, although ports in Cube are bi-directional, so technically either port can serve either function. Both ports are connected through pipes to holder cubes in the bottom plane of the diagram which represents the base case of the recursion. Note that each plane represents a dataflow diagram. In the case of the bottom plane, the diagram simply supplies default values for the ports and indicates what type of values each port can accept or produce. If the value at the input port is 0, then the bottom plane is active and the value from the right-hand holder cube, i.e. one, flows to the output port. If the input is greater than zero, the greater than predicate in the top plane is satisfied, and one is subtracted from the input by the bottom branch of the upper dataflow diagram. This difference is fed into the recursive call to the factorial function, and the result is multiplied by the original input. The product then flows to the output port. After defining the factorial function within a program, the programmer can then call it by simply connecting a predicate cube labeled by the "!" icon to holder cubes at the two ports.

4.0 CONCLUSION

Despite the move toward graphical displays and interactions embodied by VPLs, a survey of the field quickly shows that it is not worthwhile to eschew text entirely. While many VPLs could represent all aspects of a program visually, such programs are generally harder to read and work with than those that use text for labels and some atomic operations. For example, although an operation like addition can be represented graphically in VIPR, doing so results in a rather dense, cluttered display. On the other hand, using text to represent such an atomic operation produces a less complicated display without losing the overall visual metaphor.

As computer graphics hardware and processors continue to improve in performance and drop in price, three dimensional VPLs like Cube should begin to garner more attention from the research community

5.0 SUMMARY

Notable among what you learnt in this unit include:

- ARK programming Language
- VIPR programming Language,
- Prograph programming Language
- Forms/3 programming Language
- Cube programming Language

6.0 TUTOR-MARKED ASSIGNMENT

- Describe Form/3 programming Language
- Compare and Contrast VIPR and C++
- Enumerate the benefits of Cube over non visual programming Languages
- Compare and Contrast ARK and Prograph

7.0 REFERENCES/FURTHER READINGS

- Citrin, W., Doherty, M., and Zorn, B. 1994. Design of a completely visual object-oriented programming language. In Burnett, M., Goldberg, A., and Lewis, T., editors, *Visual Object- Oriented Programming*. Prentice-Hall, New York..
- Citrin, W., Hall, R., and Zorn, B. 1995 Programming with visual expressions. In *Proc. 1995 IEEE Symposium Visual Languages*, pp. 294–301.
- Citrin, W., Hall, R., and Zorn, B. 1996 Addressing the scalability problem in visual programming. In *Proc. of CHI '96.*
- Cox P.T and Pietrzykowski T. 1988. "Using a Pictorial Representation to combine DataFlow and Object-orientation in a language independent programming mechanism", Proceedings of the International Computer Science Conference, pp. 695-704
- Cox, P. T. and Pietryzkowsky, T. 1990. Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism. In Glinert, E. P., editor, *Visual Programming Environments: Paradigms and Systems.* IEEE Computer Society Press, Los Alamitos, CA.
- Najork, M. 1995. Visual programming in 3-d. *Dr. Dobb's Journal*, 20(12):18–31, December 1995.
- Najork, M. and Kaplan, S.1991. The cube language. In *Proc. 1991 IEEE Workshop Visual Languages*, pp. 218–224, Kobe, Japan, 1991
- Parker, Richard O. 1993. Easy Object Programming for the Macintosh using AppMaker and THINK Pascal, Prentice-Hall, 1993
- Schmucker, 1994 "DemoDialogs in Prograph CPX", FrameWorks, Volume 8, Number 2, (March/April 1994), pp. 8-13.
- Schmucker, Kurt, 1988. Object-Oriented Programming for the Macintosh, Hayden,
- Smith, R. 1996. The alternate reality kit : An animated environment for creating interactive simulations.In *Proc. 1986 IEEE Workshop Visual Languages*, pp. 99–106, 1986.
- Smith, R. B. 1987. Experiences with the alternate reality kit: An example of the tension between literalism and magic. *IEEE CG & A*, 7(9):42–50, September 1987.
- TGS, 1989 - TGS Systems, "Prograph Syntax and Semantics", Prograph 2.5 manuals, Appendix IV, Sept. 1989 (first printing), July 1990 (second printing)

MODULE 2 – WEBSITE DESIGN

U NIT 1 Understanding the WWW

UNIT 2 HTML and the Web

UNIT 2 Getting Started with HTML

UNIT 3 Understanding the Basics of HTML

U NIT 1 – Understanding the WWW

1.0 Introduction

2.0 Objectives

3.0 Main Content

- 3.1 How the WWW Works
- 3.2 How do URLs Work
- 3.3 How to Use a Web Browser
- 3.4 How to Use a Hypertext Link

4.0 Conclusion

5.0 Summary

8.0 Tutor Marked Assignment

7.0 Further Reading and Other Resources

1.0 INTRODUCTION

One of the best things about the World Wide Web is that it is just as easy to create Web pages as it is to browse them. The key to publishing on the Web is having a firm understanding of Hypertext Markup Language (HTML). Despite the intimidating name, HTML is extremely simple to learn and use. By the time you finish this course material, you will be well on your way to becoming an HTML wizard.

Before diving head-first into the language of HTML itself, it will help you to understand a little bit about how the World Wide Web works. After all, HTML is designed to guide users through the vast and tangled resources of the Web. As a student of this course , you will need to understand some of the basics behind the architecture of the World Wide Web. Knowing how the Web works, as well as when it doesn't and why, can help you make important decisions about how to construct your own Web pages.

It would be impossible to describe in detail the inner workings of the Web in a single Unit. With that in mind, this Unit provides you with a "refresher course" on the basics. Armed with this basic knowledge, you'll be able to move on to writing your own Web pages in a very short time.

2.0 OBJECTIVES

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

- Explain the meaning of WWW
- Describe how WWW Works
- Explain parts of URLs

3.0 MAIN CONTENT

3.1 How the WWW Works

The World Wide Web is a system of hypertext documents that are linked to each other. Internet is the means to access this set of interlinked documents. These hypertext documents can contain text, images or even audio and video data. The World Wide Web, serving as an enormous information base, has also facilitated the spread of this information across the globe. It has led to the emergence of the Internet age. It will not be an exaggeration to say that the Internet owes its popularity to the World Wide Web.

Before understanding how World Wide Web works, let us delve into the history behind the creation of this smart information base, popularly known as 'www'. It was the genius of Sir Tim Berners-Lee, an English computer scientist and MIT professor, who created the World Wide Web. While he was working at CERN in Switzerland, he built ENQUIRE, a closed database of information containing bidirectional links that could be edited from the server. ENQUIRE was, in many ways, similar to the modern-day World Wide Web. In 1989, Berners-Lee wrote a proposal describing an information management system. True, the concept of hypertext originated from projects such as the Hypertext Editing System at Brown University and similar projects by Ted Nelson and Andries van Dam, both working in the field of computers and Internet technology. But Berners-Lee accomplished the feat of combining the concepts of hypertext and Internet. He also developed a system of globally unique identifiers for web resources, which later came to be known as Uniform Resource Identifiers. On April 30, 1993, it was decided the the World Wide Web would be free to everyone. After leaving CERN, Tim Berners-Lee founded the World Wide Web Consortium at the MIT Laboratory for Computer Science.

Asking how the Internet works is not the same as asking how the world wide web works. Well, Internet and the World Wide Web are not one and the same, although they are often used as synonyms. While the Internet is an infrastructure providing interconnectivity between network computers, the web is one of the services of the Internet. It is a

collection of documents that can be shared across Internet-enabled computers.

The network of web servers serves as the backbone of the World Wide Web. The Hypertext Transfer Protocol (HTTP) is used to gain access to the web. A web browser makes a request for a particular web page to the web server, which in turn responds with the requested web page and its contents. It then displays the web page as rendered by HTML or other web languages used by the page. Each resource on the web is identified by a globally unique identifier (URI). Each web page has a unique address, with the help of which a browser accesses it. With the help of the domain name system, a hierarchical naming system for computers and resources participating in the Internet, the URL is resolved into an IP address.

Presence of hyperlinks, the worldwide availability of content and universal readership is some of the striking features of the World Wide Web. The interlinked hypertext documents form a web of information. Hyperlinks present on web pages allow the web users to choose their paths of traversal across information on the web. They provide an efficient cross-referencing system and create a non-linear form of text. Moreover, they create a different reading experience. The information on the web is available 24/7 across the globe. It is updated in real time and made accessible to web users around the world. Except for certain websites requiring user login, all the other websites are open to everyone. This all-time availability of information has made the Internet, a platform for knowledge-sharing. Thanks to the use of a common HTML format for rendering web content and a common access method using the HTTP protocol, the web has achieved universal readership.

The World Wide Web, a compilation of millions of hypertext documents, has brought together information from all over the world, 'just a click away'.

3.2 – How do URLs work

Almost every item of information on the WWW can be accessed directly. That is because every document, file and image has a specific address. These addresses are called *Uniform Resource Locators* (URLs). A URL is a formatted text string used by Web browsers, email clients and other software to identify a *network resource* on the Internet. Network resources are files that can be plain Web pages, other text documents, graphics, or programs.

URLs are used by Web browser to locate and access information on the WWW. A URL is also known as a Web address. Think of URLs as a postal addresses for the Internet.

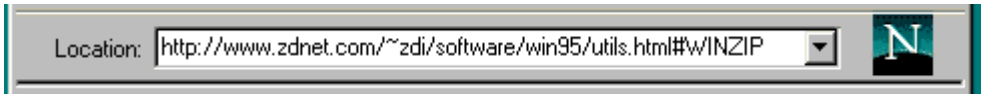


Figure 1: *Showing parts of Uniform Resource Locator.*

The first part of the URL parts is known as the *protocol*. This is almost always *http://*, which is short for Hypertext Transfer Protocol. Some URLs start with a different protocol, such as *ftp://* or *news://*. If you're accessing a document on your local machine instead of on the Web, the URL will begin with *file://*.

The second part of the URL is known as the *domain name*. If you've used e-mail on the Internet, you're probably already familiar with domains. The domain represents the name of the server that you're connecting to. A domain name, to put it simple, is your address on the World Wide Web. This is where you put up your website and it is what internet users will type in their address bar in order to locate your site while online. Your domain name should be short, simple, and easy to remember. But, one must keep in mind that domain names are only available for one individual or business. This is to maintain uniqueness and to avoid confusion among the millions of websites and internet users.

A most common example of a domain name is www yahoo.com. The first part, the www identifies the server name of the domain. Yahoo, the second element, is the name of the company, individual or organization; and the suffix .com is the domain name extension, which identifies the purpose of the website.

The most important part of the entire domain name is the second element, which states the unique name of an individual, an organization, or a company. This is what sets it apart from all the other addresses present on the web, as some people would try to change a part of the domain name in order to direct traffic to their site instead.

Another example of a domain name is www nasa.gov. This is the NASA website, and since it is a government office, it uses the extension dot gov. Users need to bear in mind that the domain name extensions are there for a purpose. It indicates the purpose why the website exists.

The third part of the URL is called the *directory path*. This is the specific area on the server where the item resides. Directory paths on Web servers work a lot like they do on your desktop computer. To locate a particular file on a server, you need to indicate its directory path first.

The fourth part of the URL is called the *document file name*. This indicates the specific file being accessed. This is usually an HTML file, but it can also be an image, sound, or another file.

Sometimes the URL contains a fifth part, known as the *anchor name*. This is a pointer to a specific part of an HTML document. It's always preceded by the pound sign (#). Anchors are especially useful for large documents.

Absolute vs. Relative URLs

Full URLs featuring all substrings are called *absolute* URLs. In some cases such as within Web pages, URLs can contain only the one location element. These are called *relative* URLs. Relative URLs are used for efficiency by Web servers and a few other programs when they already know the correct URL protocol and host.

3.3- **How to Use a Web Browser**

Many people can successfully navigate the World Wide Web without any problem at all and may even consider themselves experts of the Web. On the other hand, there are thousands of other people who do not even know the first thing about operating a web browser. If you would like to know how to operate a web browser (the tool you use to navigate the internet), such asNetscape, Microsoft's Internet Explorer or Mozilla's Firefox without taking an expensive computer-learning class then here are the basics of using a web browser of your choice.

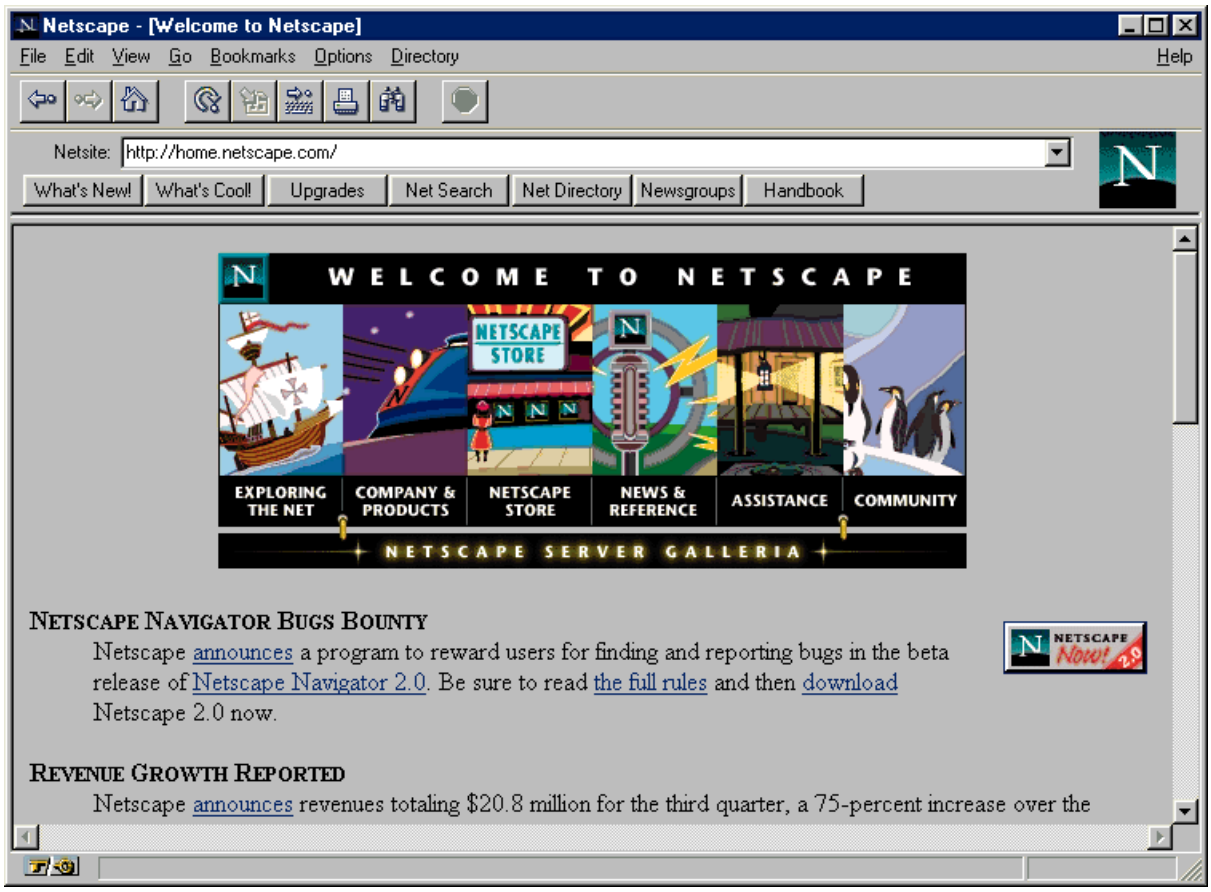


Figure 2: A Web Browser.

Instructions

1. Get acquainted with the web browser you are using. All web browsers are relatively similar, but the two most popular web browsers are Microsoft Internet Explorer and Mozilla Firefox. Open the web browser by double-

clicking on the icon on your desktop or right-clicking the icon and choosing "Open."

2. Once the browser is open, notice the Address Bar, which is the URL (website address) where you are located. To easily identify the Address Bar, it will be located at the top of the browser itself and the URL in the Address Bar will usually begin with "http://www."

3. Next, notice the tools that are surrounding the Address Bar. For example, there will usually be a "Page Back" and "Page Forward" button, usually indicated by a forward and backward arrow. There may even be a picture or icon that will stand for "Home Page," which is the first web page that is viewed when the web browser is opened.

4. Above the web browser's icons that you've just looked at, you'll notice a series of help menus, such as "File," "Edit," "View," "History," "Bookmarks" or "Favorites,", as well as "Tools," and "Help." All of these menus are placed there because they may be able to help you at one time or the other. For example, clicking once on "Bookmarks" or "Favorites" will show you a list of all the websites that you have placed in your "favorite" list.

5. Two important things to know how to do are to get to a specific website destination of your choice and make that website one of your "favorites" or "bookmarks."We will start with navigating to a specific website of your choice. If you have a specific URL of a website in mind then feel free to use that. For this example, we'll use the Google Search Engine home page. In the Address Bar which is located at the top of the web browser, type in the full URL of the website you would like to go to. In this example, we'll type in "http://www.google.com." After you've typed in the full URL, either press "Enter" on the keyboard or click on the arrow icon or "go" button located at the very right end of the Address Bar. After you have done that the website should appear on your web browser's screen.

6. We will now put this website, the website of your choice or in this example we're using www.google.com, in your "Favorites" or "Bookmarks" list. Click on the "Bookmark" or "Favorites" menu at the top of the web browser and choose the option that says either, "Bookmark this page," or "Add to Favorites." Your website is now in the "Favorites" or "Bookmarks" list and all you will need to do to get back to that website is to click on the "Bookmarks" or "Favorites" menu again and click on the website that you have favorited or bookmarked.

7. You have just learned how to do two very important tasks in two of the most popular web browsers that are used for navigating the internet. For any other help that is needed, the web browser's own "Help" section can be read by clicking on the "Help" menu at the top of the web browser and choosing the browser help section. In Mozilla Firefox, this section is called "Help

Contents," while in Microsoft's Internet Explorer it is called "Contents and Index."

Experiment with your Web browser to get an understanding of how navigation works on the World Wide Web. It is a good idea to use a few different browsers and note the differences. Knowing how users browse the Web is an important part of understanding how you should construct your own HTML pages.

3.4 How to Use a Hypertext link

Using a hypertext link to move from one place to another is one of the most common activities on the World Wide Web. In fact, hypertext links are the very essence of the Web.

The following steps explain how to use links and describe a little of what happens behind the scenes.

1. find a link on the page, look for text that's displayed in a different color. By default, hypertext links you haven't used are blue and underlined. Links you've already visited are purple. These colors can be changed, however.
2. Using your mouse, place the pointer over the hypertext link and click. There will be a brief delay after you press on the hypertext link.
3. During this delay, your browser client is contacting the Web server referenced in the hypertext link's URL. It is attempting to retrieve the referenced document.

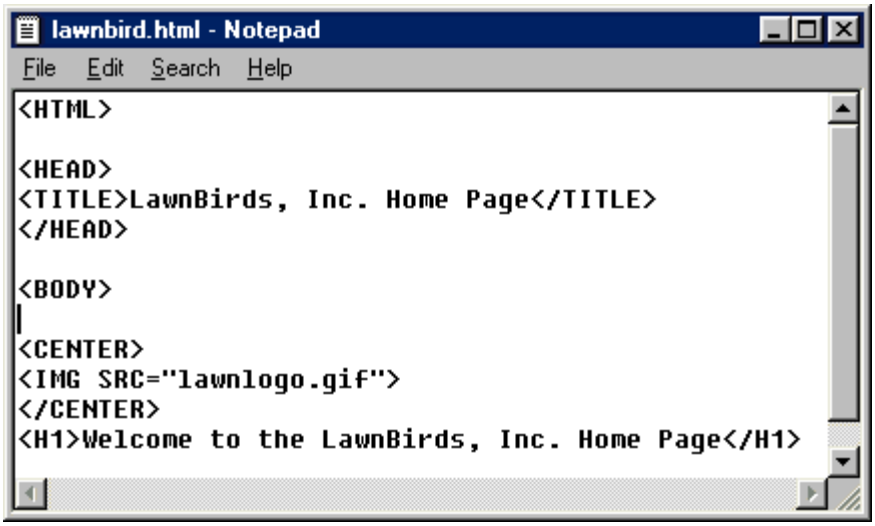


Figure 3: **An example of HTML document**.

4. Once the contact has been established, your browser begins displaying the new document.
5. Not all links appear as text. Many links appear in images, such as buttons or icons. Sometimes a colored border will appear around the image, or it will be designed to

look like a button. In many browsers, the cursor will change to a hand when it passes over a hypertext link. These visual clues help the reader understand that it is a link. However, sometimes there are no visual clues. Understanding the need to provide visual clues is an important part of being an HTML author.

4.0 CONCLUSION:

Hypertext is one the most common activities on the WWW. It is the very essence of the Web.

5.0 SUMMARY:

In this unit, you have learnt:

- The working method of WWW
- The work of the URLs in the system
- How hypertext links are used.

6.0 Tutor Marked Assignment

- What is the difference between Absolute and Relative URLs?
- Explain the term WWW
- What is the work of Hypertext Transfer Protocol (HTTP)?
- Give three examples of domain names?

7.0 Further Reading and Other Resources

- Abbate, Janet. *Inventing the Internet*. Cambridge: MIT Press, 1999.
- Bemer, Bob, "A History of Source Concepts for the Internet/Web"
- Campbell-Kelly, Martin; Aspray, William. *Computer: A History of the Information Machine*. New York: BasicBooks, 1996.
- Clark, David D., "The Design Philosophy of the DARPA Internet Protocols", Computer Communications Review 18:4, August 1988, pp. 106–114
- Graham, Ian S. *The HTML Sourcebook: The Complete Guide to HTML*. New York: John Wiley and Sons, 1995.
- Krol, Ed. *Hitchhiker's Guide to the Internet*, 1987.
- Krol, Ed. *Whole Internet User's Guide and Catalog*. O'Reilly & Associates, 1992.
- *Scientific American Special Issue on Communications, Computers, and Networks*, September, 1991

UNIT 2 –HTML

1.0 INTRODUCTION

Web design is the process of planning and creating a website. Text, images, digital media and interactive elements are shaped by the web designer to produce the page seen on the web browser. It is the creation of digital environments that facilitate and encourage human activity; reflect or adapt to individual voices and content; and change gracefully over time while always retaining their identity.

Web designers utilize markup language, most notably HTML for structure and CSS for presentation to develop pages that can be read by web browsers. This module will deal on only HTML structures. HTML is not the only way to present information on the Web, but it is the glue that holds everything together. In addition to begin a markup language for displaying text, images, and multimedia, HTML provides instructions to Web browsers in order to control how documents are viewed and how they relate to each other. For all its simplicity, HTML is a very powerful language.

In this unit, we will take a look at how HTML interacts with the Web and students are expected to explore some of the ways that it is begin used today on popular web sites.

2.0 OBJECTIVES

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

- Define HTML
- Describe the History of HTML
- Explain how HTML works

3.0 MAIN CONTENT

3.1. What is HTML?

HTML is a computer language devised to allow website creation. These websites can then be viewed by anyone else connected to the Internet. It is relatively easy to learn, with the basics being accessible to most people in one sitting; and quite powerful in what it allows you to create. It is constantly undergoing revision and evolution to meet the demands and requirements of the growing Internet audience under the direction of the World Wide Web Consortium, W3C, the organisation charged with designing and maintaining the language.

The full meaning of HTML is **HyperText Markup Language**.

- *HyperText* is the method by which you move around on the web — by clicking on special text called **hyperlinks** which bring you to the next page. The fact that it is *hyper* just means it is not linear — i.e. you can go to any place on the Internet whenever you want by clicking on links — there is no set order to do things in.
- *Markup* is what **HTML tags** do to the text inside them. They mark it as a certain type of text (*italicised* text, for example).
- HTML is a *Language*, as it has code-words and syntax like any other language.

3.1.1 How does HTML works?

HTML consists of a series of short **codes** typed into a text-file by the site author — these are the tags. The text is then **saved as a html file**, and **viewed through a** browser, like *Internet Explorer* or firefox. This browser reads the file and translates the text into a visible form, rendering the page as the author had intended. Writing your own HTML entails using tags correctly to create your vision. You can use anything from a rudimentary text-editor to a powerful graphical editor to create HTML pages.

3.1.2 What are the tags up to?

The tags are what separate normal text from HTML code. You might know them as the words between the <angle-brackets>. They allow all the cool stuff like images and tables and stuff, just by telling your browser what to render on the page. Different tags will perform different functions. The tags themselves don't appear when you view your page through a browser, but their effects do. The simplest tags do nothing more than apply formatting to some text, like this:

```
<b>These words will be bold</b>, and these will not.
```

In the example above, the <b> tags were wrapped around some text, and their effect will be that the contained text will be bolded when viewed through an ordinary web browser.

3.1.3 **Is there anything HTML cannot do?**

Of course, but since making websites became more popular and needs increased many other supporting languages have been created to allow new stuff to happen, plus HTML is modified every few years to make way for improvements.

Cascading Stylesheets are used to control how your pages are presented, and make pages more accessible. Basic special effects and interaction is provided by JavaScript, which adds a lot of power to basic HTML. Most of this advanced stuff is for later down the road, but when using all of these technologies together, you have a lot of power at your disposal.

3.2. History of HTML

A markup language combines text as well as coded instructions on how to format that text and the term "markup" originates from the traditional practice of 'marking up' the margins of a paper manuscript with printer's instructions. Nowadays, however, if you mention the term 'markup' to any knowledgeable web author, the first thing they are likely to think of is 'HTML'.

In the Beginning

HTML —which is short for **H**yper**T**ext **M**arkup **L**anguage— is the official language of the World Wide Web and was first conceived in 1990. HTML is a product of SGML (Standard

Generalized Markup Language) which is a complex, technical specification describing markup languages, especially those used in electronic document exchange, document management, and document publishing. HTML was originally created to allow those who were not specialized in SGML to publish and exchange scientific and other technical documents. HTML especially facilitated this exchange by incorporating the ability to link documents electronically using *hyper*links. Thus the name *Hyper*text Markup Language.

However, it was quickly realized by those outside of the discipline of scientific documentation that HTML was relatively easy to learn, was self contained and lent itself to a number of other applications. With the evolution of the World Wide Web, HTML began to proliferate and quickly spilled over into the mainstream.

Browser Wars

Soon, companies began creating browsers —the software required to view an HTML document, i.e., a web page— and as they gained popularity it gave rise to competition and other web browsers. It may surprise some that back in late 1995, Netscape —which now plays a distant second to the King Kong of browsers, Internet Explorer— was the dominant browser on the market. In fact, Netscape was the first browser to support Javascript, animated gifts and HTML frames.

Thus began the so-called 'browser wars' and, along with seeing who could implement more 'bells and whistles' than the other guy, browser makers also began inventing proprietary HTML elements that only worked with their browsers. Some examples of these are the <marquee>**...**</marquee> tags (scrolling text) which originally only worked with Internet Explorer and the <blink>**...**</blink> tags (blinking text) which still only works with Gecko-based browsers such as Mozilla or Netscape.

A side effect of all this competition was that HTML became fragmented and web authors soon found that their web pages looked fine in one browser but not in another. Hence it became increasingly difficult and time consuming to create a web page that would display uniformly across a number of different browsers. (This phenomenon remains to some extent to this very day.)

Meanwhile, an organization known as the World Wide Web Consortium (W3C for short) was working steadily along in the background to standardize HTML. Several recommendations were published by the W3C during the late 1990s which represented the official versions of HTML and provided an ongoing comprehensive reference for web authors. Thus the birth of HTML 2.0 in September 1995, HTML 3.2 in January 1997 and HTML 4.01 in December 1999.

By now, Internet Explorer (IE) had eclipsed Netscape Navigator as the browser to use while surfing the net due to its superior capabilities but also largely due to the fact that the IE came

bundled with the Windows operating system. Essentially when people bought computers using the Windows OS, it had the 'internet installed on it'. This tended to suit people just fine since the typical newcomer to computers was someone who was tentatively striking forth to take on this intimidating new-fangled technology that was crammed to the rafters with indecipherable acronyms, software help files that made no sense and buggy programs. Hence, the more 'instant' solutions this new technology offered, the better it was.

3.3. **How HTML works with the Web**

HTML allows the individual elements on the Web to be brought together and presented as a collection. Text, images, multimedia, and other files can all be packaged together using HTML. This section of unit three explains the basic principles behind the interaction between HTML and the WWW.

You can always view the HTML source code for a particular page through your browser. Once you've mastered the basics of HTML, this is a great way to learn how other authors put together their HTML documents. To view the source code of the current document in Netscape, choose Document Source from the View menu.

Figure 1: **Document Source from the View menu**

1. The author of the Web page assembles all of the materials necessary, including text, charts, images, and sounds.

2. All of the material for the Web page is linked together using HTML. HTML codes control the appearance, layout, and flow of the page. The amazing thing about HTML is that it is all done with simple text codes that anyone can understand.
3. When someone connects to a Web server from his or her computer, the HTML file is transferred from server to client. Because an HTML file is simple text, this usually happens very quickly.
4. The Web browsing software (the client) interprets the layout and markup commands specified in the HTML file and then display the text exactly as the HTML author intended.
5. Any images and charts on the page are retrieved as well. The HTML file tells the Web browser what images to download and how to display them on the page.

3.3.1. **How HTML works on the Web**

a) Your Computer

The browser on your computer sends a request for an HTML document from a remote server using addresses called URLs (Uniform Resource Locators). When the data is located and returned, your browser displays the web page (text and graphics) according to the HTML tags in the document.

b) Connection to the Internet

A dial up modem in your computer or a direct high speed data transmission line connects your computer to an internet service provider.

c) Internet Service Provider

Your internet service provider is probably an internet web server and is connected to all the other computers on the web. Your web server sends your request for an HTML document and sends back the file to you.

d) Internet

The internet is a collection of web servers around the world. Each server has a URL and will forward your request on until it reaches the server you are looking for. When the data is returned to you, it may travel a totally different route over different computers.

e) Remote server

The remote web server with the URL you are looking for has all the HTML files including text, graphics, sound, and video. It may also have gateway scripts that are programs running on the server to process data.

3.4. Things you can do with HTML

There are many ways you can use HTML to publish content on the World Wide Web. This unit will teach you the techniques you need to know to create timely, informative, and compelling HTML documents.

3.4.1 Six Cool Things You Can Do with HTML

1. You can create a personal home page and leave your mark on the World Wide Web.

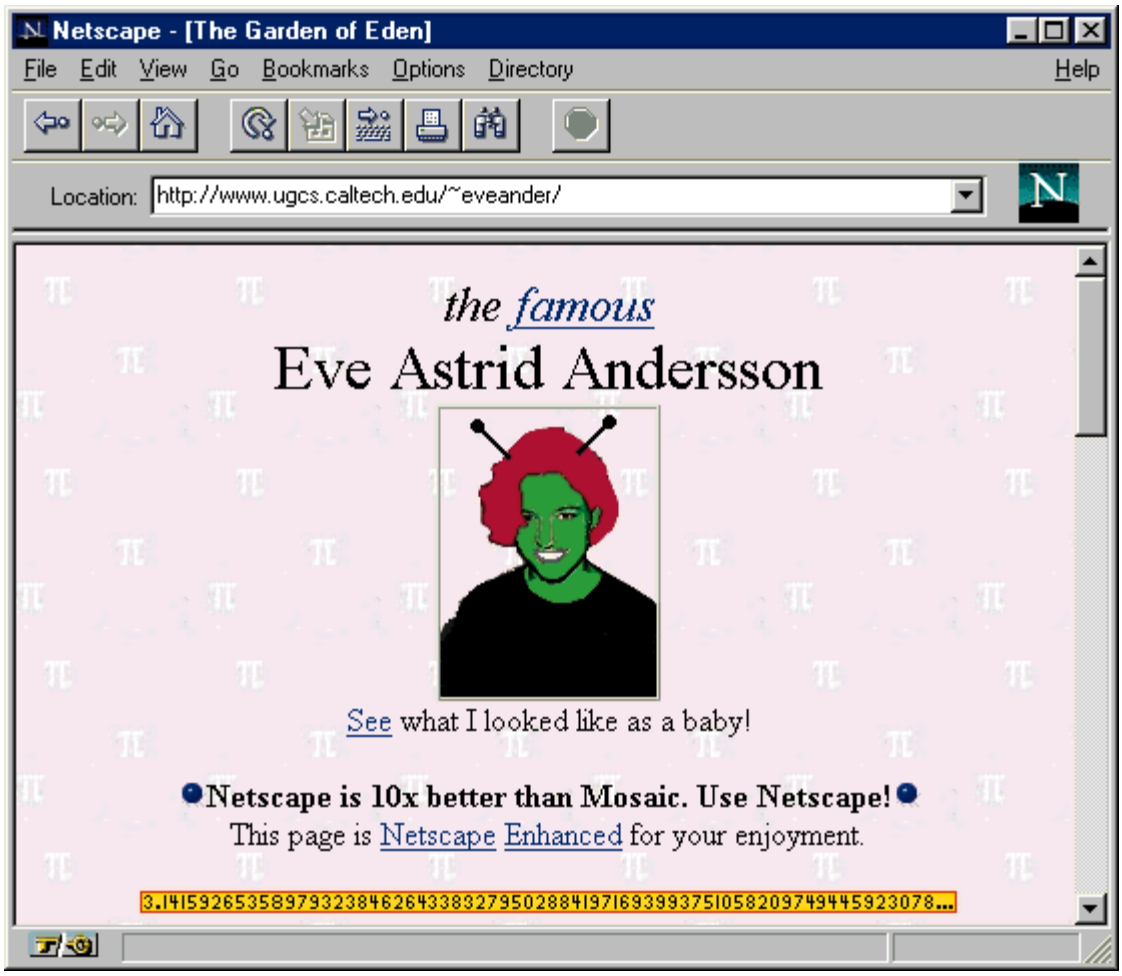


Figure 2 : An example of a personal home page

2. You can create a page for your company to advertise and promote products and services.

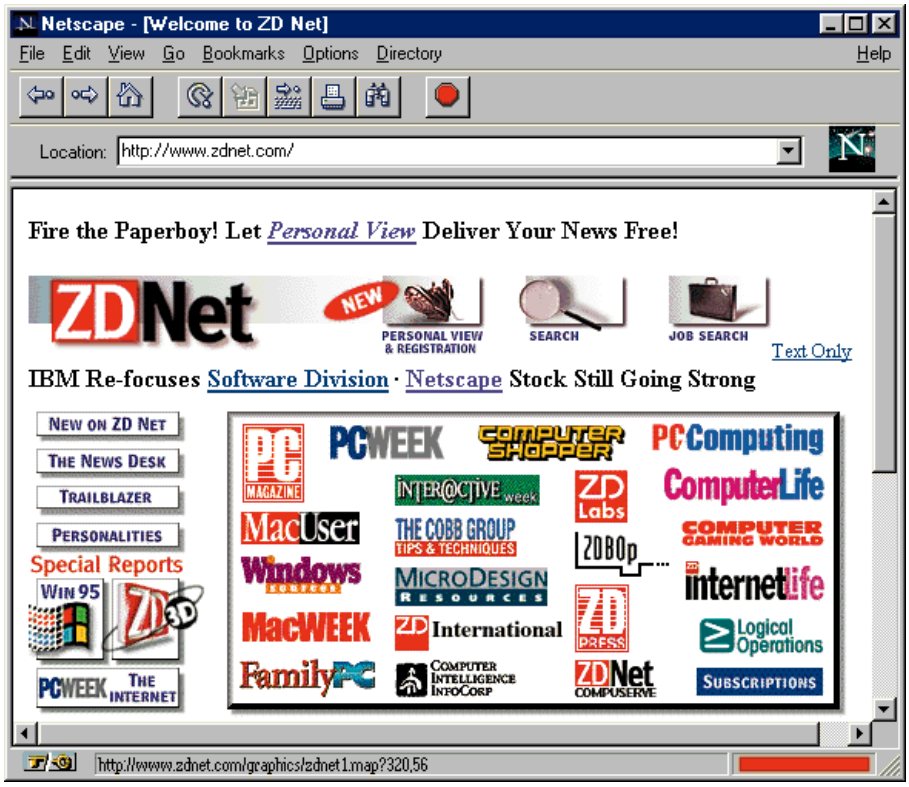


Figure 3 : An example of a home page for your company

3. You can build a catalog on the World Wide Web, complete with product descriptions and photographs. You can even incorporate fill-in order forms so that your customers can order products from you on line.

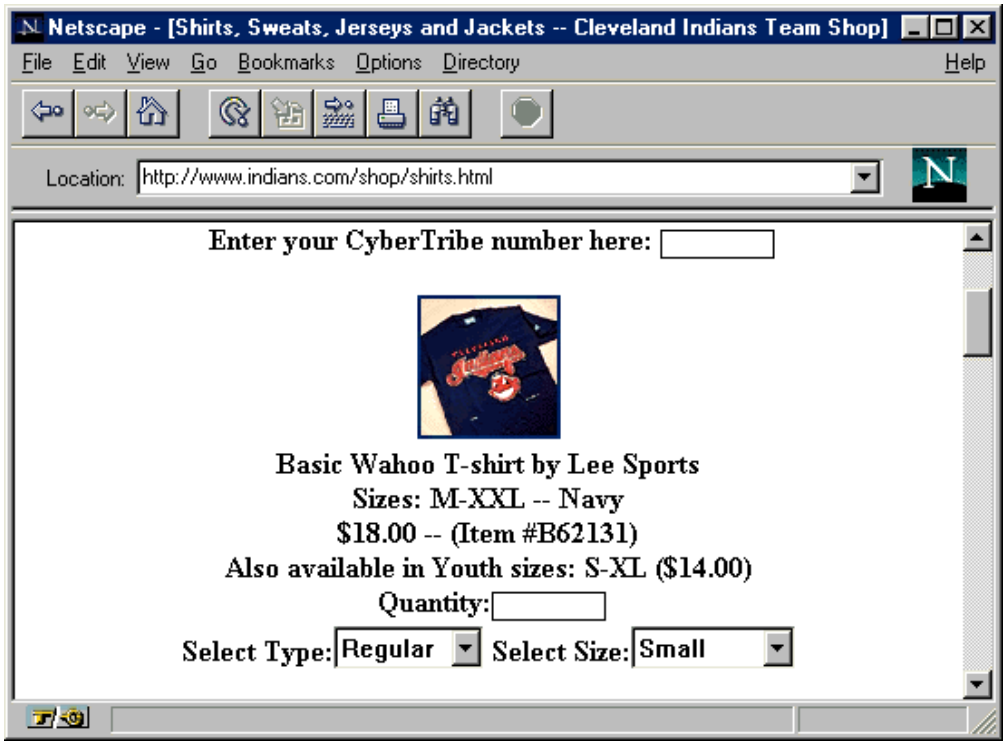


Figure 4: An example catalog on the World Wide Web

4. You can create a searchable phone directory for your company or organization.

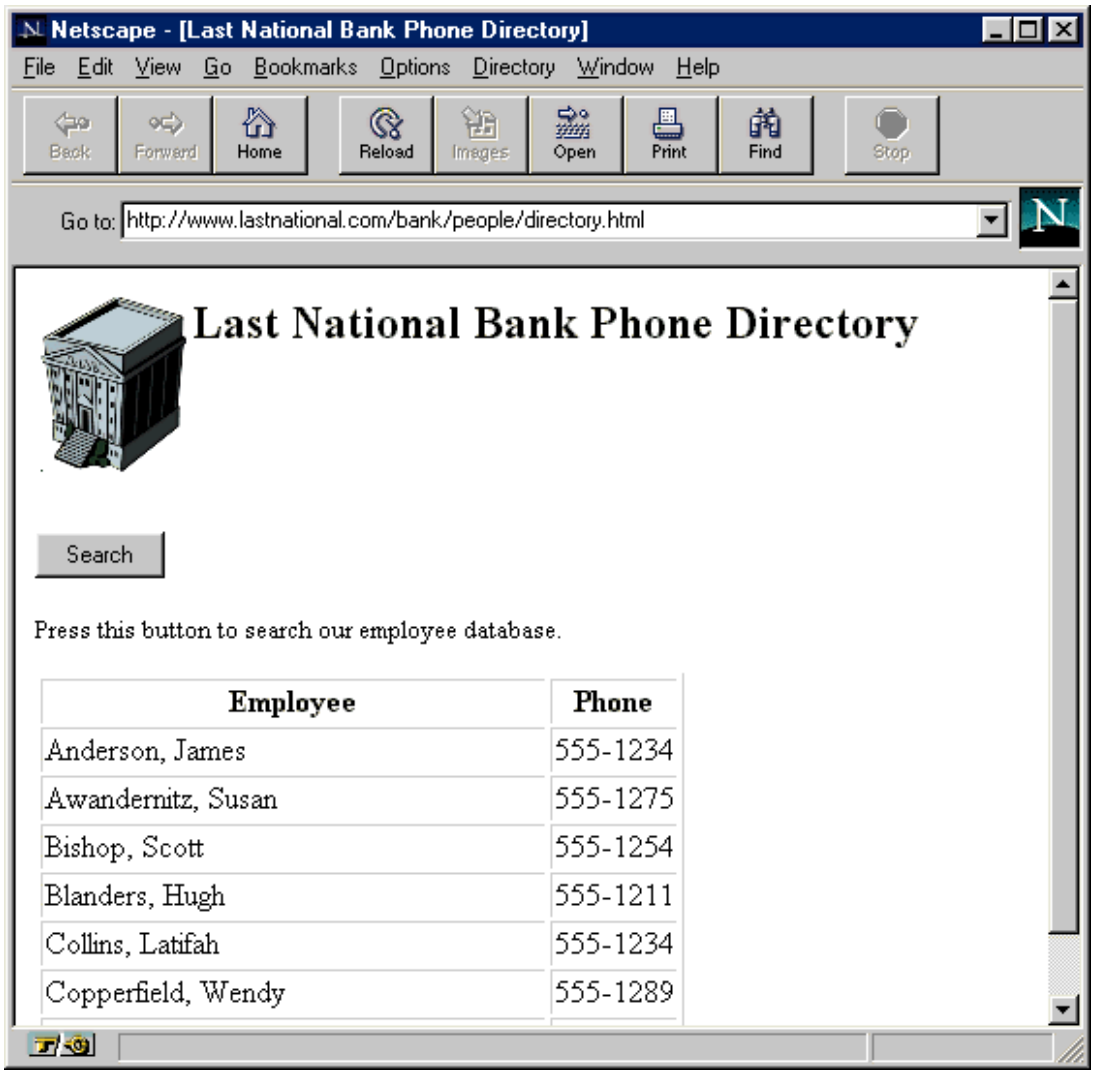


Figure 5: An example of searchable phone directory for a company

5. You can create a newsletter on the Web, with pictures and sounds. Using some of the advanced HTML tricks explained in this book, you can format the newsletter to give it a slick, professional appearance.

Figure 6: An example of how to create newsletter on the Web

4.0 CONCLUSION:

HTML is a computer language devised to allow website creation. Its tags are embedded in angle brackets. HTML is made up of different tags that perform different functions.

5.0 SUMMARY:

In this unit, you have learnt:

- The meaning of HTML
- How HTML worked with the Web
- How HTML worked on the Web
- Things that can be done with HTML

6.0 Tutor Marked Assignment

- List five Things You Can Do with HTML
- Define HTML
- Describe the History of HTML

7.0 Further Reading and Other Resources

- Abbate, Janet. *Inventing the Internet*. Cambridge: MIT Press, 1999.
- Bemer, Bob, "A History of Source Concepts for the Internet/Web"
- Campbell-Kelly, Martin; Aspray, William. *Computer: A History of the Information Machine*. New York: BasicBooks, 1996.

- Clark, David D., "The Design Philosophy of the DARPA Internet Protocols", Computer Communications Review 18:4, August 1988, pp. 106–114
- Graham, Ian S. *The HTML Sourcebook: The Complete Guide to HTML*. New York: John Wiley and Sons, 1995.
- Krol, Ed. *Hitchhiker's Guide to the Internet*, 1987.
- Krol, Ed. *Whole Internet User's Guide and Catalog*. O'Reilly & Associates, 1992.
- *Scientific American Special Issue on Communications, Computers, and Networks*, September, 1991

UNIT 3 - Getting Started with HTML

1.0 INTRODUCTION

This Unit will introduce you to the basics of HTML.We will take a quick look at Notepad, which is one of the tools you really need to write HTML documents. We will also go over the fundamentals of a basic HTML document. You will even learn how to write your first Web page!

It might be tempting to skip ahead and check out the "cool stuff" in the later Modules of this Course Material. But if you spend some time going over the basics, it will serve you well in the long run. Enough chatter. Let's get going.

2.0 OBJECTIVES

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

- Describe how to open a Notepad
- Describe markup tag in HTML
- Write a simple HTML Document using Notepad editor

3.0 MAIN CONTENT

3.1 How to Use Notepad

HTML is not really anything more than plain text. For that reason, you don't need any special editors or compilers to create HTML files. In fact, you can create all of your HTML files with the simplest of text editors. There are many specialized HTML editors and converters available, and you may decide to choose one of them based on your particular needs. But for all the HTML examples in this unit, we will use Windows Notepad to illustrate just how simple creating HTML can be.

Figure 1: A Symbol of Notepad

1. To open Notepad, click on the Start button in the lower-left corner of your screen. Then choose Programs, followed by Accessories. Click on the Notepad icon.

Figure 2: Processes of Clicking on the Notepad Icon

Notepad begins with a blank document. You can begin typing to create a new document. To open an existing text file from disk, pull down the File menu and choose Open.

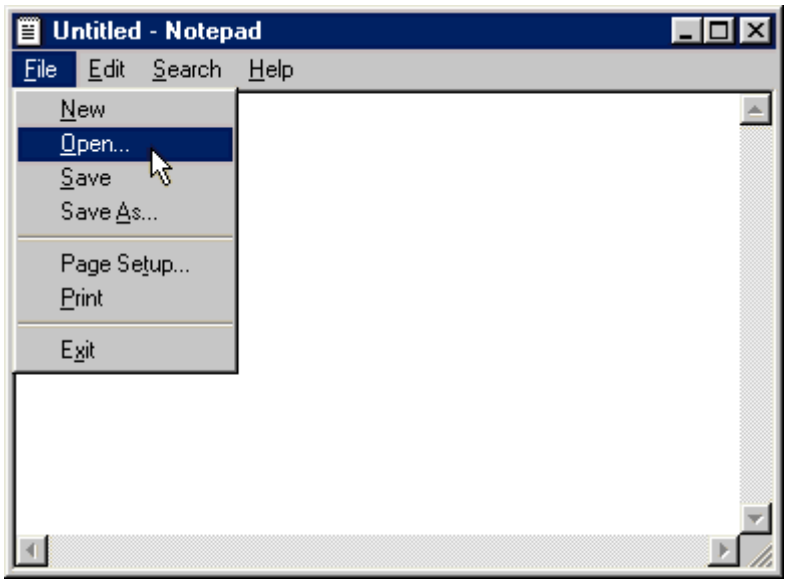


Figure 3: Notepad Environment

2. Choose the file name from the Open File dialog box. Notepad's Open File dialog window normally only shows files with the extension .txt. You'll want to change the Files of Type selection to All Files if you're opening or saving an HTML file, which uses the extension .htm or .html.
3. Once you've opened an existing file or begun typing a new one, you can easily edit your text. Notepad has all the basic editing functions of a word processor. For example, you can select blocks of text for cut and paste operations.

```
Editing text is easy using
and keyboard. Position the
then select text with your
```

4. HTML files usually contain very long lines that will run off the edge of the page. Notepad has a feature called Word Wrap that will format these lines to fit entirely within the window, making them much easier to read. To activate this feature, pull down the Edit menu and select Word Wrap.

Figure 3: Notepad Environment – Word Wrap

5. To save your HTML file, first pull down the File menu. If this is a new file that you started from scratch, choose Save As and then type a file name. Remember to use .htm or .html as the file extension. (Check with your Web server administrator to find out which extension you should use.) If this is an existing file that you opened from Notepad, you can just choose Save from the File menu.

3.2– **How to Use Mark up Tags**

The use of markup tags is what separates HTML from plain text. Markup tags are used extensively in HTML, and they provide ways to control text formatting, create links to other documents, and even incorporate images and sounds. In short, markup tags are the key to making HTML pages work. Markup tags are not case sensitive. For example, the body element tag (which you'll learn about in the next unit) can be typed as **<BODY>**, **<body>**, or even **<BoDy>**.

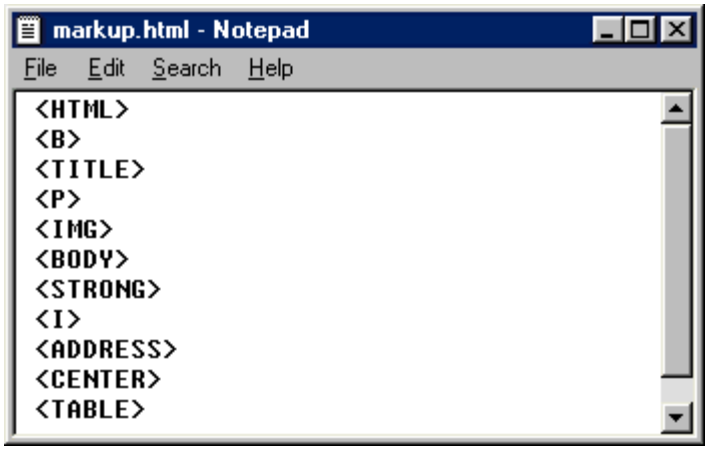


Figure 4 : **An example of body element tag**

1. Open a new file in Notepad and type in the words "**the bold new frontier"**. In this example, we'll make this text appear in boldface type from the following sentence:.

```
The pioneers ventured Westward in
hopes of blazing a trail of prosperity
in the bold new frontier.
```

2. HTML markup tags are easy to create. They consist of a left angle bracket, the name of the tag, and a right angle bracket. The left and right angle brackets are also known to some as less-than and greater-than symbols. To start a boldface markup tag, type **<B>** where you'd like the boldface type to begin.

```
<B>the bold new frontier.
```

3. Locate the place where you'd like the boldface to stop. At this point, you need to create an ending tag for the boldface type. An ending tag looks just like a starting tag, except it is preceded by a forward slash character (/). To mark the end of the boldface tag, type **</B>**.

```
new frontier.</B>
```

4. When viewed with a Web browser, the text between the **<B>** and **</B>** tags will appear in boldface.

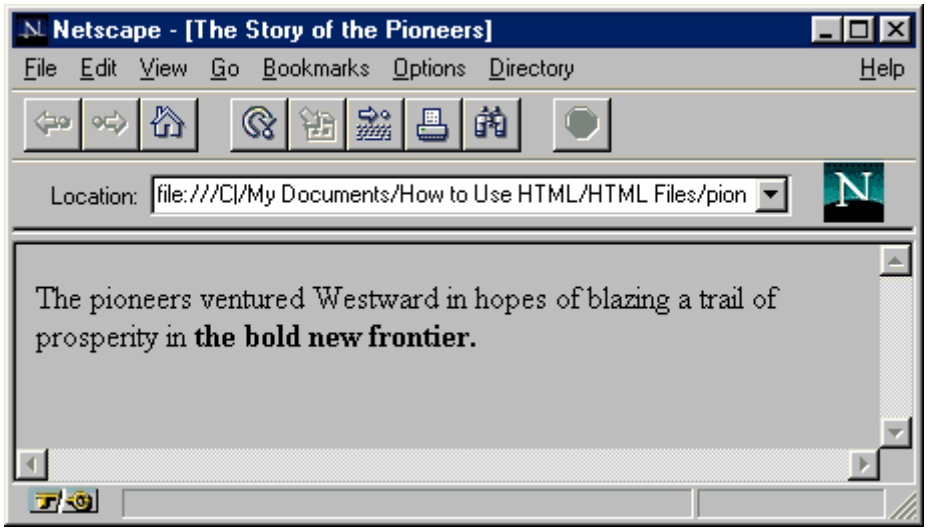


Figure 5: **An example of Bold Tag**

5. Almost every markup tag in HTML requires both a starting tag and an ending tag. One notable exception is the paragraph marker, <P>, which does not require an ending </P> tag.

3.3 How to Write a Simple HTML Document

Now that you have learned how to create markup tags, the next step is to learn how to put them together to create a simple HTML Document. The basic HTML document contains two parts: the *head* and the *body*. The head section contains important information about the document itself, such as the title. The actual text, images, and markup tags are placed in the body section. You will learn the specifics of both sections in the next unit.

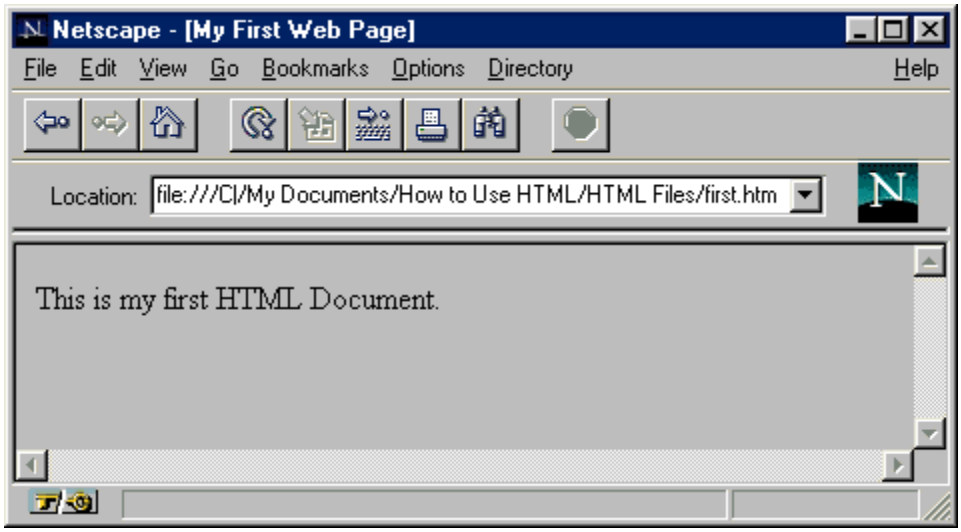


Figure 6: **A Simple HTML Document**

1. The first markup tag in every HTML document is the <HTML> tag. This lets Web browsers know that everything in the file is HTML text. Open a new blank document in Notepad. Type **<HTML>** on one line, and then on the next line, close the tag by typing **</HTML>**. From now on, everything you type in this document should go between these two tags.

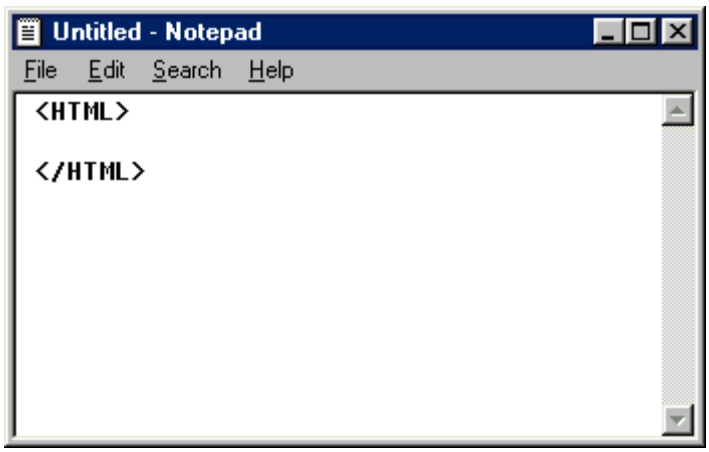


Figure 7: **HTML Tag.**

2. The head section comes next. Type **<HEAD>** on the line after the first HTML tag, followed by **</HEAD>** on the next line to create the section.

```
<HTML>

<HEAD>

</HEAD>

</HTML>
```

3. One of the key head elements is the title of your HTML document. To start the title tag, start a new line between the <HEAD> and </HEAD> tags and type **<TITLE>**. Now enter a title for your document, such as **My First Web Page**. Finally, end the title by typing **</TITLE>** on the same line.

```
<TITLE>My First Web Page</TITLE>
</HEAD>

</HTML>
```

4. The next section of your HTML document is the body. This section contains most of the elements of your document. To create the body section, type **<BODY>** on the next line. On the next line after that, type **</BODY>** to mark the end of the section. Most of your text and HTML codes will be placed between these two tags.

```
<TITLE>My First Web Page</TITLE>
</HEAD>

<BODY>

</BODY>
```

5. Right now, your HTML document is properly formatted, but it does not have any content. Fortunately, that is simple enough to change. On a new line between the beginning and ending body tags, begin typing some text, such as **This is my first HTML document**.

```
<BODY>
This is my first HTML Document.
</BODY>
```

6. Save your HTML file in Notepad with a descriptive file name, such as **first.htm**.
7. Using your Web browser, open your new HTML document. Because your file is on your local desktop machine and not on the Web, you will need to use the Open File option in your Web browser. With Netscape, choose Open File from the File menu, go to the folder where you saved your document, and select it.
8. There it is-your first HTML document. It may not look like much at this point, but you should give yourself a pat on the back. You are now an HTML author.

3.4 How to Use Special HTML Editing Software

Throughout this Course Material, you will learn to write HTML documents with the simplest of tools: a text editor. Creating HTML documents with a text editor is the best way to learn the language.

However, before you continue, you should know that there are a number of specialized HTML editing programs available. Some have graphical interfaces, others feature online help. All of them make creating HTML documents much easier. Once you have mastered the HTML basics, you may want to try out one of these programs. In this unit, we will show you what you should look for in an HTML editor.

- Many of the best HTML editors avail-able on the Internet are shareware. Shareware is a type of software marketing that allows you to try the soft-ware before you purchase it. If you decide that you like the software and want to keep it, you pay the author directly, according to the documentation supplied with the program. If you don't like the program, simply delete it and forget it. Shareware is a great way to find a program that's right for your tastes.
- Even if your favorite HTML editor does not support the latest HTML tags and features, you can always add them later using Notepad. Because HTML files are plain text, you can work on your HTML documents with just about any editor you like.

1. Make sure the HTML editing software has support for all the HTML features. If it does not, you would not be able to use all the cool HTML tricks you will learn in this Course Material.

Figure 8: **Some Features of HTML**

2. Look for toolbars and other features that make creating HTML easier. To create a markup tag, you can click on a button instead of typing it in.
3. Many WYSIWYG (What You See Is What You Get) HTML editors are now available. These allow you to see what your HTML document will look like as you're putting it together. This feature will save you the trouble of having to load your page with a Web browser every time you want to see how things are progressing.
4. Another feature to look for is HTML syntax checking. Editors with this capability can check your document for HTML errors. Some will even fix the errors for you automatically.
5. Many of the best HTML editors are available right on the World Wide Web as shareware. That means you can download them and try them out before buying them. There are plenty of places on the Web to find HTML editors. One of the best places to start looking is the HTML Editors section in the Yahoo directory.

4.0 CONCLUSION

The markup tags are the key to making HTML pages work. They are not case sensitive. Almost every markup tag in HTML requires both a starting tag and an ending tag.

5.0 SUMMARY:

In this unit, you have learnt:

- How to use Notepad
- How to use Markup tags
- How to write a simple HTML documents

6.0 Tutor Marked Assignment

- Mention 5 uses of Markup tags
- List 5 markup tags you know

7.0 Further Reading and Other Resources

- Abbate, Janet. *Inventing the Internet*. Cambridge: MIT Press, 1999.
- Bemer, Bob, "A History of Source Concepts for the Internet/Web"
- Campbell-Kelly, Martin; Aspray, William. *Computer: A History of the Information Machine*. New York: BasicBooks, 1996.

- Clark, David D., "The Design Philosophy of the DARPA Internet Protocols", Computer Communications Review 18:4, August 1988, pp. 106–114
- Graham, Ian S. *The HTML Sourcebook: The Complete Guide to HTML*. New York: John Wiley and Sons, 1995.
- Krol, Ed. *Hitchhiker's Guide to the Internet*, 1987.
- Krol, Ed. *Whole Internet User's Guide and Catalog*. O'Reilly & Associates, 1992.
- *Scientific American Special Issue on Communications, Computers, and Networks*, September, 1991

UNIT 4 - UNDERSTANDING THE BASICS OF HTML

1.0 INTRODUCTION

Now that you have learned how to use markup tags and have even written your first HTML document, you are ready to dig a little deeper and learn the basics of the HTML language.
In this Unit, you will cover the different sections of an HTML document, such as the head and body, and learn what type of information goes in each. You will also discover how to include basic paragraphs in your document, as well as insert headlines and special characters.

2.0 OBJECTIVES

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

- Describe the head and body section of HTML
- Create a headline for your document using head section
- Describe the paragraph marker.

3.0 MAIN CONTENT

3.1. How to Use the Head Section

The least thing you must include in the HEAD section of all your webpages in order to write valid HMTL documents is the **TITLE** element. The title of a webpage appears in your

browser´s title bar when you view the page. The screenshot below shows you the title of this webpage:

Figure 1: An example of HTML Head Section

The title tag is used extensively by Web search engines; search engines use the text inside a title tag as a way to determine the actual contents of your page. So make sure your title is descriptive.

- Don't type any extra text in between the <HEAD> and </HEAD> tags. In most cases, the only line you'll insert between those two tags is your document title.

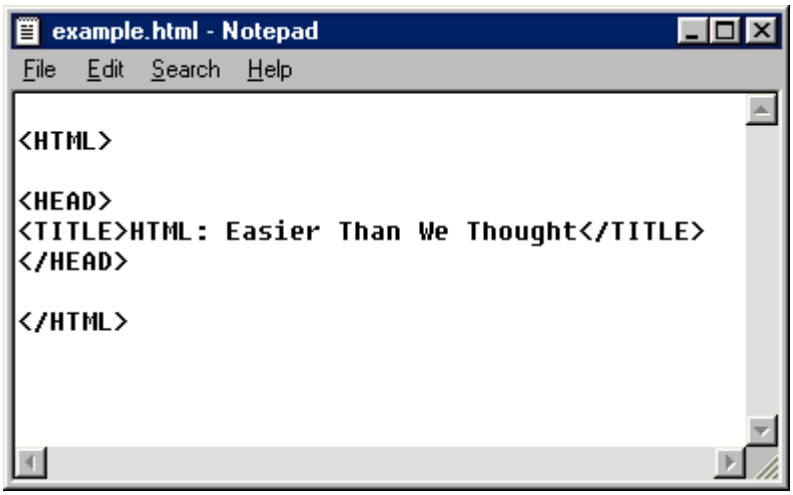


Figure 2: A Note pad Environment

1. Open a new document in Notepad and type **<HTML>**. To begin the head section, insert an opening tag into your HTML document by typing **<HEAD>**.
2. The only element required in the head section is the Title of your document. Your title should be short enough to fit in the title bar of a typical browser window, but descriptive enough to explain what your HTML document contains.
3. Insert a title tag within the head section by typing **<TITLE>**, followed by the actual title of your document. In this example, we'll name this document *HTML: Easier Than We Thought*. Go ahead and type in that title, then close the tag by typing **</TITLE>** on the same line.

```
<HTML>

<HEAD>
<TITLE>HTML: Easier Than We Thought</TITLE>
```

4. Close the head section by typing **</HEAD>** on the line below the title line.

```
<HTML>

<HEAD>
<TITLE>HTML: Easier Than We Thought</TITLE>
</HEAD>
```

3.2– How to Use the Body Section

The body section of your HTML document contains most of the text, graphics, hypertext links, and other information that will appear on the page. All of your HTML formatting tags, which describe the content and appearance of your document, are placed in the body section. These tags will be explained in detail in the next two units.

- Sometimes it is easier to type both the <BODY> and </BODY> tags on separate lines right away, and then fill in the rest of your HTML document between them.

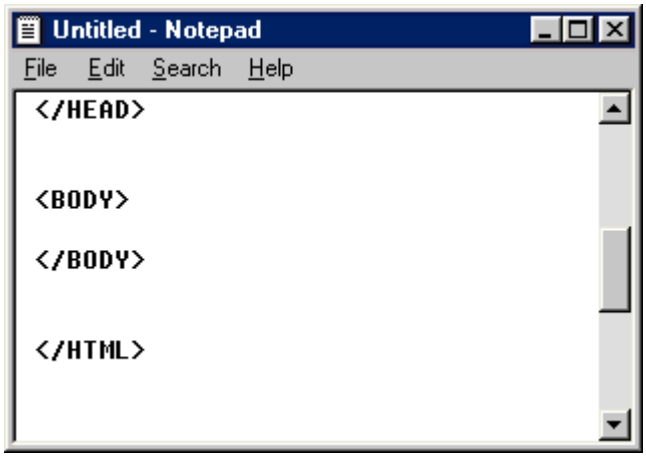


Figure 3: An example of HTML Body Section

1. Insert the opening body tag by typing **<BODY>** on a new line in your document. Make sure that the new body tag follows the end of the head section of your document.

```
<HEAD>
<TITLE>HTML: Easier Than We Thought</TITLE>
</HEAD>

<BODY>
```

2. Following the <BODY> tag, begin entering the actual text of your HTML document. For this example, we will just insert a simple sentence. Type **HTML is much easier than I thought.**

```
<BODY>
Creating HTML is much easier than I thought.
```

3. Close the body section of your document by typing **</BODY>** on a new line. Make sure that this closing tag appears before the </HTML> tag at the very bottom of your document.

```
Creating HTML is much easier than I thought.
</BODY>
```

4. Here is what your HTML document looks like so far when viewed with Netscape. Notice the placement of the document title and the body text.

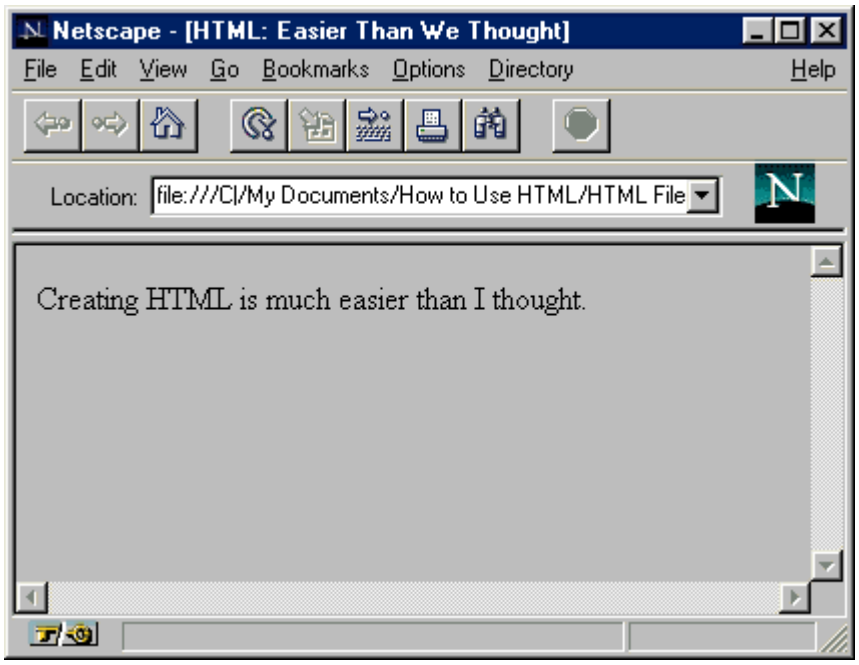


Figure 4: **A example of Head and Body Section**

5. At this point, you should save your file in Notepad. Make sure you save it with an extension of .htm or.html (it doesn't matter which-all browsers will handle both types). Keep this file open, because you will be adding to it in the next unit.

3.3 How to Use Headings

Headings are used in HTML documents to indicate different sections. There are six different Heading sizes, which range from very large to very small (smaller than the default body text). You should use headings judiciously, keeping them short and concise. The most common use for a heading is as the first line of a home page. In essence, it becomes a headline for your document.

- Headings are an excellent way to break up large amounts of text into smaller, digestible sections. But be careful not to overuse heading tags, or they'll make your document appear confusing.
- Think of heading tags as headlines. Generally, you'll only have one big headline for your document and a few smaller subheads to break the document into smaller sections.

- It is a good idea to repeat the document title as a Level 1 Heading at the very top of your page. This lets your readers know the title of the document without having to look at the title bar of their browsers.
- Headings can be compared in many ways to outlines. When structuring your documents with headings, use the same type of heading for elements of equal importance.

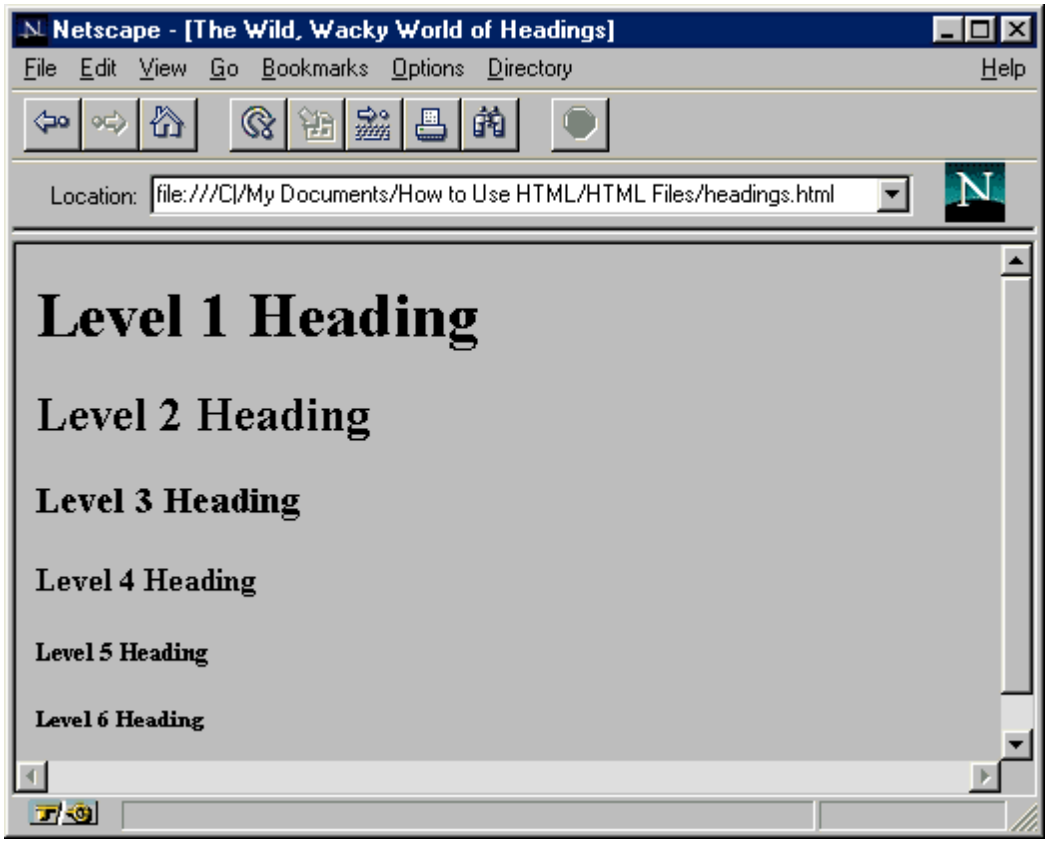


Figure 5: **Levels of Heading Tags**

1. To insert a heading into your document, place an opening tag anywhere in the body section. A heading tag follows the format of <H*x*>, where *x* is a number from 1 to 6, indicating the size from largest to smallest. Level 1 is the largest, type **<H1>**.

```
<H1>
```

2. Any text you enter immediately after the <H1> tag will be displayed in large bold type by a Web browser.

```
<H1>This is a Heading</H1>
```

3. Close the heading tag by typing **</H1>**.

```
<H1>This is a Heading</H1>
```

4. You can experiment with different sized headings by changing the number of the heading tag to any value between 1 and 6. The result will look something like this.

3.4 How to Use the Paragraph Tag

One of the most commonly used tags in HTML is the paragraph marker, which is used to break apart blocks of text into separate paragraphs. Any formatting that you perform in Notepad, such as placing carriage returns, extra spaces, or tab stops, will be ignored by Web browsers. The only way to indicate separate paragraphs is by using the paragraph marker. Unfortunately, despite its simplicity, the paragraph marker is also one of the most misunderstood tags in HTML.

- Remember that in HTML, paragraph tags are considered to be containers of text. That means each paragraph should have a starting <P> tag and an ending </P> tag. Early versions of HTML used the <P> tag as a paragraph separator.
- Paragraphs can contain more than plain text. You can place images, hyperlinks, and many other HTML elements inside paragraphs as well. You will learn more about these elements in later modules.

1. The most important thing to remember about the paragraph tag is that it marks the beginning of a paragraph, not the end. The original HTML standard used the paragraph marker differently, which has led to some confusion.
2. To insert a new paragraph, type **<P>** anywhere in the body section of your HTML document. This will tell the browser to insert a line space and start a new paragraph.

```
<BODY>

<P>

</BODY>
```

3. Enter the text of the paragraph after this tag. Remember that any carriage returns or line breaks you enter into Notepad will be ignored by a Web browser. The browser will continue to treat the text as part of the current paragraph until it sees another <P> tag.

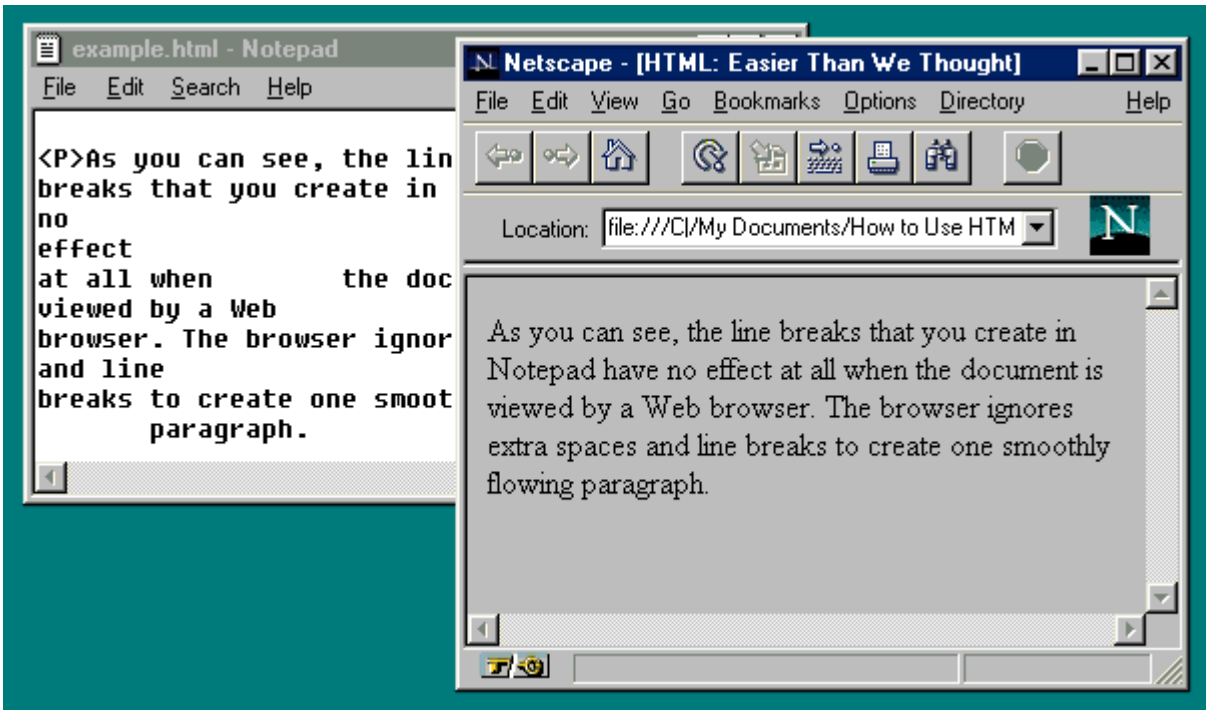


Figure 6: An Example of Paragraph Tag.

4. You can indicate the end of a paragraph by typing </P>. However, this tag is optional. The end of the current paragraph is implied whenever a new paragraph marker is found by a browser.

```
paragraph.</P>
```

5. Continue entering new paragraphs of text, using the <P> tag to indicate the beginning of each.

```
<P>This is a paragraph in HTML.</P>
<P>This is a second paragraph, only it's a
little bit longer. It still uses the same
opening and closing paragraph tags.</P>
<P>You can include all of the HTML character
formatting codes inside of paragraphs. For
example, you can make text appear <B>bold</B>
or in <I>italics</I>.</P>
```

3.5 How to Use Special Characters

By now, you may have noticed a potential problem with HTML. All of the markup tags are indicated by left and right angle brackets (greater-than and less-than symbols). These characters are reserved by HTML for use with tags. What happens when you want to include one of these characters in your text?

That's a good question, and the problem isn't limited to just those two symbols. A number of characters can't be typed directly into the body text HTML, including many foreign language symbols. Fortunately, HTML provides a solution through the use of *character entities*. By

using special codes, HTML can display all of the characters in the ISO-Latin-1 (ISO 8859) character set. HTML 3.2 also includes support for many mathematical symbols.

1. Locate your cursor at the position in the document where the character entity for the special character is to be placed.

```
<P>Goodbye, whispered Lauren. Then, without
another word, she walked out of Fabio's life
and went on to pursue her dream of becoming
a professional kazoo player.</P>
```

2. A character entity begins with an ampersand (&), followed by the code, and ends with a semicolon. To place a double quote in your document, for example, type **"**.

```
<P>&quot;Goodbye,&quot; whispered Lauren.
Then, without another word, she walked out
of Fabio's life and went on to pursue her
dream of becoming a professional kazoo
player.</P>
```

3. Other common character entities for characters that are reserved for HTML tags are *<* for the less-than symbol; *>* for the greater-than symbol; and *&* for the ampersand. Note that these named character entities are case-sensitive.
4. You can also use named character entities for many foreign language symbols. For example, to create the umlaut used in the German phrase, *über alles*, you would type in **über alles**.

```
&#174;
```

5. In addition to named character entities, you can use numbered character entities. HTML uses a subset of the ISO 8859/1 8-bit character set, and several characters, including the copyright symbol, trademark symbol, and mathematical symbols, are available when referenced by their numbered character entity.
6. To insert a numerical character entity into HTML, type an ampersand, followed by a pound sign, the number of the character and a semicolon. For example, to enter the registered trademark symbol into your document, you would type **®**. You can find a partial list of numerical character entities in the Appendix.

über alles
mañana
resumé

4.0 CONCLUSION:

The two major sections of an HTML document are the head and body sections. The two sections are always within the HTML tags.

5.0 SUMMARY:

In this unit, you have learnt:

- How to use the head and the body sections
- How to use headings and
- How to insert paragraphs tags.
- How to use special characters

6.0 Tutor Marked Assignment

- Given 5 line document, insert 3 new paragraphs on the document
- What are the uses of the head section of HTML document
- What are the steps of inserting paragraph tags in a document.

7.0 Further Reading and Other Resources

1. `Arpajian, S., and R. Mullen. 1996. ***How to use HTML 3.2.*** Emeryville, Ziff-Davis Press. 219 pp.

2. Castro, E. 1998. ***HTML 4 for the world wide web.***Berkeley, Peachpit Press. 336 pp.
3. Graham, I. S. 1997. ***HTML sourcebook, third edition***. New York, John Wiley & Sons. 620 pp.
4. Williams, R. 1994. ***The non-designer's design book. Design and typographic principles for the visual novice.*** Berkeley, Peachpit Press. 144 pp.
5. Williams, R., and J. Tollett. 1998. ***The non-designer's web book. An easy guide to creating, designing , and posting your own web site.*** Berkeley, Peachpit Press. 288 pp.

MODULE 3 PHYSICAL MARKUP TAGS, HYPERTEXT LINKS, CREATING LISTS IN HTML

Unit 1 Formatting Text

Unit 2 Using Hypertext Links

Unit 3 Creating Lists In Html

UNIT 1 FORMATTING TEXT

CONTENT

1.0 INTRODUCTION

HTML was originally designed as a markup language, not as a formatting and layout specification. The key difference is that HTML allows the author to specify how certain elements are to be used, not necessarily how they are supposed to look. The actual details of presentation are left up to the client-the Web browser.

That is how HTML was originally designed, but that is not necessarily how things turned out. Increasingly, HTML designers are demanding greater control over the look and feel of their documents. HTML provides that control, and yet still allows HTML authors to take the first approach and allow formatting to be handled entirely by the browser.

As the author of your own document, you will decide how you want your page to look. In this unit, you will learn how to handle basic formatting for text and paragraphs. You will also learn a few valuable techniques for breaking large amounts of text into readable chunks.

2.0 OBJECTIVES

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

- Describe the physical markup tags
- Describe the logical markup tags
- State the difference between formatting tags and preformatted text

3.0 MAIN CONTENT

3.1 How to Format Characters with Physical Tags

HTML provides two general ways to apply formatting to text. The first group of formatting tags is collectively known as *physical markup tags.* This type of tag gets its name because it indicates a specific change in appearance. Bold and italic tags, for example, are known as physical markup tags because they directly specify how the text should appear on screen. In this section of the unit, we will look at how you can use physical tags in HTML.

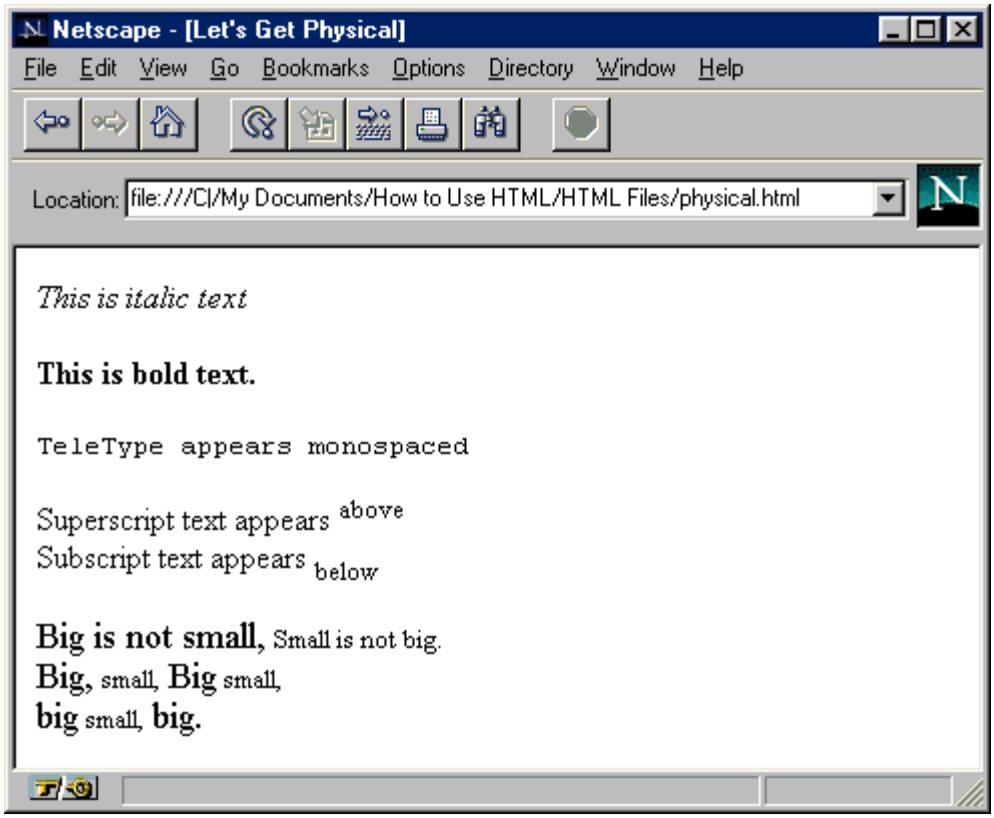


Figure 1: **An Example of formatting Tags**

1. In general, all character formatting tags work the same. Each has a starting tag and an ending tag. All of the text that falls between the two tags inherits the specified format. In addition, you can nest formatting tags inside one another to combine effects.
2. To create italic text, insert an **<I>** tag in the document, followed by a **</I>** tag. Any text between these two tags will be displayed in italics when viewed by a browser.

```
<I>This is italic text</I>
```

3. To create bold text, insert **<B>** and **</B>** tags. Any text falling between these two tags will appear in boldface type.

```
<B>This is bold text.</B>
```

4. To create text that is displayed in a monospaced font (such as Courier), use the **<TT>** and **</TT>** tags. Text falling between these two tags will be displayed in a fixed-width font, similar to the output from a teletype machine or typewriter.

```
<TT>TeleType appears monospaced</TT>
```

5. To create strike-through text, which is text with a single horizontal line running through it, use the **<STRIKE>** and **</STRIKE>** tags.

```
<S>Strike-through text</S>
```

6. Underlined text can be displayed using the **<U>** and **</U>** tag pair. You should use these tags only when absolutely necessary, as underlined text is not widely supported by Web browsers.

```
<U>Underlined Text</U>
```

7. You can change the font size of normal text. Using the **<BIG>** and **</BIG>** tags will increase the size of the indicated text relative to the default size. **<SMALL>** and **</SMALL>** will make the text smaller.

```
<BIG>Big is not small,</BIG>
<SMALL>Small is not big.</SMALL>
```

8. You can also format text as either superscript or subscript, which is text that appears slightly above or below the current line, respectively. Superscript and subscript numbers are often used in mathematical equations or to indicate footnotes. Using the **^{** and **}** tags will mark text as superscript (slightly above the current line). **_{** and **}** will mark text as subscript (slightly below the current line)

```
Superscript text appears <SUP>above</SUP>
Subscript text appears <SUB>below</SUB>
```

3.2 – How to Format Characters with Logical Markup Tags

On the previous page, you learned how to specify the appearance of text using physical markup tags. However, there is a second method for formatting text-through the use of *logical markup tags,* sometimes known as information style elements.

Logical tags take the approach that what is really important is the *type* of information being displayed, rather than exactly *how* it is displayed. Logical tags leave the actual appearance decisions-such as whether to display text in boldface, italics, or larger sizes-up to the browser (and ultimately the reader).

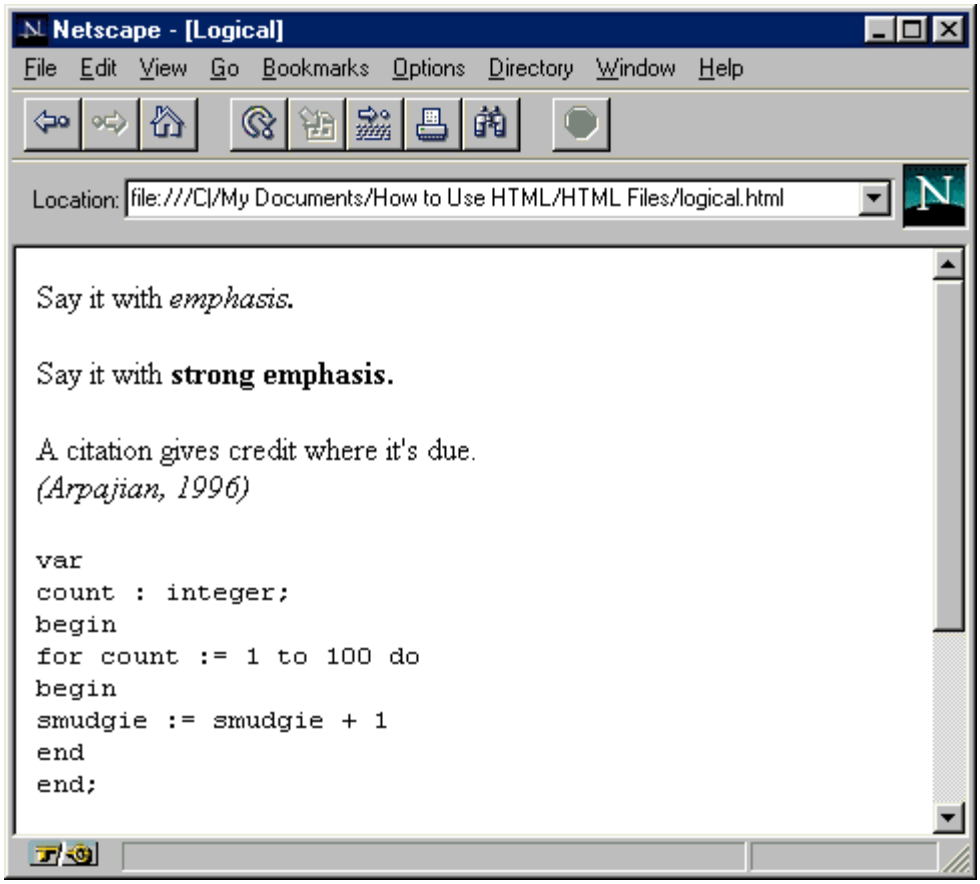


Figure 2: **An Example of Emphasis Tag.**

1. When you want to add importance to a section of text, you can use the logical style tag called *emphasis*. Using the **<EM>** and **</EM>** tags will usually display the indicated text in italics. However, remember that with logical tags, the actual appearance of the text is determined by the end user's Web browser, not your HTML document.

```
Say it with <EM>emphasis.</EM>
```

2. If a particular section of text is very important, you can mark it with *strong emphasis* by using the **<STRONG>** and **</STRONG>** tag pair. Most browsers tend to display strongly emphasized text in boldface.

```
Say it with <STRONG>strong emphasis.</STRONG>
```

3. The **<CODE>** and **</CODE>** tags indicate that the text is to be presented as an example of *programming code*. In most browsers, this text will be displayed in a monospaced font, such as Courier. The <CODE> tags are used extensively in interactive computer manuals.

```
<CODE>
var
  count : integer;
begin
 for count := 1 to 100 do
   begin
    smudgie := smudgie + 1
   end
end;
</CODE>
```

4. The **<SAMP>** and **</SAMP>** tags are very similar to the <CODE> tags, and are used to indicate sample text that isn't specifically programming code. Most Web browsers will handle both sets of tags in the same way.
5. The **<KBD>** and **</KBD>** tags indicate text that is supposed to be typed in by the reader. By default, most browsers will display this text in a similar fashion to the <CODE> and <SAMP> tags.

```
The user types in <KBD>keyboard text</KBD>.
```

6. The **<CITE>** and **</CITE>** tags are used to insert a citation to give credit for a short quotation in the body of the document. Citations are typically displayed in italics.

```
A citation gives credit where it's due.
<CITE>(Arpajian, 1996)</CITE>
```

7. The **<DFN>** and **</DFN>** tags are used to highlight the *defining instance* of a term. This is a word or phrase that is being defined in the context of the paragraph in which it appears.

```
The tag used to highlight a word or phrase that
will be defined is called the <DFN>defining
instance tag</DFN>.
```

3.3 - How to Format Paragraphs

Now that you have learned all the ways to format individual characters, words, and phrases, you are ready to examine the options you have for presenting entire sections of text. As with normal documents, the basic section of text in HTML is the paragraph. HTML provides many new ways to present, format, and align paragraphs.

1. The basic paragraph tag is always used to start a new paragraph. To indicate a paragraph, type **<P>**. This tells the Web browser to insert a line space and begin a new paragraph. The <P> tag always creates a simple, left-justified

paragraph. Although the closing <P> tag is optional, you may want to include it to help you remember where a paragraph ends.

```
<P>
```

2. You can change the justification of the paragraph with the *ALIGN* attribute. To change the alignment of a paragraph, put the ALIGN statement in the paragraph tag, followed by the type of justification you want. To create a right-justified paragraph, type **<P ALIGN=RIGHT>**.

```
<P ALIGN=RIGHT>
```

3. To create a centered paragraph, type **<P ALIGN=CENTER>**. To create a paragraph that is justified on both sides, type **<P ALIGN=JUSTIFY>**. You can also create a left-justified paragraph by typing **<P ALIGN=LEFT>**. However, since this is the default, just typing **<P>** will have the same effect.

```
<P ALIGN=CENTER>
```

1. By default, the Web browser will wrap lines of text to keep the entire paragraph in view. You have the option of turning off word wrapping by including the *NOWRAP* command in the paragraph tag. To turn off word wrapping in a paragraph, type **<P NOWRAP>**. This will allow you to explicitly place line breaks using the
 tag, which is explained in the next section.

```
<P NOWRAP>
```

2. Normally, paragraphs will wrap around an object in the margin, such as a figure or table. To force the paragraph to begin below the object, you can use the *CLEAR* attribute. Typing **<P CLEAR=LEFT>** moves the paragraph down until the left margin is clear. **CLEAR=RIGHT** forces the paragraph down to a point where the right margin is clear. **CLEAR=ALL** forces the paragraph to wait until both margins are clear.

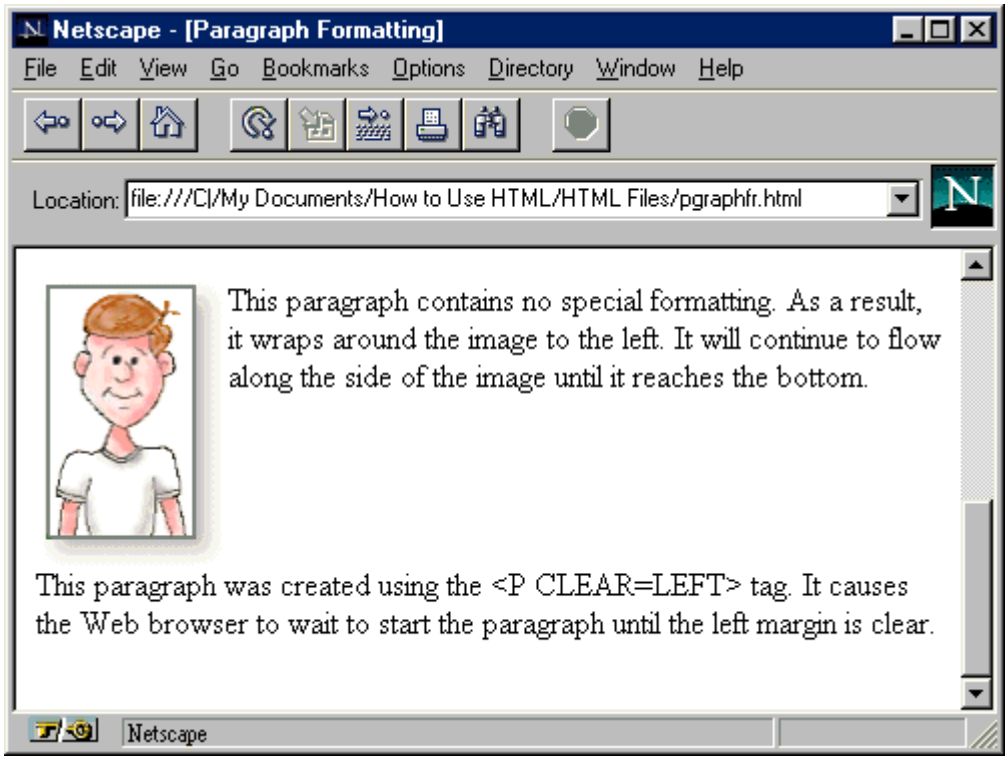


Figure 3: **An Example of Special Formatting**

3. To combine formatting commands in the same paragraph, type all the attributes together in the same <P> tag. For example, to create a centre-aligned paragraph with no word wrapping, type **<P ALIGN=CENTER NOWRAP>**.

```
<P ALIGN=CENTER NOWRAP>
```

3.4 How to Use Text Breaks

Not all text fits neatly into paragraphs. Sometimes you want the reader's Web browser to end a line of text at a specific point. If you're using HTML to display poetry, lyrics, instructional materials, or any other type of information where specific formatting is necessary, you will want to have control over the flow of text in the document.

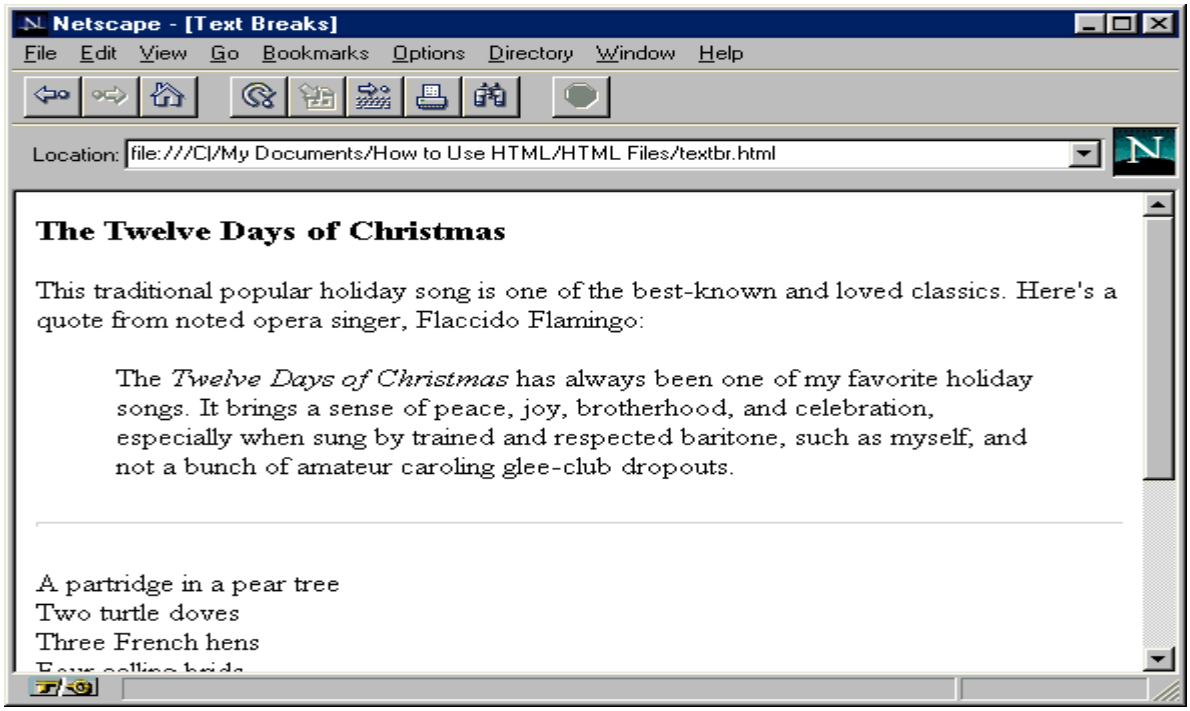


Figure 4: An Example of Line Break

1. To insert a line break at a specific point, type **
**. This instructs the Web browser to immediately end the current line and begin placing text on the next line. A line break does not start a new paragraph.

```
<P>A partridge in a pear tree<BR>
```

2. You can use multiple line breaks to create a short, informal list of items. By creating a new paragraph before and after the list, you can separate it from the rest of your text.

```
Four calling brids<BR>
Five golden rings<BR>
Six geese a-laying<BR>
Seven swans a-swimming<BR>
Eight maids a-milking<BR>
```

3. Sometimes you'll want to visually break apart sections of text using a visible line. HTML supports this through the use of *horizontal rules*. These can be added anywhere in the document by typing **<HR>**. A thin line stretching across the entire window will be placed at that point in the text. Horizontal rules, like paragraphs, support the clear attribute to allow you to begin the line when the margins are clear.

```
<HR>
```

4. To place an entire section of text apart from the rest, use the **<BLOCKQUOTE>** and **</BLOCKQUOTE>** tag pair. This tag, used in place of a paragraph tag, will offset an entire paragraph from the main body of text, usually by indenting it and adding extra spaces to the top and bottom. It is commonly used to highlight long quotations and passages.

```
<BLOCKQUOTE>
The <I>Twelve Days of Christmas</I> has
always been one of my favorite holiday
songs. It brings a sense of joyous
celebration,  especially when sung by a
group of friends.
</BLOCKQUOTE>
```

3.5 How to Use Preformatted Text

Preformatted text allows you break away from the normal rules of HTML and quickly specify exactly how a section of text will appear in the reader's Web browser. When you are using preformatted text, you don't need to use the HTML markup tags-the text will appear exactly as you have typed it, complete with spaces, line breaks, and empty lines. Preformatted text is always displayed in a monospaced, fixed-width font.

1. To begin a section of preformatted text, type **<PRE>**.

```
<PRE>
```

Now type the section of text exactly how you want it to appear. It's a good idea to limit the length of your lines to 65 characters or less, so that you can accommodate the screen width of most browsers. (Remember that browsers will not word wrap preformatted text.)

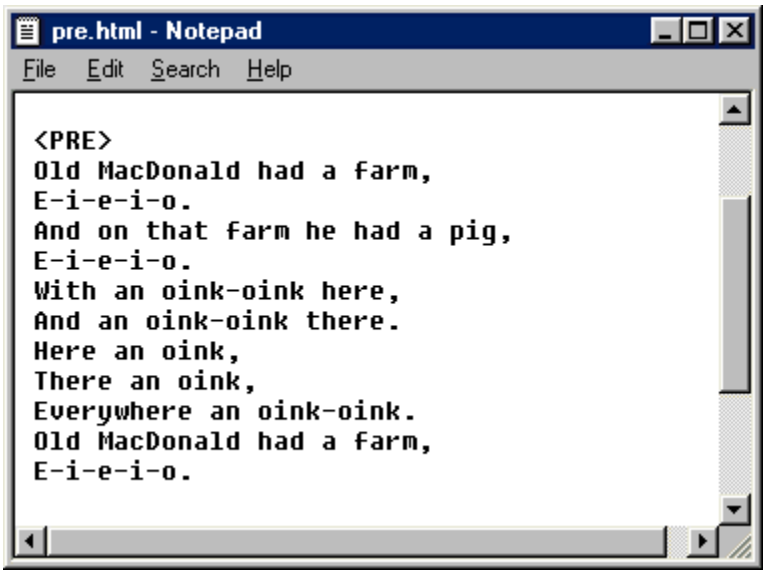


Figure 4 : **An Example of preformatted text**

2. When you're finished entering your preformatted text, type **</PRE>** to mark the end of the section.

```
</PRE>
```

3. You can apply character formatting styles, such as bold and italic, in preformatted text. Headings and paragraphs will not work in preformatted text blocks, however.

4.0 CONCLUSION:

The physical and logical markup tags are the two general ways of formatting text. They are with different markup tags.

5.0 SUMMARY:

In this unit, you have learnt the following:

- Physical Markup tags
- Logical Markup tags
- Paragraph formatting

6.0 Tutor Marked Assignment

- List the physical Markup tags you know?
- Distinguish between physical and logical markup tags

7.0 Further Reading and Other Resources

- `Arpajian, S., and R. Mullen. 1996. ***How to use HTML 3.2.*** Emeryville, Ziff-Davis Press. 219 pp.
- Castro, E. 1998. ***HTML 4 for the world wide web.***Berkeley, Peachpit Press. 336 pp.
- Graham, I. S. 1997. ***HTML sourcebook, third edition***. New York, John Wiley & Sons. 620 pp.
- Williams, R. 1994. ***The non-designer's design book. Design and typographic principles for the visual novice.*** Berkeley, Peachpit Press. 144 pp.
- Williams, R., and J. Tollett. 1998. ***The non-designer's web book. An easy guide to creating, designing , and posting your own web site.*** Berkeley, Peachpit Press. 288 pp.

UNIT 2 USING HYPERTEXT LINKS

1.0 INTRODUCTION

The single greatest feature of the World Wide Web is its diverse collection of documents, which number in the millions. All of these documents are brought together through the use of hypertext links. Users navigate the Web by clicking on the links that HTML authors provide. Hypertext links are a crucial part of HTML-which, after all, is short for *Hypertext Markup Language*.

In this unit, we will look at the simple process behind how hyperlinks work in HTML documents. You will also learn how to link to a specific point in a large document by using the ID attribute. Finally, we will take a look at the difference between using absolute and relative path names in your hyperlink references.

Linking is one of the easiest and most important parts of using HTML. So warm up your Web browser and Notepad and get ready to explore.

2.0 OBJECTIVES

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

- Define hypertext link
- List types and uses of hypertext link
- **S**tate the steps of creating hyperlinks
- Create hyperlinks

3.0 MAIN CONT

3.1 What is hypertext link

Hypertext is text displayed on a computer or other electronic device with references (hyperlinks) to other text that the reader can immediately access, usually by a mouse click or key press sequence. Apart from running text, hypertext may contain tables, images and other presentational devices. Hypertext is the underlying concept defining the structure of the World Wide Web. It is an easy-to-use and flexible format to share information over the Internet.

Types and uses of hypertext

Hypertext documents can either be static (prepared and stored in advance) or dynamic (continually changing in response to user input). Static hypertext can be used to cross-reference collections of data in documents, software applications, or books on CDs. A well-constructed system can also incorporate other user-interface conventions, such as menus and command lines. Hypertext can develop very complex and dynamic systems of linking and cross-referencing. The most famous implementation of hypertext is the World Wide Web , first deployed in 1992.

3.2 **How to Create a Hyperlink**

Hyperlinks connect two different documents. You can link to one of your own documents or to any other document on the World Wide Web. You can even link to a different section in the same document. It is very easy to create links with HTML, and you only need to follow a

few simple steps.

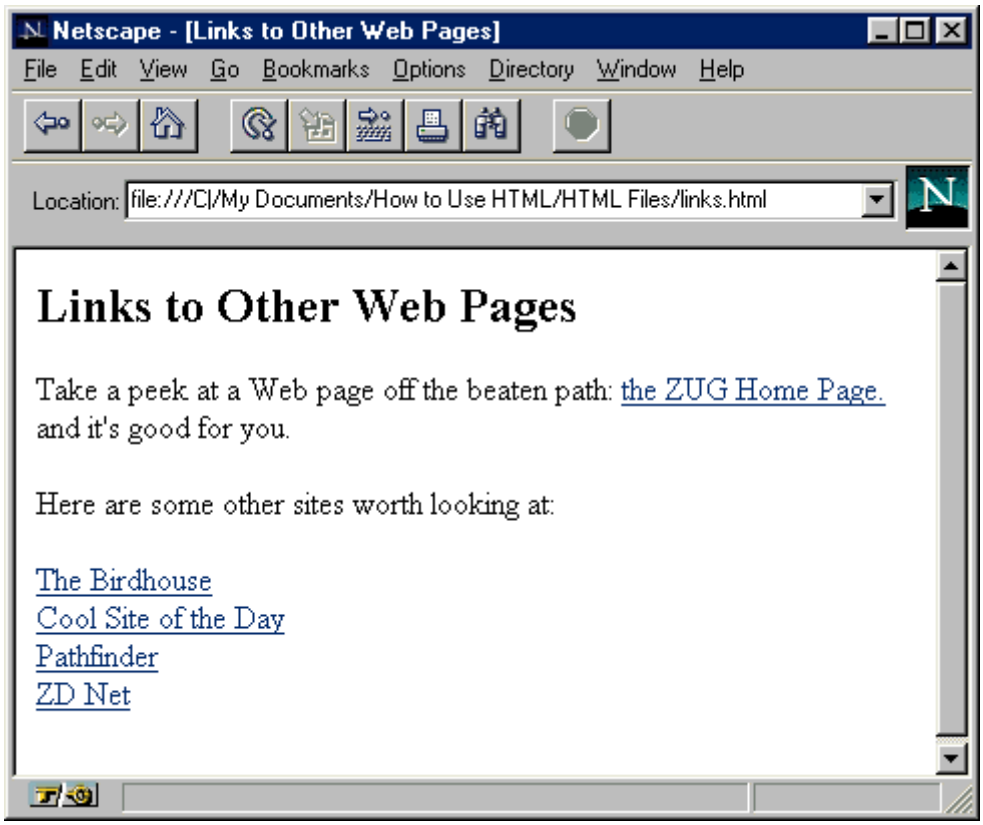


Figure 1: An Example of Hyperlink Document

1. Use your Web browser to locate the document you want to link. You can link to any other document on the World Wide Web.
2. Make a note of the Uniform Resource Locator (URL) of the document you want to link to. The URL is prominently displayed by your Web browser, usually near the top. Make sure to note the complete URL.
3. To make a link to another document, you need to use a special type of HTML tag known as an *anchor tag*, also commonly known as a *link tag*. Locate the place in your HTML document where you want to insert the hypertext link. Type <**A HREF**=", followed by the URL of the document you want to link to. Then close the tag by typing ">.

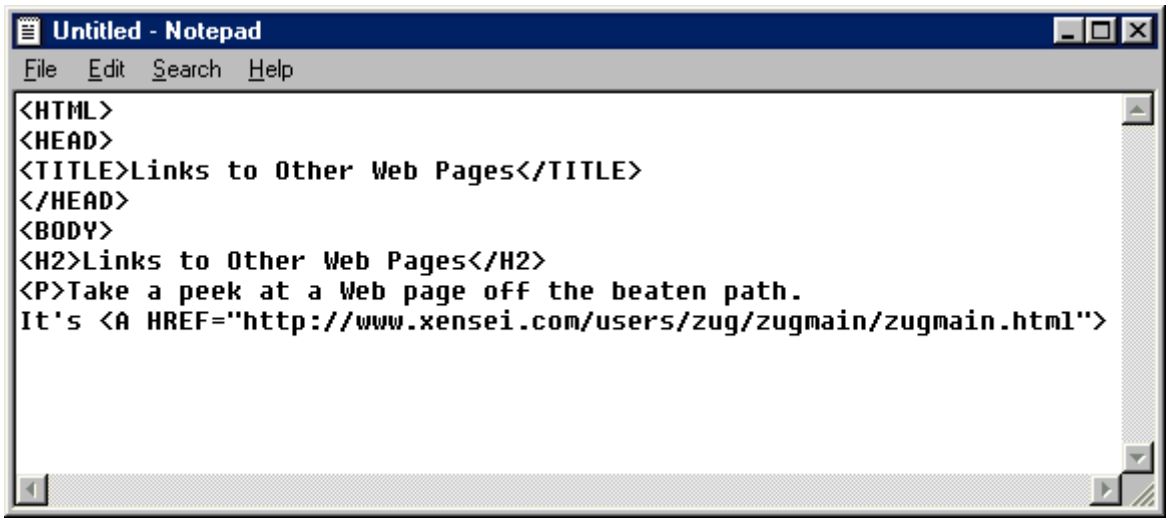


Figure 2: An Example of Link Tag

4. Type some descriptive text (also known as the *link text*) after the anchor tag to let readers know something about where this link will take them.

```
/zugmain.html">the Zug Home Page.
```

5. Finish the anchor tag by typing **</A>** on the same line.

```
/zugmain.html">the Zug Home Page.</A>
```

6. Once you've created your link, check to make sure it works by clicking on it while using your Web browser. Note that by default, most Web browsers display hypertext links as underlined text in a different colour than normal text. This lets your readers know that clicking on the text will take them to another document.

3.3 How to Use the ID Attribute

When you create a simple link to a Web page using the technique you learned in section 3.1, the reader is always taken to the top of the new page. What if you want to link to a particular section of a document and take the reader immediately to that point?

Assigning an ID to an element in your HTML document allows hyperlinks to point directly to that element instead of to the very top of the page. The HTML 3.0 ID attribute did not make it into the HTML 3.2 standard, but it is rendered by some browsers. You can use the ID attribute for most HTML elements, such as paragraphs, headings, and lists.

1. Locate the element you'd like to name with an ID. This can be almost any element in your document, but it is usually a paragraph or heading.

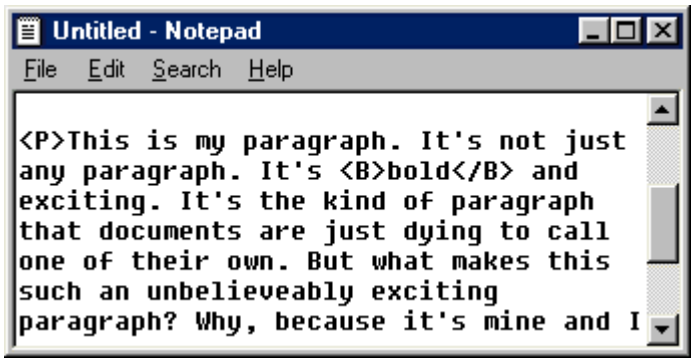


Figure 3: An Example of ID Document

2. Inside the element's opening tag, just after the letter P, insert a space and type **ID=**.
3. Your element ID needs a name. The hyperlinks will use this name to take readers directly to this section of your document. In this example, we'll simply name the element "MyParagraph".
4. Type the name of your ID, inside quotation marks.

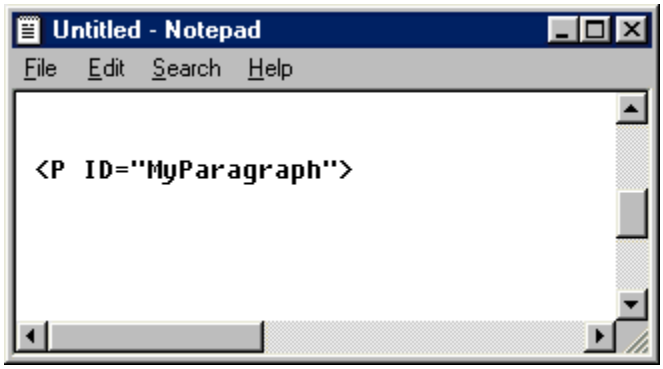


Figure 4: Another Example of ID Document

5. To create a hyperlink directly to this element, add a pound sign and the ID name inside your hyperlink tag. For example, to link directly to "MyParagraph", a typical hyperlink might look like the one above.

3.3 How to Use Relative Path Names

In the beginning of this unit, you learned how to create a hyperlink by pointing to the full URL of another document. However, if you are linking to different documents on the same Web server (usually your own), you do not always need to use the full URL. You can use *relative path names.*

Web browsers, even when running on PC or Macintosh machines, always follow UNIX style path names. This means that directories (folders) are separated by forward slash marks (/), and higher-level directories are indicated by two periods (..).

- The simplest relative path name is no path name at all. If you are linking to another document that is in the same directory, all you have to do is type in the file name of the new document in place of the full URL. For example, to link

to a document named newfile.html, type
<AHREF="newfile.html">.

```
<A HREF="newfile.html">
```

- To link to documents or files in a subdirectory, all you need to specify is the path and file name relative to the current document. For example, to link to a document called budget.html in a subdirectory named budget96, you would type
<A HREF="budget96/budget.html">.

```
<A HREF="budget96/budget.html">
```

- You can also navigate up the directory tree of your server by using two periods (..) to move up one level. For example, to link from the budget.html file in the previous example back to the main document, you would type
<A HREF="../main.html">.

```
<A HREF="../main.html">
```

- If the new document was two levels above the current one, you would separate each level with a slash, and type
<A HREF="../../main.html">.

```
<A HREF="../../main.html">
```

- The single greatest advantage to using relative path names is portability. If you do your HTML development on a local machine, and then upload your finished work to a Web server, you can save yourself the trouble of having to reset all of your hyperlinks to reflect the new location. Likewise, relative path names will save you the headache of changing your hyperlinks if you move your existing HTML files to an entirely new Web server

4.0 CONCLUSION:

Hypertext systems are particularly useful for organizing and browsing through large databases that consist of disparate types of information. It is the underlying concept defining the structure of the World Wide Web.

5.0 SUMMARY

In this unit, you have learnt the following:

- Definition of hypertext link?
- Types and Uses of hypertext

- Stated the steps in the creation of hyperlink
-

6.0 Tutor Marked Assignment

- Define hypertext link
- Distinguish between hyperlink and hypertext
- List the steps of creating a hyperlink

7.0 Further Reading and Other Resources

- Barnet, Belinda (2004). *Lost In The Archive: Vision, Artefact And Loss In The Evolution Of Hypertext*. University of New South Wales, PhD thesis.
- Bolter, Jay David (2001). *Writing Space: Computers, Hypertext, and the Remediation of Print*. New Jersey: Lawrence Erlbaum Associates. ISBN 0-8058-2919-9.
- Buckland, Michael (2006). *Emanuel Goldberg and His Knowledge Machine*. Libraries Unlimited. ISBN 0-31331-332-6.
- Byers, T. J. (April 1987). "Built by association". *PC World* **5**: 244–251.
- Cicconi, Sergio (1999). "Hypertextuality". *Mediapolis. Ed. Sam Inkinen. Berlino & New York: De Gruyter.*: 21–43. http://www.cisenet.com/cisenet/writing/essays/hypertextuality.htm.
- Conklin, J. (1987). "Hypertext: An Introduction and Survey". *Computer* **20** (9): 17–41. doi:10.1109/MC.1987.1663693.
- Crane, Gregory (1988). "Extending the boundaries of instruction and research". *T.H.E. Journal (Technological Horizons in Education)* (Macintosh Special Issue): 51–54.
- Ensslin, Astrid (2007). *Canonizing Hypertext: Explorations and Constructions*. London: Continuum. ISBN 0-8264-95583.
- Landow, George (2006). *Hypertext 3.0 Critical Theory and New Media in an Era of Globalization: Critical Theory and New Media in a Global Era (Parallax, Re-Visions of Culture and Society)*. Baltimore: The Johns Hopkins University Press. ISBN 0-8018-8257-5.
- van Dam, Andries (July 1988). "Hypertext: '87 keynote address". *Communications of the ACM* **31**: 887–895. doi:10.1145/48511.48519. http://www.cs.brown.edu/memex/HT_87_Keynote_Address.html.
- Yankelovich, Nicole; Landow, George P., and Cody, David (1987). "Creating hypermedia materials for English literature students". *SIGCUE Outlook* **20** (3): All.

UNIT 3– CREATING LISTS IN HTML

1.0 INTRODUCTION

Everyone makes lists. Whether you use them for groceries, to-do items, or holiday gifts and cards, lists are an important part of one's life. Lists are also important on the World Wide Web. The environment of the Web calls for information to be presented in a concise and timely manner. Lists are ideal vehicles for delivering all kinds of information on line. In HTML, you have many choices for how to create and present lists. In this unit, we will look at ways to create *unordered lists*, *ordered* (numbered) *lists*, and a special type of list known as a *definition list*. You will also learn how to combine multiple levels of lists.

2.0 OBJECTIVES

In this unit, you should be able to:

- Define unordered lists, ordered list, definition list and list within list
- State the steps in creating unordered lists, ordered list, definition list and list within list
- Distinguish between Ordered list and list within list

3.0 MAIN CONTENT

3.1 How to Create Unordered Lists

The simplest list in HTML is the unordered or bulleted list. This is ideal for listing items that have no particular hierarchy or order of importance. Unordered lists are very common on the Web and are used to convey items of note in a quick and concise manner. Web browsers usually place bullets or other markers in front of each item in an unordered list.

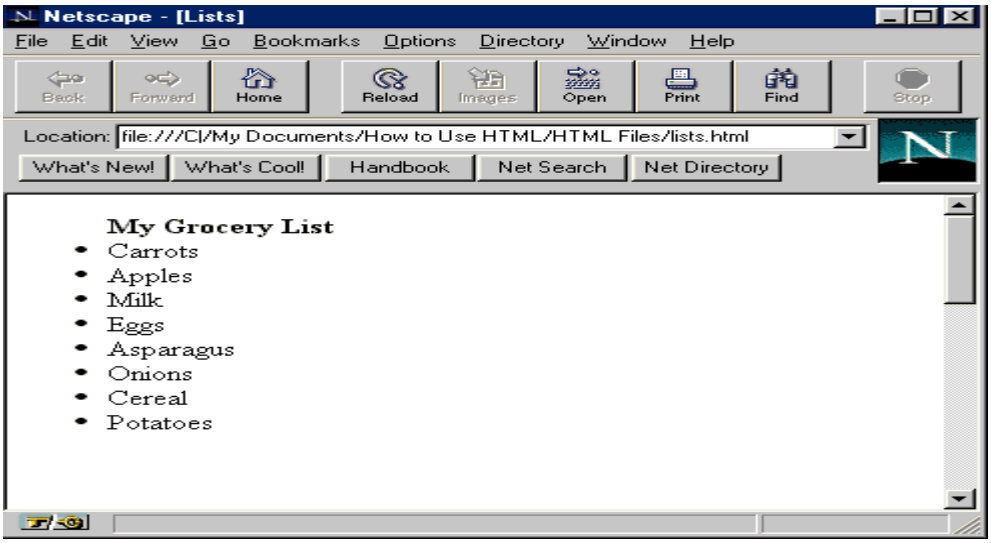


Figure 1: An Example List in HTML.

1. Locate the part of your HTML document where you want to insert a list.
2. Begin the unordered list by typing **<UL>**, and then press Enter. The *<UL>* tag tells the Web browser to treat this section of text as an unordered list. Unordered lists will usually be indented from the main document and list items will be formatted with bullets. The size and type of bullets used are determined by the Web browser.

```
<UL>
```

3. Create a heading for your list. This is an optional brief description of what your list contains. To create a list header, type **<LH>**, followed by a brief summary of the list contents. Then type **</LH>** to close the list heading tag. For example, to create a list heading for a grocery list, you would type **<LH>My Grocery List</LH>**.

```
<UL>
<LH>My Grocery List</LH>
```

4. To create the first item in your list, type **<LI>**. Then type the text of the item itself. *<LI>* is an open tag, which means that you do not need to type </LI> at the end of each item.

```
<UL>
<LH>My Grocery List</LH>

<LI>Carrots
```

5. Continue typing **<LI>** followed by text for each item in your list. Press Enter after each item.
6. Finish the unordered list by typing **</UL>**.

```
<LI>Potatoes
</UL>
```

3.2 How to Create Ordered Lists

Sometimes you need to list items in a specific order. Examples of this type of list include step-by-step instructions and "Top 10" lists. HTML provides a way to do this through ordered lists. Web browsers will place a number in front of each item, increasing the number by one for each entry down the list.

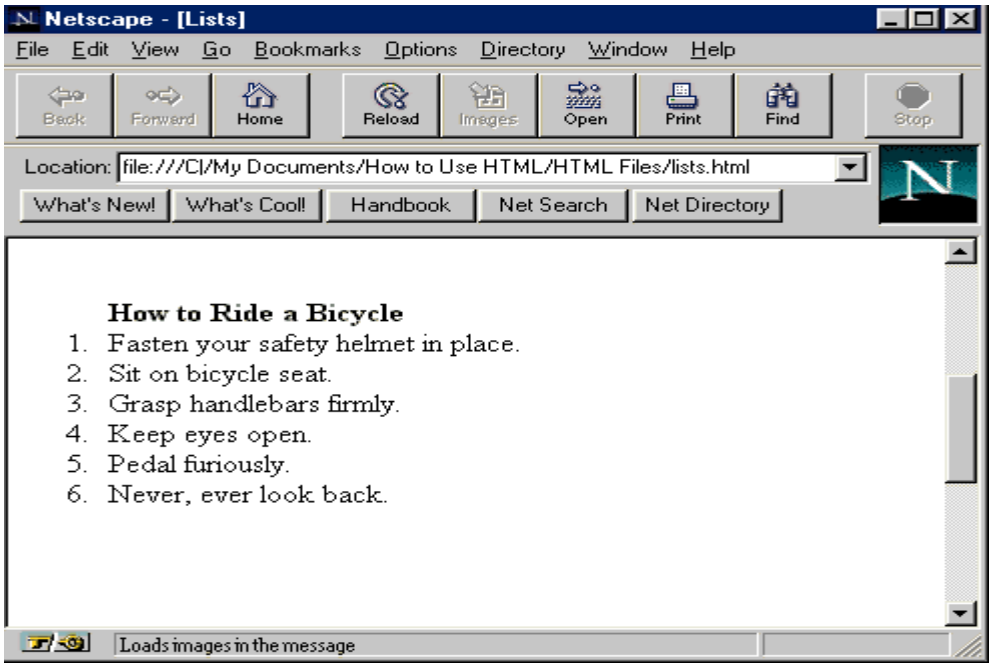


Figure 2: An Example of Ordered List in HTML

1. To create an ordered list, locate the place in your document where you'd like to begin the list and type **<OL>**.

```
<OL>
```

2. To create an optional heading for the ordered list, type **<LH>** followed by the heading. Then close the heading tag by typing **</LH>**.

```
<LH>How to Ride a Bicycle</LH>
```

3. To enter the first item of your list, type **<LI>** followed by the item. There is no need to include a closing </LI> tag.

```
<LI>Fasten your safety helmet in place.
```

4. Type **</OL>** to close the ordered list.

```
</OL>
```

3.3 How to Create Definition Lists

Definition lists are different from other lists in HTML, because each item in a definition list contains two parts: a term and a definition. Definition lists are typically used for glossaries and dictionaries. With a little creativity, however, they can be put to use in many different ways, such as product catalogs and even poetry. Definition lists are extremely flexible. The information contained in a <DD> tag is not limited to simple text. You can include images, tables, and full character formatting in your definitions.

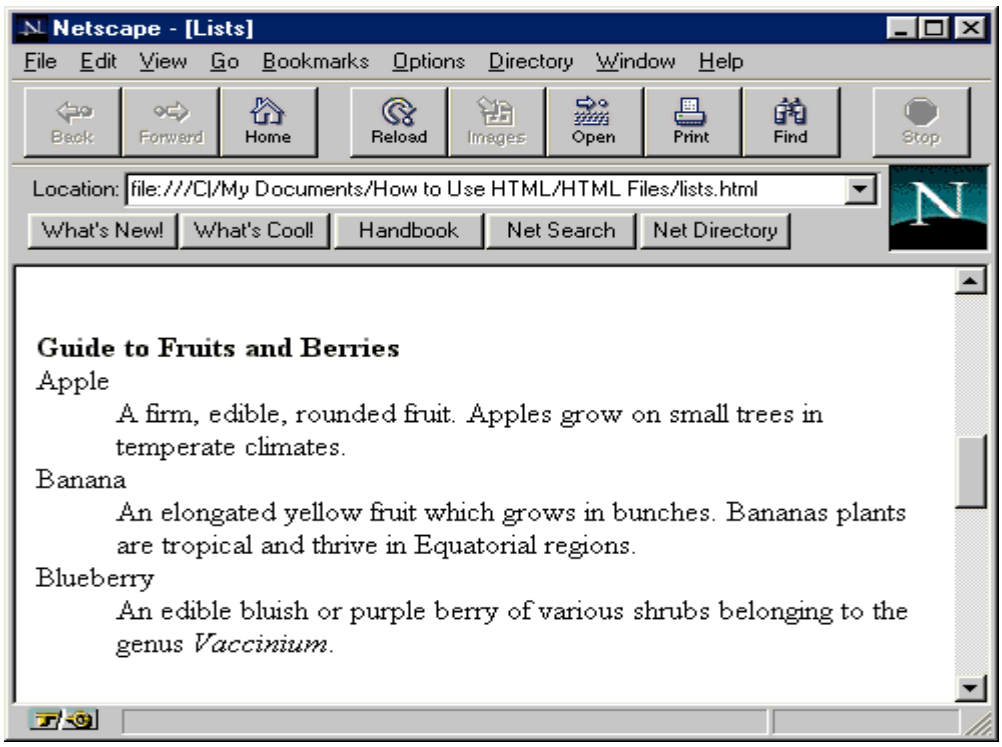


Figure 3 : A n Example of definition List in HTML

1. To create a definition list in your HTML document, type **<DL>** at the point where you'd like the list to begin.

```
<DL>
```

2. As mentioned earlier, definition lists are slightly different from ordered and unordered lists. Each item in a definition list is made up of two separate parts: the *term* and the *definition*. Typically, browsers will display the term on one line and the definition indented on the next line.

Apple
    A firm, edible, rounded fruit. Apples grow on small trees in temperate climates.
Banana
    An elongated yellow fruit which grows in bunches. Bananas plants are tropical and thrive in Equatorial regions.

3. To create a definition term, type **<DT>** followed by text describing the element being defined. For example, to begin a definition of the word *apple*, you would type **<DT>Apple**.

```
<DT>Apple
```

4. To create the definition, type **<DD>**, followed by the text of the definition. For example, to create a definition for the term in the previous step, you would type **<DD>a firm, edible, rounded fruit**.

```
<DD>A firm, edible, rounded fruit.
    Apples grow on small trees in
    temperate climates.
```

5. As with ordered and unordered lists, there are no closing tags for list items. Therefore, it is not necessary to type </DT> or </DD> at the end of your terms and definitions.
6. Type **</DL>** to close your definition list.

```
</DL>
```

3.4 How to Create Lists within Lists

In the beginning of this unit, we learned that lists are extremely flexible and powerful tools in HTML. Sometimes you will want to create lists within lists, especially when you need to create a hierarchy of items, such as in outlines or detailed instructions. Creating lists within lists is easy in HTML. It helps to keep your lists and list items indented in Notepad. Even though Web browsers will ignore the extra spaces, keeping everything organized this way will help you keep a handle on your HTML code. You can nest lists as many levels deep as you like. However, it is good practice to limit your nesting to three levels or less in order to make sure that the lists stay within the visible area of the reader's Web browser.

Begin the first list by typing <**OL**>. In this example, we're assuming that the first list is an ordered list, but in reality, it can be any type of list you want.

```
<OL>
```

1. Enter your list items one by one, beginning each item with <LI>.

```
<OL>
  <LI>Remove your LawnBird Flamingo
      from its attractive box.
  <LI>Assemble your tools. You'll
      need the following:
```

2. When you reach a step that requires a nested list, begin another list. The Web browser will automatically format this new list to fall underneath the current item in the first list. For example, to create a nested list under Step 2 in your original list, just type **<UL>**.

```
<OL>
  <LI>Remove your LawnBird Flamingo
      from its attractive box.
  <LI>Assemble your tools. You'll
      need the following:
       <UL>
```

3. Start entering items in your new list. When you're finished, close the new list by typing **</UL>**. You must close the new list before continuing to enter items in the original list.

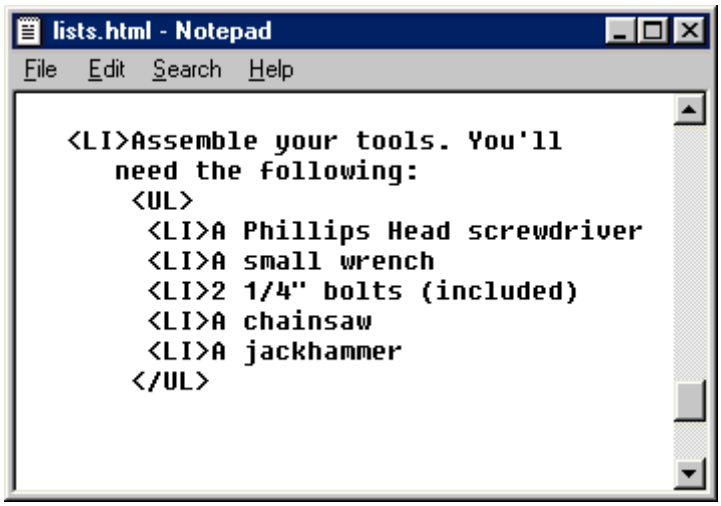


Figure 4: An Example of Lists within Lists in HTML

Enter the remaining items in the original list. Then press Enter and type **</OL>** when you're finished.

```
</OL>
```

4.0 CONCLUSION:

HTML has generally three types of lists namely Unordered lists, Ordered list and Definition list, these are sometimes called numbered lists. They are ideal vehicles for delivering all kinds of information online.

5.0 SUMMARY:

In this unit, you have learnt the following:

- Definitions of Unordered lists, Ordered list and Definition list
- Markup tags for Unordered lists, Ordered list and Definition list
- How to create Unordered lists, Ordered list and Definition list
- How to create Lists within Lists

6.0 Tutor Marked Assignment

- Define Definition list in HTML
- State the steps in creating an Ordered list
- State the steps in creating Lists within Lists

7.0 Further Reading and Other Resources

- Byers, T. J. (April 1987). "Built by association". *PC World* **5**: 244–251.
- Cicconi, Sergio (1999). "Hypertextuality". *Mediapolis. Ed. Sam Inkinen. Berlino & New York: De Gruyter.*: 21–43. http://www.cisenet.com/cisenet/writing/essays/hypertextuality.htm.
- Conklin, J. (1987). "Hypertext: An Introduction and Survey". *Computer* **20** (9): 17–41. doi:10.1109/MC.1987.1663693.
- Heim, Michael (1987). *Electric Language: A Philosophical Study of Word Processing*. New Haven: Yale University Press. ISBN 0-300-07746-7.
- Nelson, Theodor H. (September 1970). "No More Teachers' Dirty Looks". *Computer Decisions*. http://www.newmediareader.com/excerpts.html.
- Nelson, Theodor H. (1973). "A Conceptual framework for man-machine everything". *AFIPS Conference Proceedings VOL. 42*. pp. M22–M23.
- Nelson, Theodor H. (1992). *Literary Machines 93.1*. Sausalito CA: Mindful Press. ISBN 0-89347-062-7.
- van Dam, Andries (July 1988). "Hypertext: '87 keynote address". *Communications of the ACM* **31**: 887–895. doi:10.1145/48511.48519. http://www.cs.brown.edu/memex/HT_87_Keynote_Address.html.
- Yankelovich, Nicole; Landow, George P., and Cody, David (1987). "Creating hypermedia materials for English literature students". *SIGCUE Outlook* **20** (3)

MODULE 4: JAVA PROGRAMMING LANGUAGE

UNIT 1: OBJECT PROGRAMMING LANGUAGES

1.0 INTRODUCTION

Computers are used for solving problems quickly and accurately irrespective of the magnitude of the input. To solve a problem, a sequence of instructions is communicated to the computer. To communicate these instructions, *programming* languages are developed. The instructions written in a programming language comprise a *program*. A group of programs developed for certain specific purposes are referred to as *software* whereas the electronic components of a computer are referred to as *hardware*. Software activates the hardware of a computer to carry out the desired task. In a computer, hardware without software is similar to a body without a soul. Software can be system software or application software. *System software* is a collection of system programs. A *system program* is a program, which is designed to operate, control, and utilize the processing capabilities of the computer itself effectively. *System programming* is the activity of designing and implementing system programs. Almost all the operating systems come with a set of ready-to-use system programs: user management, file system management, and memory management. By composing programs it is possible to develop new, more complex, system programs. *Application software* is a collection of prewritten programs meant for specific applications.

Computer hardware can understand instructions only in the form of machine codes i.e. 0's and 1's. A programming language used to communicate with the hardware of a computer is known as *low-level language* or *machine language*. It is very difficult for humans to understand machine language programs because the instructions contain a sequence of 0's and 1's only. Also, it is difficult to identify errors in machine language programs. Moreover, low-level languages are machine-dependent. To overcome the difficulties of machine languages, *high-level languages* such as Basic, Fortran, Pascal, COBOL, and C were developed.

High-level languages allow some English-like words and mathematical expressions that facilitate better understanding of the logic involved in a program. While solving problems using high-level languages, importance was given to develop an *algorithm* (step-by-step instructions to solve a problem). While solving complex problems, a lot of difficulties were faced in the algorithmic approach. Hence, *object oriented programming languages* such as C++ and Java were evolved with a different approach to solve the problems. Object-oriented languages are also high-level languages with concepts of classes and objects that will be discussed in this unit.

2.0 OBJECTIVES

After learning the contents of this unit, the student would be able to:

- Define an Object Oriented Programming.
- Differentiate between a Class and an Object
- Enumerate five (5) features of an Object Oriented Programming Language

3.0 MAIN CONTENT

3.1 **Evolution of a New Paradigm**

The complexity of software required a change in the style of programming. It was aimed to:

- Produce reliable software
- Reduce production cost
- Develop reusable software modules
- Reduce maintenance cost
- Quicken the completion time of software development

The *Object-oriented model* was evolved for solving complex problems. It resulted in *object-oriented programming paradigms*. Object-oriented software development started in the 1980s. Object-oriented programming (OOP) seems to be effective in solving the complex problems faced by software industries. The end-users as well as the software professionals are benefited by OOP. OOP provides a consistent means of communication among analysts, designers, programmers, and end-users.

Object-oriented programming paradigm suggests new ways of thinking for finding a solution to a problem. Hence, the programmers should keep their minds tuned in such a manner that they are not to be blocked by their preconceptions experienced in other programming languages, such as structured programming. Proficiency in object-oriented programming requires talent, creativity, intelligence, logical thinking, and the ability to build and use abstractions and experience.

If procedures or functions are considered as verbs and data items are considered as nouns, a procedure oriented program is organized around verbs, while an object-oriented program is organized around nouns.

3.2. What is Object Oriented Programming (OOP)?

OOP is a type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an *object* that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can *inherit* characteristics from other objects.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

To perform object-oriented programming, one needs an *object-oriented programming language (OOPL)*. Simula was the first object-oriented programming language. Java, Python, C++, Visual Basic .NET and Ruby are the most popular OOP languages today. The Java programming language is designed especially for use in distributed applications on corporate networks and the Internet

3.3 The History of Object Oriented Programming

The basis for OOP started in the early 1960s. A breakthrough involving instances and objects was achieved at MIT with the PDP-1, and the first programming language to use objects was Simula 67. It was designed for the purpose of creating simulations, and was developed by Kristen Nygaard and Ole-Johan Dahl in Norway.

They were working on simulations that deal with exploding ships, and realized they could group the ships into different categories. Each ship type would have its own class, and the class would generate its unique behavior and data. Simula was not only responsible for introducing the concept of a class, but it also introduced the instance of a class.

The term "object oriented programming " was first used by Xerox PARC in their Smalltalk programming language. The term was used to refer to the process of using objects as the foundation for computation. The Smalltalk team was inspired by the Simula 67 project, but they designed Smalltalk so that it would be dynamic. The objects could be changed, created, or deleted, and this was different from the static systems that were commonly used. Smalltalk was also the first programming language to introduce the inheritance concept. It is this feature that allowed Smalltalk to surpass both Simula 67 and the analog programming systems. While these systems were advanced for their time, they did not use the inheritance concept.

Simula 67 was a groundbreaking system that has inspired a large number of other programming languages, and some of these include Pascal and Lisp. By the 1980s, object oriented programming had become prominent, and the primary factor in this is C++. Object oriented programming was also important for the development of Graphical user interfaces. The Cocoa structure that exists within Mac OS X is a good example of a dynamic GUI that works with an object oriented programming language. This paradigm of programming has also played an important role in the development of event-driven programming.

Niklaus Wirth and his associates were looking at areas such as modular programming and data abstraction, and they developed two systems which incorporated these elements. These two systems are Oberon and Modula-2. Oberon used a unique approach to classes and object orientation that is much different than C++ or Smalltalk. Since the introduction of OOP, a large number of modern programming languages are now using the concept. Some of these are Fortran, BASIC, and Pascal. There have been some compatibility issues, because many programs were not designed with a OOPs approach in mind. Object oriented programming languages that were "pure" did not have many of the functions that programmers needed.

To solve these problems, a number of researchers have been attempting to design new programming languages that used object oriented concepts but still retained many of the functions that programmers needed. One example of a programming language that has achieved this to some degree is Eiffel. Another programming language that has attempted to solve this problem is Java. Java has become popular because it uses a virtual machine, and it is very similar to C++ and C. The virtual machine is important because it allows code to be run on multiple platforms without having to be changed. Another system that is similar is Microsoft's .NET. Many developers now understand the importance of OOP, and are actively using it within their own programs. Many researchers have continued to make advancements by using the object oriented approach.

There are a number of other languages that have successfully combined the object oriented

approach with procedures that are useful to programmers. Python is one example, and Ruby uses a similar approach as well.

The use of an object oriented approach has led to advancements in modeling languages, design patterns, and a number of other areas. It is likely that OOP is a programming paradigm that will continue to evolve as we move forward into the future. It is a powerful language which has continued to improve over the years. It is the subject of debate within the programming community, as critics point out a number of problems with the structure. However, the popularity of programming languages such as Java demonstrate that it is a paradigm that is here to stay.

3.4 Features of Object-Oriented Programming

The fundamental features of object-oriented programming are as follows:

3.4.1 Classes and Objects

The concepts of object-oriented technology must be represented in object-oriented programming languages. Only then, complex problems can be solved in the same manner as they are solved in real-world situations.

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

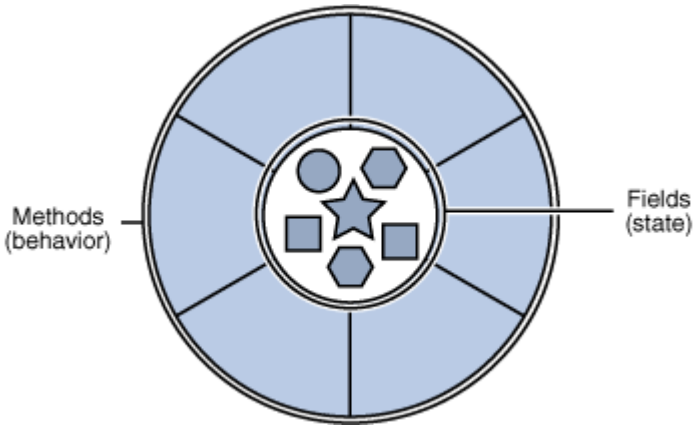


Figure 1: A software object.

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

Consider a bicycle, for example:

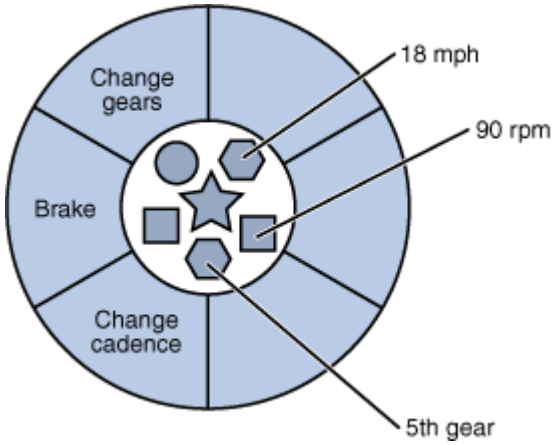


Figure 2: A bicycle modelled as a software object.

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

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

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

The class is a *prototype* or *blue print* or *model* that defines different features. A feature may be a data or an operation. Data are represented by *instance variables* or *data variables* in a class. The operations are also known as behaviors, or methods, or functions. They are represented by member functions of a class in C++ and methods in Java and C#.

A class is a data type and hence it cannot be directly manipulated. It describes a set of objects. For example,

apple is a fruit

implies that apple is an example of fruit. The term “fruit” is a type of food and apple is an instance of fruit. Likewise, a class is a type of data (data type) and object is an instance of class.

Similarly *car* represents a *class* (a model of vehicle) and there are a number of instances of car. Each instance of car is an object and the class car does not physically mean a car. An object is also known as *class variable* because it is created by the class data type. Actually, each object in an object-oriented system corresponds to a *real-world thing*, which may be a person, or a product, or an entity. The differences between class and object are given in Table 1.

Table 1. Comparison of Class and Object

CLASS	OBJECT
Class is a data type	Object is an instance of class data type.
It generates object.	It gives life to a class.
It is the prototype or model.	It is a container for storing its features. .
Does not occupy memory location.	It occupies memory location.
It cannot be manipulated because it is not available in the memory.	It can be manipulated

Instantiation of an object is defined as the process of creating an object of a particular class. An object has:

- States or properties
- Operations
- Identity

Properties maintain the internal state of an object. Operations provide the appropriate functionality to the object. Identity differentiates one object from the other. Object name is used to identify the object. Hence, object name itself is an identity. Sometimes, the object name is mixed with a property to differentiate two objects. For example, differentiation of two similar types of cars, say MARUTI 800 may be differentiated by colors. If colors are also same, the registration number is used. Unique identity is important and hence the property reflecting unique identity must be used in an object.

The properties of an object are important because the outcome of the functions depends on these properties. The functions control the properties of an object. They act and react to messages. The message may cause a change in the property of an object. Thus, the behavior of an object depends on the properties. For example, assume a property called brake condition for the class car. If the brake is not in working condition, guess the behavior of car. The outcome may be unexpected.

Similarly, in a student mark statement, the result() behavior depends on the data called marks. The property of resultStatus may be modified based on the marks.

How To Design A Class?

A class is designed with a specific goal. Its purpose must be clear to the users. An entity in solving a problem is categorized as a class if there is a need for more than one instance of this class. Also, it is very important to entrust a responsibility to an object. Presenting simply the behaviors such as reading data and displaying data in a class is a poor design of a class. To perform complex tasks, one class must jointly work with the other classes to perform the task. This approach is known as collaboration among classes. The class must be designed with essential attributes and behavior to reflect an idea in the real world.

The terms class and object are very important in object-oriented programming. A class is a prototype or blueprint or model that defies the variables and functions in it. The variables defined in a class represent the *data*, or *states*, or *properties*, or *attributes* of a visible thing of a certain type.

Classes are user-defined data types. It is possible to create a lot of objects of a class. The important advantages of classes are:

- Modularity
- Information hiding
- Reusability

3.4.2 Encapsulation

The process, or mechanism, by which you combine code and the data it manipulates into a single unit, is commonly referred to as *encapsulation*. Encapsulation provides a layer of

security around manipulated data, protecting it from external interference and misuse. In Java, this is supported by *classes* and *objects*.

3.4.3 Data Abstraction

Real-world objects are very complex and it is very difficult to capture the complete details. Hence, OOP uses the concepts of abstraction and encapsulation. Abstraction is a design technique that focuses on the essential attributes and behavior. It is a named collection of essential attributes and behavior relevant to programming a given entity for a specific problem domain, relative to the perspective of the user.

Closely related to encapsulation, data abstraction provides the ability to create user-defined data types. Data abstraction is the process of *abstracting* common features from objects and procedures, and creating a single interface to complete multiple tasks. For example, a programmer may note that a function that prints a document exists in many classes, and may *abstract* that function, creating a separate class that handles any kind of printing. Data abstraction also allows user-defined data types that, while having the properties of built-in data types, it also allows a set of permissible operators that may not be available in the initial data type. In Java, the *class* construct is used for creating user-defined data types, called Abstract Data Types (ADTs).

A good abstraction is characterized by the following properties:

1. *Meaningful way of naming* An abstraction must be named in a meaningful way. The name itself must refl ect the attributes and behaviors of the object for which the abstraction is made.
2. *Minimum features* An abstraction must have only essential attributes and behaviors, no more and no less.
3. *Complete details*
4. *Coherence*

An abstraction should define a related set of attributes and behavior to satisfy the requirement. Knowing the ISBN number of a book is irrelevant for a reader whereas for a librarian, it is very important for classification. Hence, the abstraction must be relevant to the given application.
Separation of interface and implementation is an abstraction mechanism in object-oriented programming language. Separation is useful in simplifying a complex system. It refers to distinguishing between a goal and a plan. It can be stated as separating “what” is to be done from “how” it is to be done. The separation may be well understood by the following equivalent terms, as shown in Table 2.

Table 2

WHAT:	HOW
Goals	Plans
Policy	Mechanism

Interface/requirement	Implementation

The implementation is hidden and it is important only for the developer. Separation in software design is an important concept for simplifying the development of software. Also, separation provides flexibility in the implementation. Several implementations are possible for the same interface. Sometimes, a single implementation can satisfy several interfaces.

Encapsulation is a process of hiding nonessential details of an object. It allows an object to supply only the requested information to another object and hides nonessential information. Since it packages data and methods of an object, an implicit protection from external tampering prevails. However, an entire application cannot be hidden. A part of the application needs to be accessed by users to use an application. Abstraction is used to provide access to a specific part of an application. It provides access to a specific part of data, while encapsulation hides data.

Rendering abstraction in software is an implicit goal of programming. Object-oriented programming languages permit abstractions to be represented more easily and explicitly. Object-oriented programming languages use classes and objects for representing abstractions. A class defines the specific structure of a given abstraction. It has a unique name that conveys the meaning of the abstraction. Class definition defines the common structure once. It allows "reuse" when creating new objects of the defined structure. An object's properties are exactly those described by its class. Two main parts of an object are:

- *Interface*: The user's view of the operations performed by an object is known as the interface part of that object.
- *Implementation*: The implementation of an object describes how the entrusted responsibility in the interface is achieved.

It is important to observe abstraction from the perspective of the user. Software is developed for endusers. Hence, the abstraction is captured from the user's point of view. For the same reason abstraction varies from viewer to viewer. For example, a book abstraction viewed by a librarian is different from the abstraction viewed by a reader of the book. A librarian may consider the following features:

Attributes	**Functions**
title	printBook( )
author	getDetails( )
publisher	sortTitle( )
cost	sortAuthor( )
accNumber	
ISBNnumber	

A reader may consider the following features:

Attributes	**Functions**
title	bookDetails()
author	availability()
content	tokenDetails()
examples	
exercises	
index	

Here, the attributes are data and the functions are operations or behaviors related to data. If an application software is to be developed for a library, the abstraction captured by the librarian is important. The reader's point of view is not necessary. Thus abstraction differs from viewer to viewer. Abstraction relative to the perspective of the user is very important in software development.

A simple view of an object is a combination of properties and behavior. The method name with arguments represents the interface of an object. The interface is used to interact with the outside world. Object-oriented programming is a packaging technology. Objects encapsulate data and behavior hiding the details of implementation. The concept of implementation hiding is also known as *information hiding*. Since data is important, the users cannot access this data directly. Only the interfaces (methods) can access or modify the encapsulated data. Thus, *data hiding* is also achieved. The restriction of access to data within an object to only those methods defined by the object's class is known as encapsulation. Also, implementation is independently done improving software reuse concept. Interface encapsulates knowledge about the object. Encapsulation is an abstract concept. Table 3 gives a clear picture about the different concepts.

Table 3: Comparison of Abstraction and Encapsulation

Abstraction	Encapsulation
Abstraction separates interface and implementation.	Encapsulation groups related concepts into one item.
User knows only the interfaces of the object and how to use them according to abstraction. Thus, it provides access to a specific part of data.	Encapsulation hides data and the user cannot access the same directly (data hiding).
Abstraction gives the coherent picture of what the user wants to know. The degree of relatedness of an encapsulated unit is defined as cohesion. High cohesion is achieved by means of good abstraction	Coupling means dependency. Good systems have low coupling. Encapsulation results in lesser dependencies of one object on other objects in a system that has low coupling. Low coupling may be achieved by designing a good encapsulation.
Abstraction is defined as a data type called class which separates interface from implementation.	Encapsulation packages data and functionality and hides the implementation details (information hiding).
The ability to encapsulate and isolate design from execution information is known as abstraction.	Encapsulation is a concept embedded in abstraction.

Classes and objects represent abstractions in OOP languages. Class is a common representation with definite attributes and operations having a unique name. Class can be viewed as a user-defined data type.
Data types cannot be used in a program for direct manipulation. A variable of a particular data type is defined first as a container for storage. The variables are manipulated after holding data in them. For example,

int year, mark ;

is a declaration of variables in C. This statement conveys to the compiler that *year* and *mark* are instances of integer data type. Likewise, in OOP, a class is a data type. A variable of a class data type is known as an object. An object is defined as an instance of a class. For example, if *Book* is a defined class,

Book cBook, javaBook ;

declares the variables cBook and javaBook of the Book class type. Thus, classes are software prototypes for objects. Creation of a class variable or an object is known as *instantiation* (creation of an instance of a class). The objects must be allocated in memory. Classes cannot be allocated in memory.

3.4.4 Inheritance

Inheritance allows the extension and reuse of existing code, without having to repeat or rewrite the code from scratch. Inheritance involves the creation of new classes, also called *derived* classes, from existing classes (*base* classes). Allowing the creation of new classes enables the existence of a hierarchy of classes that simulates the class and subclass concept of the real world. The new derived class inherits the members of the base class and also adds its own. For example, a banking system would expect to have customers, of which we keep information such as name, address, etc. A subclass of customer could be customers who are students, where not only we keep their name and address, but we also track the educational institution they are enrolled in.

Inheritance is mostly useful for two programming strategies: extension and specialization. Extension uses inheritance to develop new classes from existing ones by adding new features. Specialization makes use of inheritance to refine the behavior of a general class.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, Bicycle now becomes the *superclass* of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:

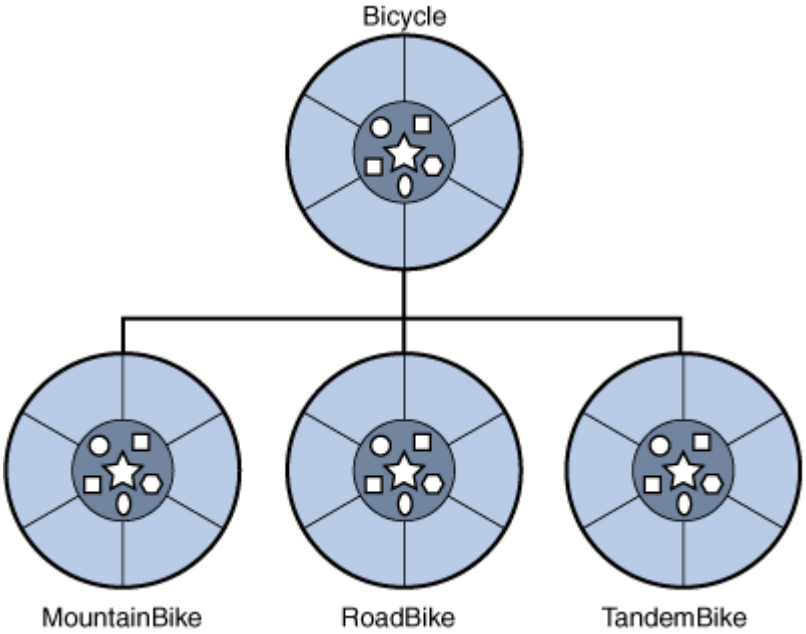


Figure 3: A hierarchy of bicycle classes.

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {

    // new fields and methods defining a mountain bike would go here

}
```

This gives MountainBike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

3.4.4.1 Multiple Inheritance

When a class is derived through inheriting one or more base classes, it is being supported by *multiple inheritance*. Instances of classes using multiple inheritance have instance variables for each of the inherited base classes. Java does not support multiple inheritance. However, Java allows any class to implement multiple interfaces, which to some extent appears like a realization of multiple inheritances.

3.4.5 What Is an Interface?

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface Bicycle {

      void changeCadence(int newValue);   // wheel revolutions per minute

      void changeGear(int newValue);

      void speedUp(int increment);

      void applyBrakes(int decrement);
}
```

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {

   // remainder of this class implemented as before

}
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

3.4.6 What Is a Package?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or

thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a String object contains state and behavior for character strings; a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

The Java Platform API Specification contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java Platform 6, Standard Edition. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

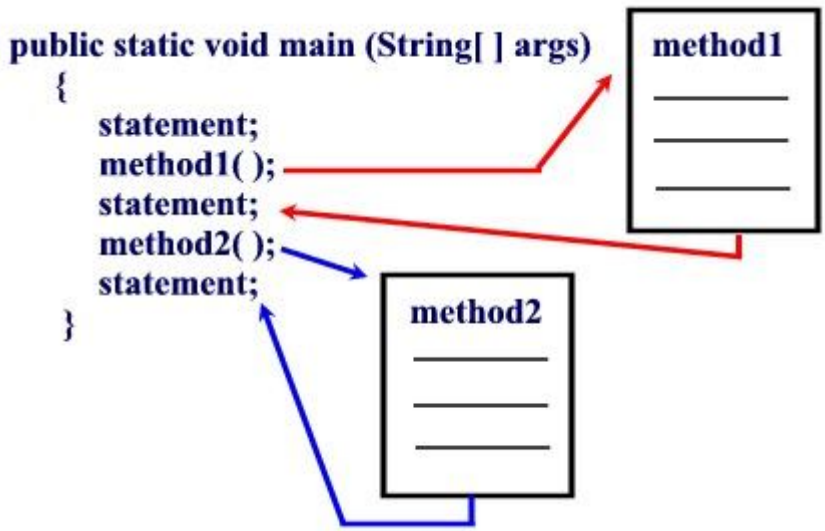
3.4.7. Polymorphism

Polymorphism allows an object to be processed differently by data types and/or data classes. More precisely, it is the ability for different objects to respond to the same message in different ways. It allows a single name or operator to be associated with different operations, depending on the type of data it has passed, and gives the ability to redefine a *method* within a *derived class*. For example, given the *student* and *business* subclasses of *customer* in a banking system, a programmer would be able to define different *getInterestRate()* methods in *student* and *business* to override the default interest *getInterestRate()* that is held in the *customer* class. While Java supports method overloading, it does not support operator overloading.

3.4.8 What are Methods?

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name. Think of a method as a subprogram that acts on data and often returns a value.

Each method has its own name. When that name is encountered in a program, the execution of the program branches to the body of that method. When the method is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.



Good programmers write in a modular fashion which allows for several programmers to work independently on separate concepts which can be assembled at a later date to create the entire project. The use of methods will be our first step in the direction of modular programming.

Methods are time savers, in that they allow for the repetition of sections of code without retyping the code. In addition, methods can be saved and utilized again and again in newly developed programs.

You are using methods when you use System.out.print( ) and System.out.println( ).

There are two basic types of methods:

Built-in: Build-in methods are part of the compiler package, such as System.out.println( ) and System.exit(0).
User-defined: User-defined methods are created by you, the programmer. These methods take-on names that you assign to them and perform tasks that you create.

How to invoke (call) a method (method invocation):

When a method is invoked (called), a request is made to perform some action, such as setting a value, printing statements, returning an answer, etc. The code to invoke the method contains the name of the method to be executed and any needed data that the receiving method requires. The required data for a method are specified in the method's parameter list.

Consider this method;

int number = Console.readInt("Enter a number"); //returns a value

The method name is "readInt" which is defined in the class "Console". Since the method is defined in the class Console, the word Console becomes the calling object. This particular method returns an integer value which is assigned to an integer variable named number.

You invoke (call) a method by writing down the calling object followed by a dot, then the name of the method, and finally a set of parentheses that may (or may not) have information for the method.

3.4.9 Delegation

Delegation is an alternative to class inheritance. Delegation allows an object composition to be as powerful as inheritance. In delegation, two objects are involved in handling a request: methods can be delegated by one object to another, but the *receiver* stays bound to the object doing the delegating, rather than the object being delegated to. This is analogous to child classes sending requests to parent classes. In Java, delegation is supported as more of a *message forwarding* concept.

3.4.10 Genericity

Genericity is a technique for defining software components that have more than one interpretation depending on the data type of parameters. Thus, it allows the abstraction of data items without specifying their exact type. These unknown (generic) data types are resolved at the time of their usage (e.g., through a function call), and are based on the data type of parameters. For example, a *sort* function can be parameterized by the type of elements it sorts. To invoke the parameterized *sort()*, just supply the required data type parameters to it and the compiler will take care of issues such as creation of actual functions and invoking that transparently.

3.4.11 Persistence

Persistence is the concept by which an object (a set of data) outlives the life of the program, existing between executions. All database systems support persistence, but, persistence is not supported in Java. However, persistence can be simulated through use of *file streams* that are stored on the file system.

3.4.12 Concurrency

Concurrency is represented in Java through threading, synchronization, and scheduling. Using concurrency allows additional complexity to the development of applications, allowing more flexibility in software applications.

3.4. 13 Events

An event can be considered a kind of interrupt: they interrupt your program and allow your program to respond appropriately. In a conventional, non object-oriented language, processing proceeds literally through the code: code is executed in a ‘top-down’ manner. The flow of code in a conventional language can only be interrupted by loops, functions, or iterative conditional statements. In an object-oriented language such as Java, events interrupt the normal flow of program execution. Objects can pass information and control from themselves to another object, which in turn can pass control to other objects, and so on. In Java, events are handled through the *EventHandler* class, which supports dynamically

generated listeners. Java also implements event functionality in classes such as the *Error* subclass: abnormal conditions are *caught* and *thrown* so they can be handled appropriately.

3.5 Design Strategies in OOP

Object-oriented programming includes a number of powerful design strategies based on software engineering principles. Design strategies allow the programmers to develop complex systems in a manageable form. They have been evolved out of decades of software engineering experience. The basic design strategies embedded in object-oriented programming are:

i. Abstraction
ii. Composition
iii. Generalization

The existing object-oriented programming languages support most of these features.

3.5.1 Abstraction

The abstract view of solving a problem is an essential requirement as we do in a real-world problem. Consider the previous example of the situation in an office. The manager passes the information about the place of destination to the driver who performs the action of moving from the office to the desired site. The manager must know the person who is capable of doing this task even though he may not know driving. The driver takes care of the execution part of driving. In the perspective of the manager, the driver is an employee who knows driving and can take him to the desired place. This abstract information about the driver is enough for the manager. The manager is an officer employed in the office. For the driver the details of the officer like name and designation are enough. This is the abstract information about the officer. The driver uses a car to perform the task. In the perspective of the manager, the type of car such as A/C or non-A/C and brand name may be important. Thus this differs from individual to individual.

The essential features of an entity are known as *abstraction.* A feature may be either an attribute reflecting a property (or state or data) or an operation reflecting a method (or behavior or function). The features such as things in the trunk of a car, the medical history of the manager travelling in the car, and the working mechanism of the car engine are not necessary for the driver. The essential features of an entity in the perspective of the user define abstraction. A good abstraction is achieved by having

- meaningful name such as driver reflecting the function
- minimum and at the same time complete features
- coherent features

Abstraction specifies necessary and sufficient descriptions rather than implementation details. It results in separation of interface and implementation.

3.5.2 Composition

A complex system is organized using a number of simpler systems. An organized collection of smaller components interacting to achieve a coherent and common behavior is known as composition. There are two types of composition:

1. Association
2. Aggregation

Aggregation considers the composed part as a single unit whereas association considers each part of composition as a separate unit. For example, a computer is an association of CPU, keyboard, and monitor. Each part is visible and manipulated by the user. CPU is an aggregation of processor memory and control unit. The individual parts are not visible and they cannot be manipulated by the user. Both types of and cannot be altered at runtime. Association offers greater flexibility because the relationships among the visible units can be redefined at run time. It adapts to changing conditions in its execution environment by replacing one or more of its components. The two types of composition are frequently used together. A computer is an example for combination of both association and aggregation.

3.5.3 Generalization

Generalization identifies the common properties and behaviors of abstractions. It is different from abstraction. Abstraction is aimed at simplifying the description of an entity, whereas generalization identifies commonalities among a set of abstractions. Generalizations are important since they are like “laws” or “theorems”, which lay the foundation for many things. Generalization helps to develop software capturing the idea of similarity.

The different types of generalization are:

1. Hierarchy
2. Genericity
3. Polymorphism
4. pattern

1. ***Hierarchy*** The first type of generalization uses a tree-structured form to organize commonalities. A generalization/specialization hierarchy is achieved with the help of inheritance in object-oriented programming languages. The advantages are:

- Knowledge representation in a particular form.
- The intermediate levels in the hierarchy provide the names that can be used among developers and between developers and application domain experts.
- A new specialization at any level can be extended.
- New attributes and behavior can be easily added.

2. ***Genericity*** It refers to a generic class, which is meant for accepting different types of parameters. A stack class can be considered as a generic class if it is capable of accepting integer data as well as fl oat or double or char data also. This type of generalization is known as genericity.

3. ***Polymorphism*** The term *poly* means *many* and the term *morph* means to *form*. Then

polymorphism concerns the possibility for a single property of exposing multiple possible states. The generally accepted definition for this term in object oriented programming is the capability of objects belonging to the same class hierarchy to react differently to the same method call. This means that a function may be defined in different forms with the same function name. It is possible to implement different functionalities using a common name for a function. Polymorphism provides a way of generalizing algorithms. Late binding or dynamic binding is required to implement polymorphism in object-oriented programming. Based on the parameters passed, the compiler dynamically identifies the function to be invoked and it is known as dynamic binding.

4. Pattern A pattern is a generalization of a solution for a common problem. An architecture or model is a large scale pattern used in computer science. Client-server model is an example of a large-scale pattern.
A pattern is a distinct form of generalization. It gives a general form of solution. A pattern need not be expressed in code at all. The elements of the pattern are represented by classes. The relationships among the elements may be defined by association, aggregation, and/or hierarchy.

3.6 Examples of Object-Oriented Programming Languages

Several object-oriented programming languages have been invented since 1960. Some well-known among them are, C++, Java, and C# are the three most commercially successful OOP languages.

Simula : Simula was the first object-oriented language with syntax similar to Algol. Concurrent processes are managed by scheduler class. This language is best suited to the simulation of parallel systems. It allows classes with attributes and procedures that are public by default. It is also possible to declare them as private. Inheritance and virtual functions are supported. Memory is managed automatically with garbage collection.

Ada Ada was developed by Jean Ichbiah and his team at Bull in the late 1970s. It was named after Augusta Ada, daughter of Byron, the famous romantic poet. It is a general-purpose language. An abstract type is implemented as a package in Ada. Each package can contain abstract types. The concept of genericity is introduced at the level of types and packages.

Smalltalk It was designed by Alan Kay at Xerox PARC during the 1970s. It is a general purpose language. It allows polymorphism. Automatic garbage collection is provided. Generalization of the object concept is another original contribution from Smalltalk.

C++
C++ was designed by Bjarne Stroustrup in the AT and T Bell Laboratories in the early 1980s. It borrowed the concepts of class, subclass, inheritance and polymorphism from Simula. The name C++ was coined by Rick Mascitti in 1983.

Objective C It is a general-purpose language designed by B. Cox. It extends C with an object model based on Smalltalk 80. It does not support metaclasses, which are classes used to describe other classes. Simple inheritance is supported. There are generic classes and no memory garbage collector.

Object Pascal It is an extension of Pascal, developed by Apple for the Macintosh in the early 1980s. Simple inheritance dynamic binding is supported. There is no automatic garbage collection.

Eiffel It was developed by Bertrand Meyer in 1992 for both scientific and commercial applications. Exception management is a feature supported by this language.

Java It is a pure object-oriented language developed by Arnold and Gosling in 1996. It helps in developing small applications called applets which can be integrated into web pages. It supports multithreading. It supports encapsulation, inheritance, polymorphism, genericity, and dynamic binding.

C# It is an object-oriented programming language developed by Microsoft Corporation for its new .NET Framework. It is derived from C and C++; appears very similar to Java. It supports encapsulation, inheritance, polimorphysm, genericity, and late binding.

3.7 Requirements of Using OOP Approach

The method of solving complex problems using OOP approach requires:

- Change in mindset of programmers, who are familiar with structured programming.
- Closer interaction between program developers and end-users.
- Much concentration on requirement, analysis, and design.
- More attention for system development than just programming.
- Intensive testing procedures.

3.8 Advantages of Object-Oriented Programming

The following are the advantages of software developed using object-oriented programming:

1. Software reuse is enhanced.
2. Software maintenance cost can be reduced.
3. Data access is restricted providing better data security.
4. Software is easily developed for complex problems.
5. Software may be developed meeting the requirements on time, on the estimated budget.
6. Software has improved performance.
7. Software quality is improved.

8. Class hierarchies are helpful in the design process allowing increased extensibility.

9. Modularity is achieved.

10. Data abstraction is possible.

3.9 Limitations of Object-Oriented Programming

1. The benefits of OOP may be realized after a long period.

2. Requires intensive testing procedures.

3. Solving a problem using OOP approach consumes more time than the time taken by structured programming approach.

3.10 Applications of Object-Oriented Programming

If there is complexity in software development, object-oriented programming is the best paradigm to solve the problem. The following areas make use of OOP:

1. Image processing

2. Pattern recognition

3. Computer assisted concurrent engineering

4. Computer aided design and manufacturing

5. Computer aided teaching

6. Intelligent systems

7. Data base management systems

8. Web based applications

9. Distributed computing and applications

10. Component based applications

11. Business process reengineering

12. Enterprise resource planning

13. Data security and management

14. Mobile computing

15. Data warehousing and data mining

16. Parallel computing

Object concept helps to translate our thoughts to a program. It provides a way of solving a problem in the same way as a human being perceives a real world problem and finds out the solution. It is possible to construct large reusable components using object-oriented techniques. Development of reusable components is rapidly growing in commercial software industries.

4.0 CONCLUSION

Object Oriented Programming (OOP) is to mean any kind of programming that uses a programming language with some object oriented constructs or programming in an environment where some object oriented principles are followed.

5.0 SUMMARY

In this unit, you have learnt the following:

- Evolution of a new Paradigm
- Features of Object-Oriented Programming
- Design Strategies in OOP
- Object-Oriented Programming Languages
- Requirements of Using OOP Approach
- Advantages of Object-Oriented Programming
- Limitations of Object-Oriented Programming
- Applications of Object-Oriented Programming

6.0 TUTOR MARKED ASSIGNMENT

- Define Object Oriented Programming Language in your own words
- List three (3) advantages of OOP.
- List three (3) features of OOP
- Enumerate three (3) places where OOP can be applied

7.0 FURTHER READING AND OTHER RESOURCES

- Cargill, Tom A.: *PI: A Case Study in ObjectOriented Programming*. SIGPLAN Notices, November 1986, pp 350360.
- Kerr, Ron: *ObjectBased Programming: A Foundation for Reliable Software.* Proceedings of the 14th SIMULA Users' Conference. August 1986, pp 159165. An abbreviated version of this paper can be found under the title *A Materialistic View of the Software ''Engineering'' Analogy* in SIGPLAN Notices, March 1987, pp 123125.
- Nygaard, Kristen: *Basic Concepts in Object Oriented Programming*. SIGPLAN Notices, October 1986, pp 128132.

.

- Snyder, Alan: *Encapsulation and Inheritance in ObjectOriented Programming Languages*. SIGPLAN Notices, November 1986, pp 3845.

.UNIT 2: JAVA LANGUAGE

1.0 INTRODUCTION

If you do not have any object-oriented (OO) programming background, don't worry; this unit does not assume any prior experience. If you do have experience with OO programming, however, be careful. The term "object-oriented" has different meanings in different languages. Don't assume that Java works the same way as your favorite OO language. This is particularly true for C++ programmers.

Java is a general-purpose, object-oriented language that looks a lot like C and C++. Its design, however, was built on those of its predecessors, making it easier, safer, and more productive than C++. While Java started out as a niche language for developing applets or small programs that run in Web browsers, it has evolved to arguably become the most important programming language for developing ecommerce and other Web-driven applications. Its area of use is growing daily and includes dynamic Web-content generation with server technology, the building of business components with Enterprise JavaBeans, the creation of cross-platform user interfaces with Swing, and much more.

3.0 OBJECTIVES

After learning the contents of this unit, the student should be able to:

- Define Java
- Enumerate the uses of Java
- Describe the History of Java

- Describe Standalone type of Java

3.1 What is Java?

Java is a computer programming language. It enables programmers to write computer instructions using English based commands, instead of having to write in numeric codes. It's known as a "high-level" language because it can be read and written easily by humans. Like English, Java has a set of rules that determine how the instructions are written. These rules are known as its "syntax". Once a program has been written, the high-level instructions are translated into numeric codes that computers can understand and execute.

Java is a high-level object-oriented programming language. Though it is associated with the World Wide Web but it is older than the origin of Web. It was only developed keeping in mind the consumer electronics and communication equipments. It came into existence as a part of web application, web services and a platform independent programming language in the 1990s.

Java is a very dynamic language that has thousands of applications online and offline. It is currently supported by almost every computer user online today.

Java is a programming language used for a variety of applications, but usually simple, lightweight programs used with the Internet.

In discussing Java, it is important to distinguish between the Java programming language, the Java Virtual Machine, and the Java platform. The Java programming language is the language in which Java applications (including applets, servlets, and JavaBeans components) are written. When a Java program is compiled, it is converted to byte codes that are the portable machine language of a CPU architecture known as the Java Virtual Machine (also called the Java VM or JVM). The JVM can be implemented directly in hardware, but it is usually implemented in the form of a software program that interprets and executes byte codes.

The Java platform is distinct from both the Java language and Java VM. The Java platform is the predefined set of Java classes that exist on every Java installation; these classes are available for use by all Java programs. The Java platform is also sometimes referred to as the Java runtime environment or the core Java APIs (application programming interfaces). The Java platform can be extended with optional standard extensions. These extension APIs exist in some Java installations, but are not guaranteed to exist in all installations.

3.1.1. The Java Programming Language

The Java programming language is a state-of-the-art, object-oriented language that has a syntax similar to that of C. The language designers strove to make the Java language powerful, but, at the same time, they tried to avoid the overly complex features that have

bogged down other object-oriented languages, such as C++. By keeping the language simple, the designers also made it easier for programmers to write robust, bug-free code. As a result of its elegant design and next-generation features, the Java language has proved wildly popular with programmers, who typically find it a pleasure to work with Java after struggling with more difficult, less powerful languages.

3.1.2. The Java Virtual Machine

The Java Virtual Machine, or Java interpreter, is the crucial piece of every Java installation. By design, Java programs are portable, but they are only portable to platforms to which a Java interpreter has been ported. Sun ships VM implementations for its own Solaris operating system and for Microsoft Windows (95/98/NT) platforms. Many other vendors, including Apple and various Unix vendors, provide Java interpreters for their platforms. There is a freely available port of Sun's VM for Linux platforms, and there are also other third-party VM implementations available. The Java VM is not only for desktop systems, however. It has been ported to set-top boxes, and versions are even available for hand-held devices that run Windows CE and PalmOS.

Although interpreters are not typically considered high-performance systems, Java VM performance is remarkably good and has been improving steadily. Of particular note is a VM technology called just-in-time (JIT) compilation, whereby Java byte codes are converted on-the-fly into native-platform machine language, boosting execution speed for code that is run repeatedly. Sun's new Hotspot technology is a particularly good implementation of JIT compilation.

3.1.3. The Java Platform

The Java platform is just as important as the Java programming language and the Java Virtual Machine. All programs written in the Java language rely on the set of predefined classes that comprise the Java platform. Java classes are organized into related groups known as *packages*. The Java platform defines packages for functionality such as input/output, networking, graphics, user-interface creation, security, and much more.

A *class* is a module of Java code that defines a data structure and a set of methods (also called procedures, functions, or subroutines) that operate on that data.

The Java 1.2 release was a major milestone for the Java platform. This release almost tripled the number of classes in the platform and introduced significant new functionality. In recognition of this, Sun named the new version the Java 2 Platform. This is a trademarked name created for marketing purposes; it serves to emphasize how much Java has grown since its first release. However, most programmers refer to the Java platform by its official version number, which, at the time of this writing, is 1.2.

Although there is currently a beta release of Java 1.3 available

It is important to understand what is meant by the term platform. To a computer programmer, a platform is defined by the APIs he or she can rely on when writing programs. These APIs are usually defined by the operating system of the target computer. Thus, a programmer writing a program to run under Microsoft Windows must use a different set of APIs than a

programmer writing the same program for the Macintosh or for a Unix-based system. In this respect, Windows, Macintosh, and Unix are three distinct platforms.

Java is not an operating system. Nevertheless, the Java platform--particularly the Java 2 Platform--provides APIs with a comparable breadth and depth to those defined by an operating system. With the Java 2 Platform, you can write applications in Java without sacrificing the advanced features available to programmers writing native applications targeted at a particular underlying operating system. An application written on the Java platform runs on any operating system that supports the Java platform. This means you do not have to create distinct Windows, Macintosh, and Unix versions of your programs, for example. A single Java program runs on all these operating systems, which explains why "Write once, run anywhere" is Sun's motto for Java.

There is a Java-based operating system, however; it is known as JavaOS.

It also explains why companies like Microsoft might feel threatened by Java. The Java platform is not an operating system, but for programmers, it is an alternative development target and a very popular one at that. The Java platform reduces programmers' reliance on the underlying operating system, and, by allowing programs to run on top of any operating system, it increases end users' freedom to choose an operating system.

3.1.4. Versions of Java

As of this writing, there have been four major versions of Java. They are:

Java 1.0

> This was the first public version of Java. It contained 212 classes organized in 8 packages. There is a large installed base of web browsers that run this version of Java, so this version is still in use for writing simple applets--Java programs that are included in web pages.

Java 1.1

> This release of Java doubled the size of the Java platform to 504 classes in 23 packages. It introduced inner classes, an important change to the Java language itself, and included significant performance improvements in the Java VM. This version of Java is out of date, but is still in use on systems that do not yet have a stable port of Java 1.2.

Java 1.2

> This is the latest and greatest significant release of Java; it tripled the size of the Java platform to 1520 classes in 59 packages. Because of the many new features included in this release, the platform was renamed and is now called the Java 2 Platform.

Java 1.3 (beta)

> This release includes minor corrections and updates to the Java platform, but does not include major changes or significant new functionality.

In addition, Sun has instituted a process for proposing and developing standard extensions to the Java platform. In the future, most new functionality is expected to take the form of a standard extension, rather than be a required part of every Java installation.

In order to work with Java 1.0 or Java 1.1, you have to obtain the Java Development Kit (JDK) for that release. As of Java 1.2, the JDK has been renamed and is now called a Software Development Kit (SDK), so we have the Java 2 SDK or, more precisely, the Java 2 SDK, Standard Edition, Version 1.2 (or Version 1.3 beta). Despite the new name, many programmers still refer to the development kit as the JDK.

Don't confuse the JDK (or SDK) with the Java Runtime Environment (JRE). The JRE contains everything you need to run Java programs, but does not contain the tools you need to develop Java programs (i.e., the compiler). You should also be aware of the Java Plug-in, a version of the Java 1.2 (and 1.3) JRE that is designed to be integrated into the Netscape Navigator and Microsoft Internet Explorer web browsers.

In addition to evolving the Java platform over time, Sun is also trying to produce different versions of the platform for different uses. The Standard Edition is the only version currently available, but Sun is also working on the Java 2 Platform, Enterprise Edition (J2EE), for enterprise developers and the Java 2 Platform, Micro Edition, for consumer electronic systems, like handheld PDAs and cellular telephones.

3.2 History of Java

In 1991, a group of Sun Microsystems engineers led by James Gosling decided to develop a language for consumer devices (cable boxes, *etc.*). They wanted the language to be small and use efficient code since these devices do not have powerful CPUs. They also wanted the language to be hardware independent since different manufacturers would use different CPUs. The project was code-named *Green*.

These conditions led them to decide to compile the code to an intermediate machine-like code for an imaginary CPU called a *virtual machine*. (Actually, there is a real CPU that implements this virtual CPU now.) This intermediate code (called *bytecode*) is completely hardware independent. Programs are run by an interpreter that converts the bytecode to the appropriate native machine code.
Thus, once the interpreter has been ported to a computer, it can run any bytecoded program.

Sun uses UNIX for their computers, so the developers based their new language on C++. They picked C++ and not C because they wanted the language to be *object-oriented*. The original name of the language was *Oak*. However, they soon discovered that there was already a programming language called Oak, so they changed the name to *Java*.

The Green project had a lot of trouble getting others interested in Java for smart devices. It was not until they decided to shift gears and market Java as a language for web applications that interest in Java took off. Many of the advantages that Java has for smart devices are even bigger advantages on the web.

Java is influenced by C, C++, Smalltalk and borrowed some advanced features from some other languages. The company promoted this software product with a slogan named “Write Once Run Anywhere” that means it can develop and run on any device equipped with Java Virtual Machine (JVM). This language is applicable in all kinds of operating systems including Linux, Windows, Solaris, and HP-UX etc.

3.3 Why Choose Java?

Java was designed with a few key principles in mind:

- **Easy to Use:** The fundamentals of Java came from a programming language called c++. Although c++ is a powerful language, it was felt to be too complex in its syntax, and inadequate for all of Java's requirements. Java built on, and improved the ideas of c++, to provide a programming language that was powerful and simple to use.
- **Reliability:** Java needed to reduce the likelihood of fatal errors from programmer mistakes. With this in mind, object-oriented programming was introduced. Once data and its manipulation were packaged together in one place, it increased Java’s robustness.
- **Secure:** As Java was originally targeting mobile devices that would be exchanging data over networks, it was built to include a high level of security. Java is probably the most secure programming language to date.
- **Platform Independent:** Programs needed to work regardless of the machine they were being executed on. Java was written to be a portable language that doesn't care about the operating system or the hardware of the computer.

The team at Sun Microsystems were successful in combining these key principles, and Java's popularity can be traced to it being a robust, secure, easy to use, and portable language.

3.4 What Is Java Used For

Java is used to, chat with friends and family, or view certain images. It is a free program that is available for download through the official Java website.

Java allows you to play online games, chat with people around the world, calculate your mortgage interest, and view images in 3D. These applications, written in the Java programming language and accessible from your browser, are called "applets". Corporations also use Java applets for intranet applications and other e-business solutions

Java allows people a large amount of interactivity online. It is used for messaging programs, online calculators, converters, and much more. Social networking sites also rely heavily on the Java programming language.

Even the United States Postal Service website uses Java, their online postage pricing forms that calculate the cost of sending parcels worldwide is a Java application.

This online automation, which would have meant a trip to the post office to get a parcel priced, is time and cost effective for both the company and their customers!

Java has thousands of online applications and is used extensively with great efficiency online

to automate a great number of tasks.

3.5 Basics of Java

Applications

There are 2 basic types of Java applications:

- Standalone

These run as a normal program on the computer. They may be a simple console application or a windowed application. These programs have the same capabilities of any program on the system. For example, they may read and write files. Just as for other languages, it is easily to write a Java console program than a windowed program. So despite the leanings of the majority of Java books, the place to start Java programming is a standalone console program, *not* an applet!

- Applets

A **Java applet** is an applet delivered to users in the form of Java bytecode. Java applets can run in a Web browser using a Java Virtual Machine (JVM), or in Sun's AppletViewer, a stand-alone tool for testing applets. Java applets were introduced in the first version of the Java language in 1995, and are written in programming languages that compile to Java bytecode, usually in Java, but also in other languages such as Jython, JRuby, or Eiffel (via SmartEiffel).

Java applets run at speeds comparable to, but generally slower than, other compiled languages such as C++, but until approximately 2011 many times faster than JavaScript. In addition they can use 3D hardware acceleration that is available from Java. This makes applets well suited for non trivial, computation intensive visualizations. When browsers have gained support for native hardware accelerated graphics in the form of Canvas and WebGL, as well as Just in Time compiled JavaScript, the speed difference has become less noticeable.

Since Java's bytecode is cross-platform or platform independent, Java applets can be executed by browsers for many platforms, including Microsoft Windows, Unix, Mac OS and Linux. It is also trivial to run a Java applet as an application with very little extra code. This has the advantage of running a Java applet in offline mode without the need for any Internet browser software and also directly from the integrated development environment (IDE).

Applets are used to provide interactive features to web applications that cannot be provided by HTML alone. They can capture mouse input and also have controls like buttons or check boxes. In response to the user action an applet can change the provided graphic content. This makes applets well suitable for demonstration, visualization and teaching. There are online applet collections for studying various subjects, from physics to heart physiology. Applets are also used to create online game collections that allow players to compete against live opponents in real-time.

An applet can also be a text area only, providing, for instance, a cross platform command-line interface to some remote system. If needed, an applet can leave the dedicated area and run as a separate window. However, applets have very little control over web page content outside the applet dedicated area, so they are less useful for improving the site appearance in general (while applets like news tickers or WYSIWYG editors are also known). Applets can also play media in formats that are not natively supported by the browser

HTML pages may embed parameters that are passed to the applet. Hence the same applet may appear differently depending on the parameters that were passed.

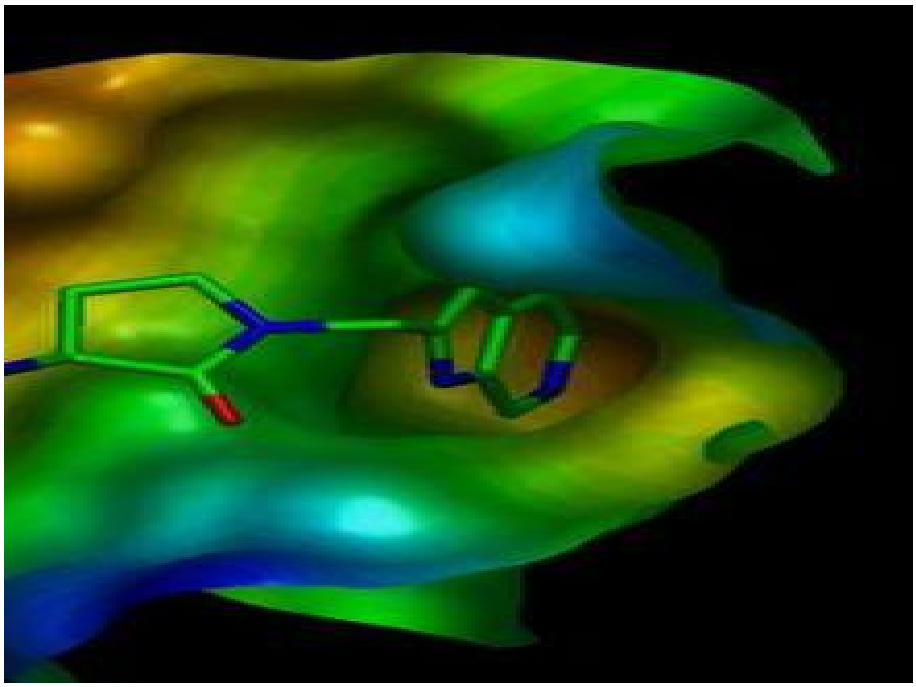


Figure 1: Java applet that uses 3D hardware acceleration, downloading from the server 3D files in .pdb format to visualize

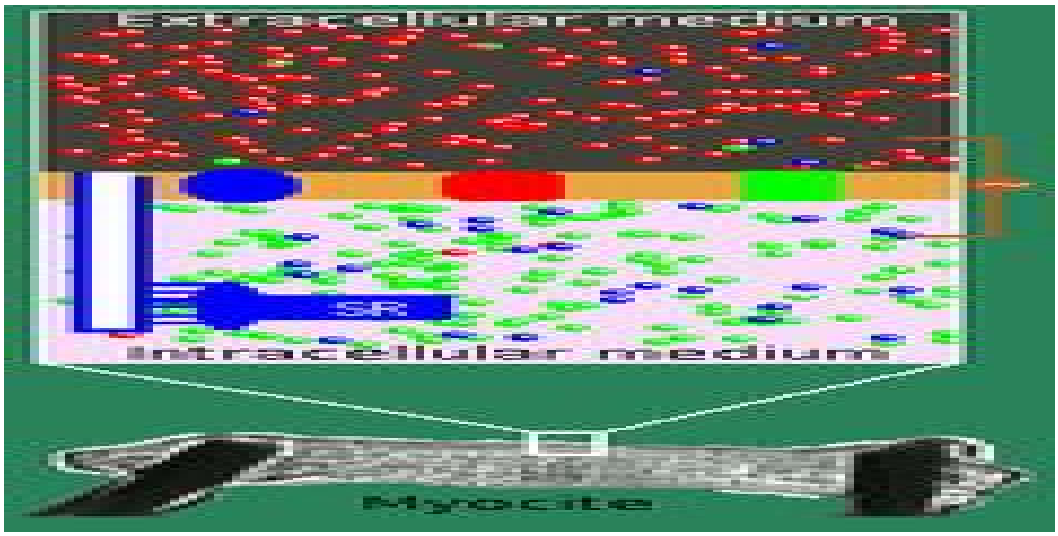


Figure 2: Using applet for nontrivial animation illustrating biophysical topic (randomly moving ions pass through voltage gates)

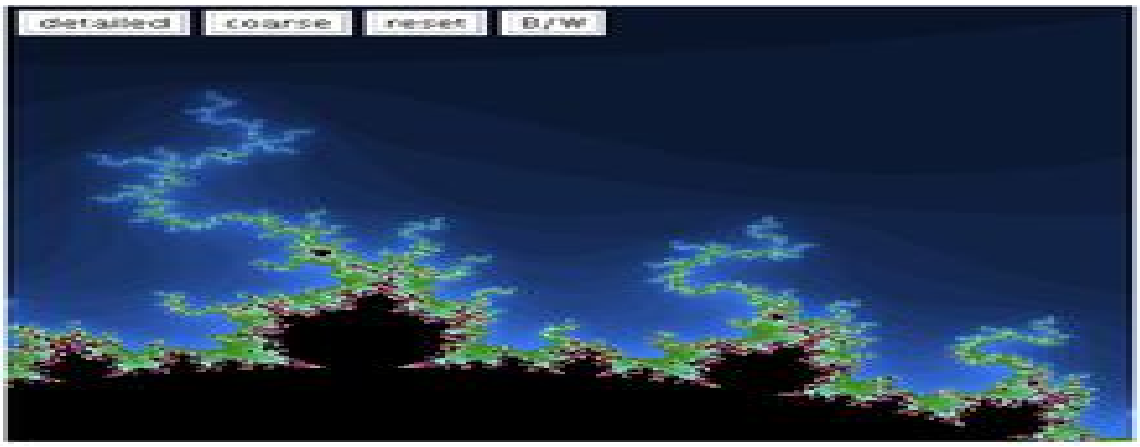


Figure 3: Using Java applet for computation - intensive visualization of the Mandelbrot set

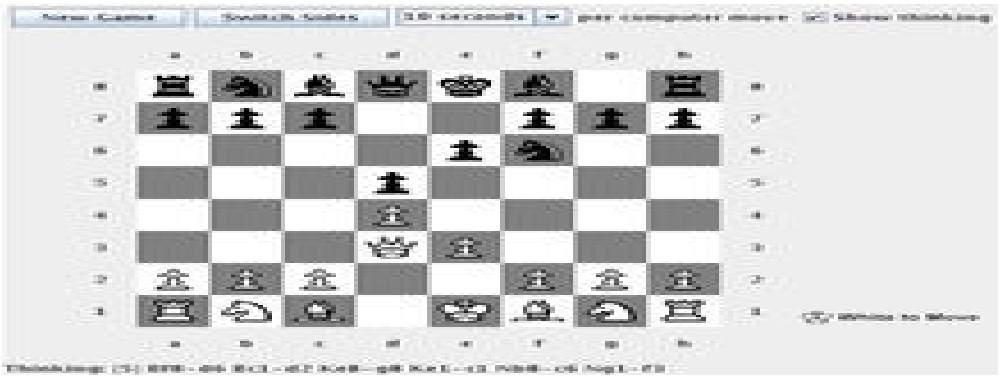


Figure 4: Sufficient running speed is also utilized in applets for playing non trivial computer games like chess

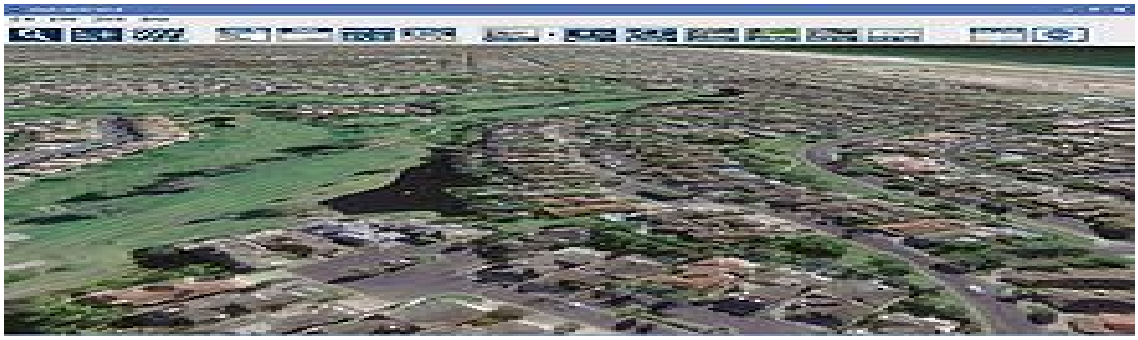


Figure 5: NASA World Wind (open source) is a second generation applet that makes heavy use of OpenGL and on-demand data downloading to provide a detailed 3D map of the world.

Figure 6: Web access to the server console at the hardware level with the help of a Java applet

Figure 7: Demonstration of image processing using two dimensional Fourier transform

3.6 How Java Works

A program written in Java programming language is first translated into Java's intermediate language (this is called "compiling", and the software used for compiling is called a "compiler"). The program is then executed on a Java virtual machine which interprets the intermediate language on some target computer.

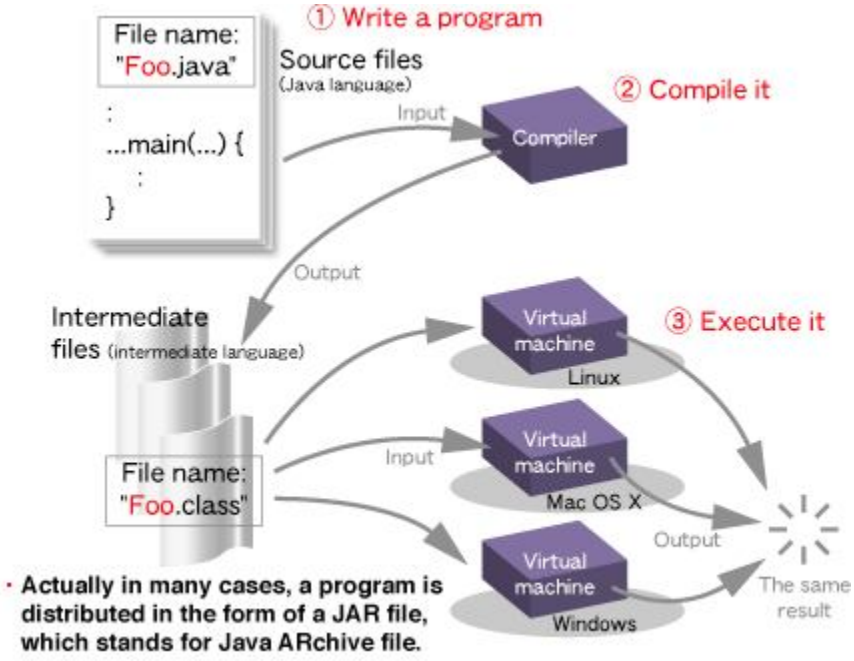


Figure 8: Java virtual machine

By preparing a Java virtual machine for each platform, one program translated into the intermediate language can be run on many different computers.

3.7 Key Benefits of Java

3.7.1 Write Once, Run Anywhere

Sun identifies "Write once, run anywhere" as the core value proposition of the Java platform. Translated from business jargon, this means that the most important promise of Java technology is that you only have to write your application once--for the Java platform--and then you'll be able to run it *anywhere*.

Anywhere, that is, that supports the Java platform. Fortunately, Java support is becoming ubiquitous. It is integrated, or being integrated, into practically all major operating systems. It is built into the popular web browsers, which places it on virtually every Internet-connected PC in the world. It is even being built into consumer electronic devices, such as television set-top boxes, PDAs, and cell phones.

3.7.2 Security

Another key benefit of Java is its security features. Both the language and the platform were designed from the ground up with security in mind. The Java platform allows users to download untrusted code over a network and run it in a secure environment in which it cannot do any harm: it cannot infect the host system with a virus, cannot read or write files from the hard drive, and so forth. This capability alone makes the Java platform unique.

The Java 2 Platform takes the security model a step further. It makes security levels and restrictions highly configurable and extends them beyond applets. As of Java 1.2, any Java code, whether it is an applet, a servlet, a JavaBeans component, or a complete Java application, can be run with restricted permissions that prevent it from doing harm to the host system.

The security features of the Java language and platform have been subjected to intense scrutiny by security experts around the world. Security-related bugs, some of them potentially serious, have been found and promptly fixed. Because of the security promises Java makes, it is big news when a new security bug is found. Remember, however, that no other mainstream platform can make security guarantees nearly as strong as those Java makes. If Java's security is not yet perfect, it has been proven strong enough for practical day-to-day use and is certainly better than any of the alternatives.

3.7.3 Network-centric Programming

Sun's corporate motto has always been "The network is the computer." The designers of the Java platform believed in the importance of networking and designed the Java platform to be network-centric. From a programmer's point of view, Java makes it unbelievably easy to work with resources across a network and to create network-based applications using client/server or multitier architectures. This means that Java programmers have a serious head start in the emerging network economy.

3.7.4 Dynamic, Extensible Programs

Java is both dynamic and extensible. Java code is organized in modular object-oriented units called *classes*. Classes are stored in separate files and are loaded into the Java interpreter only when needed. This means that an application can decide as it is running what classes it needs and can load them when it needs them. It also means that a program can dynamically extend itself by loading the classes it needs to expand its functionality.

The network-centric design of the Java platform means that a Java application can dynamically extend itself by loading new classes over a network. An application that takes advantage of these features ceases to be a monolithic block of code. Instead, it becomes an interacting collection of independent software components. Thus, Java enables a powerful new metaphor of application design and development.

3.7.5 Internationalization

The Java language and the Java platform were designed from the start with the rest of the world in mind. Java is the only commonly used programming language that has internationalization features at its very core, rather than tacked on as an afterthought. While most programming languages use 8-bit characters that represent only the alphabets of English and Western European languages, Java uses 16-bit Unicode characters that represent the phonetic alphabets and ideographic character sets of the entire world. Java's internationalization features are not restricted to just low-level character representation, however. The features permeate the Java platform, making it easier to write internationalized programs with Java than it is with any other environment.

3.7.6 Performance

Java programs are compiled to a portable intermediate form known as byte codes, rather than to native machine-language instructions. The Java Virtual Machine runs a Java program by interpreting these portable byte-code instructions. This architecture means that Java programs are faster than programs or scripts written in purely interpreted languages, but they are typically slower than C and C++ programs compiled to native machine language. Keep in mind, however, that although Java programs are compiled to byte code, not all of the Java platform is implemented with interpreted byte codes. For efficiency, computationally intensive portions of the Java platform--such as the string-manipulation methods--are implemented using native machine code.

Although early releases of Java suffered from performance problems, the speed of the Java VM has improved dramatically with each new release. The VM has been highly tuned and optimized in many significant ways. Furthermore, many implementations include a just-in-time compiler, which converts Java byte codes to native machine instructions on the fly. Using sophisticated JIT compilers, Java programs can execute at speeds comparable to the speeds of native C and C++ applications.

Java is a portable, interpreted language; Java programs run almost as fast as native, non-portable C and C++ programs. Performance used to be an issue that made some programmers avoid using Java. Now, with the improvements made in Java 1.2, performance issues should no longer keep anyone away. In fact, the winning combination of performance plus portability is a unique feature no other language can offer.

3.7.7 Programmer Efficiency and Time-to-Market

The final, and perhaps most important, reason to use Java is that programmers like it. Java is an elegant language combined with a powerful and well-designed set of APIs. Programmers enjoy programming in Java and are usually amazed at how quickly they can get results with it. Studies have consistently shown that switching to Java increases programmer efficiency. Because Java is a simple and elegant language with a well-designed, intuitive set of APIs, programmers write better code with fewer bugs than for other platforms, again reducing development time.

3.8 Disadvantages

A Java applet may have any of the following disadvantages:

- It requires the Java plug-in.
- Some browsers, notably mobile browsers running Apple iOS or Android do not run Java applets at all.
- Some organizations only allow software installed by the administrators. As a result, some users can only view applets that are important enough to justify contacting the administrator to request installation of the Java plug-in.
- As with any client-side scripting, security restrictions may make it difficult or even impossible for an untrusted applet to achieve the desired goals.
- Some applets require a specific JRE. This is discouraged.
- If an applet requires a newer JRE than available on the system, or a specific JRE, the user running it the first time will need to wait for the large JRE download to complete.
- Java automatic installation or update may fail if a proxy server is used to access the web. This makes applets with specific requirements impossible to run unless Java is

manually updated. The Java automatic updater that is part of a Java installation also may be complex to configure if it must work through a proxy.

- Unlike the older applet tag, the object tag needs workarounds to write a cross-browser HTML document.

3.9 Where Do I Start?

To start programming in Java, all you need to do is download and install the Java development Kit.
Once you have the JDK installed on your computer, there's nothing to stop you writing your first Java program.

4.0 CONCLUSION

Java was developed by Sun Microsystems as a cross platform programming language, meaning a computer language that will work across a wide variety of computer systems. It was redesigned specifically for the internet to be used on any system that is Java enabled. Early in the development of Java the catch phrase was developed "write once run anywhere" which refers to the programs robust cross platform nature.

5.0 SUMMARY

We have seen that:

- Java is a high-level object-oriented programming language.
- Java allows you to play online games, chat with people around the world, calculate your mortgage interest, and view images in 3D
- There are 2 basic types of Java applications- Standalone and Applets
- Once you have the JDK installed on your computer, there's nothing to stop you writing your first Java program.

7.0 Tutor Marked Assignment

- Define Java programming language in your own words
- List five reasons for learning Java
- Mention four uses of Java

8.0 **Resources**

- "Performance tests show Java as fast as C++," Carmine Mangione (*JavaWorld*, February 1998)
  http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html
- Java/C++ performance test
  http://www.geko.net.au/~sprack/perform/index.html
- Java 2 SDK download
  http://java.sun.com/jdk
- Programmer's File Editor (simple freeware editor)
  http://www.lancs.ac.uk/people/cpaap/pfe
- Kawa, a basic Java development environment http://www.tek-tools.com/kawa

- The Java Virtual Machine specification
  http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
- ***Java Persistence Tools*** OpenJPA, Toplink, Hibernate Suppt No Lock-in, Eclipse-Based www.myeclipseide.com

- ***Multicore Programming*** Join Cavium University Program Free Teaching Material Lecture Code Lab University.Cavium.com

- ***Generation of Source Code*** .NET, Java, C++, XSD, DDL, PHP, CORBA, Python & more. Free Trial! www.sparxsystems.com

- ***Learn Ethical Hacking*** Ethical Hacking Training Bootcamp now in Lagos by Innobuzz! www.innobuzz.in/Hacking
  - World of Fungi - page of the scientific project, serving an applet that is used as an illustration figure
  - The home site of the 3D protein viewer (Openastexviewer) under LGPL
  - The home site of the Mandelbrot set applet under GPL
  - The home site of the chess applet under BSD
  - Java.Sun.com
  - 2D FFT Java applet
  - Jython applet page
  - About Java applets in Ruby
  - A tool to produce Java applets with SmartEiffel
  - An example of the 2005 year performance benchmarking
  - Paul Falstad online applet portal
  - Jraft.com
  - ObjectPlanet.com, an applet that works as news ticker
  - Sferyx.com, a company that produces applets acting as WYSWYG editor.
  - Cortado applet to play ogg format
  - Top 13 Things Not to Do When Designing a Website
  - JavaWorld.com
  - JavaChannel.net

UNIT 3 VARIABLES AND OPERATORS IN JAVA

1.0 Introduction

2.0 Objectives

3.0 Main Content

3.1 Variables

3.1.1 Types of Variables

3.1.2 Naming of Variables

3.1.3 Primitive Data Types

3.1.4 Default Values

3.1.5 Literals

3.1.5.1 Integer Literals

3.1.5.2 Floating-Point Literals

3.1.5.3 Character and String Literals

3.1.5.4 Using Underscore Characters in Numeric Literals

3.2 Operators

3.2.1 The increment operator

3.2.2 The Simple Assignment Operator

3.2.3 Arithmetic Operators

3.2.4 Unary and binary operators

3.2.5 The Equality and Relational Operators

3.2.6 The Conditional Operators

3.2.7 The Type Comparison Operator instanceof

3.2.8 Bitwise and Bit Shift Operators

4.0 Conclusion

1.0 INTRODUCTION

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, operators, array, expressions and flow of control etc. But this unit will concentrate on variables and operators in Java programming Language.

3.0 OBJECTIVES

After learning the contents of this unit, the student would be able to:

- Display Java operators in order of precedence
- List variable types in Java
- Mention two (2) rules and conventions for naming variables

3.0 MAIN CONTENT

3.1 Variables

In the Java programming language, the terms "field" and "variable" are both used; this is a common source of confusion among new developers, since both often seem to refer to the same thing.

3.1.1 Types of Variables

- **Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.
- **Class Variables (Static Fields)** A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in

- which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameters** You've already seen examples of parameters, both in the Bicycle class and in the main method of the "Hello World!" application. Recall that the signature for the main method is public static void main(String[] args). Here, the args variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields.

3.1.2 Naming of Variables

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "_". The convention, however, is to always begin your variable names with a letter, not "$" or "_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "_", this practice is discouraged. White space is not permitted.
- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named cadence, speed, and gear, for example, are much more intuitive than abbreviated versions, such as s, c, and g. Also keep in mind that the name you choose must not be a keyword or reserved word.
- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names gearRatio and currentGear are prime examples of this convention. If your variable stores a constant value, such as static final int NUM_GEARS = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

3.1.3 Primitive Data Types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int gear = 1;
```

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other *primitive data types*. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
- **int**: The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.
- **long**: The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by int.
- **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.
- **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new String object; for example, String s = "this is a string";. String objects are *immutable*, which means that once created, their values cannot be changed. The String class is not technically a primitive data

type, but considering the special support given to it by the language, you'll probably tend to think of it as such.

3.1.4 Default Values

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

Table 1: Data Types and their Default values

Data Type	**Default Value (for fields)**
Byte	0
Short	0
Int	0
Long	0L
Float	0.0f
Double	0.0d
Char	'\u0000'
String (or any object)	Null
Boolean	False

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

3.1.5 Literals

You may have noticed that the new keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

3.1.5.1 Integer Literals

An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int. It is recommended that you use the upper case letter L because the lower case letter l is hard to distinguish from the digit 1.

Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long literals. Integer literals can be expressed these number systems:

- Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
- Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

```
int decVal = 26;      // The number 26, in decimal
int hexVal = 0x1a;    // The number 26, in hexadecimal
int binVal = 0b11010; // The number 26, in binary
```

3.1.5.2 Floating-Point Literals

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
double d2 = 1.234e2; // same value as d1, but in scientific notation
float f1  = 123.4f;
```

3.1.5.3 Character and String Literals

Literals of types char and String may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as '\u0108' (capital C with circumflex), or "S\u00ED se\u00F1or" (Sí Señor in Spanish). Always use 'single quotes' for char literals and "double quotes" for String literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \" (double quote), \' (single quote), and \\ (backslash).

There's also a special null literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types. There's little you can do with a null value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending ".class"; for example, String.class. This refers to the object (of type Class) that represents the type itself.

3.1.5.4 Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal

- Prior to an F or L suffix
- In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

```
float pi1 = 3_.1415F;      // Invalid; cannot put underscores adjacent to a decimal point
float pi2 = 3._1415F;      // Invalid; cannot put underscores adjacent to a decimal point
long socialSecurityNumber1
  = 999_99_9999_L;         // Invalid; cannot put underscores prior to an L suffix

int x1 = _52;              // This is an identifier, not a numeric literal
int x2 = 5_2;              // OK (decimal literal)
int x3 = 52_;              // Invalid; cannot put underscores at the end of a literal
int x4 = 5_______2;        // OK (decimal literal)

int x5 = 0_x52;            // Invalid; cannot put underscores in the 0x radix prefix
int x6 = 0x_52;            // Invalid; cannot put underscores at the beginning of a number
int x7 = 0x5_2;            // OK (hexadecimal literal)
int x8 = 0x52_;            // Invalid; cannot put underscores at the end of a number
```

3.2 Operators

Learning the operators of the Java programming language is a good place to start. Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When 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 from left to right; assignment operators are evaluated right to left.

Table 2: Operator Precedence

Operator Precedence	
Operators	**Precedence**
postfix	*expr*++ *expr*--

unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

3.2.1 The increment operator

An extremely important *unary* operator that is used in both Java and C++ (but does not exist in Pascal) is the increment operator (++).

This operator is used in both *prefix* and *postfix* notation. The increment operator causes the value of its operand to be increased by one.

Prefix and postfix increment operators

With the *prefix* version, the operand appears to the right of the operator (**++X**), while with the *postfix* version, the operand appears to the left of the operator (**X++**).

What's the difference in prefix and postfix?

The difference in prefix and postfix has to do with the point in time that the increment actually occurs if the operator and its operand appear as part of a larger overall expression.

Prefix behavior

With the *prefix* version, the variable is incremented before it is used to evaluate the larger overall expression.

Postfix behavior

With the *postfix* version, the variable is used to evaluate the larger overall expression and then it is incremented.

Illustration of prefix and postfix behavior

The use of both the *prefix* and *postfix* versions of the increment operator is illustrated in the following Java program.

/*File incr01.java Copyright 1997, n
Illustrates the use of the prefix and the postfix increment
operator.

The output from the program follows:

a = 5
b = 5
a + b++ = 10
b = 6

c = 5
d = 5
c + ++d = 11
d = 6

**********/

```
class incr01 { //define the controlling class
  public static void main(String[] args){ //main method
    int a = 5, b = 5, c = 5, d = 5;
    System.out.println("a = " + a );
    System.out.println("b = " + b );
```

```
    System.out.println("a + b++ = " + (a + b++) );
    System.out.println("b = " + b );
    System.out.println();

    System.out.println("c = " + c );
    System.out.println("d = " + d );
    System.out.println("c + ++d = " + (c + ++d) );
    System.out.println("d = " + d );
  }//end main
}//End incr01 class.
```

Binary operators and infix notation

Binary operators use *infix* notation, which means that the operator appears between its operands.

3.2.2 The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator "=". You saw this operator in the Bicycle class; it assigns the value on its right to the operand on its left:

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

This operator can also be used on objects to assign *object references.*

3.2.3 Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There's a good chance you'll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is "%", which divides one operand by another and returns the remainder as its result.

```
+      additive operator (also used for String concatenation)
-      subtraction operator
*      multiplication operator
/      division operator
%      remainder operator
```

The plus operator

Of particular interest in this list is the plus sign (+) . In Java, the plus sign can be used to perform arithmetic addition.

It can also be used to concatenate strings. When the plus sign is used in this manner, the operand on the right is automatically converted to a character string before being concatenated with the operand on the left.

The following program, ArithmeticDemo, tests the arithmetic operators.

```
class ArithmeticDemo {

    public static void main (String[] args){

        int result = 1 + 2; // result is now 3
        System.out.println(result);

        result = result - 1; // result is now 2
        System.out.println(result);

        result = result * 2; // result is now 4
        System.out.println(result);

        result = result / 2; // result is now 2
        System.out.println(result);

        result = result + 8; // result is now 10
        result = result % 7; // result is now 3
        System.out.println(result);

    }
}
```

You can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*. For example, x+=1; and x=x+1; both increment the value of x by 1.

The + operator can also be used for concatenating (joining) two strings together, as shown in the following ConcatDemo program:

```
class ConcatDemo {
    public static void main(String[] args){
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

By the end of this program, the variable thirdString contains "This is a concatenated string.", which gets printed to standard output.

3.2.4 **Unary and binary operators**

Both Java and C++ provide a set of operators that can be used to perform an action on one or two operands. An operator that operates on one operand is called a **unary** operator, and an operator that operates on two operands is called a **binary** operator.

Some operators can behave either as a unary or as a binary operator, the best known of which is probably the minus sign. As a binary operator, the minus sign causes its right operand to be subtracted from its left operand. As a unary operator, the minus sign causes the algebraic sign of the right operand to be changed.

```
+       Unary plus operator; indicates positive value (numbers are positive without this,
however)
-       Unary minus operator; negates an expression
++      Increment operator; increments a value by 1
--      Decrement operator; decrements a value by 1
!       Logical complement operator; inverts the value of a boolean
```

The following program, UnaryDemo, tests the unary operators:

```
class UnaryDemo {

    public static void main(String[] args){
        int result = +1; // result is now 1
        System.out.println(result);
        result--;  // result is now 0
        System.out.println(result);
        result++; // result is now 1
        System.out.println(result);
        result = -result; // result is now -1
        System.out.println(result);
        boolean success = false;
        System.out.println(success); // false
        System.out.println(!success); // true
    }
}
```

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code result++; and ++result; will both end in result being incremented by one. The only difference is that the prefix version (++result) evaluates to the incremented value, whereas the postfix version (result++) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose.

But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following program, PrePostDemo, illustrates the prefix/postfix unary increment operator:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
            i++;
            System.out.println(i);        // "4"
            ++i;
            System.out.println(i);        // "5"
            System.out.println(++i);      // "6"
            System.out.println(i++);      // "6"
            System.out.println(i);        // "7"
    }
}
```

3.2.5 The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==", not "=", when testing if two primitive values are equal.

```
==    equal to
!=    not equal to
>     greater than
>=    greater than or equal to
<     less than
<=    less than or equal to
```

The following program, ComparisonDemo, tests the comparison operators:

```
class ComparisonDemo {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2) System.out.println("value1 == value2");
        if(value1 != value2) System.out.println("value1 != value2");
        if(value1 > value2) System.out.println("value1 > value2");
```

```
        if(value1 < value2) System.out.println("value1 < value2");
        if(value1 <= value2) System.out.println("value1 <= value2");
    }
}
```

Output:

```
value1 != value2
value1 < value2
value1 <= value2
```

3.2.6 The Conditional Operators

The && and || operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

```
&& Conditional-AND
|| Conditional-OR
```

The following program, ConditionalDemo1, tests these operators:

```
class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2)) System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1)) System.out.println("value1 is 1 OR value2 is 1");

    }
}
```

3.2.7 The Type Comparison Operator instanceof

The instanceof operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, InstanceofDemo, defines a parent class (named Parent), a simple interface (named MyInterface), and a child class (named Child) that inherits from the parent and implements the interface.

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

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Output:

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

When using the instanceof operator, keep in mind that null is not an instance of anything.

3.2.8 Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand,

and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.

The bitwise & operator performs a bitwise AND operation.

The bitwise ^ operator performs a bitwise exclusive OR operation.

The bitwise | operator performs a bitwise inclusive OR operation.

The following program, BitDemo, uses the bitwise AND operator to print the number "2" to standard output.

```
class BitDemo {
    public static void main(String[] args) {
        int bitmask = 0x000F;
            int val = 0x2222;
            System.out.println(val & bitmask);  // prints "2"
    }
}
```

4.0 CONCLUSION

Operators in Java are similar to those in C++. However, there is no delete operator due to garbage collection mechanisms in Java, and there are no operations on pointers since Java does not support them. Another difference is that Java has an unsigned right shift operator (>>>). Operators in Java cannot be overloaded.

5.0 SUMMARY

In this unit you learnt the following:

- Definition of Variables
- Types of Variables
- Naming of Variables
- Primitive Data Types
- Literals
- Definition of Operators
- The Simple Assignment Operator
- Arithmetic Operators
- Unary and binary operators

- The Equality and Relational Operators
- The Conditional Operators
- The Type Comparison Operator instanceof
- Bitwise and Bit Shift Operators

6.0 TUTOR MARKED ASSIGNMENT

- An operator performs an action on what? Provide the name
- What do we call an operator that operates on only **one** operand?
- What do we call an operator that operates on **two** operands?
- Is the minus sign a **unary** or a **binary** operator, or both? Explain your answer.
- Show the symbol for the **increment** operator.
- Show and describe six different *relational* operators supported by Java.
- What is the type returned by *relational* operators in Java?

7.0 FURTHER READING AND OTHER RESOURCES

- Gosling, James; Joy Bill; Steele, Guy; and Bracha, Gillad (2005). *Java Language Specification* (3rd ed.). Addison-Wesley Professional. http://java.sun.com/docs/books/jls/index.html.
- Patrick Naughton , Herbert Schildt (1999). *Java 2: The Complete Reference*, third edition. The McGraw-Hill Companies, . ISBN 0-07-211976-4
- Vermeulen, Ambler, Bumgardner, Metz, Misfeldt, Shur, Thompson (2000). *The Elements of Java Style*. Cambridge University Press, . ISBN 0-521-77768-2

UNIT 4 ARRAYS AND EXPRESSIONS IN JAVA

CONTENT

1.0 `INTRODUCTION

This unit will deal on Arrays and Expressions. They are also part of foundation concepts in Java.

2.0 OBJECTIVES

After the end of this unit, students should be able to:

- Describe an array
- Declaring a Variable to refer to an array
- Create, Initialize and access an Array

3.0 MAIN CONTEENT

3.1 Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

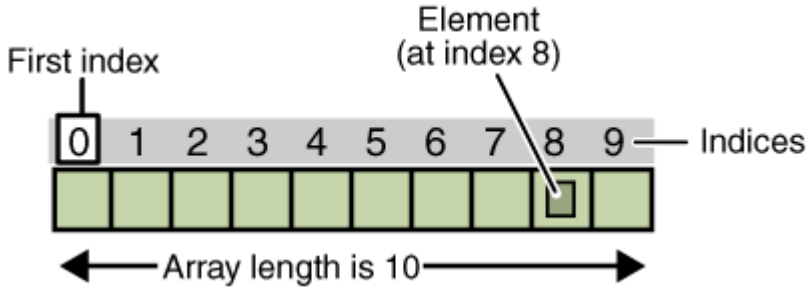


Figure 1: An array of ten elements

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

The following program, ArrayDemo, creates an array of integers, puts some values in it, and prints each value to standard output.

```
class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray;            // declares an array of integers

        anArray = new int[10];      // allocates memory for 10 integers

        anArray[0] = 100; // initialize first element
        anArray[1] = 200; // initialize second element
        anArray[2] = 300; // etc.
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: " + anArray[0]);
        System.out.println("Element at index 1: " + anArray[1]);
        System.out.println("Element at index 2: " + anArray[2]);
        System.out.println("Element at index 3: " + anArray[3]);
        System.out.println("Element at index 4: " + anArray[4]);
        System.out.println("Element at index 5: " + anArray[5]);
        System.out.println("Element at index 6: " + anArray[6]);
        System.out.println("Element at index 7: " + anArray[7]);
        System.out.println("Element at index 8: " + anArray[8]);
        System.out.println("Element at index 9: " + anArray[9]);
    }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

In a real-world programming situation, you'd probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as shown above. However, this example clearly illustrates the array syntax.

3.1.1 Declaring a Variable to Refer to an Array

The above program declares anArray with the following line of code:

```
int[] anArray;          // declares an array of integers
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as *type*[], where *type* is the data type of the contained elements; the square brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the naming section. As with variables of other types, the declaration does not actually create an array — it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the square brackets after the array's name:

```
float anArrayOfFloats[]; // this form is discouraged
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

3.1.2 Creating, Initializing, and Accessing an Array

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable.

```
anArray = new int[10];  // create an array of integers
```

If this statement were missing, the compiler would print an error like the following, and compilation would fail:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
```

Here the length of the array is determined by the number of values provided between *{* and *}*.

You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of square brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                            {"Smith", "Jones"}};
        System.out.println(names[0][0] + names[1][0]); //Mr. Smith
        System.out.println(names[0][2] + names[1][1]); //Ms. Jones
```

```
    }
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

Finally, you can use the built-in length property to determine the size of any array. The code

```
System.out.println(anArray.length);
```

will print the array's size to standard output.

3.1.3 Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

The two Object arguments specify the array to copy *from* and the array to copy *to*. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, ArrayCopyDemo, declares an array of char elements, spelling the word "decaffeinated". It uses arraycopy to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                                    'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
```

```
    }
}
```

The output from this program is:

```
caffein
```

3.2 Expressions, Statements, and Blocks

Expressions are the core components of statements; statements may be grouped into blocks.

3.2.1 Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value. You've already seen examples of expressions, illustrated in bold below:

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if(value1 == value2) System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `cadence = 0` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, no matter in which 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 an expression will be evaluated using balanced parenthesis: (and). For example, to make the previous expression unambiguous, you could write the following:

```
(x + y) / 100    // unambiguous, recommended
```

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

```
x + y / 100

x + (y / 100) // unambiguous, recommended
```

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

3.2.2 Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms 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 invocations
- Object creation expressions

Such statements are called *expression statements*. Here are some examples of expression statements.

```
aValue = 8933.234;                  // assignment statement
aValue++;                           // increment statement
System.out.println("Hello World!");     // method invocation statement
Bicycle myBike = new Bicycle(); // object creation statement
```

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements.* A *declaration statement* declares a variable. You've seen many examples of declaration statements already:

```
double aValue = 8933.234; //declaration statement
```

3.2.3 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 example, BlockDemo, illustrates the use of blocks:

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

4.0 CONCLUSION

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Statements in Java are roughly equivalent to sentences in natural languages.

5.0 SUMMARY

In this unit, you have learnt:

- Definition of an Array
- Declaring a Variable to Refer to an Array
- Creating, Initializing, and Accessing an Array
- Expressions, Statements, and Blocks

6.0 TUTOR MARKED ASSIGNMENT

- Define an Array in your own words
- Describe expression statement with examples

- Define Block in Java

7.0 FURTHER READING AND OTHER RESOURCES

- Gosling, James; Joy Bill; Steele, Guy; and Bracha, Gillad (2005). *Java Language Specification* (3rd ed.). Addison-Wesley Professional. http://java.sun.com/docs/books/jls/index.html.
- Patrick Naughton , Herbert Schildt (1999). *Java 2: The Complete Reference*, third edition. The McGraw-Hill Companies, . ISBN 0-07-211976-4
- Vermeulen, Ambler, Bumgardner, Metz, Misfeldt, Shur, Thompson (2000). *The Elements of Java Style*. Cambridge University Press, . ISBN 0-521-77768-2

UNIT 5 CONTROL FLOW STATEMENTS

CONTENT

1.0 INTRODUCTION

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements,* however, break up the flow of execution by

employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code.

2.0 OBJECTIVES

After the end of this unit, the students should be able to:

- Describe the Switch Statement
- List the types of Branching Statements
- Differentiate between if- then and if-then –else Statements
- Write simple Java program using flow of control

3.0 MAIN CONTENT

3.1 The if-then and if-then-else Statements

- The if-then Statement

The `if-then` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to `true`. For example, the `Bicycle` class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the `applyBrakes` method could be as follows:

```
void applyBrakes(){
    if (isMoving){  // the "if" clause: bicycle must be moving
        currentSpeed--; // the "then" clause: decrease current speed
    }
}
```

If this test evaluates to `false` (meaning that the bicycle is not in motion), control jumps to the end of the `if-then` statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes(){
    if (isMoving) currentSpeed--; // same as above, but without braces
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

- The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes(){
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The following program, IfElseDemo, 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.

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

       int testscore = 76;
       char grade;

       if (testscore >= 90) {
           grade = 'A';
       } else if (testscore >= 80) {
           grade = 'B';
       } else if (testscore >= 70) {
           grade = 'C';
       } else if (testscore >= 60) {
           grade = 'D';
       } else {
           grade = 'F';
       }
       System.out.println("Grade = " + grade);
   }
}
```

The output from the program is:

```
   Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

3.2 The switch Statement

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types* and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer.

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code 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;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";       break;
            case 2:  monthString = "February";      break;
            case 3:  monthString = "March";         break;
            case 4:  monthString = "April";         break;
            case 5:  monthString = "May";           break;
            case 6:  monthString = "June";          break;
            case 7:  monthString = "July";          break;
            case 8:  monthString = "August";        break;
            case 9:  monthString = "September";     break;
            case 10: monthString = "October";       break;
            case 11: monthString = "November";      break;
            case 12: monthString = "December";      break;
            default: monthString = "Invalid month"; break;
        }
        System.out.println(monthString);
    }
}
```

In this case, August is printed to standard output.

The body of a switch statement is known as a *switch block*. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, then executes all statements that follow the matching case label.

You could also display the name of the month with if-then-else statements:

```
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 if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, statements in switch blocks *fall through*: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered. The program SwitchDemoFallThrough shows statements in a switch block that fall through. The program displays the month corresponding to the integer month and the months that follow in the year:

```
public class SwitchDemoFallThrough {

  public static void main(String args[]) {
    java.util.ArrayList<String> futureMonths = new java.util.ArrayList<String>();

    int month = 8;

    switch (month) {
      case 1: futureMonths.add("January");
      case 2: futureMonths.add("February");
      case 3: futureMonths.add("March");
      case 4: futureMonths.add("April");
      case 5: futureMonths.add("May");
      case 6: futureMonths.add("June");
      case 7: futureMonths.add("July");
      case 8: futureMonths.add("August");
      case 9: futureMonths.add("September");
      case 10: futureMonths.add("October");
      case 11: futureMonths.add("November");
      case 12: futureMonths.add("December"); break;
      default: break;
    }
```

```
    if (futureMonths.isEmpty()) {
      System.out.println("Invalid month number");
    } else {
      for (String monthName : futureMonths) {
        System.out.println(monthName);
      }
    }
  }
}
```

This is the output from the code:

```
August
September
October
November
December
```

Technically, the final `break` is not required because flow falls out of the `switch` statement. Using a `break` is recommended so that modifying the code is easier and less error prone. The `default` section handles all values that are not explicitly handled by one of the `case` sections.

The following code example, `SwitchDemo2`, shows how a statement can have multiple `case` labels. The code example calculates the number of days in a particular month:

```
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))
                   || (year % 400 == 0) )
                  numDays = 29;
               else
                  numDays = 28;
               break;
            default:
               System.out.println("Invalid month.");
               break;
         }
         System.out.println("Number of Days = " + numDays);
      }
   }
```

This is the output from the code:

```
   Number of Days = 29
```

3.2.1 Using Strings in switch Statements

In Java SE 7 and later, you can use a String object in the switch statement's expression. The following code example, StringSwitchDemo, displays the number of the month based on the value of the String named month:

```
public class StringSwitchDemo {

    public static int getMonthNumber(String month) {

        int monthNumber = 0;

        if (month == null) { return monthNumber; }

        switch (month.toLowerCase()) {
            case "january":    monthNumber =  1; break;
            case "february":   monthNumber =  2; break;
            case "march":      monthNumber =  3; break;
            case "april":      monthNumber =  4; break;
            case "may":        monthNumber =  5; break;
            case "june":       monthNumber =  6; break;
```

```
            case "july":       monthNumber =  7; break;
            case "august":     monthNumber =  8; break;
            case "september":  monthNumber =  9; break;
            case "october":    monthNumber = 10; break;
            case "november":   monthNumber = 11; break;
            case "december":   monthNumber = 12; break;
            default:           monthNumber =  0; break;
        }

        return monthNumber;
    }

    public static void main(String[] args) {

        String month = "August";

        int returnedMonthNumber =
          StringSwitchDemo.getMonthNumber(month);

        if (returnedMonthNumber == 0) {
          System.out.println("Invalid month");
        } else {
          System.out.println(returnedMonthNumber);
        }
    }
}
```

The output from this code is 8.

The String in the switch expression is compared with the expressions associated with each case label as if the String.equals method were being used. In order for the StringSwitchDemo example to accept any month regardless of case, month is converted to lowercase (with the toLowerCase method), and all the strings associated with the case labels are in lowercase.

Note: This example checks if the expression in the switch statement is null. Ensure that the expression in any switch statement is not null to prevent a NullPointerException from being thrown.

3.3 The while and do-while Statements

The *while statement* continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
```

```
}
```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){
   // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
```

```
            count++;
        } while (count <= 11);
    }
}
```

3.4 The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment) {
   statement(s)
}
```

When using this version of the for statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}
```

The output of this program is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
```

```
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialization expression. 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` statement 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 initialization expression limits their life span and reduces errors.

The three expressions of the `for` loop are optional; an infinite loop can be created as follows:

```
for ( ; ; ) {    // infinite loop

    // your code goes here
}
```

The `for` statement also has another form designed for iteration through Collections and arrays This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, `EnhancedForDemo`, uses the enhanced `for` to loop through the array:

```
class EnhancedForDemo {
    public static void main(String[] args){
        int[] numbers = {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
          System.out.println("Count is: " + item);
        }
    }
}
```

In this example, the variable `item` holds the current value from the numbers array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

We recommend using this form of the `for` statement instead of the general form whenever possible.

3.5 Branching Statements

3.5.1 The `break` Statement

The `break` statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the `switch` statement. You can also use an unlabeled `break` to terminate a `for`, `while`, or `do-while` loop, as shown in the following `BreakDemo` program:

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

        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                              2000, 8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor
                               + " at index " + i);
        } else {
            System.out.println(searchfor
                               + " not in the array");
        }
    }
```

```
}
```

This program searches for the number 12 in an array. The `break` statement, shown in boldface, terminates the `for` loop when that value is found. Control flow then transfers to the print statement at the end of the program. This program's output is:

```
Found 12 at index 4
```

An unlabeled `break` statement terminates the innermost `switch`, `for`, `while`, or `do-while` statement, but a labeled `break` terminates an outer statement. The following program, `BreakWithLabelDemo`, is similar to the previous program, but uses nested `for` loops to search for a value in a two-dimensional array. When the value is found, a labeled `break` terminates the outer `for` loop (labeled "search"):

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

        int[][] arrayOfInts = { { 32, 87, 3, 589 },
                                { 12, 1076, 2000, 8 },
                                { 622, 127, 77, 955 }
                              };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor +
                               " at " + i + ", " + j);
        } else {
            System.out.println(searchfor
                               + " not in the array");
        }
```

```
    }
}
```

This is the output of the program.

```
    Found 12 at 1, 0
```

The `break` statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

3.5.2 The `continue` Statement

The `continue` statement skips the current iteration of a `for`, `while` , or `do-while` loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the `boolean` expression that controls the loop. The following program, `ContinueDemo` , steps through a `String`, counting the occurences of the letter "p". If the current character is not a p, the `continue` statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

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

        String searchMe = "peter piper picked a peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            //interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            //process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```

Here is the output of this program:

```
Found 9 p's in the string.
```

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, ContinueWithLabelDemo, uses the labeled form of continue to skip an iteration in the outer loop.

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

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++)
                        != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
                break test;
        }
        System.out.println(foundIt ? "Found it" :
                                "Didn't find it");
    }
}
```

Here is the output from this program.

```
    Found it
```

3.5.3 The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return

statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared void use the form of return that doesn't return a value.

```
return;
```

4.0 CONCLUSION

The *if* statement is the fundamental control statement that allows Java to make decisions, the *while* statement is the basic statement that allows Java to perform repetitive actions. The *for* statement provides a looping construct that is often more convenient than the while and do loops.

5.0 SUMMARY

In this unit, you have learnt:

- The if-then and if-then-else Statements
- The switch Statement
- The while and do-while Statements
- The for Statement
- Branching Statements

6.0 TUTOR ASSIGNMENT

- Distinguish between *if* Statement and *if else* Statement.
- Describe the *for* Statement
- Differentiate between the break Statement and the Continue Statement
- Write short note on Control Flow Statement

7.0 REFERNCES/FURTHER READING

- Gosling, James; Joy Bill; Steele, Guy; and Bracha, Gillad (2005). *Java Language Specification* (3rd ed.). Addison-Wesley Professional. http://java.sun.com/docs/books/jls/index.html.
- Patrick Naughton , Herbert Schildt (1999). *Java 2: The Complete Reference*, third edition. The McGraw-Hill Companies, . ISBN 0-07-211976-4
- Vermeulen, Ambler, Bumgardner, Metz, Misfeldt, Shur, Thompson (2000). *The Elements of Java Style*. Cambridge University Press, . ISBN 0-521-77768-2