FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA, NIGERIA

CENTRE FOR OPEN DISTANCE AND E-LEARNING (CODeL)

SCHOOL OF SCIENCE AND SCIENCE EDUCATION

COURSE TITLE

OBJECT-ORIENTED PROGRAMMING, I

COURSE CODE:

CPT 211

COURSE TITLE: OBJECT-ORIENTED PROGRAMMING I

COURSE CODE: CPT 211

COURSE UNIT: 3

Course Coordinator
Bashir Mohammad (Ph.D.)
Computer Science Department
FUT Minna, Nigeria.

COURSE DEVELOPMENT TEAM

Course Coordinator	Sapun Aksana (Mrs.) Potapenko Natalya Department of Information & Media Technology Federal University of Technology, Minna, Nigeria
Subject matter experts	Abraham, O. (Ph.D.) Agboola, A. K. (Ph.D.) Department of Information & Media Technology
ODL Experts	Amosa Isiaka GAMBARI (Ph.D.) Nicholas E. ESEZOBOR
Instructional System Designers	FALODE, Oluwole Caleb (Ph.D.) Bushrah Temitope OJOYE (Mrs.)
Language Editors	Chinenye Priscilla UZOCHUKWU Mubarak Jamiu ALABEDE
Centre Director	Abiodun Musa AIBINU (Ph.D.) Centre for Open Distance & e-Learning FUT Minna, Nigeria.

STUDY GUIDE

CPT211: OBJECT-ORIENTED PROGRAMMING I

1.0 Introduction

CPT 211: Object-oriented programming I is a 3 credit unit course for students studying towards acquiring a Bachelor of Technology in Information Technology and other related disciplines. The course is divided into 7 modules and 17 study units. It will first take a brief review of the concepts of Object-oriented programming. This course will then go ahead to deal with the Object-oriented programming II.

Architecture. The course goes further to deal with the concept of Objects and Classes, Polymorphism, Inheritance, Abstract Classes. The course concludes by discussing some concepts like Algorithms, Object oriented design.

2.0 Course Guide

The course guide therefore gives you an overview of what the course; CPT 211 is all about, the textbooks and other materials to be referenced, what you expect to know in each unit, and how to work through the course material. The course guide introduces to you what you will learn in this course and how to make the best use of the material. It brings to your notice the general guidelines on how to navigate through the course and on the expected actions you have to take for you to complete this course successfully. Also, the guide will hint you on how to respond to your Self Assessment Question(s) and Tutor-Marked Assignments.

3.0 What You Will Learn In This Course

The overall aim of this course, CPT 211 is to introduce you to the basic concepts of Object-oriented programming to enable students to understand the basics of Objects and Classes, Polymorphism, Inheritance, Abstract Classes.

This course highlights different primitive types and control structures. This course will introduce you to the practical terms and definitions of object-oriented programming.

4.0 Course Aim

The aim of this course is to introduce students to the basics and concepts of Object-oriented programming. It is believed the knowledge will enable the student to understand the functionalities and capabilities of Object-oriented programming to combine objects into structured networks to form a complete program and to acquaint the student with modern programming practices.

5.0 Course Objectives

It is important to note that each unit has specific objectives. Students should study them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit. However, below are overall objectives of this course. On completing this course, you should be able to:

(i) introducing the principles of object-oriented programming;
(ii) to develop students' competence in primitive types and control structures;
(iii) differentiating objects and classes;
(iv) showcase some practical issues of object-oriented programming; and
(v) dealing with inheritance, abstract classes, polymorphisms and algorithms.

6.0 Working Through This Course

To complete this course, you are required to study all the units, the recommended textbooks, and other relevant materials. Each unit contains some self-assessment exercises and tutor marked assignments, and at some point, in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

7.0 Course Materials

The major components of the course are:

1. Course Guide
2. Study Units
3. Text Books
4. Assignment File
5. Presentation Schedule

8.0 Study Units

There are 17 study units and 7 modules in this course. They are:

MODULE 1: INTRODUCTION TO PROGRAMMING

Unit 1:	Overview Paradigms of Programming
Unit 2:	Overview of Programming Languages and the Compilation Process
Unit 3:	Introduction to Object Oriented Programming

MODULE 2: FUNDAMENTALS OF OBJECTS AND CLASSES

Unit 1:	Objects and Classes
Unit 2:	Class Members and Instance Members
Unit 3:	Methods, Message Passing, Creating and Destroying Objects

MODULE 3: INHERITANCE, POLYMORPHISM AND ABSTRACT CLASSES

Unit 1:	Inheritance
Unit 2:	Polymorphism
Unit 3:	Abstract Classes

MODULE 4: PRIMITIVE DATA TYPES

Unit 1:	Primitive Data Types
Unit 2:	Control Structures

MODULE 5: ARRAYS AND STRINGS

Unit 1:	Arrays
Unit 2:	Strings

MODULE 6: ALGORITHMS

Unit 1:	Concept of an Algorithm, Problem-Solving Strategies
Unit 2:	Pseudocode and Stepwise Refinement

MODULE 7: OBJECT ORIENTED DESIGN

Unit 1: Fundamental design concept and principles
Unit 2: introduction to design patterns

9.0 RECOMMENDED TEXTS

These texts and especially the internet resource links will be of enormous benefit to you in learning this course:

http://docs.oracle.com/javase/tutorial/java/
http://www.kodejava.org/
http://objc.toodarkpark.net/objctoc.html
http://en.wikipedia.org/wiki/Object-oriented_programming

10.0 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 tutor marked assignments for this course.

11.0 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.

12.0 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.

13.0 Tutor Marked Assignments (TMAS)

There are TMAs in this course. You need to submit all the TMAs. The best 10 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.

14.0 Final Examination And Grading

The final examination for CPT 211 will be of last for a period of 2 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the Self Assessment Questions and 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.

15.0 The Following Are Practical Strategies For Working Through This Course

1. Read the course guide thoroughly

2. 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 assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather together all these information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.

3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.

4. Turn to Unit 1 and read the introduction and the objectives for the unit.

5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.

6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books

7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.

8. Review the objectives of each study unit to confirm that you have achieved them. If you are not certain about any of the objectives, review the study material and consult your tutor.

9. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you can keep yourself on schedule.

10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is

returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult you tutor as soon as possible if you have any questions or problems.

11. 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).

16.0 Tutors and Tutorials

There are 8 hours of tutorial provided in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group. Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties, you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

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

• You do not understand any part of the study units or the assigned readings.

• You have difficulty with the self test or exercise.

• You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

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

GOODLUCK!

Table of Contents

Module 1

Introduction to Object Oriented Programming

Unit 1

Overview Paradigms of Programming

CONTENTS

1.0 Introduction

In this introductory part of the material we will introduce the concept of programming paradigms. Several main styles (or paradigms, or models) of programming – imperative, functional, logic and object oriented ones – were developed during more than forty-year history of programming. Each of them is based on specific algorithmic abstractions of data, operations, and control and presents a specific mode of thinking about program and its execution. Various programming techniques (including data structures and control mechanisms) were elaborated rather independently within each style, thereby forming different scopes of their applicability. For instance, the object-oriented style and corresponding techniques are suitable for creating programs with complicated data and interface, while the logic style is convenient to program logic inference.

2.0 Learning Outcomes

At the end of this unit should be able to:

1. define a paradigm of programming;
2. know four distinct and fundamental programming paradigms;
3. distinguish a programming paradigm by its key features.

3.0 Learning Contents

3.1 What Is 'Paradigm'?

The simple definition is - "An example that serves as pattern or model."
But more scientific definitions are given in 'The American Heritage Dictionary of the English Language, Third Edition':

"A philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them is formulated".

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Programming Paradigms

Programming paradigm means:

1) a pattern that serves as a school of thoughts for programming of computers;
2) programming technique - related to an algorithmic idea for solving a particular class of problems (for examples: 'divide and conquer', 'program development by stepwise refinement');
3) programming style - the way we express ourselves in a computer program, related to elegance or lack of elegance; and
4) programming culture - The totality of programming behavior, which often is tightly related to a family of programming languages.

Paradigm= Programming styles + Certain Programming Techniques

The term paradigm refers to a pattern of thought that guides a collection of related activities. Thus, programming paradigms can be thought of as a pattern of problem solving thought that underlies a particular genre of programs and languages.

There are four distinct and fundamental programming paradigms:

1. imperative programming;
2. object-oriented programming;
3. functional programming; and
4. logic programming.

There are some programming languages that were intentionally designed to support more than one paradigm. For example, C++ is a hybrid of imperative and object-oriented language.

Imperative Programming

Imperative programming is the oldest paradigm. It is grounded in the classic “von Neumann-Ekert” model of computation.

For imperative programming, procedural abstraction is the essential building block. By procedural abstraction we mean assignments, loops, sequences, conditional statements, and exception handling.

The leading languages in this domain are Cobol, Fortran, C, Ada and Perl.

Object Oriented Programming

Object Oriented Programming (OOP) is a model based on the collection of objects that interact with each other by passing messages that transform their state. Message passing allows the data objects to become active rather than passive as in imperative paradigms.

Object classification, inheritance and message passing are the fundamental building blocks for object oriented programming paradigm.
The major languages in this paradigm are Smalltalk, C++, Java, and C#.

Functional Programming
Functional programming models a computational problem as a collection of mathematical functions, each with an input (domain) and a result (range) spaces.

Functions in program interact and combine with each other using functional composition, conditionals and recursion.

Major functional programming languages are Lisp, Scheme, Haskell..

Logic Programming
Logic programming or declarative programming allows a program to model a problem by declaring what outcome the program should accomplish rather than how it should be accomplished. These languages are also called rule-based languages because program's declarations look more like a set of rules or constraints on the problem rather than a set of commands to be carried out.

The major languages in logic programming paradigm are Prolog.
Beyond these four paradigms, several key topics such as Event-Handling, Concurrency, Distributed programming, Database programming are other key concepts. Table 1.1 depicts four paradigms, their key features and base languages.

Paradigm	Key Features	Base Languages
Event-Driven (Visual Programming)	✓ Programming is based on the set of anticipated events ✓ The base system recognizes the events as they occur and coordinates the necessary responses ✓ This paradigm is very useful in developing a good user interface	Visual Basic Visual C++
Concurrent Programming (Parallel Programming)	✓ This paradigm supports multi-threading i.e. segments of same program can execute concurrently and synchronization i.e. facilitates cooperation amongst the several threads	Concurrent Pascal ParC PARLOG Occam

Distributed Programming (Network and internet programming)	✓ Synchronization and Semantics for message passing from the core support for implementing ✓ Remote Procedure Call (RPC) or Remote Method Invocation (RMI)	Ajax JavaScript Java
Database Programming (Structured Query Languages)	✓ This paradigm provides a structured way of framing the query on RDBMS ✓ It also provides the framework for verifying and validating the query results	SQL MySql

Table 1.1 Description of four paradigms

Features of the main programming paradigms are in the table 1.2.

Paradigm	Key concept	Program	Program execution	Result
Imperative	Command (instruction)	Sequence of commands	Execution of commands	Final state of computer memory
Functional	Function	Collection of functions	Evaluation of functions	Value of the main function
Logic	Predicate	Logic formulas: axioms and a theorem	Logic proving of the theorem	Failure or Success of proving
Object-oriented	Object	Collection of classes of objects	Exchange of messages between the objects	Final state of the objects' states

Table 1.2 Features of the main programming paradigms

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

This unit took you through the basic concept of paradigms of programming. The main idea of a programming paradigm can be formulated as following: a programming paradigm is a combination of programming styles and certain programming techniques.
There are four distinct and fundamental programming paradigms: imperative programming, object-oriented programming, functional programming, logic programming.

5.0 Summary

1. Programming techniques of traditional imperative paradigm essentially differ from techniques of nontraditional ones – functional and logic. They have different scopes of applicability, and for this reason necessity to integrate techniques of different paradigms often arises in programming projects.
2. Profound education in computer science implies acquirement of programming techniques of all main paradigms, and usual learning of modern programming languages should be complemented by learning of programming paradigms and their base programming techniques.

6.0 Tutor-Marked Assignment (TMA)

1. How many base paradigms exist?
2. Name major programming paradigms.
3. Is Object a key concept of Object-oriented Paradigm? Yes or No?
4. The major language in logic programming paradigm is Prolog? True or False?
5. Programming paradigm means combination of programming styles and certain programming techniques. Yes or NO?

7.0 References/Further Reading

http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms-book.html
http://daitanmarks.sourceforge.net/or/squeak/squeak tutorial.html

Unit 2

Overview of Programming Languages and the Compilation Process

Contents

1.0 Introduction

In this unit we are going to discuss various programming languages and the compilation process. There are many more languages to use with computers than any person, project or organization might need. In the earlier days of computing, programming languages were created because they were needed. Languages of any of the types include features and libraries to solve problems of all types, but time can be saved by using a language which expresses the class of problems at hand cleanly and efficiently. Compilation may be so-called intermediate languages, i.e. languages which define virtual machines.

2.0 Learning Outcomes

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

1. define a programming language;
2. know the types of programming languages;
3. name five stages of compilation process; and
4. know the phases of the compilation process.

3.0 Learning Contents

3.1 Types of Programming Languages

A "program" was a plan of action, again usually step-by-step, but usually intended to connote something more general than a specific algorithm. An algorithm is a plan of action for solving a problem, while a program is a specific set of instructions for a computer to carry out the algorithm.

Programs, as they are actually stored and executed in most computers, are lists of numbers, sometimes expressed as decimal numbers, but more often as binary, octal, or hexadecimal numbers. This is very inconvenient to work with. In order to simplify the process of writing programs, various tools (which are, as might be expected, themselves programs) have been designed (table 1.3).

Assemblers	Assemblers allow symbolic names to be used instead of numeric machine instructions and instead of numeric machine addresses for data
Macro Assemblers	Macro assemblers allow groups of instructions to be given names, i.e. to be

	made into macro-instructions
Compilers and Interpreters	Compilers and Interpreters allow statements more oriented to the problem domain than to the structure of computers to be used to specify a program. A compiler takes large groups of statements and converts them to machine code for subsequent execution, while an interpreter translates one statement at a time and executes it immediately.

Table 1.3 Simplify the process of writing programs, various tools

Most programming languages today are specified by rigorous syntax definitions, defining the statements of the languages in terms of simpler constructs, much as an English language sentence might be defined in terms of nouns, verbs, subjects, predicates, etc.

Overview of Languages

There are many more languages to use with computers than any person, project or organization might need. In the earlier days of computing, programming languages were created because they were needed. Fortran made computers accessible to scientists who were not prepared to deal with the intricacies of machine language programming. COBOL did the same for business programmers. Arguably, many recent languages have not been creating to meet any pressing need, but to satisfy the intellectual curiosity of their creators. Some of these "language of the month" creations will introduce new ideas which will be adopted by the community, and a few will survive and prosper, but most are, at best, of academic interest. Only time will tell which are which.

The languages of most interest for those who will have an interest in the mainstream of computer science (table 1.4).

Type	Meaning	Examples
procedural (Imperative)	von Neumann model instructions executed in sequence	Fortran, Cobol, C
logic (production)	rule-based, non-procedural	Prolog
functional (applicative)	based on mappings, non-procedural	Lisp, Scheme, ML, Haskell
object-oriented	support for encapsulation, classes, objects	Simula, Smalltalk, C++, Java
pattern-matching	combine parsing rules with procedural execution	Snobol, Awk, Perl

Table 1.4 Languages in the mainstream of computer science

In most cases, languages of any of these types include features and libraries to solve problems of all types, but time can be saved by using a language which expresses the class of problems at hand cleanly and efficiently.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Compilation Process

Translating Languages

A compiler translates major blocks of statements (source code) for subsequent execution (object code). An interpreter translates small portions of code for immediate execution. There are two computational contexts to consider (table 1.5).

Translation context	The context within which the language is translated.
Execution context	The context within which the translated statements are executed.

Table 1.5 Two computational contexts

The five stages of a compiler combine to translate a high level language to a low level language, generally closer to that of the target computer. Each stage, or sub-process, fulfills a single task and has one or more classic techniques for implementation (table 1.6).

Component	Purpose	Techniques
Lexical Analyzer	✓ Analyzes the Source Code ✓ Removes "white space" and comments ✓ Formats it for easy access (creates tokens) ✓ Tags language elements with type information ✓ Begins to fill in information in the	Linear Expressions Finite State Machines LEX

	SYMBOL TABLE **	
Syntactic Analyzer	✓ Analyzes the Tokenized Code for structure ✓ Amalgamates symbols into syntactic groups ✓ Tags groups with type information Backus-Naur Form	Top-down analyzers Bottom-up analyzers Expression analyzers YACC
Semantic Analyzer	✓ Analyzes the Parsed Code for meaning ✓ Fills in assumed or missing information ✓ Tags groups with meaning information	Attribute Grammars Ad hoc analyzers
Code Generator	✓ Linearizes the Qualified Code and produces the equivalent Object Code	Generally completed by hand-written code
Optimizer	✓ Examines the Object Code to determine whether there are more efficient means of execution	Common-subexpression elimination Loop unrolling Operator reduction etc.

Table 1.6 Five stages of a compiler combine

** The Symbol Table is the data structure that all elements of the compiler use to collect and share information about symbols and groups of symbols in the program being translated.

Each context may be a physical machine or a virtual machine created in software, and these contexts need not be the same. A common practice is to make the language or a subset of the language to define the translation context, so that the translator is written in terms of the language itself. This requires manual compilation for the first system, and cross-compilation to move from system to system.
Compilation may be so-called intermediate languages, i.e. languages which define virtual machines, such a stack-oriented machine or a simple one-address machine with an accumulator.

Compilation

- source code ==> relocatable object code (binaries)
- Linking: many relocatable binaries (modules plus libraries) ==> one relocatable binary (with all external references satisfied)
- Loading: relocatable ==> absolute binary (with all code and data references bound to the addresses occupied in memory)
- Execution: control is transferred to the first instruction of the program

At compile time (CT), absolute addresses of variables and statement labels are not known. In static languages (such as Fortran), absolute addresses are bound at load time (LT). In block-structured languages, bindings can change at run time (RT).

Phases of the Compilation Process

—Lexical analysis (scanning): the source text is broken into tokens.

—Syntactic analysis (parsing): tokens are combined to form syntactic structures, typically represented by a parse tree.

—The parser may be replaced by a syntax-directed editor, which directly generates a parse tree as a product of editing.

—Semantic analysis: intermediate code is generated for each syntactic structure.

—Type checking is performed in this phase. Complicated features such as generic declarations and operator overloading (as in Ada and C++) are also processed.

—Machine-independent optimization: intermediate code is optimized to improve efficiency.

—Code generation: intermediate code is translated to relocatable object code for the target machine.

—Machine-dependent optimization: the machine code is optimized.

On some systems (e.g., C under Unix), the compiler produces assembly code, which is then translated by an assembler.

Type checking is performed in this phase. Complicated features such as generic declarations and operator overloading (as C++) are also processed.

Machine-independent optimization: intermediate code is optimized to improve efficiency.

Code generation: intermediate code is translated to reloadable object code for the target machine.

Machine-dependent optimization: the machine code is optimized.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Most programming languages are specified by rigorous syntax definitions, defining the statements of the languages in terms of simpler constructs. There are also translating Languages. A compiler translates major blocks of statements (source code) for subsequent execution (object code). An interpreter translates small portions of code for immediate execution.

5.0 Summary

A "programming language" is a language designed to describe a set of consecutive actions to be executed by a computer. Programming languages may be roughly divided into two categories: interpreted languages and compiled languages. A program written in an interpreted language requires an extra program (the interpreter) which translates the programs commands as needed. A program written in a compiled language has the advantage of not requiring an additional program to run it once it has been compiled. Furthermore, as the translation only needs to be done once, at compilation it executes much faster.

6.0 Tutor Marked Assignment (TMA)

1. An algorithm is a specific set of instructions for a computer to carry out the program: Yes or No?
2. Name phases of the compilation process.
3. Is lexical analysis the first or the last phase of the compilation process?
4. Name five stages of compilation process.

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Programming_language
http://programmers.stackexchange.com/questions/164442/human-powered-document-processing
http://www.webopedia.com/TERM/P/programming_language.html

Unit 3

Introduction to Object Oriented Programming

Contents

1.0 Introduction

Object-Oriented Programming (OOP) uses a different set of programming languages than old procedural programming languages (C, Pascal, etc.). Everything in OOP is grouped as self-sustainable "objects". In this unit we shall be discussing basic object-oriented programming concepts such as objects, classes, inheritance, data abstraction, data encapsulation, polymorphism, overloading and reusability.

2.0 Learning Outcomes

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

1. define the object oriented programming;
2. have general idea of basic object-oriented programming concepts;
3. provide an example of an object; and
4. provide an example of a class.

3.0 Learning Contents

3.1 Overview of Object Oriented Programming

Object Oriented Programming, also known as OOP, is a computer science term which is used to describe a computer application that is composed of multiple objects which are connected to each other. Traditionally, most computer programming languages were simply a group of functions or instructions.

With OOP, every object can handle data, get messages, and transfer messages to other objects. The objects will all act as independent units in their own right, and they will be responsible for carrying out a certain process.

Because the objects are not dependent on each other, OOP is seen as being more flexible than older methods of programming. It has become quite popular, and it is now used in a number of advanced software engineering projects. Many programmers feel that object oriented programming is easier for beginners to learn than previous programming methods. Because it is easier to learn, it can also be analyzed and maintained without a large amount of difficulty. However, there are some people that feel that OOP is more complicated than older programming methods. To understand object oriented programming, there are a few concepts you will need to become familiar with.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Overview of basic object-oriented programming concepts

Object oriented programming is based on the following concepts**:**

1. objects;
2. classes;
3. inheritance;
4. data Abstraction;
5. data Encapsulation;
6. polymorphism;
7. overloading; and
8. reusability.

Let us see the concept of each briefly.

What is Object?

An object can be considered as a "thing" that can perform a set of related activities. The set of activities that the object performs defines the object's behavior. In pure OOP terms an object is an instance of a class.

What is a Class?

A class is simply a representation of a type of object. It is the blueprint/ plan/ template that describe the details of an object. A class is the blueprint from which the individual objects are created. Class is composed of three things: a name, attributes (variables or fields), and operations (functions or methods). Next figure 1.1 explains it.

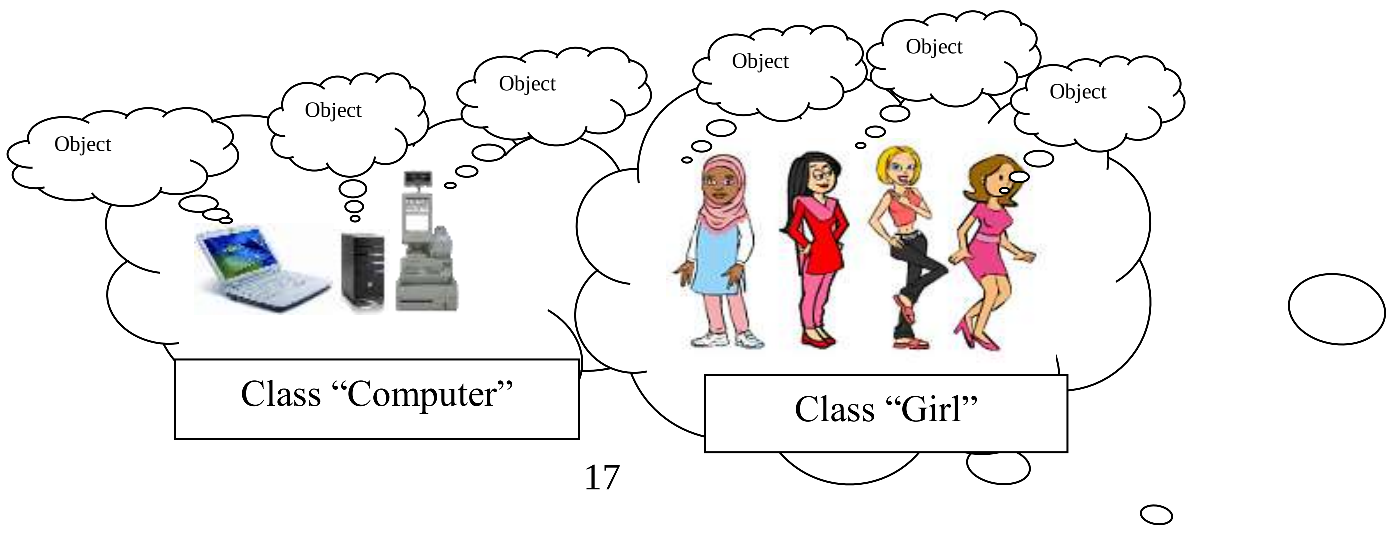

Figure 1.1 Example of class

Any blueprint of class “Computer” is object, and any blueprint of class “Girl” is object also. But each copy has individual properties and methods. Each girl has attributes. These attributes are called properties. The precise meaning of these terms depends on language /system/universe we are used.

How to identify and design a Class?
Each designer/programmer uses different techniques to identify classes. However according to Object Oriented Design Principles, there are 5 principles that you must follow when design a class (table 1.6).

№	Named	Explain
1	SRP – The Single Responsibility Principle	A class should have one, and only one, reason to change.
2	OCP – The Open Closed Principle	You should be able to extend a classes behavior, without modifying it.
3	LSP – The Liskov Substitution Principle	Derived classes must be substitutable for their base classes
4	DIP – The Dependency Inversion Principle	Depend on abstractions, not on concretions
5	ISP – The Interface Segregation Principle	Make fine grained interfaces that are client specific

Table 1.6 Principles that you must follow when design a class

Additionally to identify a class correctly, you need to identify the full list of leaf level functions/ operations/methods of the system. Then you can proceed to group each function to form classes (classes will group same types of functions/ operations). However a well-defined class must be a meaningful grouping of a set of functions and should support the re-usability while increasing expandability/ maintainability of the overall system.

In software world the concept of dividing and conquering is always recommended, if you start analyzing a full system at the start, you will find it harder to manage. So the better approach is to identify the module of the system first and then dig deep in to each module separately to seek out classes.

A software system may consist of many classes. But in any case, when you have many, it needs to be managed. Same technique can be applies to manage classes of your software system as well. In order to manage the classes of a software system, and to reduce the complexity, the system designers use several techniques, which can be grouped under four main concepts named Encapsulation, Abstraction, Inheritance, and Polymorphism.

What is Encapsulation (or information hiding)?
The encapsulation is the inclusion within a program object of all the resources need for the object to function - basically, the methods and the data. In OOP the encapsulation is mainly achieved by creating classes, the classes expose public methods and properties. The class is kind of a container or capsule or a cell, which encapsulate the set of methods, attribute and properties to provide its indented functionalities to other classes. In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the system. That idea of encapsulation is to hide how a class does it but to allow requesting what to do.

According to OOP there are several techniques, classes can use to link with each other and they are named association, aggregation, and composition. What are association, aggregation, and composition in table 1.7?

Association	relationship between two classes, where one class use another
Aggregation	the object of one class uses a part of object of other class
Composition	the object of one class uses a objects of a lot classes

Table 1.7 Base concept

So in summary, we can say that aggregation is a special kind of an association and composition is a special kind of an aggregation. (Association->Aggregation->Composition)

There are several other ways that an encapsulation can be used, as an example we can take the usage of an interface. The interface can be used to hide the information of an implemented class.

What is Abstraction?
Abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail). The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects. Abstraction is essential in the construction of programs. It places the emphasis on what an object is or does rather than how it is represented or how it works. Thus, it is the primary means of managing complexity in large programs.

What is an Abstract class?
Abstract classes, which declared with the abstract keyword, cannot be instantiated. It can only be used as a super-class for other classes that extend the abstract class. Abstract class is the concept and implementation gets completed when it is being realized by a subclass. In addition to this a class can inherit only from one abstract class (but a class may implement many interfaces) and must override all its abstract methods/ properties and may override virtual methods/ properties. Abstract classes are ideal when implementing frameworks.

This class will allow all subclass to gain access to a common exception logging module and will facilitate to easily replace the logging library.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Object-oriented programming is a way to think. It is not the classes, syntax, encapsulation, or other big words that we will learn about in the following units. The true purpose of OOP is the empowering it gives us to see our programs in a new light, a higher view, a clearer picture. It is hard to go back once you've had a taste.

Object-oriented programming was conceived to allow us to create larger projects more simply. Thus, it is very helpful in our larger projects that we do. It can also be nice in smaller projects, especially once we get the hang of it. Small projects are easily done in procedural programming though, and even once we learn how to program object-oriented we may still find ourselves using procedural at times.

5.0 Summary

1. Object is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.
2. Classes are the data types on which objects are created. So while a class is created no memory is allocated only when an object is created memory gets allocated
3. The ability to derive new classes from existing classes. A derived class (or "subclass") inherits the instance variables and methods of the base class and may add new instance variables and methods. New methods may be defined with the same names as those in the base class, in which case they override the original one.
4. Data Encapsulation is the process of combining data and functions into a single unit called class. By this method one cannot access the data directly. Data is accessible only through the functions present inside the class. Thus Data Encapsulation gave rise to the important concept of data hiding.
5. The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.
6. That is object oriented programming has the feature of allowing an existing class which is written and debugged to be used by other programmers and there by

provides a great time saving and also code efficiency to the language. Also it is possible to a have the existing class and adds new features to the existing class as pet the programmer's choice.

6.0 Tutor Marked Assignment (TMA)

1. What is object oriented programming?
2. An object is both data and function that operate on data: True or False?
3. What is known as data types? On which objects are they created?
4. What is polymorphism?

7.0 References/Further Reading

http://docs.oracle.com/javase/tutorial/java/javaOO/
http://en.wikipedia.org/wiki/Class_(computer_programming)
http://www.tutorialspoint.com/java/java_object_classes.htm
http://cnx.org/content/m11708/latest/

Module 2

Fundamentals of Objects and Classes

Unit 1

Objects and Classes

Contents

1.0 Introduction

In this unit we will move from the conceptual picture of objects and classes to a thorough study of classes and objects. You will learn that objects are closely related to classes. A class can contain variables and methods. If an object is also a collection of variables and methods, how do they differ from classes? The answer to this question will be given in this unit.

This unit will enable you to distinguish between an object and a class and be familiarized with object's variables.

2.0 Learning Outcomes

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

1. distinguish between classes and objects;
2. name the benefits of encapsulation; and
3. know and use three parts of a message.

3.0 Learning Contents

3.1 Objects, Encapsulation, Messages

In object-oriented programming we create software objects that model real world objects. Software objects are modeled after real-world objects in that they too have state and behavior. A software object maintains its state in one or more ***variables***. A variable is an item of data named by an ***identifier***. A software object implements its behavior with ***methods***. A method is a function associated with an object.

An object is a software bundle of variables and related methods.
An object is also known as an *instance*. An instance refers to a particular object. For e.g. Jim's car is an instance of a car — it refers to a particular car. Sandy Zafira is an instance of a Student.

The variables of an object are formally known as instance variables because they contain the state for a particular object or instance. In a running program, there may be many instances of an object. For e.g. there may be many Student objects.

Each of these objects will have their own instance variables and each object may have different values stored in their instance variables. For e.g. each Student object will have a different number stored in its Student Number variable.

Object diagrams show that an object’s variables make up the center, or nucleus, of the object. Methods surround and hide the object’s nucleus from other objects in the program. Packaging an object’s variables within the protective custody of its methods is called ***encapsulation***.

Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two benefits to software developers:

1. ***Modularity***: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.
2. ***Information-hiding***: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting other objects that depend on it.

Messages

Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B’s methods, object A sends a message to object B

There are three parts of a message: The three parts for the message System.out.println{‘‘Hello World’’}; are:

- The ***object*** to which the message is addressed (System.out)
- The ***name*** of the method to perform (println)
- ***Any parameters*** needed by the method (“Hello World!”)

Classes

In object-oriented software, it’s possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. A class is a software blueprint for objects. A class is used to manufacture or create objects.

The class declares the instance variables necessary to contain the state of every object. The class would also declare and provide implementations for the instance methods necessary to operate on the state of the object.

A class is a blueprint that defines the variables and the methods common to all objects of a certain kind.

A class is a kind of factory for constructing objects. The non-static parts of the class specify, or describe, what variables and methods the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

A ***class*** is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An ***object*** is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifies.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

1. private members of a class are accessible only from within other members of the same class or from their *friends*;
2. protected members are accessible from members of their same class and from their friends, but also from members of their derived class; and
3. Finally, public members are accessible from anywhere where the object is visible.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Object-oriented languages provide more powerful and flexible encapsulation mechanisms for restricting interactions between components. When used carefully, these mechanisms allow the software developers to restrict the interactions between components to those that are required to achieve the desired functionality. The management of component interactions is an important part of software design. It has a significant impact on the ease of understanding, testing, and maintenance of components.

5.0 Summary

An object is a software bundle of variables and related methods. The variables of an object are formally known as instance variables because they contain the state for a particular object or instance. Packaging an object's variables within the protective custody of its methods is called encapsulation. Software objects interact and communicate with each other by sending messages to each other.

In object-oriented software, it's possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. A class is a software blueprint for objects. A class is used to manufacture or create objects.

6.0 Tutor-Marked Assignment (TMA)

1. What is the difference between classes and objects? Give examples.
2. Name the benefits of encapsulation.
3. Three parts of a message are object, name, any class: True or False?

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)
http://en.wikipedia.org/wiki/Encapsulation
http://en.wikipedia.org/wiki/Object-oriented_programming
http://en.wikipedia.org/wiki/Encapsulation_(networking)

Unit 2

Class Members and Instance members

Contents

1.0 Introduction

In this unit you will learn what classes are made of. In many object-oriented languages, classes are objects in their own right (to a greater or lesser extent, depending on the language). Their primary function is as factories for objects in the category. A class can also hold data variable and constants that are shared by all of its objects and can handle methods that deal with an entire class rather than an individual object. These members are called class members or, in some languages (C++ and Java, for example), static members. The members that are associated with objects are called instance members.

2.0 Learning Outcomes

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

1. define static and instance class members;
2. know the purpose of use of static and instance class members; and
3. know and use static and instance class members.

3.0 Learning Contents

A class definition is made of *members* or components. A class can define variables (or fields) and methods. Variables and methods can be static or non-static i.e. they are defined with or without the keyword ***static***.

```
static double lastStudentNumber;// a s t a t i c member / v a r i a b l e / f i e l d
double studentNumber;           // a non−s t a t i c v a r i a b l e
static void printLastNumber() {...}      // a s t a t i c member / method
void printNumber() {...}                 // a non−s t a t i c method
```

The non-static members of a class (variables and methods) are also known as *instance* variables and methods while the non-static members are also known as class variables and class methods. Each instance of a class (each object) gets its own copy of all the instance variables defined in the class. When you create an instance of a class, the system allocates enough memory for the object and all its instance variables.

In addition to instance variables, classes can declare class variables. A class variable contains information that is shared by all instances (objects) of the class. If one object changes the variable, it changes for all other objects of that type. e.g. A Student number generator in a New Student class.

You can invoke a class method directly from the class, whereas you must invoke instance methods on a particular instance. e.g. The methods in the Math class are static and can be invoked without creating an instance of the Math class for e.g. we can say Math.sqrt(x).

Consider a simple class whose job is to group together a few static member variables for example a class could be used to store information about the person who is using the program:
class UserData { static String name; static int age; }

In programs that use this class, there is one copy each of the variables UserData.name and UserData.age. There can only be one "user," since we only have memory space to store data about one user. The class, UserData, and the variables it contains exist as long as the program runs. Now, consider a similar class that includes non-static variables:
class PlayerData { String name; int age; }

In this case, there is no such variable as PlayerData.name or PlayerData.age, since name and age are not static members of PlayerData. There is nothing much in the class except the potential to create objects. But, it's a lot of potential, since it can be used to create any number of objects. Each object will have its own variables called name and age. There can be many "players" because we can make new objects to represent new players on demand. A program might use this class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new PlayerData object can be created to represent that player. If a player leaves the game, the PlayerData object that represents that player can be destroyed. A system of objects in the program is being used to dynamically model what is happening in the game. You can't do this with "static" variables.

An object that belongs to a class is said to be an instance of that class and the variables that the object contains are called instance variables. The methods that the object contains are called instance methods.

For example, if the PlayerData class, is used to create an object, then that object is an instance of the PlayerData class, and name and age are instance variables in the object. It is important to remember that the class of an object determines the types of the instance variables; however, the actual data is contained inside the individual objects, not the class. Thus, each object has its own set of data.

The source code for methods is defined in the class yet it's better to think of the instance methods as belonging to the object, not to the class. The non-static methods in the class merely specify the instance methods that every object created from the class will contain. For example a draw() method in two different objects do the same thing in the sense that they both draw something. But there is a real difference between the two methods—the things that they draw can be different. You might say that the method definition in the class specifies what type of behavior the objects will have, but the specific behavior can vary from object to object, depending on the values of their instance variables.

The static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static. However, it is possible to mix static and non-static members in a single class. The “static” definitions in the source code specify the things that are part of the class itself, whereas the non-static definitions in the source code specify things that will become part of every instance object that is created from the class. Static member variables and static member methods in a class are sometimes called class variables and class methods, since they belong to the class itself, rather than to instances of that class.

It is a good idea to look at a specific example to see how classes and objects work. Consider this extremely simplified version of a Student class, which could be used to store information about students taking a course:

```
public class Student {
public String name;     / / Student ’ s name . p u b l i c double t e st 1 ,
test2, test3;                       / / Grades on t h r e e t e s t s .
public double getAverage() { / / compute average t e s t grade r e t u r n
(test1 + test2 + test3) / 3; }
}                                   / / end of c l a s s Student
```

None of the members of this class are declared to be static, so the class exists only for creating objects. This class definition says that any object that is an instance of the Student class will include instance variables named name, test1, test2, and test3, and it will include an instance method named getAverage(). The names and tests in different objects will generally have different values. When called for a particular student, the method getAverage() will compute an average using that student’s test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In JAVA, for example, a class is a type, similar to the built-in types such as int and boolean. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a method. For example, a program could define a variable named std of type Student with the statement
Student std; However, declaring a variable does not create an object!

You should think of objects as floating around independently in the computer’s memory. In fact, there is a special portion of memory called the heap where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a reference or pointer to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object.

In a program, objects are created using an operator called new, which creates an object and returns a reference to that object. For example, assuming that std is a variable of type Student, declared as above, the assignment statement *std = new Student();* would create a new object which is an instance of the class Student, and it would store a reference to that object in the variable std. The value of the variable is a reference to the object, not the object itself. It is not quite true to say that the object is the "value of the variable std". It is certainly not at all true to say that the object is "stored in the variable std." The proper terminology is that "the variable std *refers to* the object".

So, suppose that the variable std refers to an object belonging to the class Student.
That object has instance variables name, test1, test2, and test3. These instance variables can be referred to as std.name, std.test1, std.test2, and std.test3. This follows the usual naming convention that when B is part of A, then the full name of B is A.B. For example, a program might include the lines

```
System.out.println(" Hello , " + std.name + " . Your t e s t grades are: ");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which std refers. Similarly, std can be used to call the getAverage() instance method in the object by saying std.getAverage(). To print out the student's average, you could say:

```
System.out.println( " Your average i s " + std.getAverage() );
```

More generally, you could use std.name any place where a variable of type String is legal. You can use it in expressions. You can assign a value to it. You can pass it as a parameter to method. You can even use it to call methods from the String class. For example, std.name.length() is the number of characters in the student's name.

It is possible for a variable like std, whose type is given by a class, to refer to no object at all. We say in this case that std holds a null reference. The null reference is written in JAVA as "null". You can store a null reference in the variable std by saying "std = null;" and you could test whether the value of "std" is null by testing "if (std == null) . . .".

If the value of a variable is null, then it is, of course, illegal to refer to instance variables or instance methods through that variable–since there is no object, and hence no instance variables to refer to. For example, if the value of the variable st is null, then it would be illegal to refer to std.test1. If your program attempts to use a null reference illegally like this, the result is an error called *a null pointer exception*.

Let's look at a sequence of statements that work with objects:

```
Student std, std1,                      / / Decla re f o u r v a r i a b l e s of
std2, std3;                             / / t ype Student .
std = new Student();              / / C reate a new o b j e ct belonging
/ / t o the c l a s s Student , and
/ / st o r e a r ef e r e n c e t o t h a t
/ / o b j e ct i n the v a r i a b l e st d .
std1 = new Student();             / / C reate a second Student o b j e ct
/ / and st o r e a r ef e r e n c e t o
```

```
//i t i n the v a r i a b l e std1 .
std2 = std1;                              //Copy the r ef e r e n c e value i n std1
//i n t o the v a r i a b l e std2 .
std3 = null;                              //Store a null reference i n the
//v a r i a b l e std3 .
std.name = " John Smith ";          //Set value s of some i n st a n c e variables
.std1.name = "Mary Jones ";   //(Other i n st a n c e v a r i a b l e s have default
//i n i t i a l value s of zero . )
```

When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.

When the assignment “std2 = std1;” was executed, no new object was created. Instead, std2 was set to refer to the very same object that std1 refers to. This has some consequences that might be surprising. For example, std1.name and std2.name are two different names for the same variable, namely the instance variable in the object that both std1 and std2 refer to. After the string “Mary Jones” is assigned to the variable std1.name, it is also be true that the value of std2.name is “Mary Jones”. There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, “The object is not in the variable. The variable just holds a pointer to the object.”

You can test objects for equality and inequality using the operators == and !=, but here again, the semantics are different from what you are used to. The test “if (std1 == std2)”, tests whether the values stored in std1 and std2 are the same. But the values are references to objects, not objects. So, you are testing whether std1 and std2 refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether

```
std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3
 == std2.test3 && std1.name.equals(std2.name)}
```

It has been remarked previously that Strings are objects, and the strings “Mary Jones” and “John Smith” can be shown as objects. A variable of type String can only hold a reference to a string, not the string itself. It could also hold the value null, meaning that it does not refer to any string at all. This explains why using the == operator to test strings for equality is not a good idea.

The fact that variables hold references to objects, not objects themselves, has a couple of other consequences that you should be aware of. They follow logically, if you just keep in mind the basic fact that the object is not stored in the variable. The object is somewhere else; the variable points to it.

Suppose that a variable that refers to an object is declared to be final. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to the same object as long as the variable exists. However, this does not prevent the data in the object from changing. The variable is final, not the object. It's perfectly legal to say

```
final Student stu = new Student();
stu.name = " John Doe ";              / / Change data i n the o b j e ct ;
/ / The value st o r e d i n st u i s not changed !
/ / I t s t i l l r e f e r s t o the same o b j e ct .
```

Next, suppose that obj is a variable that refers to an object. Let's consider what happens when obj is passed as an actual parameter to a method. The value of obj is assigned to a formal parameter in the method, and the method is executed. The method has no power to change the value stored in the variable, obj. It only has a copy of that value. However, that value is a reference to an object. Since the method has a reference to the object, it can change the data stored in the object. After the method ends, obj still points to the same object, but the data stored in the object might have changed. Suppose x is a variable of type int and stu is a variable of type Student. Compare:

```
void dontChange(int z)              { void change(Student s) {
z = 42;                         s.name = " Fred ";
}                                       }
The lines:                              The lines:
x = 17;                         stu.name = " Jane ";
dontChange(x);                  change(stu);
System.out.println(x);          System.out.println(stu.name);
outputs the value 17.           outputs the value " Fred ".
The value of x is not           The value of stu is not changed ,
changed by the method,                  but stu.name is.
which is equivalent to          This is equivalent to
z = x;                                  s = stu;
z = 42;                         s.name = " Fred ";
```

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class. Static class members can be used to separate data and behavior that is independent of any object identity: the data and functions do not change regardless of what happens to the object. Static classes can be used when there is no data or behavior in the class that depends on object identity.

5.0 Summary

A class can be declared static, indicating that it contains only static members. Use a static class to contain methods that are not associated with a particular object. The main features of a static class are: they only contain static members; they cannot be instantiated; they are sealed; they cannot contain Instance Constructors (C# Programming Guide).

Creating a static class is therefore much the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated.

The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added.

6.0 Tutor-Marked Assignment (TMA)

1. What can be referred to static class members?
2. The non-static members of a class (variables and methods) are also known as *static* variables: YES or NO?
3. Why are static and instance class members used?

7.0 References/Further Reading

http://msdn.microsoft.com/en-us/library/79b3xss3(v=vs.80).aspx
http://www.functionx.com/cppcli/classes/Lesson12b.htm

http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr038.htm

Unit 3

Creating and Destroying Objects

Contents

1.0 Introduction

Object types in JAVA, for instance, are very different from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly constructed. For the computer, the process of constructing an object means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

2.0 Learning Outcomes

By the end of this unit you will be able to:

1. define instance variables;
2. know how to use instance variables; and
3. know and use constructors.

3.0 Learning Contents

3.1 Initializing Instance Variables

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named PairOfDice. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {
public int die1 = 3;              // Number showing on the f i r s t d i e .
public int die2 = 4;              // Number showing on the second d i e .
public void roll() {
// R o l l the d i c e by s e t t i n g each of the d i c e t o be
// a random number between 1 and 6.
die1 = (int)(Math.random()*6) + 1;
die2 = (int)(Math.random()*6) + 1;
}
} // end c l a s s Pai rOfDi ce
```

The instance variables die1 and die2 are initialized to the values 3 and 4 respectively. These initializations are executed whenever a PairOfDice object is constructed. It is important to understand when and how this happens. Many PairOfDice objects may exist. Each time one is created, it gets its own instance variables, and the assignments "die1 = 3" and "die2 = 4" are executed to fill in the values of those variables. To make this clearer, consider a variation of the PairOfDice class:

```
public class PairOfDice {
```

```
public int die1 = (int)(Math.random()*6) + 1;
public int die2 = (int)(Math.random()*6) + 1;

public void roll() {
die1 = (int)(Math.random()*6) + 1;
die2 = (int)(Math.random()*6) + 1;
}
} // end c l a s s Pai rOfDi ce
```

Here, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of static member variables, of course, the situation is quite different. There is only one copy of a static variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (int, double, etc.) are automatically initialized to zero if you provide no other values; boolean variables are initialized to false; and char variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is null. (In particular, since Strings are objects, the default initial value for String variables is null.).

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Constructors

Objects are created with the operator, new. For example, a program that wants to use a PairOfDice object could say:

```
PairOfDice dice;          // Decla re a v a r i a b l e of t ype Pai rOfDi ce .
dice = new PairOfDice();       // C o n st r u ct a new o b j e ct and st o r e a
// r ef e r e n c e t o i t i n the v a r i a b l e .
```

In this example, "new PairOfDice()" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, dice, so that after the assignment statement is executed, dice refers to the newly created object. Part of this expression, “PairOfDice()”, looks like a method call, and that is no accident. It is, in fact, a call to a special type of method called a constructor. This might puzzle you, since there is no such method in the class definition. However, every class has at least one constructor. If the programmer doesn’t write a constructor definition in a class, then the system will provide a default constructor for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other method, with three differences.

1. A constructor does not have any return type (not even void).
2. The name of the constructor must be the same as the name of the class in which it is defined.
3. The only modifiers that can be used on a constructor definition are the access modifiers public, private, and protected. (In particular, a constructor can’t be declared static.)

However, a constructor does have a method body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the PairOfDice class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:
The constructor is declared as “public PairOfDice(int val1, int val2)...”, with no return type and with the same name as the name of the class. This is how the JAVA compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression “new PairOfDice(3,4)” would create a PairOfDice object in which the values of the instance variables die1 and die2 are initially 3 and4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;                 // Decla re a v a r i a b l e of t ype Pai rOfDi ce .
dice = new PairOfDice(1,1);     // Let d i c e r e f e r t o a new Pai rOfDi ce
// o b j e ct t h a t i n i t i a l l y shows 1 , 1.
```

Now that we’ve added a constructor to the PairOfDice class, we can no longer create an object by saying “new PairOfDice()”! The system provides a default constructor for a class only if the class definition does not already include a constructor, so there is only one constructor in the class, and it requires two actual parameters. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the PairOfDice class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {
public int die1;  / / Number showing on the f i r s t d i e .
public int die2;  / / Number showing on the second d i e .
public PairOfDice() {
/ / C o n st r u ct o r . R o l l s the dice , so t h a t the y i n i t i a l l y
/ / show some random value s .
roll(); /              / C a l l the r o l l ( ) method t o r o l l the d i c e .
}
public PairOfDice(int val1, int val2) {
/ / C o n st r u ct o r . C reates a p a i r of d i c e t h a t
/ / a re i n i t i a l l y showing the value s v a l 1 and v a l 2 .
die1 = val1;               / / Assign s p e c i f i e d value s
die2 = val2;               / / t o the i n st a n c e v a r i a b l e s .
}
public void roll() {
/ / R o l l the d i c e by s e t t i n g each of the d i c e t o be
/ / a random number between 1 and 6.
die1 = (int)(Math.random()*6) + 1;
die2 = (int)(Math.random()*6) + 1;
}
} / / end c l a s s Pai rOfDi ce
```

Now we have the option of constructing a PairOfDice object with “new PairOfDice()” or with “new PairOfDice(x,y)”, where x and y are int-valued expressions. This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation “(int)(Math.random()*6)+1”, because it’s done inside the PairOfDice class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the PairOfDice class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:

```
public class RollTwoPairs {
public static void main(String[] args) {
PairOfDice firstDice;            / / Refe rs t o the f i r s t p a i r of d i c e .
firstDice = new PairOfDice();
PairOfDice secondDice;                / / Refe rs t o the second p a i r of d i c e .
secondDice = new PairOfDice();
int countRolls;            / / Counts how many time s the two p a i r s of
/ / d i c e have been r o l l e d .
int total1;                        / / T ot a l showing on f i r s t p a i r of d i c e .
int total2; /                      / T ot a l showing on second p a i r of d i c e .
countRolls = 0;
do {               / / R o l l the two p a i r s of d i c e u n t i l t o t a l s a re the same .
firstDice.roll();            / / R o l l the f i r s t p a i r of d i c e .
total1 = firstDice.die1 + firstDice.die2;        / / Get t o t a l .
System.out.println(" F i r s t pa i r comes up "            + total1);
secondDice.roll();                  / / R o l l the second p a i r of d i c e .
total2 = secondDice.die1 + secondDice.die2; / / Get t o t a l .
System.out.println(" Second pa i r comes up " + total2);
countRolls++;             / / Count t h i s r o l l .
System.out.println();    / / Blank l i n e .
} while (total1 != total2);
System.out.println(" I t took " + countRolls
+ " r o l l s u n t i l the t o t a l s were the same. ");
} / / end main ( )
} / / end c l a s s RollTwoPairs
```

Constructors are methods, but they are methods of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like static member methods, but they are not and cannot be declared to be static. In fact, according to the JAVA language specification, they are technically not members of the class at all. In particular, constructors are not referred to as "methods".

Unlike other methods, a constructor can only be called using the new operator, in an expression that has the form new class−name{parameter−list} where the parameter−list is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a method call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created. A constructor call is more complicated than an ordinary method call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. first, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type;
2. it initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used;
3. the actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor;
4. the statements in the body of the constructor, if any, are executed; and
5. a reference to the object is returned as the value of the constructor call. The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

For another example, let's rewrite the Student class. I'll add a constructor, and we'll also take the opportunity to make the instance variable, name, private.

```
public class Student {
private String name; / / Student ' s name .
public double test1, test2, test3; / / Grades on t h r e e t e s t s .
// C o n st r u ct o r f o r Student o b j e ct s−p r o v i d e s a name f o r the Student .
Student(String theName) {
name = theName;
}
// G ett e r method f o r the p r i v a t e i n st a n c e v a r i a b l e , name .
public String getName() {
return name;
}
// Compute average t e s t grade .
public double getAverage() {
return (test1 + test2 + test3) / 3;
}
} // end of c l a s s Student
```

An object of type Student contains information about some particular student. The constructor in this class has a parameter of type String, which specifies the name of that student. Objects of type Student can be created with statements such as:

std = new Student(" John Smith ");
std1 = new Student("Mary Jones ");

In the original version of this class, the value of name had to be assigned by a program after it created the object of type Student. There was no guarantee that the programmer would always remember to set the name properly. In the new version of the class, there is no way to create a Student object except by calling the constructor, and that constructor automatically sets the name. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the private modifier. Since the instance variable, name, is private, there is no way for any part of the program outside the Student class to get at the name directly. The program sets the value of name, indirectly, when it calls the constructor.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created. You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even void.

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

5.0 Summary

1. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deal location of memory for objects.
2. Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:
3. Constructors and destructors do not have return types nor can they return values. References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
4. Constructors cannot be declared with the keyword virtual.
5. Constructors and destructors cannot be declared static, const, or volatile.

6.0 Tutor-Marked Assignment (TMA)

1. What is a constructor?
2. A constructor has some return type (not even void): True or False?
3. What functionality does a destructor fulfill?

7.0 References/Further Reading

http://www.daniweb.com/software-development/cpp/threads/3138/creating-and-destroying-objects
http://en.wikipedia.org/wiki/Object_to_Be_Destroyed
http://wiki.answers.com/Q/When_object_is_created_and_destroyed_in_java

Module 3

Inheritance, Polymorphism, Abstract Classes

Unit 1

Inheritance

Contents

1.0 Introduction

A class represents a set of objects which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea.

However, something like this can be done in most programming languages. The central new idea in object-oriented programming–the idea that really distinguishes it from traditional programming–is to allow classes to express the similarities among objects that share some, but not all, of their structure and behavior. Such similarities can be expressed using inheritance and polymorphism.

2.0 Learning Outcomes

By the end of this unit you will be able to:

1. define the inheritance;
2. know about the class hierarchy; and
3. illustrate the power of inheritance with case studies.

3.0 Learning Contents

3.1 Extending Existing Classes

In day-to-day programming, especially for programmers who are just beginning to work with objects, sub classing is used mainly in one situation: There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be extended to make a subclass. The syntax for this is

```
public class Subclass−name
extends Existing−class−name {
 // Changes and a d d i t i o n s .
}
```

As an example, suppose you want to write a program that plays the card game, Blackjack. You can use the Card, Hand, and Deck classes developed previously. However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the “value” of a Blackjack hand according to the rules of the game. The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand.

The value of a numeric card such as a three or a ten is its numerical value.

The value of a Jack, Queen, or King is 10.

The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing Hand class by adding a method that computes the Blackjack value of the hand. Here's the definition of such a class:

```
public class BlackjackHand extends Hand {
/ * *
∗ Computes and r et u r n s the value of t h i s hand i n the game
∗ of B l a c k j a c k .
∗/
public int getBlackjackValue() {
int val;          / / The value computed f o r the hand .
boolean ace;          / / Thi s w i l l be s et t o t r u e i f the
/ / hand c o nt a i n s an ace .
int cards;          / / Number of ca rds i n the hand .

val = 0;
ace = false;
cards = getCardCount();
for ( int i = 0;          i < cards; i++ ) {
/ / Add the value of the i −t h ca rd i n the hand .
Card card;          / / The i −t h ca rd ;
int cardVal;          / / The b l a c k j a c k value of the i −t h ca rd .
card = getCard(i);
cardVal = card.getValue(); / / The normal value , 1 t o 13.
if (cardVal > 10) {
cardVal = 10;          / / For a Jack , Queen , o r King .
}
if (cardVal == 1) {
ace = true;          / / There i s at l e a s t one ace .
}
val = val + cardVal;
}
/ / Now, v a l i s the value of the hand , c o u nt i n g any ace as 1.
/ / I f t h e r e i s an ace , and i f changing i t s value from 1 t o
/ / 11 would lea ve the sco re l e s s than o r equal t o 21 ,
/ / then do so by adding the e xt r a 10 p o i nt s t o v a l .
if ( ace == true && val + 10 <= 21 )
val = val + 10;
return val;
} / / end g etB l a c k j a c kV a l u e ( )
} / / end c l a s s BlackjackHand
```

Since BlackjackHand is a subclass of Hand, an object of type BlackjackHand contains all the instance variables and instance methods defined in Hand, plus the new instance method named getBlackjackValue(). For example, if bjh is a variable of type BlackjackHand, then all of the following are legal: bjh.getCardCount(), bjh.removeCard(0), and bjh.getBlackjackValue(). The first two methods are defined in Hand, but are inherited by BlackjackHand.

Inherited variables and methods from the Hand class can also be used in the definition of BlackjackHand (except for any that are declared to be private, which prevents access even by subclasses). The statement "cards = getCardCount();" in the above definition of getBlackjackValue() calls the instance method getCardCount(), which was defined in Hand.

Extending existing classes is an easy way to build on previous work. We'll see that many standard classes have been written specifically to be used as the basis for making subclasses.

Access modifiers such as public and private are used to control access to members of a class. There is one more access modifier, protected, that comes into the picture when subclasses are taken into consideration. When protected is applied as an access modifier to a method or member variable in a class, that member can be used in subclasses – direct or indirect – of the class in which it is defined, but it cannot be used in non-subclasses. (There is one exception: A protected member can also be accessed by any class in the same package as the class that contains the protected member. Recall that using no access modifier makes a member accessible to classes in the same package, and nowhere else. Using the protected modifier is strictly more liberal than using no modifier at all: It allows access from classes in the same package and from subclasses that are not in the same package.)

When you declare a method or member variable to be protected, you are saying that it is part of the implementation of the class, rather than part of the public interface of the class. However, you are allowing subclasses to use and modify that part of the implementation.

For example, consider a PairOfDice class that has instance variables die1 and die2 to represent the numbers appearing on the two dice. We could make those variables private to make it impossible to change their values from outside the class, while still allowing read access through getter methods. However, if we think it possible that PairOfDice will be used to create subclasses; we might want to make it possible for subclasses to change the numbers on the dice.

For example, a GraphicalDice subclass that draws the dice might want to change the numbers at other times besides when the dice are rolled. In that case, we could make die1 and die2 protected, which would allow the subclass to change their values without making them public to the rest of the world. (An even better idea would be to define protected setter methods for the variables. A setter method could, for example, ensure that the value that is being assigned to the variable is in the legal range 1 through 6.).

Self -Assessment Questions

Please Insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Inheritance and Class Hierarchy

The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived class and base class are used instead of subclass and superclass; this is the common terminology inC++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure).

In Java, for example, to create a class named “B” as a subclass of a class named “A”, you would write

```
class B extends A {
 / / a d d i t i o n s to , and m o d i f i c a t i o n s of ,
 / / s t u f f i n h e r i t e d from c l a s s A
}
```

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as “sibling classes,” share some structures and behaviors – namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several “generations” of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small class hierarchy.

Let’s look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named Vehicle to represent all types of vehicles. Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the Vehicle class, as shown in this class hierarchy diagram:

The Vehicle class would include instance variables such as registration Number and owner and instance methods such as transferOwnership(). These are variables and methods common to all vehicles. The three subclasses of Vehicle – Car, Truck, and Motorcycle – could then be used to hold variables and methods specific to particular types of vehicles. The Car class might add an instance variable numberOfDoors, the Truck class might have numberOfAxels, and the Motorcycle class could have a boolean variable hasSidecar. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in Java program would look, in outline, like this (although in practice, they would probably be public classes, defined in separate files):

```
class Vehicle {
int registrationNumber;
Person owner; / / ( Assuming t h a t a Person c l a s s has been d ef i n e d ! )
void transferOwnership(Person newOwner) {
. . .
}
. . .
}
class Car extends Vehicle {
int numberOfDoors;
. . .
}
class Truck extends Vehicle {
int numberOfAxels;
. . .
}
class Motorcycle extends Vehicle {
boolean hasSidecar;
. . .
}
```

Suppose that myCar is a variable of type Car that has been declared and initialized with the statement Car myCar = new Car(); Given this declaration, a program could refer to myCar.numberOfDoors, since numberOfDoors is an instance variable in the class Car. But since class Car extends class Vehicle, a car also has all the structure and behavior of a vehicle. This means that myCar.registrationNumber, myCar.owner, and myCar.transferOwnership() also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type Caror Truck or Motorcycle is automatically an object of type Vehicle too. This brings us to the following Important Fact: A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.

The practical effect of this is that an object of type Car can be assigned to a variable of type Vehicle; i.e. it would be legal to say Vehicle myVehicle = myCar; or even Vehicle myVehicle = new Car();. After either of these statements, the variable myVehicle holds a reference to a Vehicle object that happens to be an instance of the subclass, Car. The object "remembers" that it is in fact a Car, and not just a Vehicle. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the instanceof operator. The test: if (myVehicle instanceof Car) ... determines whether the object referred to by myVehicle is in fact a car.

On the other hand, the assignment statement myCar = myVehicle; would be illegal because myVehicle could potentially refer to other types of vehicles that are not cars. This is similar to a problem we saw previously: The computer will not allow you to assign an int value to a variable of type short, because not every int is a short. Similarly, it will not allow you to assign a value of type Vehicle to a variable of type Car because not every vehicle is a car. As in the case of int s and shorts, the solution here is to use type-casting. If, for some reason, you happen to know that myVehicle does in fact refer to a Car, you can use the type cast (Car)myVehicle to tell the computer to treat myVehicle as if it were actually of type Car. So, you could say myCar = (Car)myVehicle; and you could even refer to ((Car)myVehicle).numberOfDoors. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

```
System.out.println(" Vehicle Data : ");
System.out.println(" Reg i s t ra t ion number: "+ myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
System.out.println(" Type of vehicle : Car ");
Car c;
c = (Car)myVehicle;
System.out.println("Number of doors : " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
System.out.println(" Type of vehicle : T ruck ");
Truck t;
t = (Truck)myVehicle;
System.out.println("Number of axe l s : " + t.numberOfAxels);
}
else if (myVehicle instanceof Motorcycle) {
System.out.println(" Type of vehicle : Motorcycle ");
Motorcycle m;
m = (Motorcycle)myVehicle;
System.out.println(" Has a sideca r : " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if myVehicle refers to an object of type Truck, then the type cast (Car)myVehicle would be an error. When this happes, an exception of type ClassCastException is thrown.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

In object-oriented programming , inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support. In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, parent classes or ancestor classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance gives rise to a hierarchy. In prototype-based programming, objects can be defined directly from other objects without the need to define any classes, in which case this feature is called differential inheritance.

5.0 Summary

1. A method defined in a class is inherited by all descendants of that class.
2. When a message is sent to an object to use method m(), any messages that m() sends will also be sent to the same object.
3. If the object receiving a message does not have a definition of the method requested, an inherited definition is invoked.
4. If the object receiving a message has a definition of the requested method, that definition is invoked.

6.0 Tutor-Marked Assignment (TMA)

The term inheritance refers to the fact that one object can inherit part or all of its structure and behavior from another object: True or False?

1. Is the class that does the inheriting said to be a subclass of the class from which it inherits?
2. Give an example of class hierarchy
3. Illustrate the power of inheritance with case study

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
http://expressionflow.com/2007/04/02/inheritance-and-class-hierarchies-in-object-oriented-programming/
http://www.clear.rice.edu/comp215/handouts/Lecture6.pdf
http://staff.science.uva.nl/~heck/JAVAcourse/ch3/s1.html

Unit 2

Polymorphism

Contents

1.0 Introduction

Polymorphism allows subclasses to have methods with the same names as methods in their superclasses. It gives the ability for a program to call the correct method depending on the type of object that is used to call it.

The term polymorphism refers to an object's ability to take different forms. It is a powerful feature of object-oriented programming. Here, we will look at two essential ingredients of polymorphic behavior:

1. The ability to define a method in a superclass, and then define a method with the same name in a subclass. When a subclass method has the same name as a superclass method, it is often said that the subclass method overrides the superclass method.

2. The ability to call the correct version of an overridden method, depending on the type of object that is used to call it. If a subclass object is used to call an overridden method, then the subclass's version of the method is the one that will execute. If a superclass object is used to call an overridden method, then the superclass's version of the method is the one that will execute.

2.0 Learning Outcomes

At the end of this unit you will be able to:
1. define polymorphism;
2. know three types of polymorphism; and
3. elucidate the difference between types of polymorphism.

3.0 Learning Contents

The word *polymorphism* comes from Greek and means *having several different forms*. This is one of the essential concepts of object-oriented programming. Where inheritance is related to classes and (their hierarchy), polymorphism is related to object methods.
In general there are three types of polymorphism:
1. overloading polymorphism;
2. Parametric polymorphism (also called *template polymorphism*); and
3. Inclusion polymorphism (also called *redefinition* or *overriding*).

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.1 Overloading Polymorphism

Overloading polymorphism is where functions of the same name exist, with similar functionality, in classes which are completely independent of each other (these do not have to be children of the object class). For example, the complex class, the image class and the link class may each have the function "display". This means that we do not need to worry about what type of object we are dealing with if all we want to do is display it on the screen.

Overloading polymorphism therefore allows us to define operators whose behaviour will vary depending on the parameters that are applied to them. Therefore it is possible, for example, to add the + operator and make it behave differently according to whether it refers to an operation between two integers (addition) or between two character strings (concatenation).

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Parametric Polymorphism

Parametric polymorphism is the ability to define several functions using the same name, but using different parameters (name and/or type). Parametric polymorphism automatically selects the correct method to be adopted according to the type of data passed in the parameter.

We can therefore define several addition() methods using the same name which calculates a sum of values.

The ***int*** addition (int, int) method would return the sum of two integers.

The ***float*** addition (float, float) would return the sum of two floats.

The ***char*** addition (char, char) would result as the sum of two characters as defined by the author.

A signature is the name and type (static) given to the arguments of a function. Therefore it is a method's signature which determines what is called on.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 Inclusion Polymorphism

The ability to redefine a method in classes that are inherited from a base class is called specialisation. One can therefore call on an object's method without having to know its intrinsic type: this is inclusion polymorphism. This makes it possible to disregard the specialised class details of an object family, by masking them with a common interface (this being the basic class).

Imagine a game of chess, with the objects king, queen, bishop, knight, rook and pawn, each inheriting the piece object.

The movement method could, using inclusion polymorphism, make the corresponding move according to the object class that is called on. This therefore allows the program to perform piece movement without having to be concerned with each piece's class.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Polymorphism is an object oriented strategy used when designing object models, to help simplify the code. At its core polymorphism is the ability to define two similar yet different objects, and to then treat the two objects as if they are the same.

5.0 Summary

1. Polymorphism is one of the primary characteristics (concept) of object-oriented programming.
2. Polymorphism is the characteristic of being able to assign a different meaning specifically, to allow an entity such as a variable, a function, or an object to have more than one form.
3. Polymorphism is the ability to process objects differently depending on their data types.
4. Polymorphism is the ability to redefine methods for derived classes.

6.0 Tutor-Marked Assignment (TMA)

1. What is polymorphism?
2. How many types of polymorphism you know?
3. Which types of polymorphism do you know?
4. Overloading polymorphism is where functions of the same name exist, with similar functionality: YES or NO?

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming
http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep
http://en.kioskea.net/contents/poo/polymorp.php3
http://www.dotnetfunda.com/articles/article1005-basic-concepts-of-oop-polymorphism.aspx

Unit 3

Abstract Classes

Contents

1.0 Introduction

Abstract classes are partial classes, in other words, they are classes that needs to be inherited by other classes to be used. The word abstract means that methods inside such class are not completely implemented, an abstract class can mark some methods as abstract, so, the child class needs to implement them according to its behavior. An abstract class works as a generalized type that needs specialization for each subtype.

2.0 Learning Outcomes

By the end of this unit you will be able to:

1. define abstract classes;
2. know and use variable this;
3. know and use variable super; and
4. elucidate on the use of constructors in subclasses.

3.0 Learning Contents

Whenever a Rectangle, Oval, or RoundRect object has to draw itself, it is the redraw() method in the appropriate class that is executed. This leaves open the question, What does the redraw() method in the Shape class do? How should it be defined?

The answer may be surprising: We should leave it blank. The fact is that the class Shape represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a redraw() method in the Shape class? Well, it has to be there, or it would be illegal to call it in the setColor() method of the Shape class, and it would be illegal to write “oneShape.redraw() ;”, where oneShape is a variable of type Shape. The compiler would complain that oneShape is a variable of type Shape and there’s no redraw() method in the Shape class. Nevertheless the version of redraw() in the Shape class itself will never actually be called. In fact, if you think about it, there can never be any reason to construct an actual object of type Shape . You can have **variables** of type Shape, but the objects they refer to will always belong to one of the subclasses of Shape.

Shape is an **abstract class**. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses. A class that is not abstract is said to be concrete. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Similarly, the redraw() method in class Shape is an **abstract** method, since it is never meant to be called. In fact, there is nothing for it to do – any actual redrawing is done by redraw() methods in the subclasses of Shape. The redraw() method in Shape has to be there. But it is there only to tell the computer that all Shapes understand the redraw message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of redraw() in the subclasses of Shape. There is no reason for the abstract redraw() in class Shape to contain any code at all.

Shape and its redraw() method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "**abstract**" to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must be provided for the abstract method in any concrete subclass of the abstract class.

Here's what the Shape class would look like as an abstract class:

```
public abstract class Shape {
Color color; / / co l o r of shape .
void setColor(Color newColor) { / / method to change the co l o r of the shape
color = newColor; / / change value of instance v a r i a b l e
redraw(); / / redraw shape , which w i l l appear i n new co l o r
}
abstract void redraw();
/ / a bs t rac t method−−must be def ined i n concrete subclasses
. . . / / more instance v a r i a b l e s and methods
} / / end of class Shape
```

Once you have declared the class to be **abstract**, it becomes illegal to try to create actual objects of type Shape, and the computer will report a syntax error if you try to do so.

Recall that a class that is not explicitly declared to be a subclass of some other class is automatically made a subclass of the standard class Object. That is, a class declaration with no "**extends**" part such as **public class** myClass { . . . is exactly equivalent to **public class** myClass **extends** Object { . . ..

This means that class Object is at the top of a huge class hierarchy that includes every other class. (Semantially, Object is an abstract class, in fact the most abstract class of all. Curiously, however, it is not declared to be **abstract** syntactially, which means that you can create objects of type Object.)

Since every class is a subclass of Object, a variable of type Object can refer to any object whatsoever, of any type. Java has several standard data structures that are designed to hold Object s, but since every object is an instance of class Object, these data structures can actually hold any object whatsoever. One example is the "ArrayList" data structure, which is defined by the class ArrayList in the package java.util. An ArrayList is simply a list of Object

s. This class is very convenient, because an ArrayList can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type Object, the list can actually hold objects of any type.

A program that wants to keep track of various Shape s that have been drawn on the screen can store those shapes in an ArrayList. Suppose that the ArrayList is named listOfShapes. A shape, oneShape for example, can be added to the end of the list by calling the instance method “listOfShapes.add(oneShape);” and removed from the list with the instance method “listOfShapes.remove(oneShape);”. The number of shapes in the list is given by the method “listOfShapes.size()”. It is possible to retrieve the ith object from the list with the call “listOfShapes.get(i)”.

(Items in the list are numbered from 0 to listOfShapes.size()−1.) However, note that this method returns an Object, not a Shape. (Of course, the people who wrote the ArrayList class didn’t even know about Shapes, so the method they wrote could hardly have a return type of Shape!) Since you know that the items in the list are, in fact, Shapes and not just Objects, you can type-cast the Object returned by

listOfShapes.get(i) to be a value of type Shape by saying:
oneShape = (Shape)listOfShapes.get(i);.

Let’s say, for example, that you want to redraw all the shapes in the list. You could do this with a simple **for** loop, which is lovely example of object-oriented programming and of polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
Shape s; / / i−th element of the l i s t , considered as a Shape
s = (Shape)listOfShapes.get(i);
s.redraw(); / /What ’ s drawn here depends on what type of shape s i s !
}
```

The sample source code file ShapeDraw.java uses an abstract Shape class and an ArrayList to hold a list of shapes. The file defines an applet in which the user can add various shapes to a drawing area. Once a shape is in the drawing area, the user can use the mouse to drag it around.

You might want to look at this file, even though you won’t be able to understand all of it at this time. Even the definitions of the shape classes are somewhat different from those that I have described in this section. (For example, the draw() method has a parameter of type Graphics. This parameter is required because of the way Java handles all drawing.) It is worthwhile to look at the definition of the Shape class and its subclasses in the source code. You might also check how an ArrayList is used to hold the list of shapes.

If you click one of the buttons along the bottom of this applet, a shape will be added to the screen in the upper left corner of the applet. The color of the shape is given by the “pop-up menu” in the lower right. Once a shape is on the screen, you can drag it around with the mouse. A shape will maintain the same front-to-back order with respect to other shapes on the screen, even while you are dragging it. However, you can move a shape out in front of all the other shapes if you hold down the shift key as you click on it.

In the applet the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as an abstract shape. The method that implements dragging, for example, works only with variables of type Shape. As the Shape is being dragged, the dragging method just calls the Shape's draw method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If you wanted to add a new type of shape to the program, you would define a new subclass of Shape, add another button to the applet, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

The Special Variable this

A static member of a class has a simple name, which can only be used inside the class. For use outside the class, it has a full name of the form **class**−name.simple−name. For example, "System.out" is a static member variable with simple name "out" in the class "System". It's always legal to use the full name of a static member, even within the class where it's defined. Sometimes it's even necessary, as when the simple name of a static member variable is hidden by a local variable of the same name.

Instance variables and instance methods also have simple names. The simple name of such an instance member can be used in instance methods in the class where the instance member is defined. Instance members also have full names, but remember that instance variables and methods are actually contained in objects, not classes. The full name of an instance member has to contain a reference to the object that contains the instance member. To get at an instance variable or method from outside the class definition, you need a variable that refers to the object. Then the full name is of the form variable−name.simple−name. But suppose you are writing the definition of an instance method in some class. How can you get a reference to the object that contains that instance method? You might need such a reference, for example, if you want to use the full name of an instance variable, because the simple name of the instance variable is hidden by a local variable or parameter.

Java, for example, provides a special, predefined variable named "this " that you can use for such purposes. The variable, this, is used in the source code of an instance method to refer to the object that contains the method. This intent of the name, this, is to refer to "this object," the one right here that this very method is in. If x is an instance variable in the same object, then this.x can be used as a full name for that variable. If otherMethod() is an instance method in the same object, then this.otherMethod() could be used to call that method. Whenever the computer executes an instance method, it automatically sets the variable, this, to refer to the object that contains the method. One common use of thisis in constructors. For example:

```
public class Student {
private String name; / / Name of the student .
public Student(String name) {
```

```
// Const ructor . Create a student wi th s p e c i f i e d name .
this.name = name;
}
 // More v a r i a b l e s and methods .
}
```

In the constructor, the instance variable called name is hidden by a formal parameter. However, the instance variable can still be referred to by its full name, this.name. In the assignment statement, the value of the formal parameter, name, is assigned to the instance variable, this.name. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There are other uses for this. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a method, as an actual parameter. In that case, you can use this as the actual parameter. For example, if you wanted to print out a string representation of the object, you could say "System.out.println(this);". Or you could assign the value of this to another variable in an assignment statement. In fact, you can do anything with this that you could do with any other variable, except change its value.

The Special Variable super
Java, for example, also defines another special variable, named "super", for use in the definitions of instance methods. The variable super is for use in a subclass. Like this, super refers to the object that contains the method. But it's forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. super doesn't know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Let's say that the class you are writing contains an instance method doSomething(). Consider the method call statement super.doSomething(). Now, super doesn't know anything about the doSomething() method in the subclass. It only knows about things in the superclass, so it tries to execute a method named doSomething() from the superclass. If there is none – if the doSomething() method was an addition rather than a modification – you'll get a syntax error.

The reason super exists is so you can get access to things in the superclass that are hidden by things in the subclass. For example, super.x always refers to an instance variable named x in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the subclass does not replace the variable of the same name in the superclass; it merely hides it. The variable from the superclass can still be accessed, using super.

When you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass overrides the method from the superclass. Again, however, super can be used to access the method from the superclass.

The major use of super is to override a method with a new method that extends the behavior of the inherited method, instead of replacing that behavior entirely. The new method can use super to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a PairOfDice class that includes a roll() method. Suppose that you want a subclass, GraphicalDice, to represent a pair of dice drawn on the computer screen. The roll() method in the GraphicalDice class should do everything that the roll() method in the PairOfDice class does. We can express this with a call to super.roll(), which calls the method in the superclass. But in addition to that, the roll() method for a GraphicalDice object has to redraw the dice to show the new values. The GraphicalDice class might look something like this:

```
public class GraphicalDice extends PairOfDice {
public void roll() {
/ / Rol l the dice, and redraw them.
super.roll(); / / Cal l the r o l l method from Pai rOfDice .
redraw(); / / Cal l a method to draw the dice .
}
 / / More s t u f f , i n c l u d i n g d e f i n i t i o n of redraw ( ) .
}
```

Note that this allows you to extend the behavior of the roll() method even if you don’t know how the method is implemented in the superclass!

Constructors in Subclasses

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do not become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don’t define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you.

This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass.
This could be a real problem if you don’t have the source code to the superclass, and don’t know how it works, or if the constructor in the superclass initializes private member variables that you don’t even have access to in the subclass.

Obviously, there has to be some fix for this, and there is. It involves the special variable, super. As the very first statement in a constructor, you can use super to call a constructor from the superclass. The notation for this is a bit ugly and misleading, and it can only be used in this one particular circumstance: It looks like you are calling super as a method (even though super is not a method and you can’t call constructors the same way you call other methods anyway). As an example, assume that the PairOfDice class has a constructor that takes two integers as parameters.

```
 public class GraphicalDice extends PairOfDice {
public GraphicalDice() { / / Const ructor f o r t h i s class .
super(3,4); / / Cal l the cons t ruc tor from the
/ / Pai rOfDice class , wi th parameters 3 , 4.
initializeGraphics(); / / Do some i n i t i a l i z a t i o n s p e c i f i c
/ / to the GraphicalDice class .
}
 / / More const ructor s , methods , v a r i a b l e s . . .
}
```

The statement "**super** (3,4);" calls the constructor from the superclass. This call must be the first line of the constructor in the subclass. Note that if you don't explicitly call a constructor from the superclass in this way, then the default constructor from the superclass, the one with no parameters, will be called automatically.

This might seem rather technical, but unfortunately it is sometimes necessary. By the way, you can use the special variable **this** in exactly the same way to call another constructor in the same class. This can be useful since it can save you from repeating the same code in several constructors.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Abstract types are an important feature in statically typed OO languages. Many dynamically typed languages have no equivalent feature (although the use of duck typing makes abstract types unnecessary); however traits are found in some modern dynamically-typed languages. Some authors argue that classes should be leaf classes (have no subtypes), or else be abstract. Abstract types are useful in that they can be used to define and enforce a protocol; a set of operations which all objects that implement the protocol must support.

5.0 Summary

In programming languages, an abstract type is a type in a nominative type system which cannot be instantiated. (However, it may have concrete subtypes that do have instances.) An abstract type may have no implementation, or an incomplete implementation. It may include abstract methods or abstract properties that are shared by its subtypes. A type that is not abstract is called a concrete type.

Abstract classes can be created, signified, or simulated in several ways:

By use of the explicit keyword abstract in the class definition, as in Java, D or C#.

By including, in the class definition, one or more abstract methods (called pure virtual functions in C++), which the class is declared to accept as part of its protocol, but for which no implementation is provided.

By inheriting from an abstract type, and not overriding all missing features necessary to complete the class definition.

In many dynamically typed languages such as Smalltalk, any class which sends a particular method to this, but doesn't implement that method, can be considered abstract. (However, in many such languages, like Objective-C, the error is not detected until the class is used, and the message returns results in an exception error message such as "Does not recognize selector: xxx" as - [NSObject doesNotRecognizeSelector:(SEL)selector] is invoked upon detection of an unimplemented method).

6.0 Tutor-Marked Assignment (TMA)

1. What is known as an abstract class?
2. When is variable this used?
3. The variable super is not used in a subclass: True or False?
4. Constructors are inherited: Yes or No?

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Abstract_type
http://www.dotnetfunda.com/interview/exam90-what-is-abstract-class.aspx
http://msdn.microsoft.com/en-us/library/k535acbf(v=vs.71).aspx

Module 4

Primitive Data Types

Unit 1:	Primitive Data Types
Unit 2:	Control Structures

Unit 1

Variables and the Primitive Types

Contents

1.0 Introduction

Names are fundamental to programming. In programs, names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules for giving names to things and the rules for using the names to work with those things. That is, the programmer must understand the syntax and the semantics of names.

2.0 Learning Outcomes

By the end of this unit you will be able to:

1. define variables;
2. know and use primitive types and literals; and
3. know and use variables in programs.

3.0 Learning Contents

According to the syntax rules of Java, a name is a sequence of one or more characters. It must begin with a letter or underscore and must consist entirely of letters, digits, and underscores. ("Underscore" refers to the character '_'.) For example, here are some legal names:

N
n
rate
x15
quite_a_long_name
HelloWorld

No spaces are allowed in identifiers; HelloWorld is a legal identifier, but "Hello World" is not. Upper case and lower case letters are considered to be different, so that HelloWorld, helloworld, HELLOWORLD, and hElloWorLD are all distinct names. Certain names are reserved for special uses in Java, and cannot be used by the programmer for other purposes. These reserved words include: class, public, static, if, else, while, and several dozen other words.

Java is actually pretty liberal about what counts as a letter or a digit. Java uses the Unicode character set, which includes thousands of characters from many different languages and different alphabets, and many of these characters count as letters or digits. However, I will be sticking to what can be typed on a regular English keyboard.

The pragmatics of naming includes style guidelines about how to choose names for things. For example, it is customary for names of classes to begin with upper case letters, while names of variables and of subroutines begin with lower case letters; you can avoid a lot of confusion by following the same convention in your own programs. Most Java programmers do not use underscores in names, although some do use them at the beginning of the names of certain kinds of variables. When a name is made up of several words, such as HelloWorld or interestRate, it is customary to capitalize each word, except possibly the first; this is sometimes referred to as camel case, since the upper case letters in the middle of a name are supposed to look something like the humps on a camel's back.

Finally, it's necessary to note that things are often referred to by compound names which consist of several ordinary names separated by periods. (Compound names are also called qualified names.) You've already seen an example: System.out.println. The idea here is that things in Java can contain other things. A compound name is a kind of path to an item through one or more levels of containment. The name System.out.println indicates that something called "System" contains something called "out" which in turn contains something called "println". Non-compound names are called simple identifiers. I'll use the term identifier to refer to any name -- simple or compound -- that can be used to refer to something in Java. (Note that the reserved words are not identifiers, since they can't be used as names for things.)

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.1 Variables

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where it is stored. In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way - to refer to data stored in memory - is called a variable.

Variables are actually rather subtle. Properly speaking, a variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box. Confusion can arise, especially for beginning programmers, because when a variable is used in a program in certain ways, it refers to the container, but when it is used in other ways, it refers to the data in the container. You'll see examples of both cases below. (In this way, a variable is something like the title, "The President of the United States." This title can refer to different people at different times, but it always refers to the same office. If I say "the President is playing basketball," I mean that Barack Obama is playing basketball. But if I say "Sarah Palin wants to be President" I mean that she wants to fill the office, not that she wants to be Barack Obama.)

In Java, the only way to get data into a variable -- that is, into the box that the variable names - is with an assignment statement. An assignment statement takes the form:
variable = expression; where expression represents anything that refers to or computes a data value. When the computer comes to an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. For example, consider the simple assignment statement
rate = 0.07; The variable in this assignment statement is rate, and the expression is the number 0.07. The computer executes this assignment statement by putting the number 0.07 in the variable rate, replacing whatever was there before. Now, consider the following more complicated assignment statement, which might come later in the same program:
*interest = rate * principal;* Here, the value of the expression "rate * principal" is being assigned to the variable interest. In the expression, the * is a "multiplication operator" that tells the computer to multiply rate times principal. The names rate and principal are themselves variables, and it is really the values stored in those variables that are to be multiplied. We see that when a variable is used in an expression, it is the value stored in the variable that matters; in this case, the variable seems to refer to the data in the box, rather than to the box itself. When the computer executes this assignment statement, it takes the value of rate, multiplies it by the value of principal, and stores the answer in the box referred to by interest. When a variable is used on the left-hand side of an assignment statement, it refers to the box that is named by the variable. (Note, by the way, that an assignment statement is a command that is executed by the computer at a certain time. It is not a statement of fact. For example, suppose a program includes the statement "rate = 0.07;". If the statement "interest = rate * principal;" is executed later in the program, can we say that the principal is multiplied by 0.07? No. The value of rate might have been changed in the meantime by another statement. The meaning of an assignment statement is completely different from the meaning of an equation in mathematics, even though both use the symbol "=".).

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Types and Literals

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule. We say that Java is a strongly typed language because it enforces this rule. There are eight so-called primitive types built into Java. The primitive types are named byte, short, int, long, float, double, char, and boolean. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The float and double types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type char holds a single character from the Unicode character set. And a variable of type boolean holds one of the two logical values: true or false.

Any data value stored in the computer's memory must be represented as a binary number, that is as a string of zeros and ones. A single zero or one is called a bit. A string of eight bits is called a byte. Memory is usually measured in terms of bytes. Not surprisingly, the byte data type refers to a single byte of memory. A variable of type byte holds a string of eight bits, which can represent any of the integers between -128 and 127, inclusive. (There are 256 integers in that range; eight bits can represent 256 - two raised to the power eight -- different values.) As for the other integer types, short corresponds to two bytes (16 bits). Variables of type short have values in the range -32768 to 32767. ***int*** corresponds to four bytes (32 bits). Variables of type int have values in the range -2147483648 to 2147483647. ***long*** corresponds to eight bytes (64 bits). Variables of type long have values in the range -9223372036854775808 to 9223372036854775807. You don't have to remember these numbers, but they do give you some idea of the size of integers that you can work with. Usually, for representing integer data you should just stick to the int data type, which is good enough for most purposes.

The float data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a float is about 10 raised to the power 38. A float can have about 7 significant digits. (So that 32.3989231134 and 32.3989234399 would both have to be rounded off to about 32.398923 in order to be stored in a variable of type float.) A double takes up 8 bytes, can range up to about 10 to the power 308, and has about 15 significant digits. Ordinarily, you should stick to the double type for real values.

A variable of type char occupies two bytes in memory. The value of a char variable is a single character such as A, *, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. When a character is typed into a program, it must be surrounded by single quotes; for example: 'A', '*', or 'x'. Without the quotes, A would be an identifier and * would be a multiplication operator. The quotes are not part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program.

A name for a constant value is called a literal. A literal is what you have to type in a program to represent a value. 'A' and '*' are literals of type char, representing the character values A and *. Certain special characters have special literals that use a backslash, \, as an "escape character". In particular, a tab is represented as '\t', a carriage return as '\r', a linefeed as '\n', the single quote character as '\'', and the backslash itself as '\\'. Note that even though you type two characters between the quotes in '\t', the value represented by this literal is a single tab character.

Numeric literals are a little more complicated than you might expect. Of course, there are the obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential form such as 1.3e12 or 12.3737e-108. The "e12" and "e-108" represent powers of 10, so that 1.3e12 means 1.3 times 1012 and 12.3737e-108 means 12.3737 times 10-108. This format can be used to express very large and very small numbers. Any numerical literal that contains a decimal point or exponential is a literal of type double. To make a literal of type float, you have to append an "F" or "f" to the end of the number. For example, "1.2F" stands for 1.2 considered as a value of type float. (Occasionally, you need to know this because the rules of Java say that you can't assign a value of type double to a variable of type float, so you might be confronted with a ridiculous-seeming error message if you try to do something like "x = 1.2;" when x is a variable of type float. You have to say "x = 1.2F;". This is one reason why I advise sticking to type double for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type byte, short, or int, depending on their size. You can make a literal of type long by adding "L" as a suffix. For example: 17L or 728476874368L. As another complication, Java allows octal (base-8) and hexadecimal (base-16) literals. I don't want to cover base-8 and base-16 in detail, but in case you run into them in other people's programs, it's worth knowing a few things: Octal numbers use only the digits 0 through 7. In Java, a numeric literal that begins with a 0 is interpreted as an octal number; for example, the literal 045 represents the number 37, not the number 45. Hexadecimal numbers use 16 digits, the usual digits 0 through 9 and the letters A, B, C, D, E, and F. Upper case and lower case letters can be used interchangeably in this context. The letters represent the numbers 10 through 15. In Java, a hexadecimal literal begins with 0x or 0X, as in 0x45 or 0xFF7A.

Hexadecimal numbers are also used in character literals to represent arbitrary Unicode characters. A Unicode literal consists of \u followed by four hexadecimal digits. For example, the character literal '\u00E9' represents the Unicode character that is an "e" with an acute accent. Java 7 introduces a couple of minor improvements in numeric literals. First of all, numeric literals in Java 7 can include the underscore character ("_"), which can be used to separate groups of digits. For example, the integer constant for one billion could be written 1_000_000_000, which is a good deal easier to decipher than 1000000000. There is no rule about how many digits have to be in each group. Java 7 also supports binary numbers, using the digits 0 and 1 and the prefix 0b (or OB). For example: 0b10110 or 0b1010_1100_1011.

For the type boolean, there are precisely two literals: true and false. These literals are typed just as I've written them here, without quotes, but they represent values, not variables. Boolean values occur most often as the values of conditional expressions. For example,
rate > 0.05 is a boolean-valued expression that evaluates to true if the value of the variable rate is greater than 0.05, and to false if the value of rate is not greater than 0.05. As you'll see oolean-valued expressions are used extensively in control structures. Of course, boolean values can also be assigned to variables of type boolean.

Java has other types in addition to the primitive types, but all the other types represent objects rather than "primitive" data values. For the most part, we are not concerned with objects for the time being. However, there is one predefined object type that is very important: the type String. A String is a sequence of characters. You've already seen a string literal: "Hello World!". The double quotes are part of the literal; they have to be typed in the program. However, they are not part of the actual string value, which consists of just the characters between the quotes. Within a string, special characters can be represented using the backslash notation. Within this context, the double quote is itself a special character. For example, to represent the string value

    I said, "Are you listening!"

    with a linefeed at the end, you would have to type the string literal:

    "I said, \"Are you listening!\"\n"

You can also use \t, \r, \\, and Unicode sequences such as \u00E9 to represent other special characters in string literals. Because strings are objects, their behavior in programs is peculiar in some respects (to someone who is not used to objects). I'll have more to say about them in the next section.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 Variables in Programs

A variable can be used in a program only if it has first been declared. A variable declaration statement is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable's name with that memory. A simple variable declaration takes the form:
type-name variable-name-or-names;

The variable-name-or-names can be a single variable name or a list of variable names separated by commas. (We'll see later that variable declaration statements can actually be somewhat more complicated than this.) Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way. For example:

```
int numberOfStudents;
        String name;
        double x, y;
        boolean isFinished;
        char firstInitial, middleInitial, lastInitial;
```

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader. For example:

```
        double principal;    // Amount of money invested.
        double interestRate; // Rate as a decimal, not percentage.
```

In this chapter, we will only use variables declared inside the main() subroutine of a program. Variables declared inside a subroutine are called local variables for that subroutine. They exist only inside the subroutine, while it is running, and are completely inaccessible from outside. Variable declarations can occur anywhere inside the subroutine, as long as each variable is declared before it is used in any expression. Some people like to declare all the variables at the beginning of the subroutine. Others like to wait to declare a variable until it is needed. My preference: Declare important variables at the beginning of the subroutine, and use a comment to explain the purpose of each variable. Declare "utility variables" which are not important to the overall logic of the subroutine at the point in the subroutine where they are first used. Here is a simple program using some variables and assignment statements:

```
/**
 * This class implements a simple program that
 * will compute the amount of interest that is
 * earned on $17,000 invested at an interest
 * rate of 0.07 for one year.  The interest and
 * the value of the investment after one year are
 * printed to standard output.
 */
        public class Interest {
            public static void main(String[] args) {
               /* Declare the variables. */
               double principal;     // The value of the investment.
             double rate;          // The annual interest rate.
             double interest;      // Interest earned in one year.
                  /* Do the computations. */
                  principal = 17000;
             rate = 0.07;
```

```
    interest = principal * rate;   // Compute the interest.
          principal = principal + interest;
         // Compute value of investment after one year, with interest.
         // (Note: The new value replaces the old value of principal.)
       /* Output the results. */
     System.out.print("The interest earned is $");
    System.out.println(interest);
    System.out.print("The value of the investment after one year is $");
    System.out.println(principal);
  } // end of main()
} // end of class Interest
```

This program uses several subroutine call statements to display information to the user of the program. Two different subroutines are used: System.out.print and System.out.println. The difference between these is that System.out.println adds a linefeed after the end of the information that it displays, while System.out.print does not. Thus, the value of interest, which is displayed by the subroutine call "System.out.println(interest);", follows on the same line after the string displayed by the previous System.out.print statement. Note that the value to be displayed by System.out.print or System.out.println is provided in parentheses after the subroutine name. This value is called a parameter to the subroutine.

A parameter provides a subroutine with information it needs to perform its task. In a subroutine call statement, any parameters are listed in parentheses after the subroutine name. Not all subroutines have parameters. If there are no parameters in a subroutine call statement, the subroutine name must be followed by an empty pair of parentheses.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined.

5.0 Summary

1. Primitive variables are places in the computer where primitive data is stored. You can think of them as boxes.
2. Each variable or box must have a data type associated with it, which describes the kind of data that it can store. Some examples of primitive types are int, double, boolean, and char.
3. Variables are created by 'declaring' them to the computer. You have to specify the name of the variable and its data type.
4. Giving a value to a variable to store is called assigning the variable. If it is the first value it is going to hold, then it is called initializing the variable.

6.0 Tutor-Marked Assignment (TMA)

1. A variable is not a name for the data itself but for a location in memory that can hold data: True or False?
2. When can a variable be used in a program?
3. It is customary for names of classes to begin with lower case letters, while names of variables and of subroutines begin with upper case letter: Yes or No?
4. How many are there so-called primitive types built into Java?

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Primitive_data_type
http://sydney.edu.au/engineering/it/~jchan3/soft1001/jme/primitive_variables/index.html
http://msdn.microsoft.com/en-us/library/k535acbf(v=vs.71).aspx

Unit 2

Control Flow Statements

Contents

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. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

2.0 Learning Outcomes

By the end of this unit you will be able to:

1. know control flow statements;
2. use if-then and if-then-else Statements; and
3. know and use the switch Statement.

3.0 Learning Contents

3.1 The if-then and if-then-else Statements

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() {
   // the "if" clause: bicycle must be moving
   if (isMoving){
      // the "then" clause: decrease current speed
      currentSpeed--;
   }
}
```

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() {
    // same as above, but without braces
    if (isMoving)
        currentSpeed--;
}
```

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.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 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.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 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 (discussed in Enum Types), the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer (discussed in Numbers and Strings).

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

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.4 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.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.5 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);
        }
     }
```

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.6 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:

```
// infinite loop
for ( ; ; ) {
    // 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.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Statements are the ``steps" of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another. We can, however, modify that sequence by using *control flow constructs* which arrange that a statement or group of statements is executed only if some condition is true or false, or executed over and over again to form a *loop*.

5.0 Summary

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. The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. The while and do-while statements continually execute a block of statements while a particular condition is true. 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. The for statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.

6.0 Tutor-Marked Assignment (TMA)

1. Each break statement terminates the enclosing switch statement: True or False?
2. What continues with the first statement following the switch block?

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Control_flow
http://www.karlin.mff.cuni.cz/network/prirucky/javatut/java/nutsandbolts/while.html
http://www.eskimo.com/~scs/cclass/notes/sx3.html

Module 5

Arrays and Strings

Unit 1:	Arrays
Unit 2:	Strings

Unit 1

Arrays

Contents

1.0 Introduction

In JAVA there are two types in programming language theory, a reference type is a data type that can only be accessed by references. Unlike objects of value types, objects of reference types cannot be directly embedded into composite objects and are always dynamically allocated. They are usually destroyed automatically after they become unreachable.
For immutable objects, the distinction between reference type to an immutable object type and value type is sometimes unclear, because a reference type variable to an immutable object behaves with the same semantics as a value type variable—for example, in both cases the "value" of the data the variable represents can only be changed by direct assignment to the variable (whereas for mutable objects, the data could be changed by modifying the object through another reference to the object).

2.0 Learning Outcomes

By the end of this unit you will be able to:

1. Define an array
2. Know and use arrays
3. Create and initialize arrays

3.0 Learning Contents

There are four kinds of reference types: class types, interface types, type variables, and array types. In this module we'll talk about 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. You've seen an example of arrays already, in the main method of the "Hello World!" application (figure 5.1). This unit discusses arrays in greater detail.

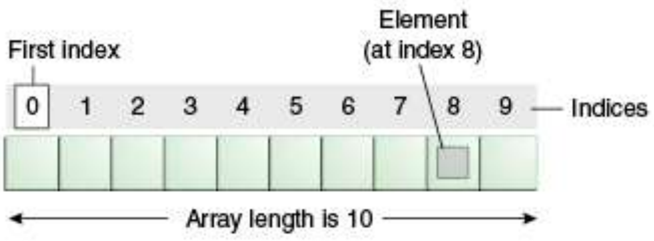

Figure 5.1 Array of ten elements

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) {
      // declares an array of integers
      int[] anArray;
      // allocates memory for 10 integers
      anArray = new int[10];
     // initialize first element
      anArray[0] = 100;
      // initialize second element
      anArray[1] = 200;
      // etc.
      anArray[2] = 300;
      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. You'll learn about the various looping constructs (for, while, and do-while) in the Control Flow unit.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.1 Declaring a Variable to Refer to an Array

The above program declares anArray with the following line of code:

```
// declares an array of integers
int[] anArray;
```

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:

```
// this form is discouraged
float anArrayOfFloats[];
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 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.

```
// create an array of integers
anArray = new int[10];
```

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 MultiDimArrayDemoprogram:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"}
        };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

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.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 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:

Caffeine.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Arrays are ordered, integer-indexed collections of any object. Array indexing starts at 0, as in C or Java. A negative index is assumed to be relative to the end of the array—that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.

5.0 Summary

A collection of data items that can be selected by indices computed at run-time, including:

Array data structure, an arrangement of items at equally spaced addresses in computer memory

Array data type, used in a programming language to specify a variable that can be indexed

Associative array, an abstract data structure model that generalizes arrays to arbitrary indices.

6.0 Tutor-Marked Assignment (TMA)

1. What is an array?
2. How is each item in an array called?
3. One way to create an array is with the new operator: Yes or No?
4. Write first element of array B[9]?

7.0 References/Further Reading

http://en.wikipedia.org/wiki/Array
http://www.ruby-doc.org/core-1.9.3/Array.html
http://www.boost.org/doc/libs/1_51_0/doc/html/array.html
http://tldp.org/LDP/abs/html/arrays.html

Unit 2

Strings

Contents

1.0 Introduction

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects. The Java platform provides the String class to create and manipulate strings.

2.0 Learning Outcomes

At the end of this unit you will be able to:

1. define strings;
2. know how to create strings;
3. concatenate strings; and
4. convert strings to numbers.

3.0 Learning Contents

3.1 Creating Strings

The most direct way to create a string is to write:

String greeting = "Hello world!";

In this case, "Hello world!" is a string literal—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a String object with its value—in this case, Hello world!.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays hello.

Note: The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 String Length

Methods used to obtain information about an object are known as accessory methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A palindrome is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string. It invokes the String method charAt(i), which returns the ith character in the string, counting from 0.

```
        public class StringDemo {
           public static void main(String[] args) {
              String palindrome = "Dot saw I was Tod";
              int len = palindrome.length();
              char[] tempCharArray = new char[len];
              char[] charArray = new char[len];

              // put original string in an
              // array of chars
              for (int i = 0; i < len; i++) {
                 tempCharArray[i] =
                    palindrome.charAt(i);
              }

              // reverse array of chars
              for (int j = 0; j < len; j++) {
                 charArray[j] =
                    tempCharArray[len - 1 - j];
              }
             String reversePalindrome =
          new String(charArray);
       System.out.println(reversePalindrome);
   }
}
```

Running the program produces this output: doT saw I was to D

To accomplish the string reversal, the program had to convert the string to an array of characters (first for loop), reverse the array into a second array (second for loop), and then convert back to a string. The String class includes a method, getChars(), to convert a string, or a portion of a string, into an array of characters so we could replace the first for loop in the program above with palindrome.getChars(0, len, tempCharArray, 0);

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 Concatenating Strings

The String class includes a method for concatenating two strings:
string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end.

You can also use the concat() method with string literals, as in:

"My name is ".concat("Rumplestiltskin");

Strings are more commonly concatenated with the + operator, as in

"Hello," + " world" + "!"

which results in

"Hello, world!"

The + operator is widely used in print statements. For example:

String string1 = "saw I was ";
System.out.println("Dot " + string1 + "Tod");

which prints

Dot saw I was Tod

Such a concatenation can be a mixture of any objects. For each object that is not a String, its toString() method is called to convert it to a String.

Note: The Java programming language does not permit literal strings to span lines in source files, so you must use the + concatenation operator at the end of each line in a multi-line string. For example:

```
String quote =
  "Now is the time for all good " +
  "men to come to the aid of their country.";
```

Breaking strings between lines using the + concatenation operator is, once again, very common in print statements.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.4 Creating Format Strings

You have seen the use of the printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of

```
System.out.printf("The value of the float " +
                  "variable is %f, while " +
                  "the value of the " +
                  "integer variable is %d, " +
                  "and the string is %s",
                  floatVar, intVar, stringVar);
```

you can write

```
String fs;
fs = String.format("The value of the float " +
                   "variable is %f, while " +
                   "the value of the " +
                   "integer variable is %d, " +
                   " and the string is %s",
                   floatVar, intVar, stringVar);
System.out.println(fs);
```

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.5 Converting Strings to Numbers

Frequently, a program ends up with numeric data in a string object—a value entered by the user, for example.

The Number subclasses that wrap primitive numeric types (Byte, Integer, Double, Float, Long, and Short) each provide a class method named valueOf that converts a string to an object of that type. Here is an example, ValueOfDemo , that gets two strings from the command line, converts them to numbers, and performs arithmetic operations on the values:

```
public class ValueOfDemo {
   public static void main(String[] args) {
      // this program requires two
      // arguments on the command line
      if (args.length == 2) {
         // convert strings to numbers
         float a = (Float.valueOf(args[0])).floatValue();
         float b = (Float.valueOf(args[1])).floatValue();
         // do some arithmetic
         System.out.println("a + b = " +
                        (a + b));
         System.out.println("a - b = " +
                        (a - b));
         System.out.println("a * b = " +
                        (a * b));
         System.out.println("a / b = " +
                        (a / b));
         System.out.println("a % b = " +
                        (a % b));
      } else {
         System.out.println("This program " +
            "requires two command-line arguments.");
      }
   }
}
```

The following is the output from the program when you use 4.5 and 87.2 for the command-line arguments:

```
a + b = 91.7
a - b = -82.7
a * b = 392.4
a / b = 0.0516055
a % b =
```

Note: Each of the Number subclasses that wrap primitive numeric types also provides a parseXXXX() method (for example, parseFloat()) that can be used to convert strings to primitive numbers. Since a primitive type is returned instead of an object, the parseFloat() method is more direct than the valueOf() method. For example, in the ValueOfDemo program, we could use:

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.6 Converting Numbers to Strings

Sometimes you need to convert a number to a string because you need to operate on the value in its string form. There are several easy ways to convert a number to a string:

```
int i;
// Concatenate "i" with an empty string; conversion is handled for you.
String s1 = "" + i;
or
// The valueOf class method.
String s2 = String.valueOf(i);
```

Each of the Number subclasses includes a class method, toString(), that will convert its primitive type to a string. For example:

```
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

The ToStringDemo example uses the toString method to convert a number to a string. The program then uses some string methods to compute the number of digits before and after the decimal point:

```
public class ToStringDemo {
      public static void main(String[] args) {
      double d = 858.48;
      String s = Double.toString(d);
            int dot = s.indexOf('.');
            System.out.println(dot + " digits " +
         "before decimal point.");
      System.out.println( (s.length() - dot - 1) +
         " digits after decimal point.");
   }
```

}

The output of this program is:

3 digits before decimal point.

2 digits after decimal point.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.7 Manipulating Characters in a String

The String class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.8 Getting Characters and Substrings by Index

You can get the character at a particular index within a string by invoking the charAt() accessor method. The index of the first character is 0, while the index of the last character is length()-1. For example, the following code gets the character at index 9 in a string:

String anotherPalindrome = "Niagara. O roar again!";

char aChar = anotherPalindrome.charAt(9);

Indices begin at 0, so the character at index 9 is 'O', as illustrated in the following figure 5.2.

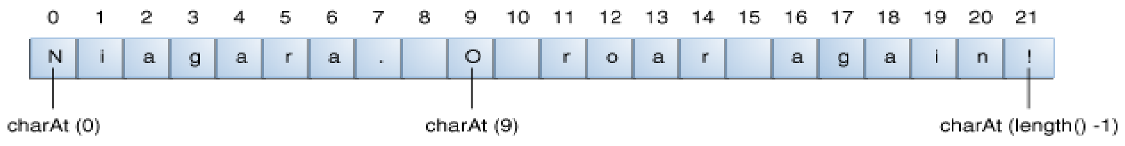

Figure 5.2 Indices begin at 0, so the character at index 9 is 'O'

If you want to get more than one consecutive character from a string, you can use the substring method. The substring method has two versions, as shown in the following table 5.1.

Method	Description
String substring(int beginIndex, int endIndex)	Returns a new string that is a substring of this string. The first integer argument specifies the index of the first character. The second integer argument is the index of the last character - 1.
String substring(int beginIndex)	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

Table 5.1 The substring Methods in the String Class

The following code gets from the Niagara palindrome the substring that extends from index 11 up to, but not including, index 15, which is the word "roar":

String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);

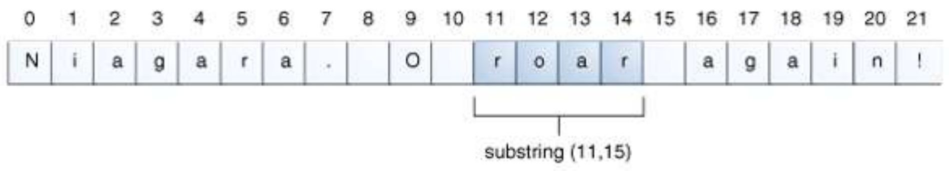

Figure 5.3 Following code.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.9 Other Methods for Manipulating Strings

Here are several other String methods for manipulating strings (table 5.2).

Method	**Description**
String[] split(String regex) String[] split(String regex, int limit)	Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly. The optional integer argument specifies the maximum size of the returned array. Regular expressions are covered in the lesson titled "Regular Expressions."
CharSequence subSequence(int beginIndex, int endIndex)	Returns a new character sequence constructed from beginIndex index up untilendIndex - 1.
String trim()	Returns a copy of this string with leading and trailing white space removed.
String toLowerCase() String toUpperCase()	Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.

Table 5.2 Other Methods in the String Class for Manipulating Strings.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.10 Searching for Characters and Substrings in a String

Here are some other String methods for finding characters or substrings within a string. The String class provides accessor methods that return the position within the string of a specific character or substring: indexOf() and lastIndexOf(). The indexOf() methods search forward from the beginning of the string, and the lastIndexOf()methods search backward from the end of the string. If a character or substring is not found, indexOf() and lastIndexOf() return -1.
The String class also provides a search method, contains, that returns true if the string contains a particular character sequence. Use this method when you only need to know that the string contains a character sequence, but the precise location isn't important.

The following table describes the various string search methods (table 5.3).

Method	**Description**
int indexOf(int ch) int lastIndexOf(int ch)	Returns the index of the first (last) occurrence of the specified character.
int indexOf(int ch, int fromIndex) int lastIndexOf(int ch, int fromIndex)	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.
int indexOf(String str) int lastIndexOf(String str)	Returns the index of the first (last) occurrence of the specified substring.
int indexOf(String str, int fromIndex) int lastIndexOf(String str, int fromIndex)	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.
boolean contains(CharSequence s)	Returns true if the string contains the specified character sequence.

Table 5.3 The Search Methods in the String Class

Note: CharSequence is an interface that is implemented by the String class. Therefore, you can use a string as an argument for the contains() method.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.11 Replacing Characters and Substrings into a String

The String class has very few methods for inserting characters or substrings into a string. In general, they are not needed: You can create a new string by concatenation of substrings you have *removed* from a string with the substring that you want to insert.
The String class does have four methods for *replacing* found characters or substrings, however. They are table 5.4:

Method	Description
String replace(char oldChar, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String replace(CharSequence target, CharSequence replacement)	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
String replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement.
String replaceFirst(String regex, String replacement)	Replaces the first substring of this string that matches the given regular expression with the given replacement.

Table 5.4 Methods in the String Class for Manipulating Strings

An Example

The following class, Filename, illustrates the use of lastIndexOf() and substring() to isolate different parts of a file name.

Note: The methods in the following Filename class don't do any error checking and assume that their argument contains a full directory path and a filename with an extension. If these methods were production code, they would verify that their arguments were properly constructed.

```
public class Filename {
   private String fullPath;
   private char pathSeparator,
               extensionSeparator;
   public Filename(String str, char sep, char ext) {
      fullPath = str;
      pathSeparator = sep;
      extensionSeparator = ext;
   }
   public String extension() {
      int dot = fullPath.lastIndexOf(extensionSeparator);
      return fullPath.substring(dot + 1);
   }
   // gets filename without extension
   public String filename() {
      int dot = fullPath.lastIndexOf(extensionSeparator);
      int sep = fullPath.lastIndexOf(pathSeparator);
```

```
            return fullPath.substring(sep + 1, dot);
        }
        public String path() {
            int sep = fullPath.lastIndexOf(pathSeparator);
            return fullPath.substring(0, sep);
        }
    }
```

Here is a program, FilenameDemo, that constructs a Filename object and calls all of its methods:

```
    public class FilenameDemo {
        public static void main(String[] args) {
            final String FPATH = "/home/user/index.html";
            Filename myHomePage = new Filename(FPATH, '/', '.');
            System.out.println("Extension = " + myHomePage.extension());
            System.out.println("Filename = " + myHomePage.filename());
            System.out.println("Path = " + myHomePage.path());
        }
    }
```

And here's the output from the program:

```
    Extension = html
    Filename = index
    Path = /home/user
```

As shown in the following figure 6.3, our extension method uses lastIndexOf to locate the last occurrence of the period (.) in the file name. Then substring uses the return value of lastIndexOf to extract the file name extension — that is, the substring from the period to the end of the string. This code assumes that the file name has a period in it; if the file name does not have a period, lastIndexOf returns -1, and the substring method throws a StringIndexOutOfBoundsException.

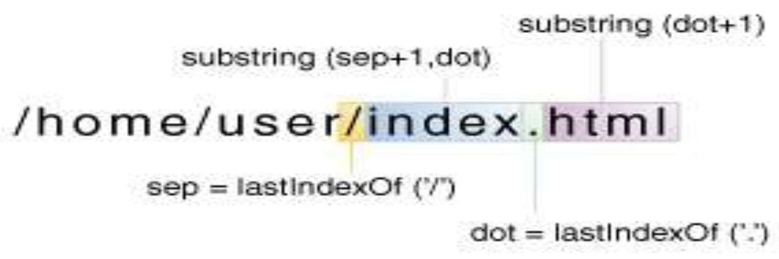

Figure 5.4 Example of extension method

Also, notice that the extension method uses dot + 1 as the argument to substring. If the period character (.) is the last character of the string, dot + 1 is equal to the length of the string, which is one larger than the largest index into the string (because indices start at 0). This is a legal argument to substring because that method accepts an index equal to, but not greater than, the length of the string and interprets it to mean "the end of the string."

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.12 Comparing Strings and Portions of Strings

The String class has a number of methods for comparing strings and portions of strings. The following table lists these methods (table 5.5).

Method	**Description**
boolean endsWith(String suffix) boolean startsWith(String prefix)	Returns true if this string ends with or begins with the substring specified as an argument to the method.
boolean startsWith(String prefix, int offset)	Considers the string beginning at the index offset, and returns true if it begins with the substring specified as an argument.
int compareTo(String anotherString)	Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
int compareToIgnoreCase(String str)	Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
boolean equals(Object anObject)	Returns true if and only if the argument is a String object that represents the same sequence of characters as this object.
boolean equalsIgnoreCase(String anotherString)	Returns true if and only if the argument is a String object that represents the same sequence of characters as this object, ignoring differences in case.
boolean regionMatches(int toffset, String	Tests whether the specified region of this

other, int ooffset, int len)	string matches the specified region of the String argument. Region is of length len and begins at the index toffset for this string and ooffsetfor the other string.
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length len and begins at the index toffset for this string and ooffsetfor the other string. The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.
boolean matches(String regex)	Tests whether this string matches the specified regular expression. Regular expressions are discussed in the lesson titled "Regular Expressions."

Table 5.5 Methods for Comparing Strings

The following program, RegionMatchesDemo, uses the regionMatches method to search for a string within another string:

```
public class RegionMatchesDemo {
    public static void main(String[] args) {
        String searchMe = "Green Eggs and Ham";
        String findMe = "Eggs";
        int searchMeLength = searchMe.length();
        int findMeLength = findMe.length();
        boolean foundIt = false;
        for (int i = 0;
             i <= (searchMeLength - findMeLength);
             i++) {
           if (searchMe.regionMatches(i, findMe, 0, findMeLength)) {
              foundIt = true;
              System.out.println(searchMe.substring(i, i + findMeLength));
              break;
           }
        }
        if (!foundIt)
            System.out.println("No match found.");
    }
}
```

The output from this program is Eggs. The program steps through the string referred to by searchMe one character at a time. For each character, the program calls the regionMatches method to determine whether the substring beginning with the current character matches the string the program is looking for.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

A string is generally understood as a data type and is often implemented as an array of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding. A string may also denote more general array data types and/or other sequential data types and structures; terms such as <data type> string or string of <data types> are sometimes used to denote strings in which the stored data represents other data types.

Depending on programming language and/or precise data type used, a variable declared to be a string may either cause storage in memory to be statically allocated for a predetermined max length or employ dynamic allocation to allow it to hold chronologically variable number of elements.

When a string appears literally in source code, it is known as a string literal and has a representation that denotes it as such.

5.0 Summary

1. Strings are a sequence of characters and are widely used in Java programming. In the Java programming language, strings are objects. The String class has over 60 methods and 13 constructors.
2. Most commonly, you create a string with a statement like
3. String s = "Hello world!";
4. Rather than using one of the String constructors.
5. The String class has many methods to find and retrieve substrings; these can then be easily reassembled into new strings using the + concatenation operator.

6. A string can be converted to a string builder using a StringBuilder constructor. A string builder can be converted to a string with the toString() method.

6.0 Tutor-Marked Assignment (TMA)

1. What is a string?
2. The String class is immutable, so that once it is created a String object cannot be changed: Yes or No?
3. How can numbers be converted into strings?
4. A program never ends up with numeric data in a string object—a value entered by the user: True or False?

7.0 References/Further Reading

http://en.wikipedia.org/wiki/String_(computer_science)
http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/oo.html
http://www.landofcode.com/java-tutorials/object-oriented-java2.php

Module 6

Algorithms

Unit 1

Algorithm Concepts

Contents

1.0 Introduction

While there is no generally accepted formal definition of "algorithm," an informal definition could be "a set of rules that precisely defines a sequence of operations." For some people, a program is only an algorithm if it stops eventually; for others, a program is only an algorithm if it stops before a given number of calculation steps.

Algorithms are essential to the way computers process data. Many computer programs contain algorithms that detail the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system. Authors who assert this thesis include Minsky (1967), Savage (1987) and Gurevich (2000):

Minsky: "But we will also maintain, with Turing . . . that any procedure which could "naturally" be called effective, can in fact be realized by a (simple) machine. Although this may seem extreme, the arguments . . . in its favor are hard to refute".

Gurevich: "...Turing's informal argument in favor of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine ... according to Savage [1987], an algorithm is a computational process defined by a Turing machine".

Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing. Stored data is regarded as part of the internal state of the entity performing the algorithm. In practice, the state is stored in one or more data structures.

For some such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

2.0 Learning Outcomes

A the end of this unit you will be able to:

1. define an algorithm;
2. know the structure of the algorithm; and
3. know and use different types of an algorithm.

3.0 Learning Contents

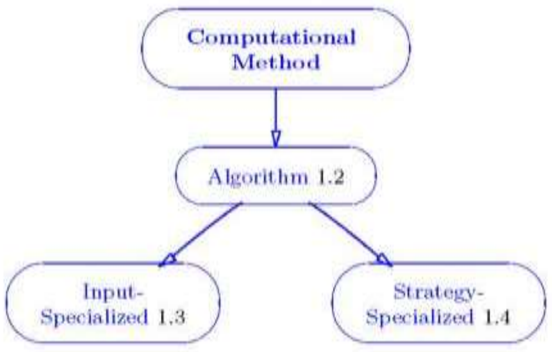

Figure 6.1 Definition of algorithm

A computational method is a method for solving a specific type of problem by means of a finite set of steps operating on inputs, which are quantities given to it before execution of the steps begins or during executing, and producing one or more outputs, which have a specified relation to the inputs. The number of steps in the method is required to be not only finite but also independent of the inputs. (The program does not grow or shrink in response to the inputs, but it might have different variations for different types of inputs.) The method is also required to be resource constrained, which means there are requirements on all operations of all steps of the method that constrain the resources (time, space) that can used in executions of the method.

Execution of steps may repeat other steps, so that although the set of steps is finite, executions of them may produce an infinite sequence of steps—finite termination is not a requirement (it is a requirement of the algorithm) concept). Some nonterminating computational methods are useful, such as computer operating systems or event-driven simulation systems. Even though the execution of such methods does not terminate, we are still generally interested in bounding the number of steps taken in producing some partial output (as in proving response-time guarantees for an operating system).

In order to bound the resources—time and space—consumed during an execution of the method, we first need bounds on the resources consumed by individual steps. This motivates the resource-constraint requirement on computational methods.
Effectiveness of a computational method is the property that all operations of all steps of the method “must be sufficiently basic that they can in principle be done exactly and in a finite length of time.” As defined here, effectiveness of computational methods follows from their resource-constraint requirement.

Definiteness of a computational method is the property that each step of the method “must be precisely defined; the actions to be carried out must be rigorously and unambiguously

specified for each case" . This includes the property that it must be unambiguous which step, if any, follows the current step in any execution of the method.
Again, resource-constraint requirements place some limitations on just how "indefinite" the steps of a method may be.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.1 Algorithm

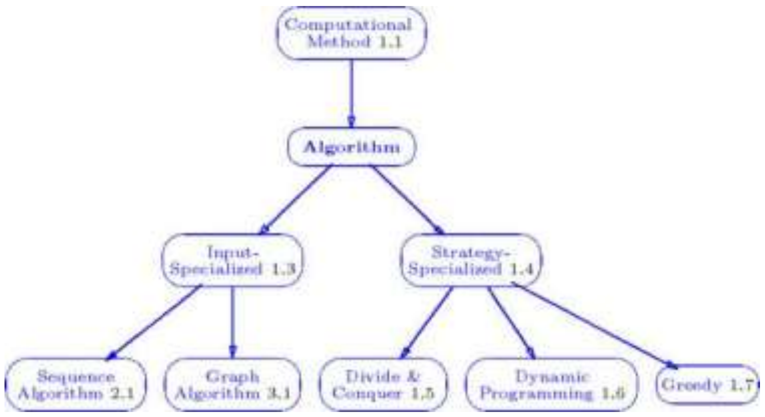

Figure 6.2 Refinement of: Computational Method

Finiteness of a computational method is the property that the number of steps in any execution of the method must be finite. The finiteness property is also called termination, and the method is said to be terminating. Algorithm is a synonym for finite computational method, a computational method with the additional property of finiteness. Every abstraction that belongs to an algorithm concept must have the termination property.
Among the abstractions belonging to a computational method concept, some might be terminating while others are nonterminating.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Algorithm Specialized by Input

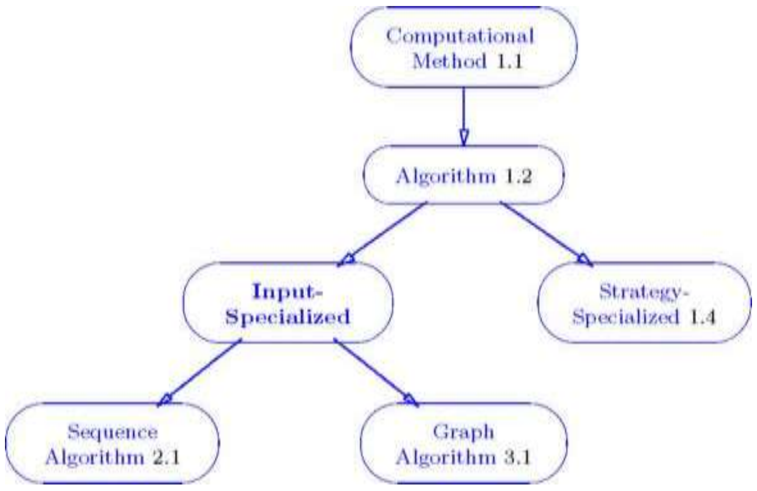

Figure 6.3 Refinement of: Algorithm

This concept is a narrowing of the algorithm concept by restrictions on the form of input. Subconcepts restrict their input to some particular domain, such as sets, graphs, or linear sequences.

Refinements: Set Algorithm, Sequence Algorithm, Polynomial Algorithm, Matrix Algorithm, Graph Algorithm.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 Algorithm Specialized by Strategy

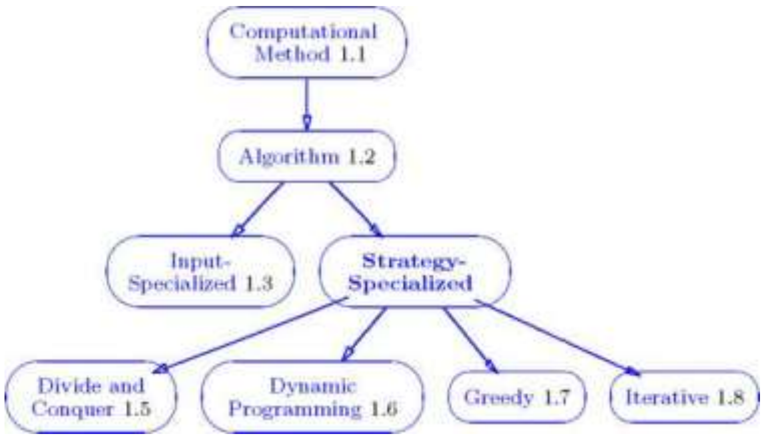

Figure 6.4 Refinement of: Algorithm

This concept is a narrowing of the algorithm concept in terms of strategies used in structuring the steps of the algorithm.

Refinements: Divide-and-Conquer Algorithm, Dynamic Programming Algorithm, Greedy Algorithm, Iterative Algorithm.

1.5 Divide-and-Conquer Algorithm

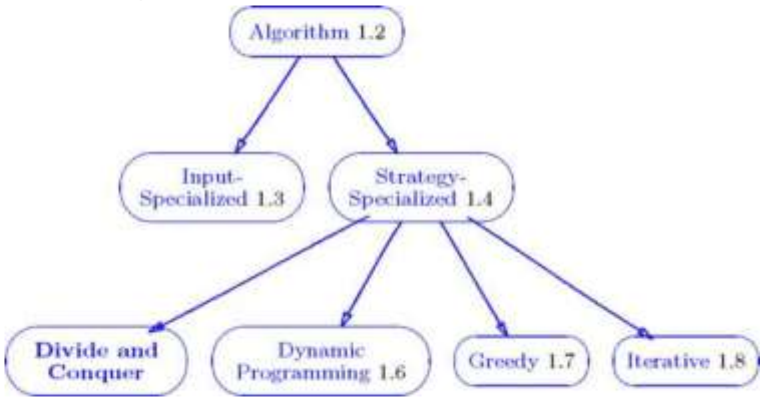

Figure 6.5 Refinement of: Algorithm

A divide-and-conquer algorithm is an algorithm whose steps are structured according to the following strategy:

1. construct the output directly and return it, if the input is simple enough. Otherwise;
2. divide the input into two or more (a finite number) of smaller inputs; and
3. recursively apply the algorithm to each of the smaller inputs produced in the first step.
4. Combine the outputs from the recursive applications to produce the output corresponding to the original input.

This concept is one of many known ways of narrowing the algorithm concept in terms of a strategy, which gives a specific structure to the steps of the algorithm.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.4 Dynamic Programming Algorithm

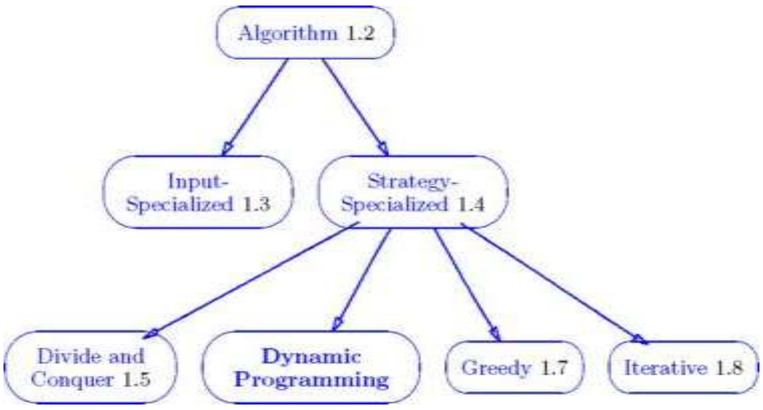

Figure 6.6 Refinement of: Algorithm Specialized by Strategy

A dynamic programming algorithm is an algorithm which solves a given problem by combining solutions to smaller subproblems. The strategy depends on two characteristics of the problem to be solved, optimal substructure and overlapping subproblems.
Optimal substructure: A problem is said to have optimal substructure if the optimal solution to the problem contains within it optimal solutions to the contained subproblems.

Overlapping subproblems: A problem exhibits overlapping subproblems if the total number of subproblems required to assemble and solve the complete problem is “small,” generally polynomial in the input size. In other words, a naive recursive (top down) approach to the problem would recompute the solution to the subproblems many times.

Taking advantage of the above properties, a dynamic programming algorithm functions in a bottom up fashion. The overall strategy can be descibed as:

1. Compute and store the solutions to all of the simplest subproblems.
2. Repeat until the full problem has been solved:

(a) Combine the solutions to the subproblems of a given size to compute and store the solutions to the next largest subproblems.

As can be seen, the solutions to the various subproblems are stored for repeated access in computing the solutions to larger subproblems. This storage is often done in some table, and dynamic programming is sometimes referred to as a tabular method.
This concept if one of many known ways of narrowing the algorithm concept in terms of a strategy , which gives a specific strategy to the steps of the algorithm.
Greedy Algorithm

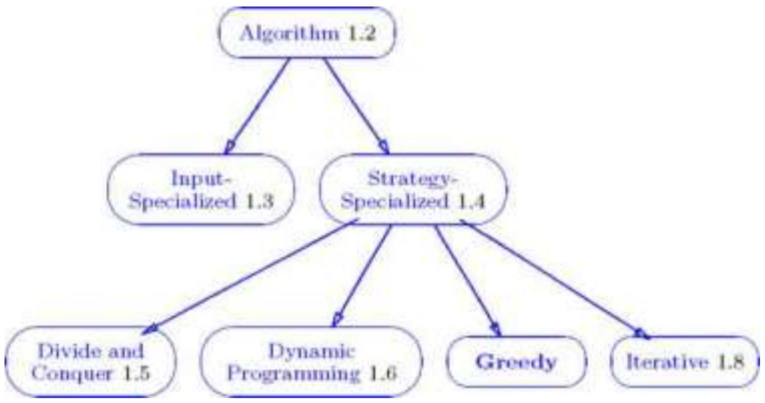

Figure 6.7 Refinement of: Algorithm Specialized by Strategy

A greedy algorithm is an algorithm which always makes locally optimal choices during its execution to produce a globally optimal solution to some problem. For such a strategy to work, the problem must exhibit the greedy choice property, and optimal substructure.
Greedy choice property: A problem exhibits the greedy choice property if a globally optimal solution can be arrived at by making locally optimal decisions at every decision point. In other words, the subproblems which would result from various decisions, and their resulting solutions to the whole problem, are irrelevant.

Optimal substructure: A problem is said to have optimal substructure if the optimal solution to the problem contains within it optimal solutions to the contained subproblems.

Having seen this, a greedy algorithm is simply an algorithm which makes a sequence of locally optimal decisions. Generally, the structure of the algorithm follows this pattern:
1. Repeat until the problem has been reduced to an empty or trivial base case.
(a) Augment the solution in some locally optimal fashion.
(b) Apply the local choice made to reduce or contract the problem.
This concept if one of many known ways of narrowing the algorithm concept in terms of a strategy, which gives a specific strategy to the steps of the algorithm.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.5 Iterative Algorithm

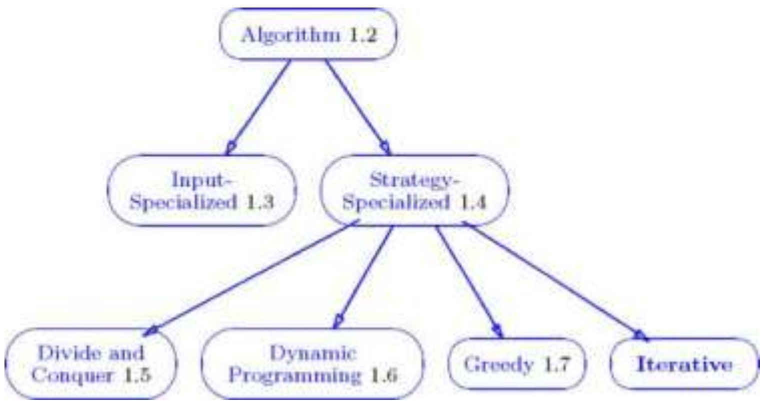

Figure 6.8 Refinement of: Algorithm Specialized by Strategy

An iterative algorithm is an algorithm which, throughout the course of execution, maintains some approximate output. As the name implies, the primary step in the strategy is to recalculate a new approximate output based on the previous approximation. In general, the approximate output grows closer to the final output (or solution) with each iteration, but this condition is not necessary.

Another important note is that an iterative algorithm must include some termination criteria. There are many useful iterative procedures which are not algorithms without a change in their formulation. Without termination, they must be considered iterative computational methods.

This concept if one of many known ways of narrowing the algorithm concept in terms of a strategy, which gives a specific strategy to the steps of the algorithm.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Currently there is a serious conceptual and technical gap between ideas emphasized in object-oriented programming. The former leads to “data-oriented” software artifacts while the latter leads to “procedure-oriented” software artifacts.

Attempts to rectify this situation by re-expressing algorithms in terms of classes and objects need to be grounded in sound fundamentals, less the cure be worse than the ailment. The machine paradigm represents one approach to the challenge of re-expressing algorithms in terms of classes and objects.

5.0 Summary

An algorithm is a method that can be used by a computer for the solution of a problem, a sequence of computational steps that transform the input into the output.

Types of algorithms:

1. Greedy
2. Divide & conquer
3. Dynamic
4. Specialized by Input
5. Strategy specialized
6. Iterative

6.0 Tutor-Marked Assignment (TMA)

1. What is an algorithm?
2. What types of algorithm do you know?
3. An iterative algorithm is an algorithm which, throughout the course of execution, maintains some approximate output: True or False?
4. A Divide-and-Conquer algorithm is an algorithm which always makes locally optimal choices during its execution to produce a globally optimal solution to some problem: Yes or No?

7.0 References/Further Reading

Donald E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Third Edition, Addison-Wesley, Reading, MA, 1997.
http://en.wikipedia.org/wiki/Algorithm
http://www.cs.rpi.edu/~musser/gp/algorithm-concepts/algorithms-screen.pdf
http://163.22.21.49/course/biology/slide2_algorithm.pdf

Unit 2

Programming Algorithms

Contents

1.0 Introduction

Programming is difficult (like many activities that are useful and worthwhile -- and like most of those activities, it can also be rewarding and a lot of fun). When you write a program, you have to tell the computer every small detail of what to do. And you have to get everything exactly right, since the computer will blindly follow your program exactly as written. How, then, do people write any but the most simple programs? It's not a big mystery, actually. It's a matter of learning to think in the right way.

2.0 Learning Outcomes

At the end of this unit you will be able to:

1. know and use pseudocode and stepwise refinement;
2. apply coding, testing, debugging.

3.0 Learning Contents

A program is an expression of an idea. A programmer starts with a general idea of a task for the computer to perform. Presumably, the programmer has some idea of how to perform the task by hand, at least in general outline. The problem is to flesh out that outline into a complete, unambiguous, step-by-step procedure for carrying out the task. Such a procedure is called an "algorithm." (Technically, an algorithm is an unambiguous, step-by-step procedure that terminates after a finite number of steps; we don't want to count procedures that go on forever.) An algorithm is not the same as a program. A program is written in some particular programming language. An algorithm is more like the idea behind the program, but it's the idea of the steps the program will take to perform its task, not just the idea of the task itself. When describing an algorithm, the steps don't necessarily have to be specified in complete detail, as long as the steps are unambiguous and it's clear that carrying out the steps will accomplish the assigned task. An algorithm can be expressed in any language, including English. Of course, an algorithm can only be expressed as a program if all the details have been filled in.

So, where do algorithms come from? Usually, they have to be developed, often with a lot of thought and hard work. Skill at algorithm development is something that comes with practice, but there are techniques and guidelines that can help.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.1 Pseudocode and Stepwise Refinement

When programming in the small, you have a few basics to work with: variables, assignment statements, and input/output routines. You might also have some subroutines, objects, or other building blocks that have already been written by you or someone else. (Input/output routines fall into this class.) You can build sequences of these basic instructions, and you can also combine them into more complex control structures such as while loops and if statements.

Suppose you have a task in mind that you want the computer to perform. One way to proceed is to write a description of the task, and take that description as an outline of the algorithm you want to develop. Then you can refine and elaborate that description, gradually adding steps and detail, until you have a complete algorithm that can be translated directly into programming language. This method is called stepwise refinement, and it is a type of top-down design. As you proceed through the stages of stepwise refinement, you can write out descriptions of your algorithm in pseudocode -- informal instructions that imitate the structure of programming languages without the complete detail and perfect syntax of actual program code.

As an example, let's see how one might develop the program from the previous section, which computes the value of an investment over five years. The task that you want the program to perform is: "Compute and display the value of an investment for each of the next five years, where the initial investment and interest rate are to be specified by the user." You might then write - or at least think - that this can be expanded as:

```
Get the user's input
Compute the value of the investment after 1 year
Display the value
Compute the value after 2 years
Display the value
Compute the value after 3 years
Display the value
Compute the value after 4 years
Display the value
Compute the value after 5 years
Display the value
```

This is correct, but rather repetitive. And seeing that repetition, you might notice an opportunity to use a loop. A loop would take less typing. More important, it would be more general: Essentially the same loop will work no matter how many years you want to process. So, you might rewrite the above sequence of steps as:

```
Get the user's input
while there are more years to process:
   Compute the value after the next year
   Display the value
```

Following this algorithm would certainly solve the problem, but for a computer we'll have to be more explicit about how to "Get the user's input," how to "Compute the value after the next year," and what it means to say "there are more years to process." We can expand the step, "Get the user's input" into

```
Ask the user for the initial investment
Read the user's response
Ask the user for the interest rate
Read the user's response
```

To fill in the details of the step "Compute the value after the next year," you have to know how to do the computation yourself. (Maybe you need to ask your boss or professor for clarification?) Let's say you know that the value is computed by adding some interest to the previous value. Then we can refine the while loop to while there are more years to process*:*

```
   Compute the interest
   Add the interest to the value
   Display the value
```

As for testing whether there are more years to process, the only way that we can do that is by counting the years ourselves. This displays a very common pattern, and you should expect to use something similar in a lot of programs: We have to start with zero years, add one each time we process a year, and stop when we reach the desired number of years. So the while loop becomes:

```
years = 0
while years < 5:
   years = years + 1
   Compute the interest
   Add the interest to the value
   Display the value
```

We still have to know how to compute the interest. Let's say that the interest is to be computed by multiplying the interest rate by the current value of the investment. Putting this together with the part of the algorithm that gets the user's inputs, we have the complete algorithm:

```
Ask the user for the initial investment
Read the user's response
```

```
Ask the user for the interest rate
Read the user's response
years = 0
while years < 5:
   years = years + 1
   Compute interest = value * interest rate
   Add the interest to the value
   Display the value
```

Finally, we are at the point where we can translate pretty directly into proper programming-language syntax. We still have to choose names for the variables, decide exactly what we want to say to the user, and so forth. Having done this, we could express our algorithm in Java as:

```
double principal, rate, interest;  // declare the variables
int years;
System.out.print("Type initial investment: ");
principal = TextIO.getlnDouble();
System.out.print("Type interest rate: ");
rate = TextIO.getlnDouble();
years = 0;
while (years < 5) {
  years = years + 1;
  interest = principal * rate;
  principal = principal + interest;
  System.out.println(principal);
}
```

This still needs to be wrapped inside a complete program, it still needs to be commented, and it really needs to print out more information in a nicer format for the user. But it's essentially the same program as the one in the previous section. (Note that the pseudocode algorithm uses indentation to show which statements are inside the loop. In Java, indentation is completely ignored by the computer, so you need a pair of braces to tell the computer which statements are in the loop. If you leave out the braces, the only statement inside the loop would be "years = years + 1;". The other statements would only be executed once, after the loop ends. The nasty thing is that the computer won't notice this error for you, like it would if you left out the parentheses around "(years < 5)". The parentheses are required by the syntax of the while statement. The braces are only required semantically. The computer can recognize syntax errors but not semantic errors.)

One thing you should have noticed here is that my original specification of the problem -- "Compute and display the value of an investment for each of the next five years" -- was far from being complete. Before you start writing a program, you should make sure you have a complete specification of exactly what the program is supposed to do. In particular, you need to know what information the program is going to input and output and what computation it is going to perform. Here is what a reasonably complete specification of the problem might look like in this example:

"Write a program that will compute and display the value of an investment for each of the next five years. Each year, interest is added to the value. The interest is computed by multiplying the current value by a fixed interest rate. Assume that the initial value and the rate of interest are to be input by the user when the program is run."

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 The 3N+1 Problem

Let's do another example, working this time with a program that you haven't already seen. The assignment here is an abstract mathematical problem that is one of my favorite programming exercises. This time, we'll start with a more complete specification of the task to be performed:

"Given a positive integer, N, define the '3N+1' sequence starting from N as follows: If N is an even number, then divide N by two; but if N is odd, then multiply N by 3 and add 1. Continue to generate numbers in this way until N becomes equal to 1. For example, starting from N = 3, which is odd, we multiply by 3 and add 1, giving N = 3*3+1 = 10. Then, since N is even, we divide by 2, giving N = 10/2 = 5. We continue in this way, stopping when we reach 1, giving the complete sequence: 3, 10, 5, 16, 8, 4, 2, 1.

"Write a program that will read a positive integer from the user and will print out the 3N+1 sequence starting from that integer. The program should also count and print out the number of terms in the sequence."

A general outline of the algorithm for the program we want is:

```
Get a positive integer N from the user.
Compute, print, and count each number in the sequence.
Output the number of terms.
```

The bulk of the program is in the second step. We'll need a loop, since we want to keep computing numbers until we get 1. To put this in terms appropriate for a while loop, we need to know when to continue the loop rather than when to stop it: We want to continue as long as the number is not 1. So, we can expand our pseudocode algorithm to:

```
Get a positive integer N from the user;
```

```
while N is not 1:
   Compute N = next term;
   Output N;
   Count this term;
Output the number of terms;
```

In order to compute the next term, the computer must take different actions depending on whether N is even or odd. We need an if statement to decide between the two cases:

```
Get a positive integer N from the user;
while N is not 1:
   if N is even:
      Compute N = N/2;
   else
      Compute N = 3 * N + 1;
   Output N;
   Count this term;
Output the number of terms;
```

We are almost there. The one problem that remains is counting. Counting means that you start with zero, and every time you have something to count, you add one. We need a variable to do the counting. (Again, this is a common pattern that you should expect to see over and over.) With the counter added, we get:

```
Get a positive integer N from the user;
Let counter = 0;
while N is not 1:
   if N is even:
      Compute N = N/2;
   else
      Compute N = 3 * N + 1;
   Output N;
   Add 1 to counter;
Output the counter;
```

We still have to worry about the very first step. How can we get a positive integer from the user? If we just read in a number, it's possible that the user might type in a negative number or zero. If you follow what happens when the value of N is negative or zero, you'll see that the program will go on forever, since the value of N will never become equal to 1. This is bad. In this case, the problem is probably no big deal, but in general you should try to write programs that are foolproof. One way to fix this is to keep reading in numbers until the user types in a positive number:

```
Ask user to input a positive number;
Let N be the user's response;
while N is not positive:
   Print an error message;
   Read another value for N;
Let counter = 0;
while N is not 1:
   if N is even:
      Compute N = N/2;
   else
      Compute N = 3 * N + 1;
   Output N;
   Add 1 to counter;
Output the counter;
```

The first while loop will end only when N is a positive number, as required. (A common beginning programmer's error is to use an if statement instead of a while statement here: "If N is not positive, ask the user to input another value." The problem arises if the second number input by the user is also non-positive. The if statement is only executed once, so the second input number is never tested, and the program proceeds into an infinite loop. With the while loop, after the second number is input, the computer jumps back to the beginning of the loop and tests whether the second number is positive. If not, it asks the user for a third number, and it will continue asking for numbers until the user enters an acceptable input.)

Here is a Java program implementing this algorithm. It uses the operators <= to mean "is less than or equal to" and != to mean "is not equal to." To test whether N is even, it uses "N % 2 == 0". All the operators used here were discussed in Section 2.5.

```
/**
 * This program prints out a 3N+1 sequence starting from a positive
 * integer specified by the user.  It also counts the number of
 * terms in the sequence, and prints out that number.
 */
 public class ThreeN1 {
       public static void main(String[] args) {
         int N;       // for computing terms in the sequence
         int counter; // for counting the terms
         TextIO.put("Starting point for sequence: ");
         N = TextIO.getlnInt();
         while (N <= 0) {
            TextIO.put("The starting point must be positive. Please try again: ");
            N = TextIO.getlnInt();
         }
         // At this point, we know that N > 0
         counter = 0;
         while (N != 1) {
             if (N % 2 == 0)
                N = N / 2;
             else
            N = 3 * N + 1;
          TextIO.putln(N);
          counter = counter + 1;
        }

        TextIO.putln();
        TextIO.put("There were ");
        TextIO.put(counter);
        TextIO.putln(" terms in the sequence.");

    }  // end of main()
  }  // end of class ThreeN1
```

As usual, you can try this out in an applet that simulates the program. Try different starting values for N, including some negative values:

Two final notes on this program: First, you might have noticed that the first term of the sequence -- the value of N input by the user -- is not printed or counted by this program. Is this an error? It's hard to say. Was the specification of the program careful enough to decide? This is the type of thing that might send you back to the boss/professor for clarification. The problem (if it is one!) can be fixed easily enough. Just replace the line "counter = 0" before the while loop with the two lines:

```
TextIO.putln(N);   // print out initial term
counter = 1;       // and count it
```

Second, there is the question of why this problem is at all interesting. Well, it's interesting to mathematicians and computer scientists because of a simple question about the problem that they haven't been able to answer: Will the process of computing the 3N+1 sequence finish after a finite number of steps for all possible starting values of N? Although individual sequences are easy to compute, no one has been able to answer the general question. To put this another way, no one knows whether the process of computing 3N+1 sequences can properly be called an algorithm, since an algorithm is required to terminate after a finite number of steps! (This discussion assumes that the value of N can take on arbitrarily large integer values, which is not true for a variable of type int in a Java program. When the value of N in the program becomes too large to be represented as a 32-bit int, the values output by the program are no longer mathematically correct.)

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 Coding, Testing, Debugging

It would be nice if, having developed an algorithm for your program; you could relax, press a button, and get a perfectly working program. Unfortunately, the process of turning an algorithm into Java source code doesn't always go smoothly. And when you do get to the stage of a working program, it's often only working in the sense that it does something. Unfortunately not what you want it to do.

After program design comes coding: translating the design into a program written in Java or some other language. Usually, no matter how careful you are, a few syntax errors will creep in from somewhere, and the Java compiler will reject your program with some kind of error message. Unfortunately, while a compiler will always detect syntax errors, it's not very good about telling you exactly what's wrong. Sometimes, it's not even good about telling you where the real error is. A spelling error or missing "{" on line 45 might cause the compiler to choke on line 105. You can avoid lots of errors by making sure that you really understand the syntax rules of the language and by following some basic programming guidelines. For example, I never type a "{" without typing the matching "}". Then I go back and fill in the statements between the braces. A missing or extra brace can be one of the hardest errors to find in a large program. Always, always indent your program nicely. If you change the program, change the indentation to match. It's worth the trouble. Use a consistent naming scheme, so you don't have to struggle to remember whether you called that variable interestrate or interestRate. In general, when the compiler gives multiple error messages, don't try to fix the second error message from the compiler until you've fixed the first one. Once the compiler hits an error in your program, it can get confused, and the rest of the error messages might just be guesses. Maybe the best advice is: Take the time to understand the error before you try to fix it. Programming is not an experimental science.

When your program compiles without error, you are still not done. You have to test the program to make sure it works correctly. Remember that the goal is not to get the right output for the two sample inputs that the professor gave in class. The goal is a program that will work correctly for all reasonable inputs. Ideally, when faced with an unreasonable input, it should respond by gently chiding the user rather than by crashing. Test your program on a wide variety of inputs. Try to find a set of inputs that will test the full range of functionality that you've coded into your program. As you begin writing larger programs, write them in stages and test each stage along the way. You might even have to write some extra code to do the testing - for example to call a subroutine that you've just written. You don't want to be faced, if you can avoid it, with 500 newly written lines of code that have an error in there somewhere.

The point of testing is to find bugs - semantic errors that show up as incorrect behavior rather than as compilation errors. And the sad fact is that you will probably find them. Again, you can minimize bugs by careful design and careful coding, but no one has found a way to avoid them altogether. Once you've detected a bug, it's time for debugging. You have to track down the cause of the bug in the program's source code and eliminate it. Debugging is a skill that, like other aspects of programming, requires practice to master. So don't be afraid of bugs. Learn from them. One essential debugging skill is the ability to read source code - the ability to put aside preconceptions about what you think it does and to follow it the way the computer does -- mechanically, step-by-step -- to see what it really does. This is hard. I can still remember the time I spent hours looking for a bug only to find that a line of code that I had looked at ten times had a "1" where it should have had an "i", or the time when I wrote a subroutine named WindowClosing which would have done exactly what I wanted except that the computer was looking for windowClosing (with a lower case "w"). Sometimes it can help to have someone who doesn't share your preconceptions look at your code.

Often, it's a problem just to find the part of the program that contains the error. Most programming environments come with a debugger, which is a program that can help you find bugs. Typically, your program can be run under the control of the debugger. The debugger allows you to set "breakpoints" in your program. A breakpoint is a point in the program where the debugger will pause the program so you can look at the values of the program's variables. The idea is to track down exactly when things start to go wrong during the program's execution. The debugger will also let you execute your program one line at a time, so that you can watch what happens in detail once you know the general area in the program where the bug is lurking.

A more traditional approach to debugging is to insert debugging statements into your program. These are output statements that print out information about the state of the program. Typically, a debugging statement would say something like
System.out.println("At start of while loop, N = " + N);

You need to be able to tell from the output where in your program the output is coming from, and you want to know the value of important variables. Sometimes, you will find that the computer isn't even getting to a part of the program that you think it should be executing. Remember that the goal is to find the first point in the program where the state is not what you expect it to be. That's where the bug is. And finally, remember the golden rule of debugging: If you are absolutely sure that everything in your program is right, and if it still doesn't work, then one of the things that you are absolutely sure of is wrong.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

In order for a computer to carry out some task, it has to be supplied with a program, which is an implementation of an algorithm. This is expressed in a computer programming language; however it is possible (and desirable) to talk and reason about algorithms in higher-level terms.

• Developing a correct algorithm can be a significant intellectual challenge – by contrast, coding it should be straightforward (although coding it well may not be).

• The most widely used notations for developing algorithms are flowcharts and pseudo-code. These are independent of the programming language to be used to implement the algorithm.

• A flowchart is a diagram containing lines representing all the possible paths through the program.

• Pseudo-code is a form of "stylised" (or "structured") natural language.

5.0 Summary

Pseudocode - informal instructions that imitate the structure of programming languages without the complete detail and perfect syntax of actual program code. After program design comes coding: translating the design into a program written in Java or some other language.

When your program compiles without error, you are still not done. You have to test the program to make sure it works correctly. Remember that the goal is not to get the right output for the two sample inputs that the professor gave in class. The goal is a program that will work correctly for all reasonable inputs. Ideally, when faced with an unreasonable input, it should respond by gently chiding the user rather than by crashing. Test your program on a wide variety of inputs.

The point of testing is to find bugs - semantic errors that show up as incorrect behavior rather than as compilation errors.

Most programming environments come with a debugger, which is a program that can help you find bugs. Typically, your program can be run under the control of the debugger. The debugger allows you to set "breakpoints" in your program.

6.0 Tutor-Marked Assignment (TMA)

1. An algorithm is the same as a program: True or False?
2. Informal instructions that imitate the structure of programming languages without the complete detail and perfect syntax of actual program code named…?
3. An algorithm can be expressed in any language, including English. Yes or No?

7.0 References/Further Reading

http://www.csi.ucd.ie/staff/jmurphy/fecs/5_algodev.pdf
http://www.cs.fsu.edu/~jestes/cop3014/notes/stepwise.html

Module 7

Software Design

Unit 1: Fundamental design concept and principles

Unit 2: introduction to design patterns

Unit 1

Design Concepts and Principles

Contents

1.0 Introduction

A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces and components.

The design process is very important. From a practical standpoint, as a labourer, one would not attempt to build a house without an approved blueprint thereby risking the structural integrity and customer satisfaction. In the same manner, the approach to building software products is no different. The emphasis in design is on quality; this phase provides us with representation of software that can be assessed for quality. Furthermore, this is the only phase in which the customer's requirements can be accurately translated into a finished software product or system. As such, software design serves as the foundation for all software engineering steps that follow regardless of which process model is being employed. Without a proper design we risk building an unstable system – one that will fail when small changes are made, one that may be difficult to test; one whose quality cannot be assessed until late in the software process, perhaps when critical deadlines are approaching and much capital has already been invested into the product.

2.0 Learning Outcomes

By the end of this unit you will know how software designed. You will know basic design concepts to provide the criteria for design quality

3.0 Learning Contents

3.1 Design - Concepts and Principles

During the design process the software specifications are transformed into design models that describe the details of the data structures, system architecture, interface, and components. Each design product is reviewed for quality before moving to the next phase of software development. At the end of the design process a design specification document is produced. This document is composed of the design models that describe the data, architecture, interfaces and components.

At the data and architectural levels the emphasis is placed on the patterns as they relate to the application to be built. Whereas at the interface level, human ergonomics often dictate the design approach employed. Lastly, at the component level the design is concerned with a "programming approach" which leads to effective data and procedural designs.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Design Specification Models

1. **Data design** – created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software. Part of the data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.
2. **Architectural design** - defines the relationships among the major structural elements of the software, the "design patterns" than can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which the architectural patterns can be applied. It is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model (DFD).
3. **Interface design** - describes how the software elements communicate with each other, with other systems, and with human users; the data flow and control flow diagrams provide much of the necessary information required.
4. **Component-level design** - created by transforming the structural elements defined by the software architecture into procedural descriptions of software components using information obtained from the process specification (PSPEC), control specification (CSPEC), and state transition diagram (STD).

   These models collectively form the design model, which is represented diagrammatically as a pyramid structure with data design at the base and component level design at the pinnacle. Note that each level produces its own documentation, which collectively form the design specifications document, along with the guidelines for testing individual modules and the integration of the entire package. Algorithm description and other relevant information may be included as an appendix.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 Design Guidelines

In order to evaluate the quality of a design (representation) the criteria for a good design should be established. Such a design should:

1. exhibit good architectural structure;
2. be modular;
3. contain distinct representations of data, architecture, interfaces, and components (modules);
4. lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns;
5. lead to components that exhibit independent functional characteristics;
6. lead to interfaces that reduce the complexity of connections between modules and with the external environment; and
7. be derived using a reputable method that is driven by information obtained during software requirements analysis.

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology and through review.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.4 Design Principles

Software design can be viewed as both a process and a model. "The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. However, it is not merely a cookbook; for a competent and successful design, the designer must use creative skill, past experience, a sense of what makes "good" software, and have a commitment to quality.

The design model is equivalent to the architect's plans for a house. It begins by representing the totality of the entity to be built (e.g. a 3D rendering of the house), and slowly refines the entity to provide guidance for constructing each detail (e.g. the plumbing layout). Similarly the design model that is created for software provides a variety of views of the computer software." – adapted from book by R Pressman.

The set of principles which has been established to aid the software engineer in navigating the design process are:

1. The design process should not suffer from tunnel vision – A good designer should consider alternative approaches. Judging each based on the requirements of the problem, the resources available to do the job and any other constraints.
2. The design should be traceable to the analysis model – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model
3. The design should not reinvent the wheel – Systems are constructed using a set of design patterns, many of which may have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. The design should minimise intellectual distance between the software and the problem as it exists in the real world – That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
5. The design should exhibit uniformity and integration – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
6. The design should be structured to degrade gently, even with bad data, events, or operating conditions are encountered – Well-designed software should never "bomb". It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
7. The design should be reviewed to minimize conceptual (semantic) errors – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax if the design model.
8. Design is not coding, coding is not design – Even when detailed designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made of the coding level address the small implementation details that enable the procedural design to be coded.
9. The design should be structured to accommodate change
10. The design should be assessed for quality as it is being created

When these design principles are properly applied, the design exhibits both external and internal quality factors. External quality factors are those factors that can readily be observed by the user, (e.g. speed, reliability, correctness, usability). Internal quality factors relate to the technical quality (which is important to the software engineer) more so the quality of the design itself. To achieve internal quality factors the designer must understand basic design concepts.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.5 Fundamental Software Design Concepts

A set of fundamental software design concepts has evolved over the past four decades, each providing the software designer with a foundation from which more sophisticated design methods can be applied. Each concept helps the soft ware engineer to answer the following questions:

1. what criteria can be used to partition software into individual components?
2. how is function or data structure detail separated from a conceptual representation of software?
3. there are uniform criteria that define the technical quality of a software design?

The fundamental design concepts are:

1. **Abstraction** - allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural abstraction - named sequence of events, data abstraction - named collection of data objects);
2. **Refinement** - process of elaboration where the designer provides successively more detail for each design component;
3. **Modularity** - the degree to which software can be understood by examining its components independently of one another;
4. **Software architecture** - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system;
5. **Control hierarchy** or **program structure** - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences);
6. **Structural partitioning** - horizontal partitioning defines three partitions (input, data transformations, and output); vertical partitioning (factoring) distributes control in a top-down manner (control decisions in top level modules and processing work in the lower level modules);
7. **Data structure** - representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design);
8. **Software procedure** - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure);

9. **Information hiding** - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

Software Design - an iterative process transforming requirements into a "blueprint" for constructing the software.
Design is the core of software engineering
Design concepts provide the basic criteria for design quality
Modularity, abstraction and refinement enable design simplification
A design document is an essential part of the process

5.0 Summary

Recall that software design can be viewed as a process and a model. In general, the process will be broken down into three broad stages:

1. Architectural design - specifications are analysed and the desired module structure is produced.
2. Detailed design – each module is designed in detailed and specific algorithms and data structures are selected.
3. Design testing – this is an activity that must be continuously conducted in parallel with all software production activities.

6.0 Tutor-Marked Assignment (TMA)

1. What is an iterative process transforming requirements into a "blueprint" for constructing the software?
2. What design specification models exit?
3. Architectural design is created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software: Yes or No?

4. In order to evaluate the quality of a design (representation) the criteria for a good design should be established. Name some of the criteria.

7.0 References/Further Reading

http://www.cavehill.uwi.edu/staff/eportfolios/paulwalcott/courses/comp2145/2010/design_-_concepts_and_principles.htm

http://www.nskinfo.com/ppt%5CCSE%5CSEM-5%5CCS2301-SOFTWARE%20ENGINEERING%5CCS2301-software%20design-3RDUNIT%5CCS2301-lecture-3RDUNIT-4.pdf

http://courses.cs.tamu.edu/cpsc431/lively/431_ppt/CH-13.PPT

Unit 2

Introduction to Design Patterns

Contents

1.0 Introduction

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?

One thing expert designers know *not* to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating obj ects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

An analogy will help illustrate the point. Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns like "represent states with objects" and "decorate objects so you can easily add/remove features." Once you know the pattern, a lot of design decisions follow automatically.

We all know the value of design experience. How many times have you had design *deja-vu*—that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it. However, we don't do a good job of recording experience in software design for others to use.

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

2.0 Learning Outcomes

By the end of this unit you will be able to:

1. define a design pattern;
2. know four essential elements of a pattern;
3. know and use Pattern Names and Classifications; and
4. know how to Determine Object Granularity.

3.0 Learning Contents

3.1 What is a Design Pattern?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns.

Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block. For this book we have concentrated on patterns at a certain level of abstraction . *Design patterns* are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.*

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use. A design pattern also provides sample C++ code to illustrate an implementation.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.2 Design Patterns in Smalltalk MVC

The Model/View/Controller (MVC) triad of classes [KP88] is used to build user interfaces in Smalltalk-80. Looking at the design patterns inside MVC should help you see what we mean by the term "pattern."

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these object s together. MVC decouples them to increase flexibility and reuse.

MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

The following figure 7.1 shows a model and three views. (We've left out the controllers for simplicity.) The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views communicate with the model to access these values.

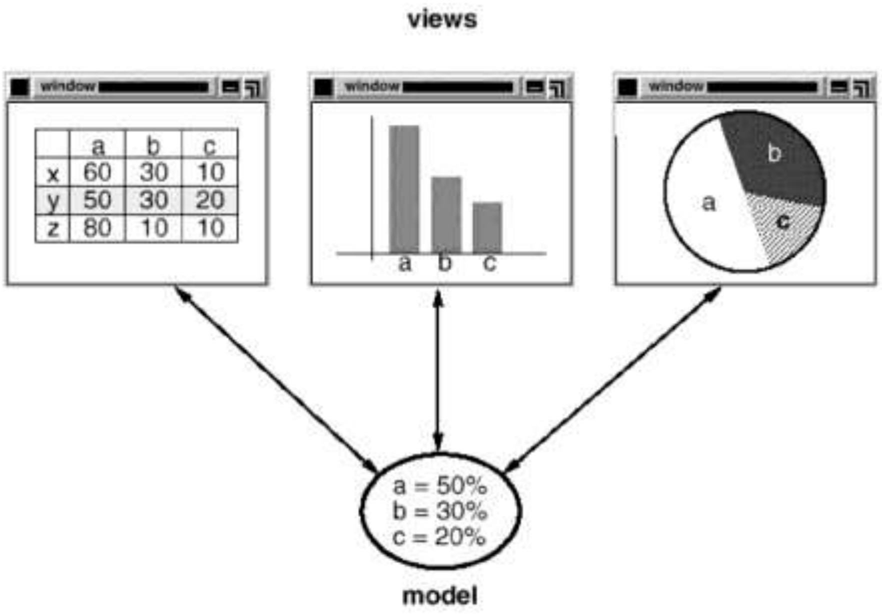

Figure 7.1 Model and three views

Taken at face value, this example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the Observer design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger. MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

Again, we could think of this as a design that lets us treat a composite view just like we treat one of its components. But the design is applicable to a more general problem, which occurs whenever we want to group objects and treat the group like an individual object. This more general design is described by the Composite design pattern. It lets you create a class hierarchy in which some subclasses define primitive objects (e.g., Button) and other classes define composite objects (CompositeView) that assemble the primitives into more complex objects.

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way it responds to the keyboard, for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.

A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller. It's even possible to change a view's controller at run-time to let the view change the way it responds to user input. For example, a view can be disabled so that it doesn't accept input simply by giving it a controller that ignores input events.

The View-Controller relationship is an example of the Strategy design pattern. A Strategy is an object that represents an algorithm. It's useful when you want to replace the algorithm either statically or dynamically, when you have a lot of variants of the algorithm, or when the algorithm has complex data structures that you want to encapsulate.

MVC uses other design patterns, such as Factory Method (121) to specify the default controller class for a view and Decorator (196) to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.3 Describing Design Patterns

How do we describe design patterns? Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action.

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.4 Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent
A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As
Other well-known names for the pattern, if any.

Motivation
A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability
What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure
A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

Participants
The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation
What pitfalls, hints, or techniques should you be aware of when implementing the pattern?

Are there language-specific issues?

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Organizing the Catalog

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. This section classifies design patterns so that we can refer to families of related patterns. The classification helps you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

We classify design patterns by two criteria (Table 7.1). The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

Scope	**Class**	Factory Method	Adapter	Interpreter	
				Template	Method
	Object	Abstract Factory (99)	Adapter (15 7)	Chain of Responsibility	
		Builder (110)	Bridge (171)	(251)	
		Prototype (133)	Composite (183)	Command	(263)
		Singleton (144)	Decorator (196)	Iterator	(289)
			Facade (208)	Mediator	(305)
			Flyweight (218)	Memento	(316)
			Proxy (233)	Observer	(326)
				State (338)	
				Strategy	(349)
				Visitor	(366)

Table 7.1 Classification of design patterns

Clearly there are many ways to organize design patterns. Having multiple ways of thinking about patterns will deepen your insight into what they do, how they compare, and when to apply them (figure 7.2).

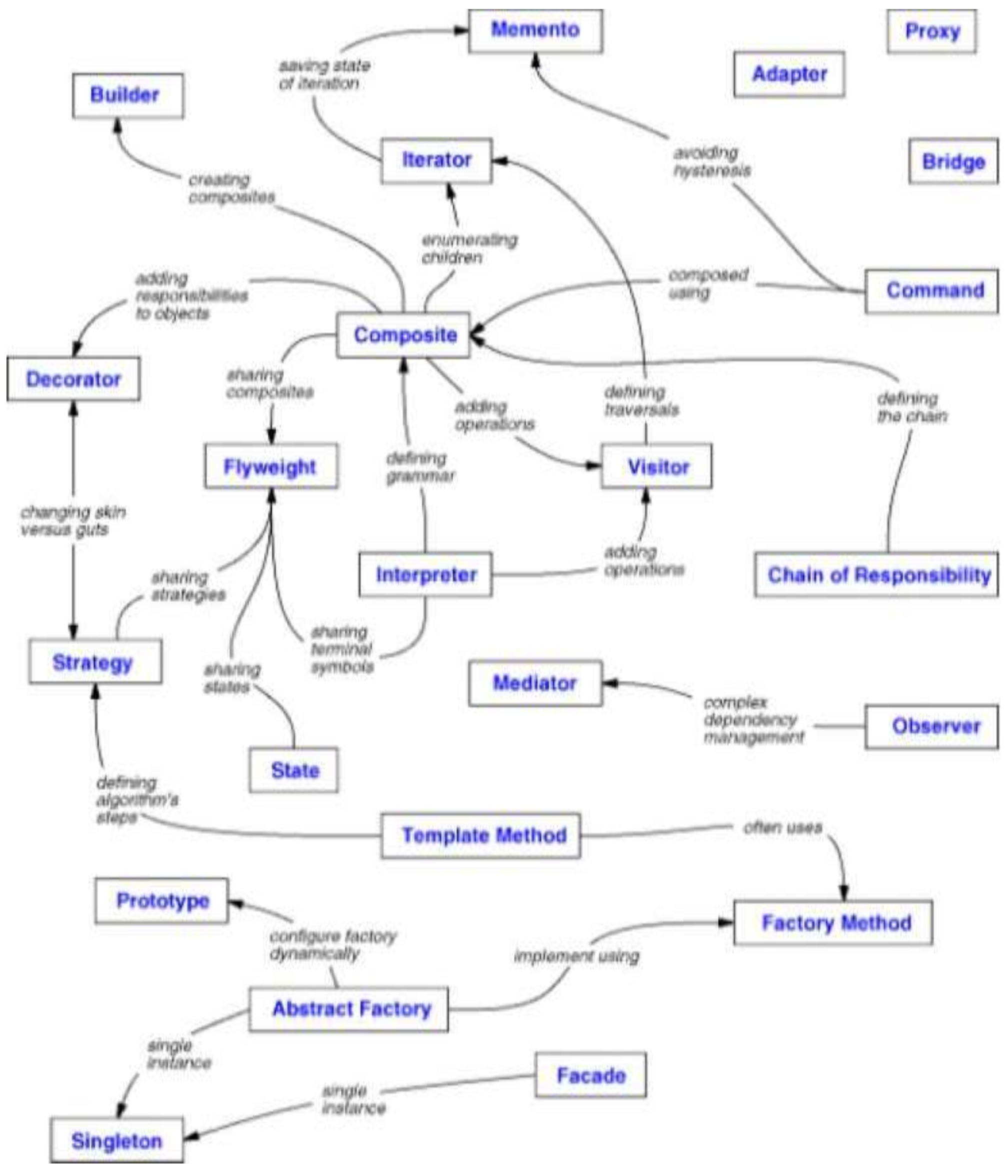

Figure 7.2 Design pattern relationships

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.5 How Design Patterns Solve Design Problems

Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. Here are several of these problems and how design patterns solve them.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.6 Finding Appropriate Objects

Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations. An object performs an operation when it receives a request (or message) from a client.

Requests are the *only* way to get an object to execute an operation. Operations are the *only* way to change an object's internal data. Because of these restrictions, the object's internal state is said to be encapsulated; it cannot be accessed directly, and its representation is invisible from outside the object.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on. They all influence the decomposition, often in conflicting ways.

Object-oriented design methodologies favor many different approaches. You can write a problem statement, single out the nouns and verbs, and create corresponding classes and operations. Or you can focus on the collaborations and responsibilities in your system. Or you can model the real world and translate the objects found during analysis into design. There will always be disagreement on which approach is best.

Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterparts in the real world. Some of these are low-level classes like arrays. Others are much higher-level. For example, the Composite pattern introduces an abstraction for treating objects uniformly that doesn't have a physical counterpart. Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.

Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy (349) pattern describes how to implement interchangeable families of algorithms. The State (338) pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.7 Determining Object Granularity

Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide what should be an object?

Design patterns address this issue as well. The Facade pattern describes how to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities. Other design patterns describe specific ways of decomposing an object into smaller objects. Abstract Factory and Builder yield objects whose only responsibilities are creating other objects. Visitor and Command yield objects whose only responsibilities are to implement a request on another object or group of objects.

Many of the design patterns depend on this distinction. For example, objects in a Chain of Responsibility (251) must have a common type, but usually they don't share a common implementation. In the Composite (183) pattern, Component defines a common interface, but Composite often defines a common implementation. Command (263), Observer (326), State (338), and Strategy (34 9) are often implemented with abstract classes that are pure interfaces.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.8 Programming to an Interface, not an Implementation

Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes. It lets you define a new kind of object rapidly in terms of an old one. It lets you get new implementations almost for free, inheriting most of what you need from existing classes.

However, implementation reuse is only half the story. Inheritance's ability to define families of objects with *identical* interfaces (usually by inheriting from an abstract class) is also important. Why? Because polymorphism depends on it.

When inheritance is used carefully (some will say *properly)*, all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. *All* subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
2. Clients remain unaware of the classes that implement these objects . Clients only know about the abstract class(es) defining the interface.

This greatly reduces implementation dependencies between subsystems.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.9 Application Programs

If you're building an application program such as a document editor or spreadsheet, then *internal* reuse, maintainability, and extension are high priorities. Internal reuse ensures that you don't design and implement any more than you have to. Design patterns that reduce dependencies can increase internal reuse. Looser coupling boosts the likelihood that one class of object can cooperate with several others. For example, when you eliminate dependencies on specific operations by isolating and encapsulating each operation, you make it easier to reuse an operation in different contexts. The same thing can happen when you remove algorithmic and representational dependencies too.

Design patterns also make an application more maintainable when they're used to limit platform dependencies and to layer a system. They enhance extensibility by showing you how to extend class hierarchies and how to exploit object composition. Reduced coupling also enhances extensibility. Extending a class in isolation is easier if the class doesn't depend on lots of other classes.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.10 How to Select a Design Pattern

With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you. Here are several different approaches to finding the design pattern that's right for your problem:

1. *Consider how design patterns solve design problems.* Determine object granularity; specify object interfaces, and several other ways in which design patterns solve design problems. Referring to these discussions can help guide your search for the right pattern.
2. *Scan Intent sections.* Read through each pattern's intent to find one or more that sound relevant to your problem.
3. *Study how patterns interrelate.* Studying relationships can help direct you to the right pattern or group of patterns.
4. *Study patterns of like purpose.*

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

3.11 How to Use a Design Pattern

Once you've picked a design pattern, how do you use it? Here's a step-by-step approach to applying a design pattern effectively:

1. *Read the pattern once through for an overview.* Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.
2. *Go back and study the Structure, Participants, and Collaborations sections.* Make sure you understand the classes and objects in the pattern and how they relate to one another.
3. Look at the Sample Code section to see a concrete example of the pattern in code. *Studying the code helps you learn how to implement the pattern.*
4. *Choose names for pattern participants that are meaningful in the application context.* The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application. That helps make the pattern more explicit in the implementation. For example, if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TeXLayoutStrategy.
5. *Define the classes.* Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify

existing classes in your application that the pattern will affect, and modify them accordingly.

6. *Define application-specific names for operations in the pattern.* Here again, the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method.

7. *Implement the operations to carry out the responsibilities and collaborations in the pattern.* The Implementation section offers hints to guide you in the implementation. The examples in the Sample Code section can help as well.

These are just guidelines to get you started. Over time you'll develop your own way of working with design patterns.

No discussion of how to use design patterns would be complete without a few words on how *not* to use them. Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection, and that can complicate a design and/or cost you some performance.

A design pattern should only be applied when the flexibility it affords is actually needed. The Consequences sections are most helpful when evaluating a pattern's benefits and liabilities.

Self -Assessment Questions

Please insert Self-Assessment Questions

Self-Assessment Answers

Please insert Self-Assessment Answers

4.0 Conclusion

It's possible to argue that this book hasn't accomplished much. After all, it doesn't present any algorithms or programming techniques that haven't been used before. It doesn't give a rigorous method for designing systems, nor does it develop a new theory of design—it just documents existing designs. You could conclude that it makes a reasonable tutorial, perhaps, but it certainly can't offer much to an experienced object-oriented designer.

We hope you think differently. Cataloging design patterns is important. It gives us standard names and definitions for the techniques we use.

5.0 Summary

In general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary.
2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

6.0 Tutor-Marked Assignment (TMA)

1. What are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context?
2. Are there any essential elements of a design pattern?
3. How to Select a Design Pattern?
4. Design Patterns Solve Design Problems: Yes or No?

7.0 References/Further Reading

http://www.cavehill.uwi.edu/staff/eportfolios/paulwalcott/courses/comp2145/2010/design_-_concepts_and_principles.htm http://www.itswtech.org/Lec/Rand(SoftwareEng)/Software%20Engineering%20%20%20%209.pdf

http://www.nskinfo.com/ppt%5CCSE%5CSEM-5%5CCS2301-SOFTWARE%20ENGINEERING%5CCS2301-software%20design-3RDUNIT%5CCS2301-lecture-3RDUNIT-4.pdf

ANSWERS TO SELF ASSESSMENT QUESTIONS

MODULE 1. INTRODUCTION TO PROGRAMMING

Unit 1: Overview paradigms of programming

1. 4
2. Imperative, Object-oriented, Functional, Logic
3. True
4. Yes
5. Yes

Unit 2: Overview of programming languages and the compilation process

1. Yes
2. Translation, execution
3. Yes
4. Lexical, Syntactic, Semantic, Code Generator, Optimizer

Unit 3: Introduction to object oriented programming

1. Computer application that is composed of multiple objects which are connected to each other.
2. True
3. Association, aggregation, composition
4. Encapsulation

MODULE 2. FUNDAMENTALS OF OBJECTS AND CLASSES

Unit 1: Objects and classes

1. A class is a kind of factory for constructing objects
2. Modularity, Information-hiding
3. False

Unit 2: Class members and instance members

1. Variables, methods
2. Yes
3. Job is to group together

Unit 3: Methods, message passing, Creating and Destroying Objects

1. Methods, but they are methods of a special type
2. True
3. Constructors and destructors cannot be declared static, const or volatile

MODULE 3 INHERITANCE, POLYMORPHISM AND ABSTRACT CLASSES

Unit 1: Inheritance

1 False

2 Yes, it is

3 Example of program

class B extends A {

```
//additions to, and modifications of,
//stuff inherited from class A
}
class Vehicle {
int registrationNumber;
Person owner; // (Assuming that a Person class has been defined!)
void transferOwnership(Person newOwner) {
. . .
}
. . .
}
class Car extends Vehicle {
int numberOfDoors;
. . .
}
class Truck extends Vehicle {
int numberOfAxels;
. . .
}
class Motorcycle extends Vehicle {
boolean hasSidecar;
. . .
}
```

Unit 2: Polymorphism

1. Polymorphism is related to object methods
2. 3
3. Overloading, Parametric, Inclusion polymorphism
4. No

Unit 3: Abstract classes

1. Shape
2. type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.
3. False
4. No

MODULE 4. PRIMITIVE TYPES

Unit 1: Primitive data types

1. Yes
2. When variable declare
3. No

4. 8

Unit 2: Control structures

1. True
2. Switch block

MODULE 5. ARRAYS AND STRINGS

Unit 1: Arrays

1. Container object that holds a fixed number of values of a single type
2. Element
3. No
4. B[0]

Unit 2: Strings

1. Strings are objects
2. Yes
3. toString method
4. True

MODULE 6. ALGORITHMS

Unit 1: Concept of an algorithm; problem-solving strategies

1. A set of rules that precisely defines a sequence of operations
2. Greedy, Divide & conquer, Dynamic, Specialized by Input, Strategy specialized, Iterative
3. True
4. No

Unit 2: Pseudocode and Stepwise Refinement

1. Falshe
2. Pseudocode
3. Yes

MODULE 7. OBJECT ORIENTED DESIGN

Unit 1: Fundamental design concept and principles

1. Software Design
2. Process and a model
3. Yes
4. Design Guidelines

Unit 2: introduction to design patterns

1. Design pattern

2. Pattern name, problem, solution, consequences

3. Consider how design patterns solve design problems, Scan Intent sections, Study how patterns interrelate, Study patterns of like purpose

4. Yes