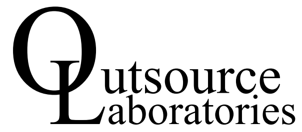


Introduction to Programming with Java™

OL302 VERSION 011703

Outsource Laboratories Press™
a division of IGS, Inc.

Copyright © 2002 - 2003 Outsource Laboratories Press, a division of IGS, Inc.
Printed in the United States of America

ISBN 0-9725199-3-9 Hard Cover
1 2 3 4 5 6 7 8 9   10 09 08 07 06 05 04 03

OUTSOURCE LABORATORIES PRESS
PO Box 187, Matawan, NJ 07747-0187 USA

For more information about Outsource Laboratories Press' products and services, contact us:

Toll free: 888-GO-OLABS (888-466-5227)
Email: training@olabs.com
http://www.olabs.com

Technical Support: Outsource Laboratories will offer limited technical support to instructors who have purchased classroom sets and have registered with Outsource Laboratories. Contact your representative for more information.

All rights reserved. This product and related documentation is protected by copyright and distributed under license restricting its use, copying, distribution, and decompilation. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written authorization from the publisher.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for any errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the information contained in this book.

Outsource Laboratories Press, Outsource Laboratories, and OLABS are trademarks of IGS, Inc. All other products referenced herein are trademarks of their respective holders.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other counties.

About Olabs' Text

This text is organized into units, and within a unit it is organized by up to three levels of headings. The unit title and first level heading are shown in top header of each page following the first page of a unit. This allows you to quickly relate a second or third level heading that occurs in a page to the main topic it is related to by simply looking at the page header.

The figures in the text coordinate with presentation files used by the instructor. Thus, for example, if the instructor is showing a slide labeled Figure 2-20, you will be able to find this in your text labeled with the same number (the figure number 2-20 references the 20th figure in Unit 2).

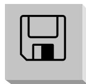
All the units conclude with review questions. Answers to these questions are provided in the .pdf file labeled "InstructorSetupGuide" and can be found on the CD provided with the instructor's copy of the text.

Exercises are placed throughout the units, close to relevant topics. Exercises may require coding, however some require analysis only. Icons next to the Exercise heading indicate what type of exercise it is. Icons are used in other situations within the text as well.

The following icons are used in the text, however each one may not be used in every manual:

This icon represents a hands-on exercise that requires compiling a program and running it.

This icon represents a hands-on exercise which is done on paper or as a discussion; it does not require editing, compiling or running a program.



This icon represents a running code sample available on line that is to be studied.

This icon represents that the material is related to a sample application.

Accompanying this text is a a set of electronic files which contain exercise templates and solutions organized to follow the unit structure of the text, sample programs referenced in the text, as well as solutions to analysis type exercises. Please read the "InstructorSetupGuide" for complete set up instructions. Third-party software required for the course is not distributed by Outsource Laboratories Press.

1 Basics of Programming (Programming Concepts)

Introduction

In this unit we provide some background on computer programming. We will start you thinking about programming by describing the six steps involved in program development. We will also introduce you to pseudocode, an English language-like notation for outlining a programming solution. You will become familiar with the procedure for compiling and running a simple Java program and you will also write your first Java Application.

The intended learning outcomes for this unit are listed in Figure 1-1.

At the end of this unit, the student will be able to:

1. Understand what programming is.
2. Identify programming problems.
3. List the steps in developing a program.
4. Identify actions expressed in pseudocode.
5. Compile and run a simple Java program.
6. Create a simple Java Application.

Figure 1-1: Unit Objectives

What is Programming?

Programming has been around for over a century. However, it did not really become popular as a field of study until the last 40 years. During this time, we saw the widespread rise in popularity of the computer due to the dramatic decrease in its cost and the significant improvements in computer technology that improved its performance.

Today computers are an important part of every day life. We see them in action when we withdraw money from bank ATMs, when we make calls using a cell phone, when we nuke our food using a microwave. Also, the advent of the Internet and the World Wide Web has given rise to a plethora of online businesses and the adoption of email as a form of communication.

The use of computers in so many different areas has generated increased interest in programming as a field of study and as a career. So what exactly does programming involve? Programming involves writing programs. This begs the question: What is a program? A program is a sequence of instructions that can be understood by a computer so that it can be made to perform some task or solve some problem. We also use the terms **software** or **application** to refer to a program.

Programs are not written in English. Programs are written in programming languages which are understood by a computer. The programming language that we will use in this course is Java but examples of others include C, C++ and Perl.

Figure 1-2 provides a summary of what programming is about.

- Programming involves writing programs.
- A program is a sequence of instructions that can be understood by a computer.
- Programs are also referred to as software or applications.
- Programs are written in a programming language.

Figure 1-2: What is Programming?

Steps for Developing a Program

Figure 1-3 lists the six steps that you would typically take in designing and writing a computer program.

1. Identify a task or problem.
2. Outline the solution by writing the algorithm.
3. Implement the algorithm using a programming language.
4. Compile the program.
5. Run the program.
6. Maintain the program.

Figure 1-3: Steps for Developing a Program

We will now discuss each of the steps above in more detail.

Step 1: Identify a Task or Problem

First we need to identify a task that we want performed or a problem that we want solved by a computer. We need to understand the problem and work out what sort of result we want produced.

Let's look at some examples of problems that could be solved using a computer.

Example 1:

Your friend from Europe comes to visit you in America. You are watching the weather forecast. The temperature is given in Fahrenheit, but your friend is used to temperature measures in Celsius. The problem here is that you have to convert the Fahrenheit temperature into a Celsius temperature so that your friend knows what sort of weather to expect. The formula to convert Fahrenheit into Celsius is: Celsius = 5/9 * (Fahrenheit - 32).

You could write a computer program to perform the temperature conversion. Typically a program has three main parts as listed in Figure 1-4.

A program consists of three main parts:

1. Take input.
2. Process the input.
3. Provide output.

Figure 1-4: Three Main Parts of a Program

The three main parts of the temperature conversion program would be:

1. Input: The temperature in Fahrenheit.
2. Process: Apply the formula to convert the Fahrenheit value into a Celsius value.
3. Output: The temperature in Celsius.

Programs should be designed to be as general as possible. In other words, they are not written to solve just one specific problem but for many problems that are similar. Given this, the temperature conversion program should be able to convert any Fahrenheit value into a Celsius value.

Example 2:

While shopping, you want to add up the prices of the goods that you intend to buy and also calculate the total including the tax. Let's consider what the three main parts of this program would be.

1. Input: Take in prices of goods that you are buying.
2. Process: Calculate the sum of the prices, calculate the tax you need to pay on that sum and add tax to sum of prices.
3. Output: The total cost including sales tax.

Example 3:

Another program could provide you with the name of the month given a number between 1 and 12. For instance, if you gave the program the number 5, it should print "May" to the screen. Again, let's think about what the three main parts of this program would be.

1. Input: A number between 1 and 12.
2. Process: Compare the number that was given with a list of numbers in the program that have names corresponding to the number. Find the respective name of the month.
3. Output: Provide the name of the month.

The three example programs that we just described are listed in Figure 1-5.

Problems that could be solved using a computer program:

1. Convert Fahrenheit Temperature into Celsius Temperature.
2. Calculate the total amount at a grocery store including sales tax.
3. Given a digit, provide the name of a month.

Figure 1-5: Example Problems for Computer Programs

STEP 2: OUTLINE THE SOLUTION BY WRITING AN ALGORITHM

Now that we understand the problem or task at hand and we know what it is that we want to achieve, it is time to outline a solution. To do this, we identify the sequence of steps that must be performed in order to produce the desired result. This sequence of steps is called an **algorithm**.

An algorithm helps us to think through the solution without having to worry about programming language considerations. You can think of it as a rough draft or a guide to producing the program.

Algorithms are written using an English-like notation called **pseudocode**. As you will see later in this unit, pseudocode has very restricted use of words and symbols. Hence, it allows us to be very specific and concise in describing the tasks we want performed by a computer.

Once we have our steps listed in an algorithm, we can test it with example input values.

STEP 3: IMPLEMENT THE ALGORITHM USING A PROGRAMMING LANGUAGE

After we have defined an algorithm to solve our problem, it can be implemented in any programming language. In the third step of the program development process, we need to decide on a programming language and then use it to write the program.

In this course we will be using Java to write our programs. Each programming language has its own set of rules, symbols and special words. Learning a programming language is mainly about learning these rules and symbols and how they are used in a program.

Once an algorithm is translated into a programming language it is called **code**. The process of writing a program in a programming language is called coding. People tend to think that coding is a difficult thing but typically the most difficult part of programming is in coming up with the algorithm.

STEP 4: COMPILE THE PROGRAM

Once a program has been implemented in a programming language, it is still not yet in a format that can be understood by a computer. A computer only understands **machine instructions**, which is a low-level programming language consisting of sequences of zeros and ones. A program must first be translated into machine instructions using a program called a **compiler**. Each programming language comes with its own compiler.

Machine instructions and compilers will be discussed in more detail in Unit 2.

STEP 5: RUN THE PROGRAM

After a program has been compiled, it can be executed by a computer using a program called an **interpreter**. Each programming language comes with its own interpreter.

More information about interpreters is provided in Unit 2. In this unit, we will be focusing more on the procedure to compile and run a Java program.

STEP 6: MAINTAIN THE PROGRAM

Most programs are written to be used not just once but many times. Over time maintenance is usually required. Code changes might be necessary because the requirements of the application might change with time. Also, there might be some bugs in the program because not every possible test has been performed on the program.

Program maintenance makes it necessary to include comments in a program. When you write a complex program, it is difficult after a long (or even short) period of time to look at it again and know exactly what it is that your code does. For this reason, and for the more likely reason that another programmer might need to understand your code, it is necessary to incorporate comments into your code to explain what it does.

Writing Algorithms with Pseudocode

This section looks more closely at the second step of the program development process, namely, the writing of algorithms using pseudocode.

There is no one standard for pseudocode. It is simply a way to formulate the programming solution using prose that is as close as possible to the programming language. Pseudocode allows us to define a solution without having to use programming language-specific syntax. It is a step in between the English language and the language that the computer understands. By learning pseudocode, you will start to become familiar with some of the technical terms associated with the programming language.

Review Questions

1. What are the three main parts of most programs?
2. Below are the steps for developing a program. They are jumbled up. Bring them into the correct sequence.

   Run the program.
   Outline the solution (algorithm).
   Identify a problem.
   Compile the program.
   Maintain the program.
   Implement the algorithm using a programming language.

3. What is pseudocode?
4. Below are two commands. Mark which of the commands compiles the program and which one runs the program.

```
java Greeting
javac Greeting.java
```

5. What is the extension of a Java program that is not yet compiled?
6. What is the extension of the file that is created when you compile a Java program?
7. Which DOS command lists all files in the current folder?
8. What is the DOS command to change into a different folder?
9. Below there is a table with example programming problems on the left hand side and statements written in pseudocode on the right-hand side. Match the problems to the pseudocode.

Programming Problems	Pseudocode

Output the result of a calculation	Cookies = 15
	WHILE (cookies > 0) DO
	Eat one cookie
	cookie - 1
	ENDWHILE

Eat cookies until they are all gone given 15 cookies.	Print result

Input the name of a person	IF age < 21
	Print “Sorry you can’t enter this place”
	ELSE
	Print “Please come in”
	ENDIF

Store the price of an item you buy in a shop	price = 67.80
Check if the age is less than 21. If that is the case print “Sorry, you can’t enter this place”. If the value is equal or more than 21 print “Please come in”	result = 7 * 45
Multiply 7 buy 45 and store the result.	Read name

2 Java Programming

Introduction

In this unit, we introduce you to the Java programming language. We explain some low-level programming concepts to help you understand more fully how a programming language like Java works. We then describe some of the different types of programs that can be written in Java. The concept of modularity, which is an important technique used in programming, is discussed. Following this, we step you through the basic structure of a Java program and show you how to document your programs using comments.

The intended learning outcomes for this unit are listed in Figure 2-1.

At the end of the unit, the student will be able to:

1. Understand some facts about the Java language
2. Understand the concepts of Machine Instructions, Statements and Compilers.
3. Understand the purpose of a Java Virtual Machine.
4. Distinguish between Java Applications, Applets and Servlets.
5. Understand the concept of modularity.
6. Understand the basic structure of a Java program.
7. Use comments to document a program.
8. Create a second simple Java Application.

Figure 2-1: Unit Objectives

3 Variables, Data Types and Operators

Introduction

Before you are able to write a program that performs a simple calculation, you need to learn the basic building blocks of a program: variables, data types and operators.

In this unit, we will discuss how to use variables to store data. We look at how data types are used to inform the computer of the type of data it is dealing with. You will learn the main operators used in Java to perform calculations on data. We will also introduce you to the `LearningIO` class, a helper class that enables us to write programs for taking in dynamic input, rather than static input specified at compile time.

At the end of this unit you will be able to write your own simple programs using these basic building blocks. The intended learning outcomes for this unit are listed in Figure 3-1.

At the end of this unit, the student will be able to:

1. Perform variable declaration and assignment.
2. Know why there are different data types.
3. Distinguish and use different data types.
4. Use the LearningIO methods to get keyboard input.
5. Understand variables and literals.
6. Understand and use operators.

Figure 3-1: Unit Objectives

The Add2Numbers Program

Let's start by looking at an example program that we will be using at many points throughout this unit.

Our example program sums up two numbers and prints out the result.

The problem we are solving with this program is as follows:

1. Declare three variables of type integer.
2. Assign numbers to two of those variables.
3. Add these two numbers together and assign the value to the third variable.
4. Print the the two numbers and the result.

The program would be written in pseudocode as follows:

```
Add2Numbers
   Read number1
   Read number2
   result = number1 + number2
   Print number1, number2 and result
END
```

The lines '`Read  number1`' and '`Read  number2`' in the pseudocode above involve the declaration of variables and assigning of values to the variables.

In Figure 3-2 the complete Java program Add2Numbers is shown. Recall that we explored the basic structure of a Java program in the Unit 2. Concentrate on the structure of the Add2Numbers program for the moment.

Can you answer the following three questions:

1. What is the name of the program?
2. In which line does the main method start?
3. In which line does the main method end?

```
1  class Add2Numbers {
2     public static void main(String[] args) {
3        // Declare three variables of type integer
4        int number1;
5        int number2;
6        int result;
7
8        // Initialize two of the variables.
9        // number1 represents the first value
10       // number2 represents the second value
11       number1 = 678;
12       number2 = 345;
13
14       // Sum the two numbers and store the result in 'result'
15       result = number1 + number2;
16
17       // Print out the result
18       System.out.println("The result of adding "
19                     + number1 + " and " + number2
20                     + " is " + result);
21    }
22 }
```

Figure 3-2: Add2Numbers Program

RUNNING THE PROGRAM

You have already compiled and run a few programs in earlier units. Remember the steps you need to take? To remind you, the steps are listed in Figure 3-3

1. Compile the program. At the command prompt type:

   ```
   javac Add2Numbers.java
   ```

2. Check if the class file Add2Numbers.class has been created. At the command prompt type:

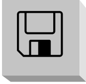
   ```
   dir
   ```

   If the file Add2Numbers.class is listed, the program compiled without errors.

3. Run the program. At the command prompt type:

   ```
   java Add2Numbers
   ```

Figure 3-3: Running the Program



Compile and run Unit03\Add2Numbers.java by following the steps of Figure 3-3. Below is a screenshot that shows the two commands and the result of running the program:

```
C:\ol302\Unit03>javac Add2Numbers.java

C:\ol302\Unit03>java Add2Numbers
The result of adding 678 and 345 is 1023

C:\ol302\Unit03>_
```

Statements and Expressions

We've already had a brief look at statements. Let's take a closer look at this topic and explore the concept of an expression, as these relate closely to the areas that we are going to address in this unit.

A **statement** is an instruction in a high-level programming language that tells the computer what operations need to be performed. Each statement can represent a sequence of several machine instructions.

In Java, every statement must be terminated by a semi-colon (;). Figure 3-4 shows some examples of statements that were part of the TempConverter Program which we looked at in Unit 1.

- Statements in Java must be terminated by a semi-colon (;).
- Example statements from TempConverter Program:

```
line 14: double fahrenheit;
line 15: fahrenheit = 80;
line 18: double celsius;
line 22: celsius = 0.55 * (fahrenheit - 32);
```

- Example expressions:

```
example 1: fahrenheit - 32
example 2: "Hello World"
example 3: 3.1459
```

Figure 3-4: Statements and Expressions

Statements are often made up of **expressions**. An expression is a combination of symbols that represent a value. Figure 3-4 provides some examples of expressions.

The symbols that combine to form expressions are called **operands** and **operators**. An operand is an entity that has value like an number. An operator is a symbol that represents an action like subtraction.

In the first expression example of Figure 3-4, `fahrenheit` and `32` are operands, and – is an operator.

Exercise 3-1: Find Valid Variable Names

Purpose: Practice identifying valid variable names.

Instructions: In Figure 3-10 valid and invalid variable names are listed. Study the rules for naming variable names that are given above. Make a list of the valid variable names.

$name	celsius	class	vegetarian
7mark	_counter	mark	$a
1234	number1	counter	bi

Figure 3-10: "Find Valid Variable Names" Exercise

Note: A list of the valid variable names is provided in Unit03\Solutions\Exercise3-1.txt.

Data Types

In the real world, we deal with different forms of data all the time. For example, we use integers to keep score in a game, decimal numbers to handle money and words to assemble sentences.

Similarly in programming, we deal with different forms of data. Data types provide a way for us to classify the different forms of data that we might use in a program. They tell the computer what sort of data it is going to deal with. Figure 3-11 lists some important points about data types. These points are discussed in the remainder of this section.

- Data types are a way of classifying data used in a program.
- A data type can be either a primitive type (e.g. int, double, char, boolean) or a reference type (e.g. String).
- Different data types take up different amounts memory.
- Java is a strongly typed language, i.e. every variable must be declared to be a particular data type.

Figure 3-11: Data Types

4 Flow Control – Selection

Introduction

So far, we know a program to be a set of instructions or statements, each of which is executed in sequence by a computer. This notion works well when the solution to a problem or task consists of a list of steps that needs to be performed in sequence.

However, in most real world problems, solutions cannot be broken down into a set of sequential steps. In many cases, you will want to perform a certain action only if a certain condition occurs. In other cases, you may want to perform a certain action repeatedly until some condition is met. We need some way to control the flow of statements in a program.

There are two types of flow control constructs in programming. The Selection constructs allow for a set of statements to be executed only if a certain condition is true or false. The Repetition constructs allow for a set of statements to be executed repeatedly while a certain condition is true. This unit focuses on Selection constructs. Repetition constructs are explored in Unit 5.

The intended learning outcomes for this unit are listed in Figure 4-1.

At the end of this unit, the student will be able to:

1. Understand flow control.
2. Give examples of the use of selection in a program.
3. Distinguish various types of selection structures.
4. Understand what a flow chart is.
5. Use and understand the different types of selection

Figure 4-1: Unit Objectives

What is Flow Control?

It is often not the case that a task which we want to program consists of a straightforward set of sequential actions. Flow control enables a program to deviate from the sequence of execution. Two types of flow control constructs can be employed in our programs.

The first type, **Selection**, is used when we want to select a course of action depending on a particular condition. For instance, an airline reservation system needs to check the availability of a flight before allowing a reservation to be made. If the flight is full, it needs to notify the user that there are no seats. Otherwise, it will let the user continue to make their reservation.

The other flow control construct is **Repetition**. It is used when we want an action to be performed repeatedly while a certain condition is true. An example is a photocopier that prints multiple copies of a page. The action of printing a particular page is repeated until the specified number of copies has been produced.

Figure 4-2 provides a summary of our discussion on flow control.

- Problems and tasks do not always consist of a set of sequential actions.
- Flow control enables a program to deviate from the sequence of execution.
- Two types of flow control constructs:
  - **Selection**: allows a program to select a course of action that is dependent on a certain condition.
  - **Repetition**: allows a program to repeat a set of statements while a certain condition is true.

Figure 4-2: What is Flow Control?

Selection Statements

Selection statements provide us with a mechanism for writing programs that can choose between a set of statements to execute. The set of statements chosen depends on a particular condition.

Consider a very simplistic AirlineReservations program which checks flight availability and prints out "Reservation rejected" if the flight is full, or "Reservation accepted" if the flight is not full. Recall from Unit 1 that the pseudocode for choosing between alternative actions is:

```
IF ... THEN
...
ELSE
...
ENDIF
```

Exercise 4-2: Using Logical Operators

Purpose: Practice evaluating logical expressions.

Instructions: Complete the exercise by following the instructions of Figure 4-8.

Evaluate the following logical expressions given the following values:

age = 28, female = true, height = 155, smokes = false, children = 1

Identify the expressions in which the evaluation is short-circuited.

1. (age < 21) && !female
2. smokes && (children > 0) && female
3. (age > 25) || (height > 150)
4. female && ((age > 30) || (children >= 1))
5. female || smokes || (height > 170)

Figure 4-8: "Using Logical Operators" Exercise

Note: A solution for this exercise can be found in Unit04\Solutions\Exercise4-2.txt.

Types of if-statements

There are three types of if-statements.

1. The if/else statement that tests a condition and then has actions for the true and the false case.

2. An if-statement without an else condition for scenarios in which an action exists for the true condition but not for the false condition.

3. A multiple if-statement for scenarios in which there are not two, but multiple courses of action that can be taken depending on a particular condition.

Let's look at each of these cases in more detail.

THE IF/ELSE STATEMENT

As the logic of our programs gets more complicated, it can be helpful to represent the logic in a picture. One way to do this is using a **flowchart**, like the one shown in the next in Figure 4-9. It represents the decision point in our AirlineReservations program.

The diamond represents making a decision in our code. In this case we are deciding whether the variable `fullFlight` is true or not. If the flight is full (fullFlight=true) the reservation is rejected. However, if it is not full (fullFlight=false), we accept the reservation. After that, the program continues on with any subsequent statements.

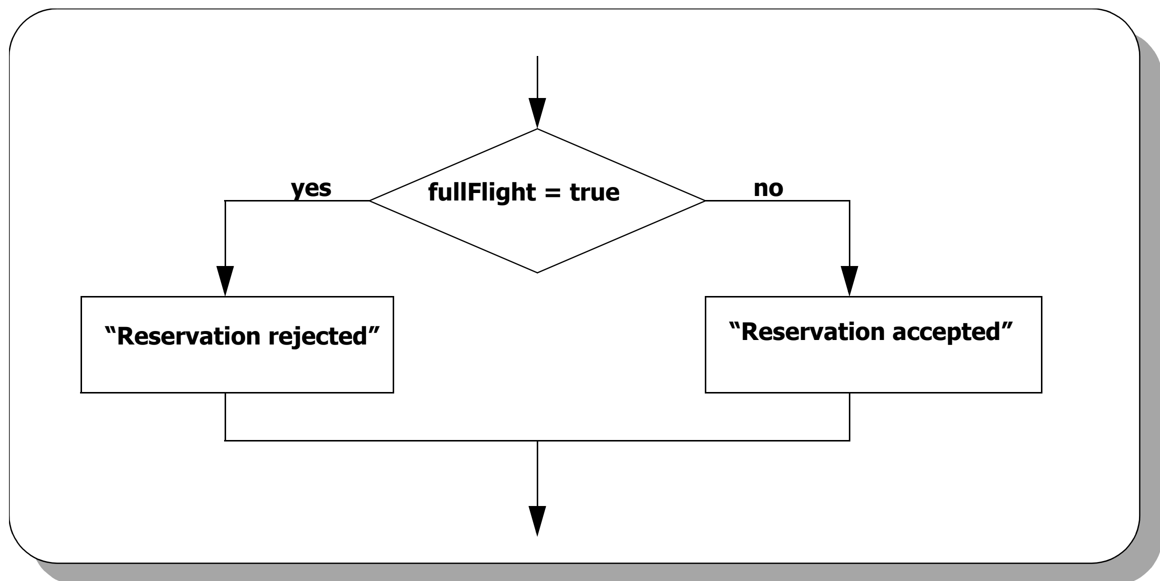


Figure 4-9: Flowchart: if/else Statement - AirlineReservations

Flowcharts are often used to illustrate the flow of a program when we are outlining the solution for a program. In Appendix B of this book we provide the meaning of the flowchart symbols that we will be using in this text.

An example that uses the if/else statement is the following weather program. Let's study the flowchart of the program first and then the pseudocode. The flowchart is depicted in Figure 4-10.

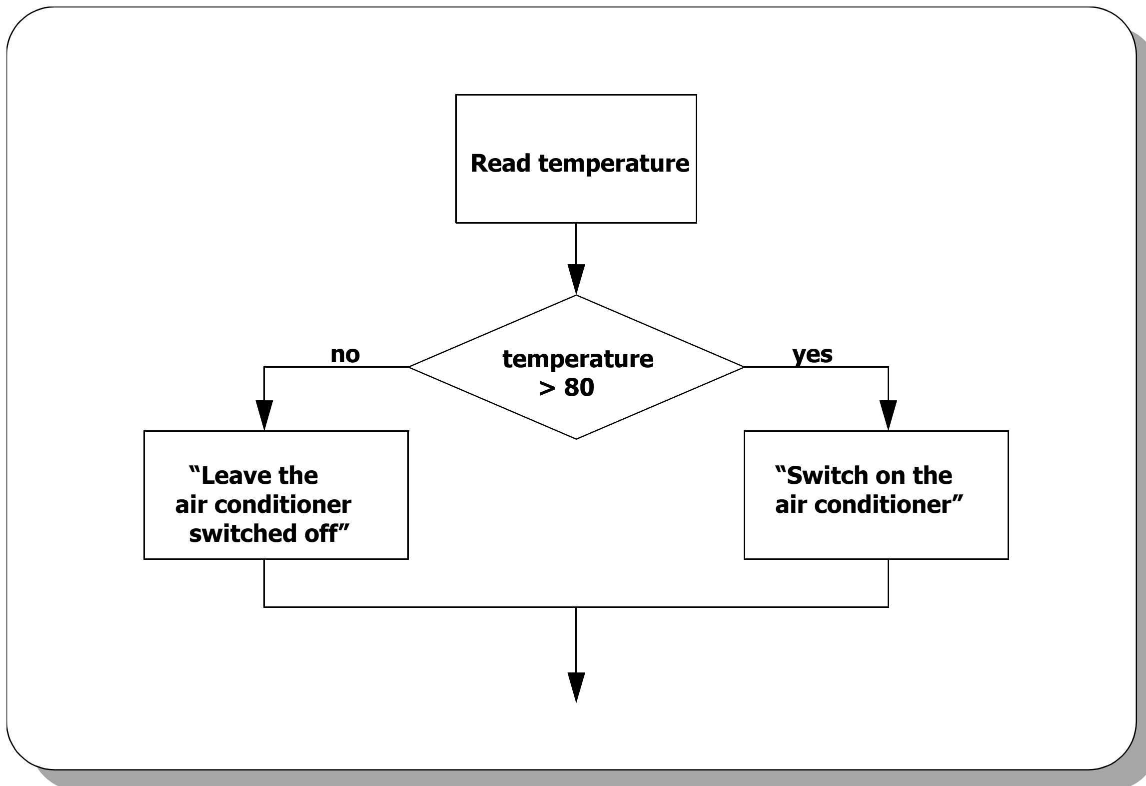


Figure 4-10: Flowchart: if/else statement - WeatherData

The pseudocode for the WeatherData program is as follows:

```
WeatherData
   Read temperature
   IF temperature > 80 THEN
      print "Switch on the air conditioner."
   ELSE
      print "Leave the air conditioner switched off."
   ENDIF
END
```

Let us study the Java code for this program that is displayed in Figure 4-11.

8 Using Arrays

Introduction

In Unit 2, you learned how to use variables in a program. A variable can only be used to store a single value. There are situations in which you may need to store a number of values of the same type. You could store each value in a different variable, however, Java and other programming languages provide a data structure called an array that enables a set of values of the same type to be stored conveniently.

In this unit you will learn how to use arrays and where they can be applied in your program.

The intended learning outcomes for this unit are listed in Figure 8-1.

At the end of this unit, the student will be able to:

1. Declare, create and use arrays.
2. Identify scenarios in which arrays would be used.
3. Search through the elements of an array for a value.
4. Use command-line parameters in a Java application.

Figure 8-1: Unit Objectives

What is an Array?

An array is a data structure used in programming to store and manipulate a collection of elements of the same data type. More formally, an array is a linear sequence of one or more variables with the same variable name, but distinguished by a positional number called an **index**. The term commonly used to describe a variable of an array is **element**.

Figure 8-3 shows a graphical representation of an array and some of its more important characteristics.

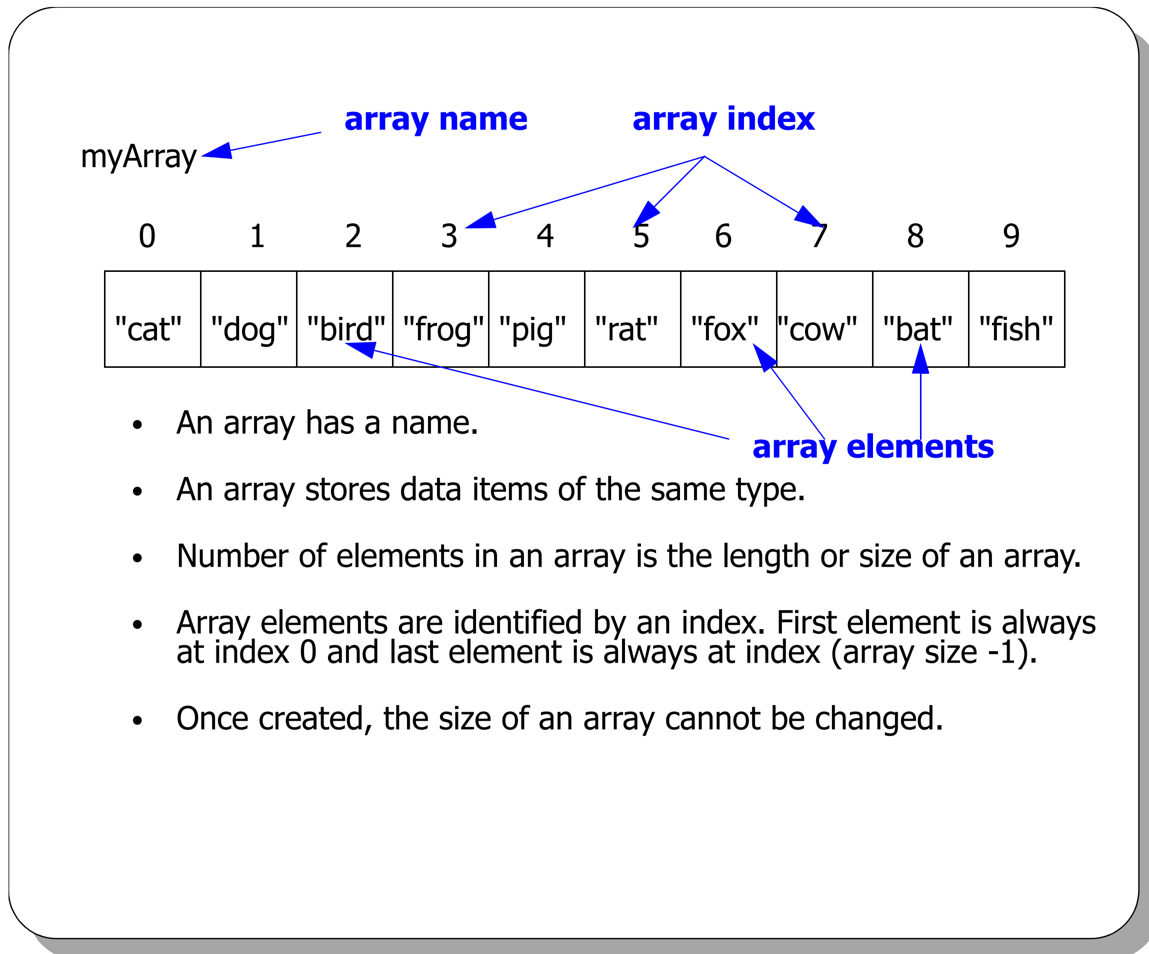


Figure 8-2: What is an Array?

Let's look at an example in which the use of an array would be appropriate. Consider a program that keeps track of the names of the people per seat in the rows of a movie theatre. To identify each person you need to store the name of the person associated with the seat number. You could create a variable for each seat. But that would make the handling of the data in the program difficult. Instead you can use an array to store the names of the people in one row. Then you would have an array for each row of the theatre.

We will use row number 7 as an example. In this row there are 10 seats; each seat is occupied by one person. Let's call the array 'row7'. It will have a length of 10 with the index starting at 0 and ending at 9. The elements of the array will be of type `String`.

Common Problem: Array Index Out of Bounds

The most common problem experienced when using arrays is an error called an `ArrayIndexOutOfBoundsException`. This exception is generated whenever an attempt is made to access an array element that is outside the index range of the array.

For example:

Declare an array of 5 Strings.

```
String[] names;
names = new String[5];
```

The next statement will generate an exception as the element 6 does not exist in the array.

```
names[6] = "Test";
```

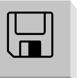
An example program that generates an `ArrayIndexOutOfBoundsException` is presented in Figure 8-19. The program tries to access an element of an array that does not exist.

Be very careful when accessing array elements as attempting to access an invalid element will be fatal for your application.

```
1  class IndexOutOfBoundsExample {
2
3      public static void main(String[] args) {
4          int[] scores;
5          scores = new int[10];
6          scores[10] = 5; //causes an error
7
8      } //main
9
10 } //IndexOutOfBoundsExample
```

Figure 8-19: Array Index Out of Bounds

Compile and run the program IndexOutOfBoundsExample. The output is displayed below:



```
C:\ol302\Unit08>javac IndexOutOfBoundsExample.java

C:\ol302\Unit08>java IndexOutOfBoundsExample
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at IndexOutOfBoundsExample.main(IndexOutOfBoundsExample.java:16)

C:\ol302\Unit08>_
```

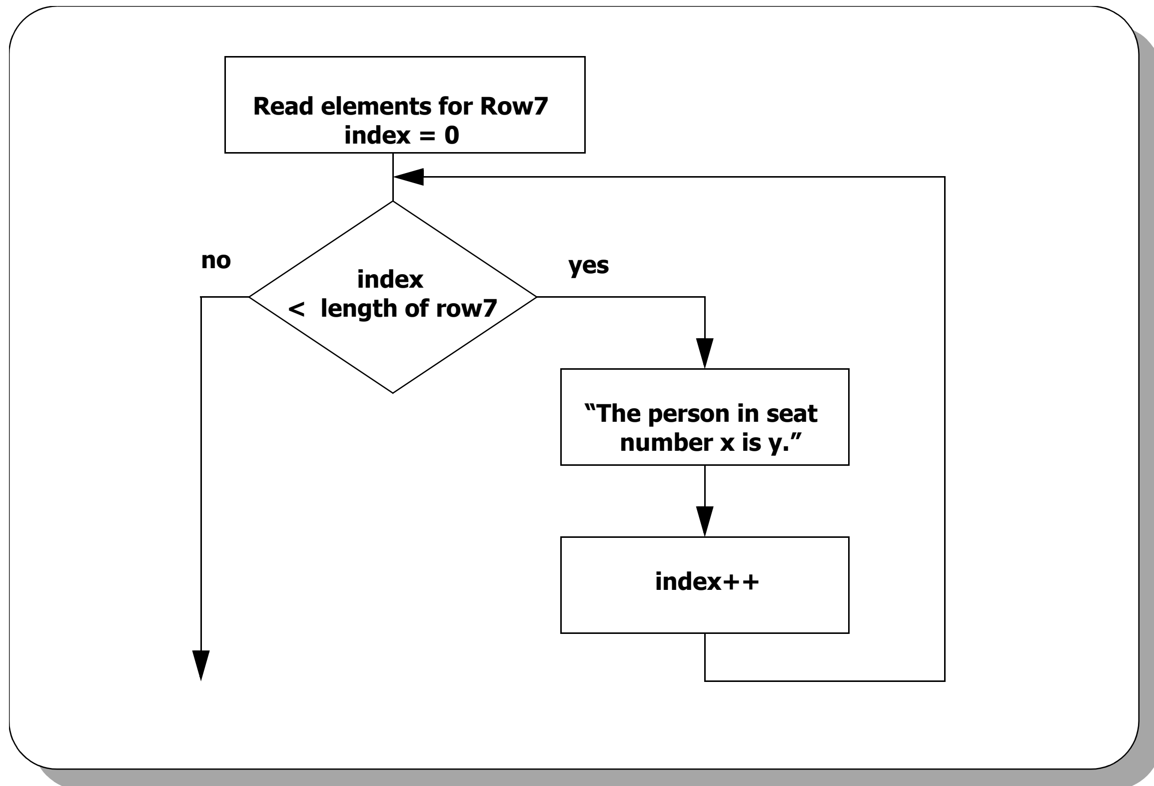


Figure 8-21: Flowchart - TheaterRow Program

The pseudocode for such a program could be expressed as follows:

```
TheaterRow
   Fill array row with elements
   index = 0
   WHILE index < length of array DO
      Print "The person in seat number x is y.
      index++
   ENDWHILE
END
```

In Figure 8-22 we show the code for the TheaterRow program.

9 Introduction to OO Programming and UML

Introduction

In this course, we have been examining, writing, compiling and executing Java programs written in a procedural style. Within each program, we focused on a sequential set of steps from beginning to end, sometimes breaking some of the steps out into a method. Java however is an object-oriented (OO) programming language, designed to be structured in an object-oriented rather than a purely procedural approach.

In this unit, we introduce you to several key concepts of OO programming, like encapsulation, inheritance and interfaces. The goal of this unit is for you to understand OO concepts, *not* to understand how to write an OO program on your own. That is left for later units.

Further, the pieces of an OO program can be represented graphically using a notation called the Unified Modeling Language (UML). So, when we present the OO concepts, we will use the UML notation. At the end of this unit, you should be able to read a simple UML diagram and explain what the UML symbols stand for.

At the end of this unit, the student will be able to:

1. Explain the difference between an object and a class.
2. Define the attributes and methods of a class, given a short description of the class.
3. Explain encapsulation.
4. Explain inheritance.
5. Explain abstract classes and interfaces.
6. Read a simple UML diagram.
7. Use UML notation represent a class and to express relationships between classes.

Figure 9-1: Unit Objectives

What is OO Programming?

In Unit 6, we examined a program written using the Java programming language that helped with the company payroll. This program took a procedural approach and used methods to organize some of the code. Even though the program was written with an object-oriented programming language, it didn't use object-oriented programming techniques.

Object-oriented programming techniques allow you to modularize your program using classes. The use of classes allows you to have smaller, more maintainable units of code. Further, within your program, the classes (for which you write the code) are used to create objects for (which you provide names). The objects interact with each other using messages. The interacting objects achieve the functionality of your program.

A class defines a template for all objects created from it. You write the code that defines the class; you don't write code for an object. You use the Java language to create an object from the class; the object is always given a name. The JVM, which we studied in Unit 2, creates the object using the class as a template and associates it with the name you specified.

Figure 9-2 illustrates these points with a diagram. On the left-hand side, we see a rectangle that represents our procedural program named CompanyPayroll. It consists of Java statements that may be organized functionally using methods. In comparison, on the right-hand side we see a rectangle that represents an object-oriented version of the procedural program; it is called OOPayrollApp. This object-oriented program contains statements that are organized into two classes and other statements that use these classes.

The two classes that make up our `OOPayrollApp` program are called `OOPayrollApp` and `Employee`. In our OO application, we use `OOPayrollApp` and `Employee` to create two objects called `payroll` and `employee` respectively. The `payroll` and `employee` objects interact in the program to produce the output you would have seen with the CompanyPayroll program. The `payroll` object interacts with the `employee` object by calling its methods *computeRaise()*, *computeNewSalary()*, *computeBonus()*, *printEmployeeInfo()*. You may remember these methods from our procedural payroll program. Note that procedural programming is still important despite the fact that we are using OO techniques. We will be using the procedural approach to implement the methods of the classes.

A class is a stand-alone unit of code. It can be compiled and small test programs can be written to interact with the class to test its methods. Now imagine that you have a large application. By being able to design the application as a set of classes, you are able to have smaller, testable units of code. This way when an error occurs in your unit of code, there is a smaller amount of code to debug.

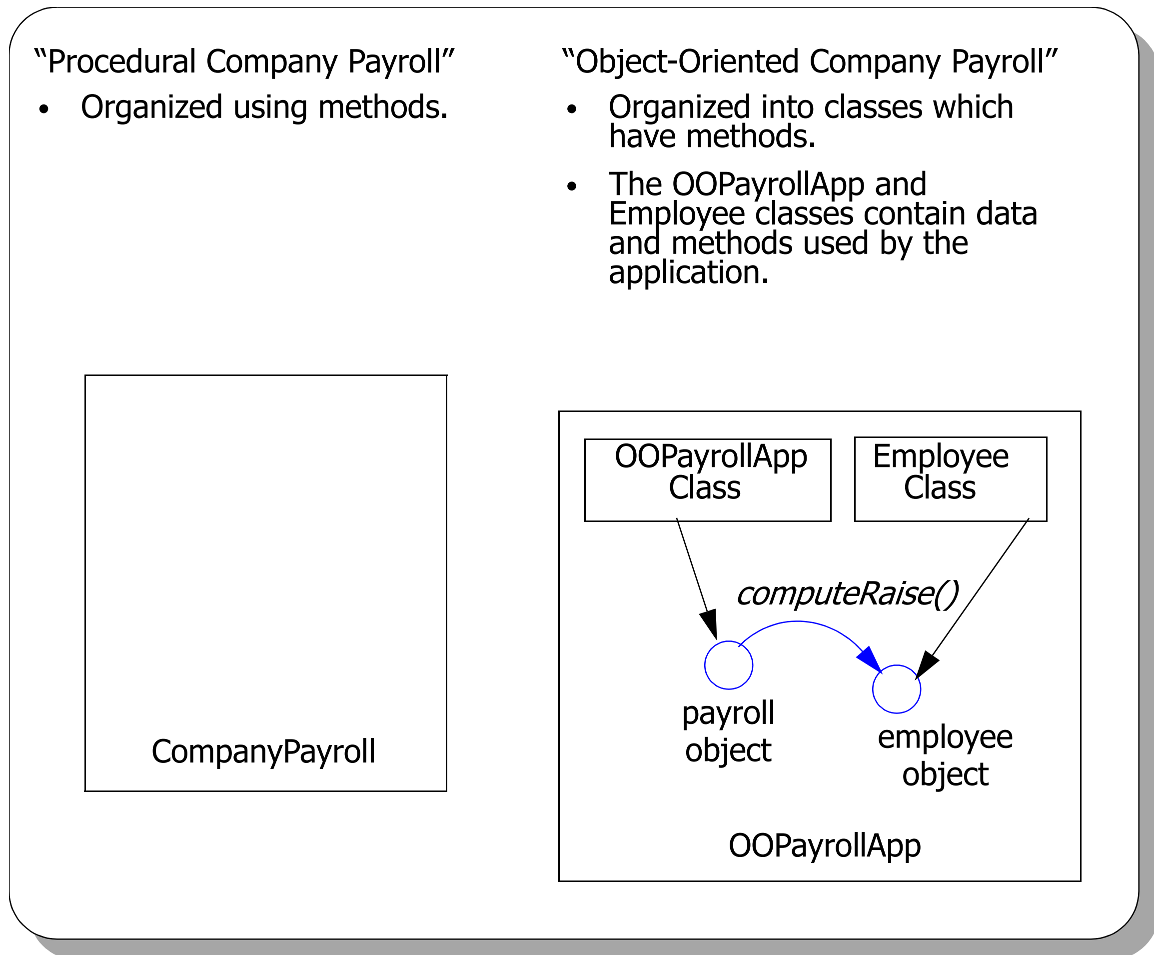


Figure 9-2: Procedural vs. Object-Oriented

Classes

Classes and objects are at the heart of object oriented programming. Recall from previous discussion that a class serves as a template for creating an object.

An important characteristic of a class and thus an object is that it "encapsulates" data and behavior. A class defines a set of attributes and operations. Each object created from the class will have a value for each attribute defined in the class and will be capable of performing the operations defined by the class. The object encapsulates a specific set of data and the operations that can be performed on that data. The object and class provide the means to treat the data and operations as a unit, which helps when building, testing and maintaining the code.

Further, many classes are based on the real world entities that they represent. Hence your program can model real world things that interact, thus making it easier to design and understand. We'll see this shortly.

Representing Class-Object Relationship

We've already worked with diagrams that shows the relationship between an object and the class from which it was created. We review this type of diagram in Figure 9-28.

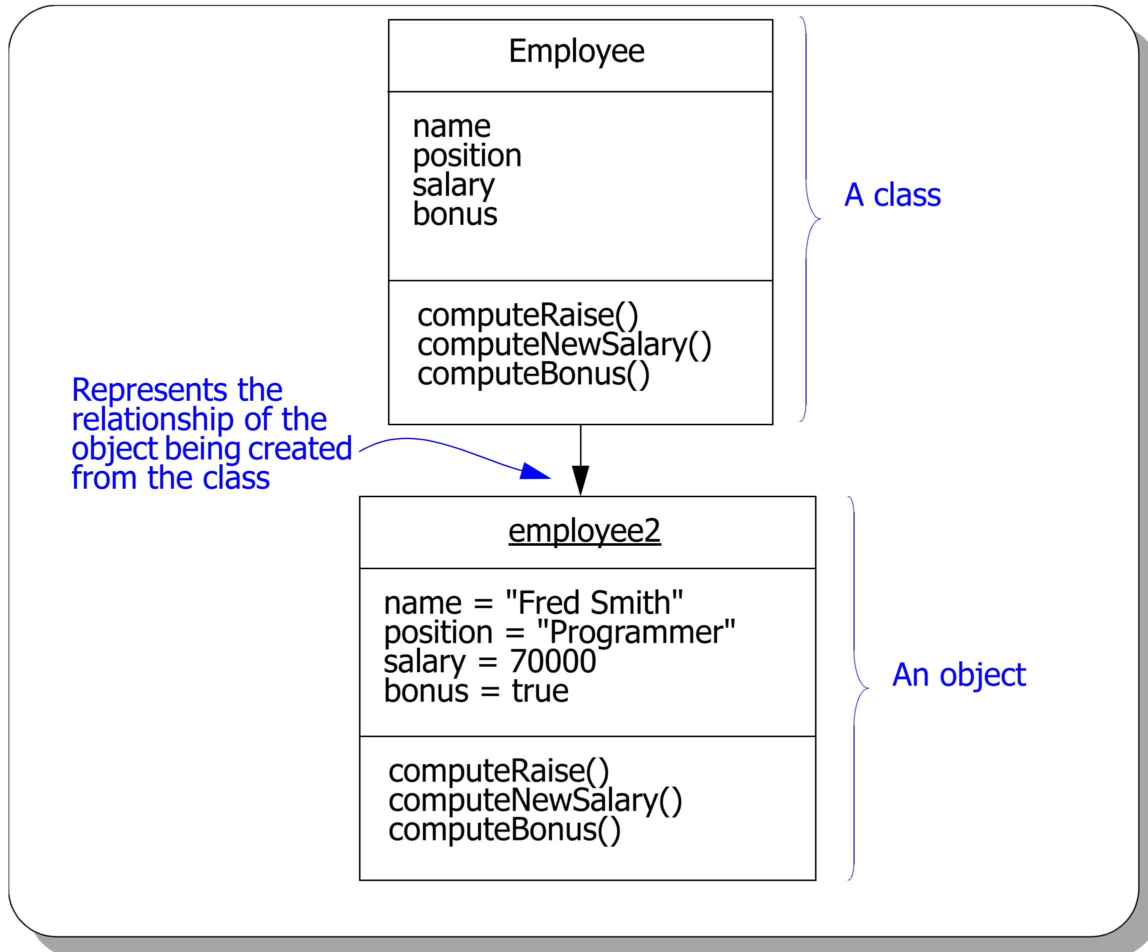


Figure 9-28: Class-Object Relationship

In the top part of the figure we see the class, with its name, attributes and operations. Then we see the object in the lower half of the figure. The object also has a name, attributes and operations. However, the object's attributes also have values. The convention for naming the object is different from that for naming the class. The class name is written beginning with a capital letter, while the object name begins with a lowercase letter and is shown underlined.

10 Using Objects

Introduction

In this unit you will learn how an object can be created from a class in Java and how an object can be used once created. At the end of the unit, you will use a pre-defined Database class to implement a Compact Disk library management system. The intended learning outcomes for this unit are listed in Figure 10-1.

At the end of the unit, the student will be able to:

1. Understand more about how objects are derived from classes.
2. Know what a constructor is and what they do.
3. Understand the new operator, garbage collection and other memory issues.
4. Use simple objects in a Java program.

Figure 10-1: Unit Objectives

More About Objects

In the last unit we introduced the notion of an object in relation to the concept of a class and learnt that an object is an instance of a class. In this unit, we will look more closely at what objects are and how they are used in Java programming.

An **object** can be defined as a combined indivisible collection of variables (attributes) together with a set of methods (operations) that operate on the attributes belonging to the object. See Figure 10-2 for an example of an object and its member attributes and operations.

object = combined indivisible collection of variables (**attributes**) together with a set of methods (**operations**) that operate on the attributes.

For Example: A bank account object members:

Object Attributes	Object Operations
accountNumber	credit
balance	debit
interestRate	isOverDrawn
clientNumber	addInterestForMonth
dateOpened	calculateGovernmentCharges

Figure 10-2: Introducing Objects

So far, we have looked at the use of primitive types of values in Java (i.e. simple values such as numbers and characters). Although these types have been effective in permitting us to demonstrate simple applications, more realistic applications require more detailed data representation. A banking or e-commerce system, for example, would be difficult to construct solely from `int`, `long`, `double`, `boolean` or `char` variables. Realistically, such systems require the ability to represent more worldly concepts and information, such as accounts, loans, clients, and transactions. An object enables us to represent these forms of composite information.

WHY ARE OBJECTS IMPORTANT?

In almost all commercial and popular programming environments, objects are typically the only, if not solely, recommended way of representing composite information. Programming languages, like Java, that focus on the use of objects for development are called object-oriented programming languages (OOPLs). Almost all modern systems are completely object-oriented. If you want to learn how to program in a modern day development environment, you need to understand object-orientation.

WHERE DO OBJECTS COME FROM?

Real world objects are constructed from designs or blue prints, whether human made, theological, genetic designs or otherwise. As in the real world, objects don't just come from thin air. Objects in programming languages such as Java are constructed from code containing the design or blue print for the objects. You may construct as many objects as you like from that same "blue print", at least until you run out of available memory.

See Figure 10-3 for how an object in Java relates to a class.

In Java:

- The "blue print" for an object is called a class.
- An individual object is created from its class.
- An individual class may be used to create many objects.
- The class from which an object is created is referred to as "the object's class", "the class of the object", or "the type of the object."
- An object is referred to as an instance of a class. A class is used to instantiate an object.

Figure 10-3: Where do Objects come from?

To create objects in Java, we first need to create a class. This class can then be used to create objects. We could, for example, write a class representing a bank account holding the bank account number and the balance. The code for this class is shown in Figure 10-4.

The class above represents information about a bank account. It contains four methods; two methods to manipulate and retrieve the bank account number (`getAccountNumber`, `setAccountNumber`) and two methods to manipulate and retrieve the current balance (`getBalance`, `setBalance`).

The `BankAccount` class represents the blue print for a bank account (although a very simple one). We need to create instances (objects) of this class. In your lab folder we have provided the program Unit10\TestBankAccount.java which creates two instances of the class `BankAccount` and manipulates the bank account number and the balance of each of the two bank accounts.

A World of Objects

The world is made up of millions of objects, all with their own attributes and operations. There are essentially two forms of objects: concrete objects (that you and I can touch, e.g. cars, planes) and conceptual objects (that represent concepts, e.g. phone calls, transactions). The form of object that is more challenging to identify is the conceptual object. However, both types typically represent nouns (i.e. people, places, things or concepts) from the real world. They rarely represent adjectives or verbs.

For example, a car is an object; stop and fast are not.

An object is generally defined by:

- Nouns that describe what an object is like (i.e. data/attributes). These nouns correspond to an object's variables.
- Verbs that describe what the object does or is capable of doing (behavior/operations). These verbs correspond to the object's methods.

An object therefore encapsulates both data and behavior. In the following sections, we will go through in detail the common processes that are performed on objects, such as accessing an object's attributes or invoking one of its methods. Figure 10-5 lists the common processes discussed in upcoming sections.

The processes typically performed on objects are:

- Declare object variables.
- Create objects.
- Access object attributes.
- Invoke object methods.

Figure 10-5: A World of Objects

Exercise 10-1: Objects

Purpose: Understanding objects, their attributes, and their behaviour.

Instructions: Follow the instructions in Figure 10-6 to complete this exercise.

For the "systems" below:

1. Identify the objects commonly found in the following "systems".
2. Categorize the objects you find into either the concrete or the conceptual forms.
3. Identify relevant attributes and operations for each object you discover.
   - An Airport
   - A School
   - A Video Shop
   - An E-Commerce Internet Site
   - A Car

Figure 10-6: "Objects" Exercise

Note: Unit10\Solutions\Exercise10-1.txt contains a solution to the exercise.

12 Introduction to GUI Programming in Java

Introduction

The programs that you have come across so far have all been examples of console applications that use a command-line interface for data input and output. Such applications are easy to create, debug and execute. Hence, they have their place in an introductory programming course.

In the real world, however, people prefer to use Graphical User Interfaces (GUIs) which provide colour, graphics, sound and user-friendly interactions. Java provides extensive support for building GUIs.

In this unit, we will introduce the Swing and AWT packages, which provide classes and interfaces for the development of GUIs in Java. We look in detail at the graphical components, containers and layout managers that a GUI can comprise, and discuss event-driven programming, a concept that is fundamental to GUI development. You will be developing parts of a GUI for a banking application using the knowledge gained from this unit. By the end of this unit, you will have developed your very first GUI!

The intended learning outcomes of this unit are listed in Figure 12-1.

At the end of this unit, the student will be able to

1. Understand GUI concepts.
2. List the steps in creating a simple Swing GUI.
3. Identify components and containers.
4. Use layout managers.
5. Explain event-driven programming.
6. Create a simple Swing GUI.

Figure 12-1: Unit Objectives

What is a GUI?

A **GUI** stands for Graphical User Interface. It is a program that interfaces with a user via graphical components. Graphical components include such things as buttons, textfields and scrollpanes. GUIs are event-driven. This means that when a user interacts with a component, like clicking on a button, the component fires an event which causes some operation to be performed. The notion of event-driven programming is central to GUI programming and will be discussed in depth later in this unit.

The features of a GUI are summarized in Figure 12-22.

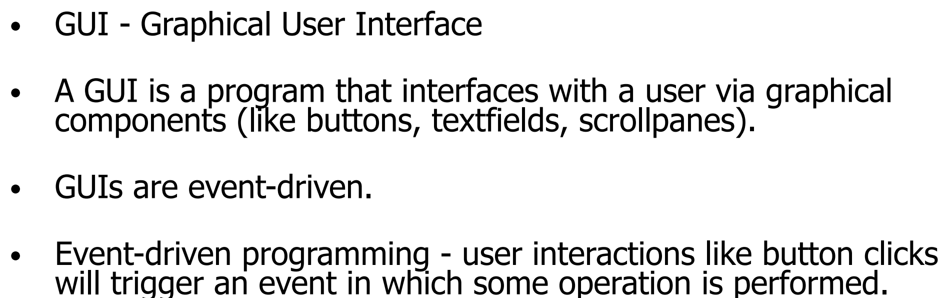


Figure 12-2: What is a GUI?

You have probably come across GUI applications before. The calculator program that you can run on a personal computer is a GUI application, as is the program that allows you to set the date/time properties of your computer. These are not programmed in Java though. Examples of Java GUIs are Applets and stand-alone GUI applications.

Figure 12-23 shows an example of a stand-alone Java GUI application. We see a pane with two parts. The left hand part contains a list of image names. When an image name is selected by clicking with the mouse, an image associated with the name is shown in the right hand part of the pane.

Swing and the AWT

The Swing and AWT packages provide a suite of classes that abstract away the primitive and platform-specific tasks associated with the construction of GUIs in Java. Let's take a closer look at these packages.

THE ABSTRACT WINDOWING TOOLKIT

The **Abstract Windowing Toolkit (AWT)** is the original set of classes used for creating GUIs. The AWT is implemented using native code. In other words, it is implemented using platform-specific code. For example, when you create a java.awt.Button object, the AWT will request the platform that the application is running in to create the button. This results in some inconsistencies across platforms.

Figure 12-4 shows a simple GUI constructed with AWT components.

- **java.awt** - the original package used for creating GUIs in Java.
- **Implementation is native. Uses GUI components from the underlying platform.**
- Enables the construction of GUIs that can enjoy Java's portability and platform independence.
- Supports:
  - Simple 2D graphics.
  - Buttons, Labels, TextFields, Containers, Windows, Dialog boxes.
  - Layout Managers.

- Code is available in Unit01\HelloAWT.java

Figure 12-4: The Abstract Windowing Toolkit

JAVA GUI COMPONENT HIERARCHY

All Java GUI components fit into an inheritance hierarchy. Figure 12-6 shows an excerpt from the Java GUI component hierarchy.

All **components** inherit from the `java.awt.Component` class. A component is an object which has a graphical representation that can be displayed on the screen. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface. Examine the API of `java.awt.Component` and look at some of the methods that are common to all components.

There is also a class called `java.awt.Container` that inherits from the `Component` class. A **container** is a special type of component since it can contain other components. All Swing components inherit from `javax.swing.JComponent` which is a subclass of `Container`.

Many Java GUI components derive from java.awt.Component

All Swing components derive from javax.swing.JComponent

Java Swing Components:
- JButton, JCheckBox, JLabel, JScrollPane, JTextField, JList,...
- All share inherited methods for manipulating colors, sizes etc.

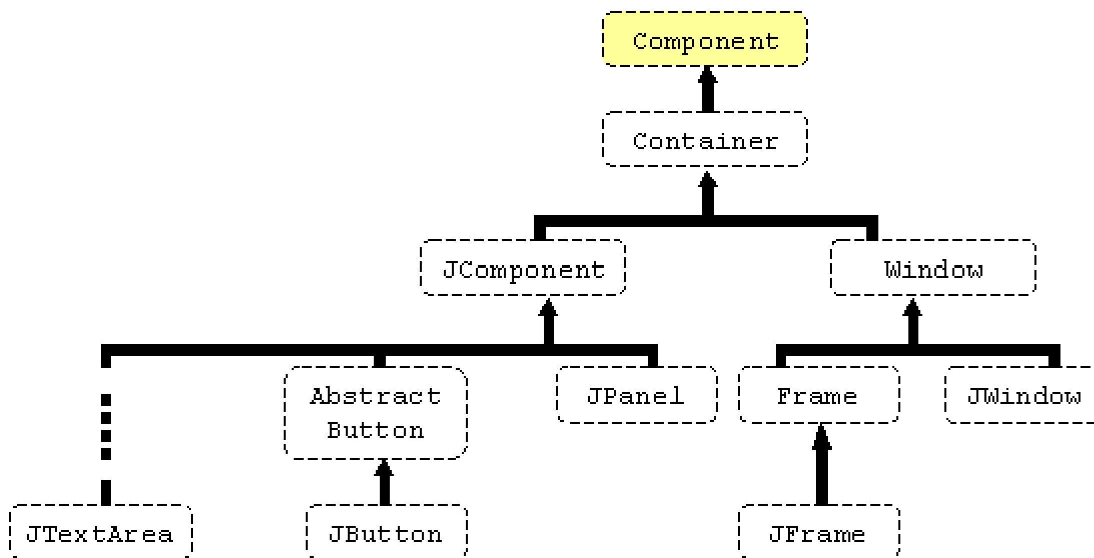


Figure 12-6: Java GUI Components

Exercise 12-5: Creating a 'Create Customer' Screen

Purpose: This exercise requires you to use `JComboBox`, `JRadioButton` and `JList` to create a 'Create Customer' screen for the First Bank of Java GUI.

Background: The 'Create Customer' screen of the First Bank of Java GUI is used to create a new customer record. The screen features a combo box for specifying the title of the customer, text fields for entering the customer's name, address and email, radio buttons for specifying the address type and also, a list is used to display the types of promotional information that a customer can choose to receive by email.

In this exercise, you will develop a single class that creates the 'Create Customer' screen. This class, like the `LoginPanel` class you developed in the previous exercise, is designed to be created within an application and not run on its own. However, we will show you how a `main` method can be included in such a class for the purposes of testing.

The exercise consists of four tasks. The first involves declaring components for the screen. The second task creates the components. The third uses layout managers to organize the components in the screen. And the fourth task sets up the main method which allows the code of the class to be easily tested.

Instructions: Figure 12-53 to Figure 12-56 lists the steps required to complete this exercise.

Task 1: Declare components.

1. Open the file Unit12\Ex05\CreateCustomerPanel.java.
2. Complete the class as indicated by the Task1 // TODO comments:

```
// TODO: Declare a combo box called "titleCB".

// TODO: Declare 2 radiobuttons called "homeRB" and "businessRB".
// TODO: Declare a button group variable called "buttonGroup".

// TODO: Declare a list called "promoJL".
// TODO: Declare a string array and initialize with
//       the following 5 elements:
//       1. "Product announcements."
//       2. "Promotional offers."
//       3. "FBJ monthly newletter."
//       4. "FBJ announcements."
//       5. "Offers from other organizations."
```

Figure 12-53: "Creating a 'Create Customer' Screen - Task 1" Exercise

13 Data Structures

Introduction

The purpose of this unit is to introduce the student to classic programming data structures. We explore the concept of a stack, queue, linked list and binary tree and cover the implementation of each data structure. We also discuss some of the more commonly used data structures that are implemented in the Java programming language. The student will become familiar with the use of Java's `ArrayList`, `LinkedList`, `TreeSet`, `Vector` and `Hashtable` classes.

The intended learning outcomes for this unit are listed in Figure 13-1.

At the end of the unit, the student will be able to:

- Understand the concept of a data structure.
- Understand stacks and queues.
- Understand linked lists.
- Understand binary trees.
- Understand how to use Java's ArrayList, LinkedList, Vector and Hashtable classes.
- Enumerate over the elements of a Java data structure.

Figure 13-1: Unit Objectives

What are Data Structures?

A data structure is a mechanism that provides a means for storing and manipulating a collection of data. There are many different ways in which data may be organized and operated on. Hence, there are many different types of data structures to represent these methods of organization. One data structure that you should already be familiar with is the array, which we covered in Unit 8. Other classic programming data structures are the stack, queue, linked list and binary tree. We will be studying these in the following sections. We will see how each stores data, how each behaves and how each can be implemented in Java.

In programming today, the craft of constructing classic data structures is becoming a declining prerequisite for application developers. The reason is that programming languages such as Java provide most of the data structures that a developer might need. Later in the unit, we will explore some of the commonly used Java data structures.

See Figure 13-2 for the main points about data structures.

Data structures:

- In general terms, a data structure is a mechanism that provides a means for storing and manipulating a collection of data (items).
- In Java terms, a data structure is typically an object that manages the storage of some collection of objects (items).
- As there are many different ways in which data may be organized and operated upon, correspondingly there are many different data structures to represent such methods of organization.
- Examples: array, linked list, stack, queue, binary tree, vector, hashtable

Figure 13-2: What are data structures?

Stacks

Stacks can be thought of as a linear data structure whose items are stacked one on top of the other. The top of the stack is its single point of access; new items are always inserted at the top of the stack and only the most recently inserted item can be accessed. For this reason, a stack is said to have **Last-In-First-Out (LIFO)** ordering.

Figure 13-3 shows a diagram of a stack that contains integers.

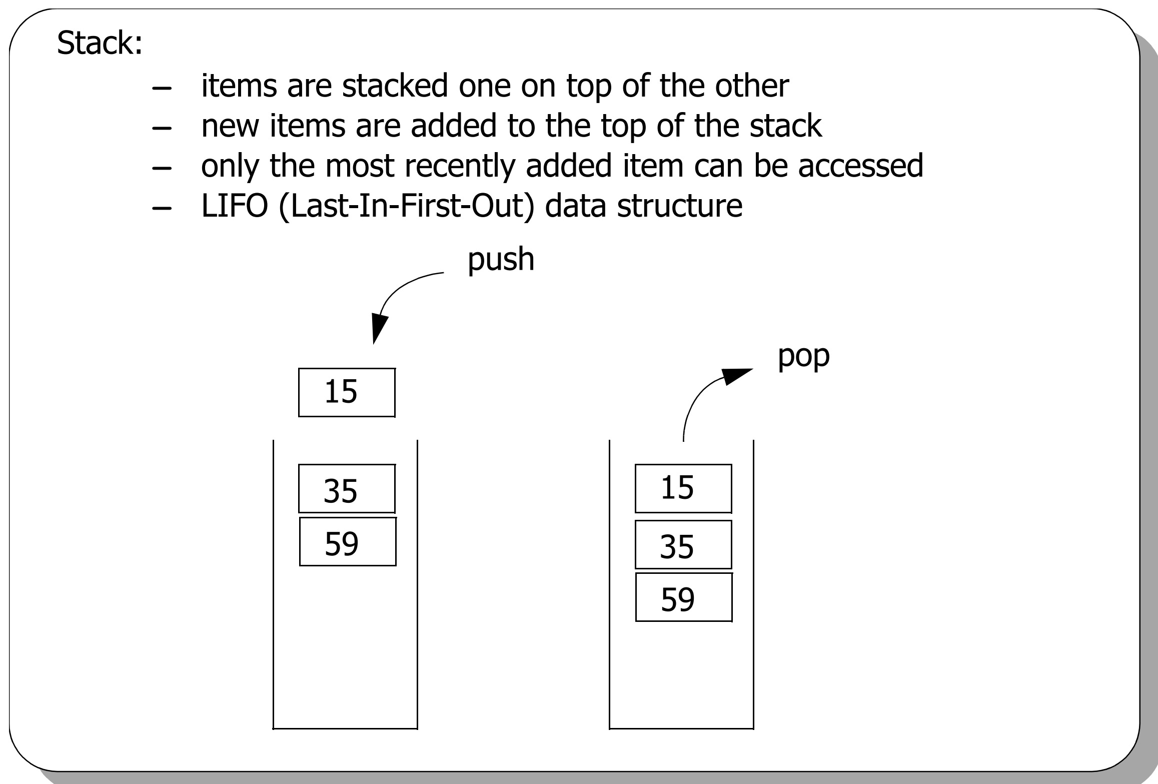


Figure 13-3: The Stack Data Structure

A good example of the way a stack works is the stack of plates that you might find in a buffet restaurant. When a plate is needed, a customer will take the top-most plate from the stack. When new clean plates are available, a waiter or waitress will add these to the top of the stack.

The stack data structure has many uses in programming. For example, it is used for evaluating expressions, keeping track of method calls and passing parameters to a method.

IMPLEMENTING A QUEUE USING AN ARRAY

A queue can be implemented using an array, just like a stack. However, the queue implementation is more complex because we need to use a **circular array**. Figure 13-10 shows a diagram representing a queue implemented as a circular array.

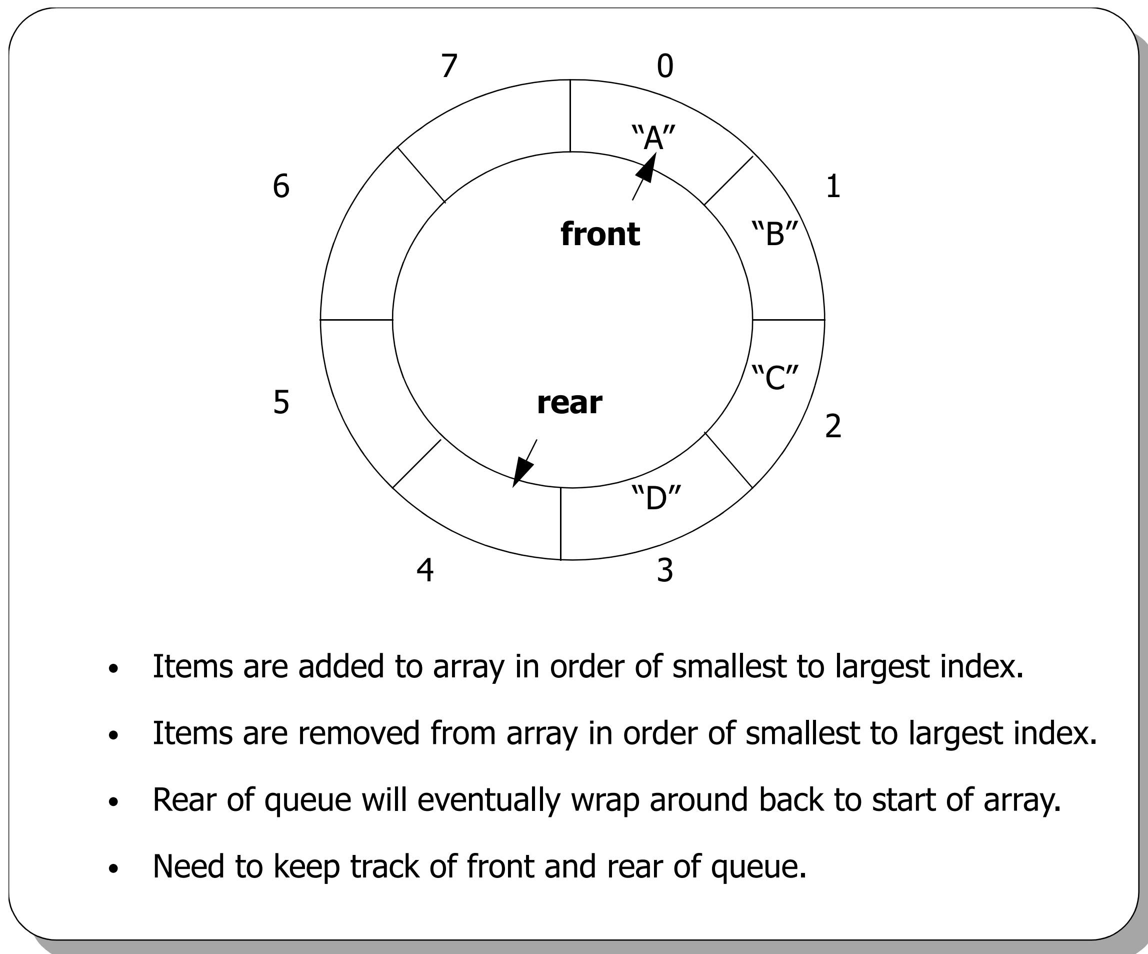


Figure 13-10: Queue implemented as Circular Array

The circular array above stores items of type `String`. We keep track of the array element that stores the front item in the queue as well as the array element after the last item in the queue. Items are added to the array in order of smallest to largest index. They are also removed from the array in order of smallest to largest index.

The front and rear of the queue will move around the array in a clockwise pattern as items are removed and added. Eventually, both the rear and front will wrap around back to the start of the array. When the rear and front of the queue are the same, this indicates either that the array is empty or that it is full. To determine whether the empty or full case has occurred, we need to keep track of the number of items in the array.

Binary Trees

A data structure that is commonly used in programming is a **tree**. The tree data structure, like a linked list, is made up of nodes. Tree nodes are organized in a hierarchical structure, similar to that of a family tree.

The node at the top of the hierarchy is known as the **root node**. Each node of a tree must have two or more pointers. A pointer of a tree node is also known as a **branch**. The node that a branch leads to is known as a **child node**. A tree node that has children is called a **parent node**. A tree node that has no children is called a **leaf node**.

There are different types of tree data structures. The type of a tree is determined by the number of pointers that belong to each node of the tree. The simplest and most commonly used type of tree is one whose nodes have two pointers. Such a tree is known as a **binary tree**. Binary trees are good for storing data that needs to be ordered. They are also good for efficent searching, inserting and deleting of items. You will see why this is so later in the section.

Figure 13-33 shows a diagram of a binary tree.

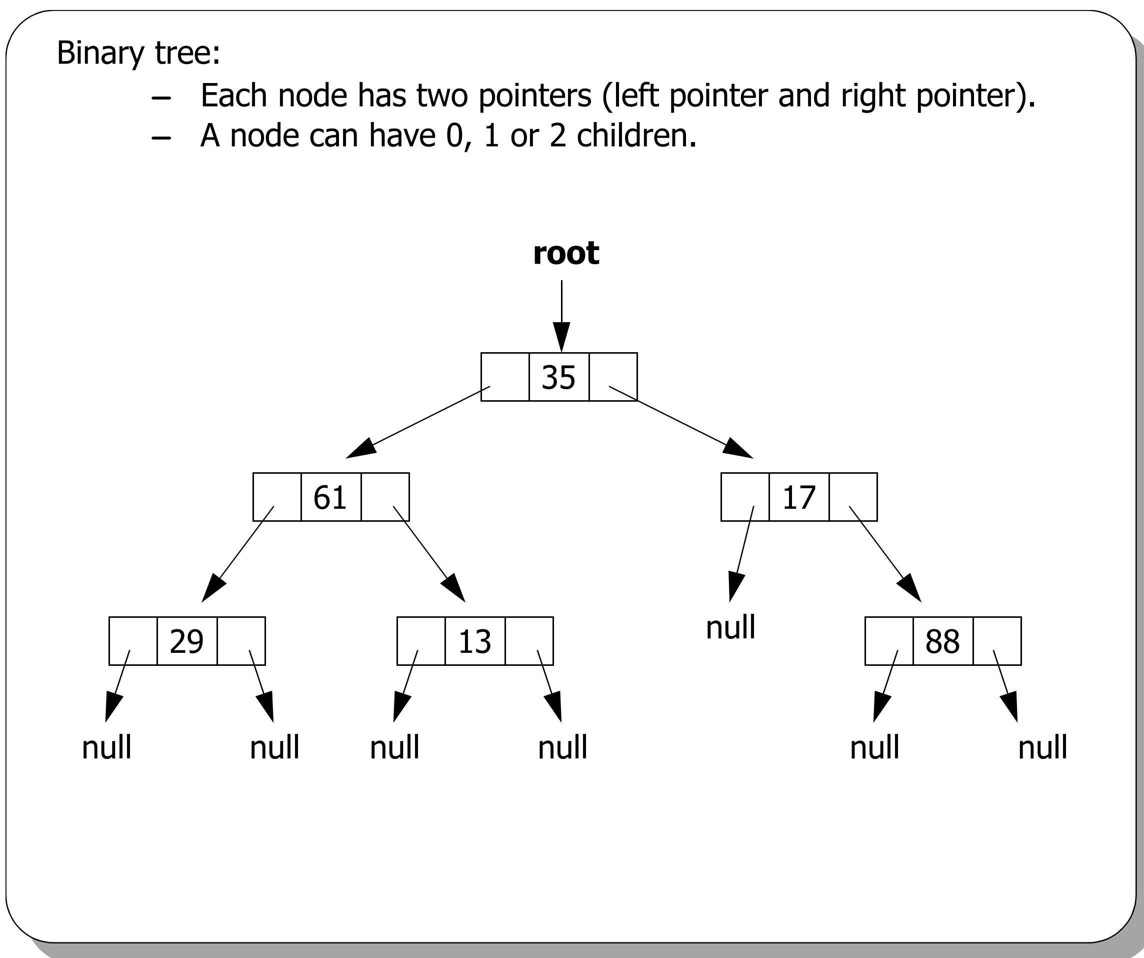


Figure 13-33: Binary Tree

Consider the tree in Figure 13-36. Let's look at the comparison and branching process that occurs when we insert item 32 into the tree.

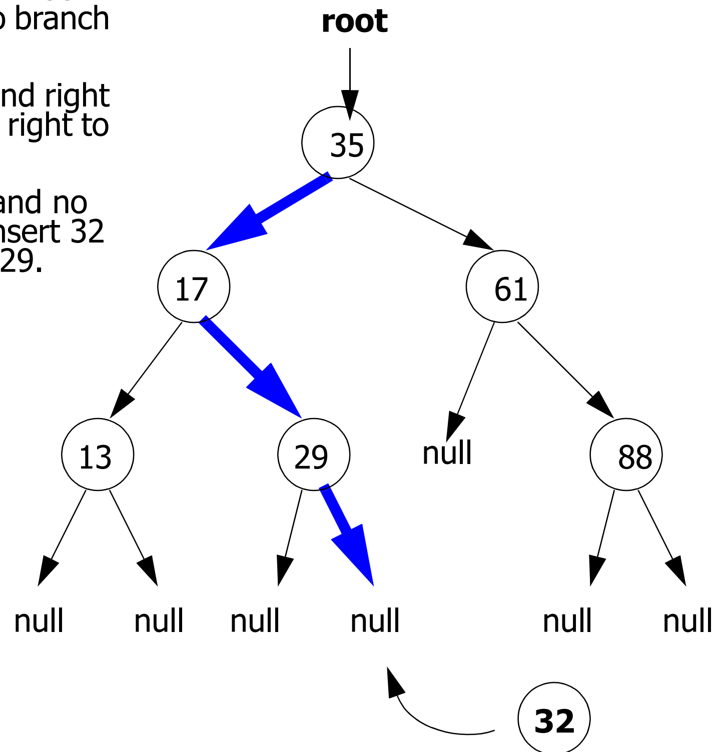


Figure 13-36: Inserting an Item into a Binary Search Tree

To insert node 32 into the tree above, we start by comparing 32 to the value of the root node which is 35. Since 32 is less than 35, we need to branch left which leads us to node 17. We compare 32 with 17 and find that it is greater so this time we branch right to node 29. Since 32 is greater than 29, we want to branch to the right child of node 29 but it does not exist. Hence, we insert 32 as the right child of node 29.

Exercise 13-8: Traversing a Binary Search Tree

Purpose: Practice using the different binary tree traversal algorithms.

Instructions: Follow the instructions outlined in Figure 13-60 to complete the exercise.

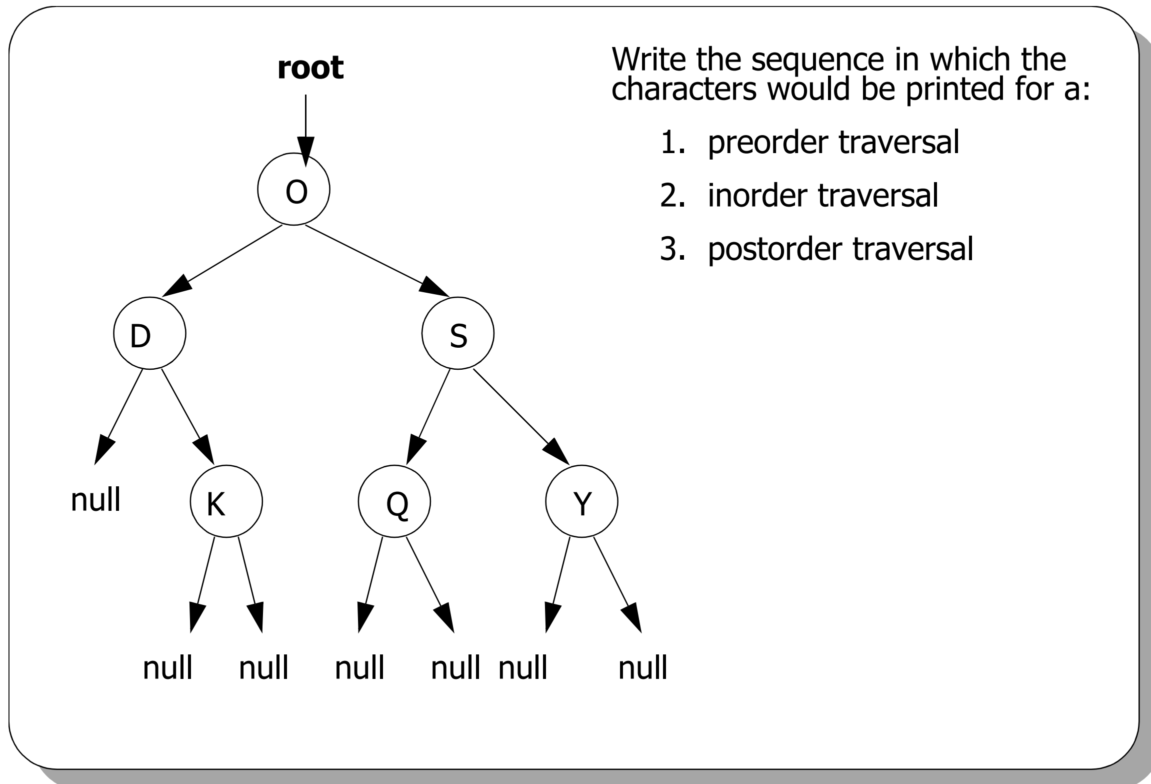


Figure 13-60: "Traversing a Binary Search Tree" Exercise

Note: Unit13\Solutions\Exercise13-8.txt contains the solution.

Java Data Structures

The Java programming language originally provided only two data structure classes: `Vector` and `Hashtable`. Java version 1.2 introduced the Collections Framework which was built on the original data structure classes but included more data structure implementations. In addition to `Vector` and `Hashtable`, the Collections Framework also features useful classes such as `ArrayList`, `LinkedList` and `TreeSet`.

Each of the classes mentioned here will be explored in the following sections.

The ArrayList Class

We discussed one of the most frequently used data structures, the array, earlier in Unit 8. Recall that arrays are **static data structures** that are built into the Java language. Arrays are useful for