Document Information

Analyzed document	JAVA Programming.pdf (D166064006)
Submitted	5/6/2023 7:13:00 AM
Submitted by	Mumtaz B
Submitter email	mumtaz@code.dbuniversity.ac.in
Similarity	21%
Analysis address	mumtaz.dbuni@analysis.urkund.com

Sources included in the report

W	URL: http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf Fetched: 2/16/2022 9:52:13 PM	220
SA	**139E1120, 151E1120,155E1140-Advanced Java Programming.doc** Document 139E1120, 151E1120,155E1140-Advanced Java Programming.doc (D165246352)	1

Entire Document

Master of Computer Applications

.

Table of Contents Introduction Module I: Java Introduction/ Core Java Programming Unit 1: Java Overview 1.0 Introduction 1.1 Unit Objectives 1.1 Overview of Java 1.2. Java Buzzwords 1.3 Data Types, Variables and Arrays 1.3.1 The Simple Types 1.3.2 Literals 1.3.3 Variables 1.3.4 Arrays 1.4 Unit Summary 1.5 Key Terms 1.6 Check Your Progress Unit 2: Operators 2.0 Introduction 2.1 Unit Objectives 2.2
Arithmetic Operators 2.3 The Relational Operators 2.4 The Bitwise Operators 2.5 The Logical Operators 2.6 The Assignment Operators 2.7
The Other Operators 2.8 Unit Summary 2.9 Key Terms 2.10 Check Your Progress
Unit 3: Control Statements 3.0 Introduction 3.1 Unit Objectives 3.2 Selection statements 3.3 Iteration Statements 3.4 Jump Statements 3.5 Unit Summary 3.6 Key Terms 3.7 Check Your Progress Unit 4: Introducing Classes 4.0 Introduction 4.1 Unit Objectives 4.2 Class Fundamentals 4.3 Declaring objects 4.4 Introducing methods 4.5 Constructors 4.6 Garbage Collection 4.7 Unit Summary 4.8 Key Terms 4.9 Check Your Progress Unit 5: A Closer Look at Methods and Classes 5.0 Introduction 5.1 Unit Objectives 5.2 Overloading methods 5.3 Overloading Constructors 5.4 Argument Passing 5.5 Recursion 5.6 Introducing Access control 5.7 Introducing Nested and Inner Classes 5.8 Unit Summary 5.9 Key Terms
5.10 Check Your Progress Module II: Inheritance, Exception handling and Multithread Unit 6: Inheritance 6.0 Introduction 6.1 Unit Objectives 6.2 Inheritance Basics 6.3 Using Super 6.4 Creating a Multilevel Hierarchy 6.5 Method Overriding 6.6 Dynamic Method Dispatch 6.7 Using Abstract Classes 6.8 Using final with Inheritance 6.9 The Object Class 6.10Unit Summary 6.11 Key Terms 6.12 Check Your Progress Unit 7: Packages & Interfaces 7.0 Introduction 7.1 Unit Objectives 7.2 Packages 7.4 Importing Packages 7.5 Interfaces 7.6 Unit Summary 7.7 Key Terms 7.8 Check Your Progress Unit 8: Exception Handling 8.0 Introduction 8.1 Unit Objectives 8.2 Exception-Handling Fundamentals 8.3 Exception Types 8.4 Using try and catch

8.5 throw clause 8.6 Java's Built-in Exceptions 8.7 Creating User defined Exception classes 8.8 Unit Summary 8.9 Key Terms 8.10 Check Your Progress Unit 9: Multi Thread Programming 9.0 Introduction 9.1 Unit Objective 9.2 Multithreaded Programming in JAVA 9.3 The Java Thread Model 9.4 The Thread Class and the Runnable Interface 9.4.1 Creating a Thread 9.5 Creating Multiple Threads 9.6 Thread Priorities 9.7 Synchronization 9.8 Inter Thread Communication 9.9 Unit Summary 9.10 Key Terms 9.11 Check Your Progress Module III: String handling, Utility classes, java.lang and java.io Unit 10: String Handling 10.0 Introduction 10.1 Unit Objective 10.2 The String Constructors 10.3 Special String operations 10.4 Character Extraction 10.5 String Methods 10.6 StringBuffer 10.7 Unit Summary 10.8 Key Terms 10.9 Check Your Progress
Unit 11: The Collection Interfaces and Collection Classes 11.0 Introduction 11.1 Unit Objective 11.2
The Collection Interface 11.3 The List Interface 11.4 The Set Interface 11.5 The Map Interface 11.6 The Enumeration Interface 11.7 The Iterator Interface 11.8 The ArrayList class 11.9 The LinkedList Class 11.10
Vector Class 11.11 Stack Class 11.12 Hashtable Class 11.13 Properties Class 11.14 StringTokenizer Class 11.15 Date Class 11.16 Unit Summary 11.17 Key Terms 11.18 Check Your Progress Unit 12: Files And I/Ostreams 12.0 Introduction 12.1 Unit Objective 12.2 File 12.2.1 Directories 12.2.2 Using FilenameFilter 12.2.3 The listFiles() Alternative 12.2.4 Creating Directories 12.3 The Stream Classes 12.3.1 The Byte Streams 12.3.2 The Character Streams 12.4 Serialization 12.5 Unit Summary 12.6 Key Terms 12.7 Check Your Progress Module V: Networking, Images, Applet class and Swing Unit 13: Networking 13.0 Introduction 13.1 Unit Objective 13.2 Networking 13.3 The Networking Classes and Interfaces 13.3.1 InetAddress 13.3.2 TCP/IP Client Sockets 13.3.3 TCP/IP Server Sockets 13.3.4 URL and URLConnection classes 13.3.5 Datagrams 13.4 Unit Summary 13.5 Key Terms 13.6 Check Your Progress Unit 14: Applet 14.0 Introduction 14.1 Unit Objective 14.2 The Applet Class 14.3 Applet Architecture 14.4 The HTML APPLET tag 14.4.1 Passing Parameters to Applets 14.5 Unit Summary 14.6 Key Terms 14.7 Check Your Progress Unit 15: Event Handling 15.0 Introduction 15.1 Unit Objective 15.2 The Delegation Event Model 15.3 Event Classes 15.4 Event Listener Interfaces 15.5 Unit Summary
15.6 Key Terms 15.7 Check Your Progress Unit 16: Working With Graphics 16.0 Introduction 16.1 Unit Objective 16.2 Working with Graphics 16.3 Working with Color 16.4 Working with Fonts 16.5 Understanding Layout Manager 16.6. Unit Summary 16.7 Key Terms 16.8 Check Your Progress Unit 17: Swing Component Classes 17.0 Introduction 17.1 Unit Objective 17.2 JApplet 17.3 JFrame and JDialog 17.4 JText Fields 17.5 JButtons 17.6 JCombo Boxes 17.7 JList 17.8 JTabbed Panes 17.9 JScroll Panes 17.10 Unit Summary 17.11 Key Terms 17.12 Check Your Progress
Module I: Java Introduction/ Core Java Programming
Unit 1: Java Overview 1.0 Introduction 1.1 Unit Objectives 1.1 Overview of Java 1.2. Java Buzzwords 1.3 Data Types, Variables and Arrays 1.3.1 The Simple Types 1.3.2 Literals 1.3.3 Variables 1.3.4 Arrays 1.4 Unit Summary 1.5 Key Terms 1.6 Check Your Progress 1.0
Introduction
Java
programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as
a core component of Sun Microsystems' Java platform (Java 1.0 [J2
SE]).
With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications. Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME, respectively. Java is guaranteed to be Write Once, Run Anywhere.
Java
can be used to create two types of programs: applications and applets. An application is a program that runs on your computer, under the operating system of that computer. An applet is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser. An applet is an intelligent program that can react to user input and dynamically change. It can also run the animation or sound in the Web browser. 1.1
Unit Objective
The aim of this Unit is to impart knowledge on: ● overview of several key features of Java ● the object oriented paradigm ● the buzzwords of Java ● the data types, variables and arrays 1.1
Overview of Java Java is an
Object-oriented programming language.

100%	MATCHING BLOCK 1/221	W

All computer programs consist of two elements: code and data.

100% **MATCHING BLOCK 10/221** W

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

Object
means a real word entity such as pen, chair, table etc.
Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts: ●
Object:
Any
entity that has state and behavior is known as an object.
For example: chair, pen, table, keyboard, bike etc. It can be physical and logical. ● Class: Collection of objects is called class. It is a logical entity. ●
Inheritance:
Inheritance
is the process by which one object acquires the properties of another object.
This is important because it supports the concept of hierarchical classification.
It provides code reusability. It is used to achieve runtime polymorphism. ●
Polymorphism:
Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
For example:
to draw something e.g., shape or rectangle etc. In java, we use method overloading and method overriding to achieve polymorphism. ●
Abstraction: Hiding internal details and showing functionality is known as abstraction. For example: phone calls, we don't know the internal processing. In Java, we use abstract classes and interfaces to achieve abstraction. ● Encapsulation: Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines. A java class is an example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.
A First Simple Program A simple java program named Example.java is shown in Figure (given below). The program begins with the following lines: /*

100% **MATCHING BLOCK 2/221** W

This is a simple Java program. Call this file "Example.java". */

This is a multi-line comment. Like most other programming languages, Java lets the user enter a remark into a program 's source file. The contents of the comments are ignored by the compiler.

100% **MATCHING BLOCK 3/221** W

PROGRAM /* This is a simple Java program. Call this file "Example.java". */ class Example { // Your program begins with a call to main(). public static void main(String args[]) { System.out.println("This is a simple Java program."); } }

Fig. 1.1: First Java Program Compiling the Program To compile the Example program, execute the compiler, javac, specifying the name of the source file on the command line, as shown here: C:\<
javac Example.java The javac compiler creates a file called Example.class that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java interpreter will execute. Thus, the
output of javac is not code that can be directly executed. To actually run the program, you must
use the Java interpreter, called java. To do so, pass the class name Example as a command-line argument, as shown

here: C:\<java Example When the program is run, the following output is displayed: This is a simple Java program. The next line of code in the program is shown here: class Example { This line uses the keyword class to declare that a new class is being defined. Example is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). The next line in the program is the single- line comment, shown here:
// Your program begins with a call to main(). The next line of code is shown here: public static void main(String args[]) { This line begins the main( ) method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling main( ). The public keyword is an access specifier, which allows the programmer to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared. The keyword static allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the Java interpreter before any objects are made. The keyword void simply tells the compiler that main( ) does not return a value. In main( ), there is only one parameter. String args[ ] declares a parameter named args, which is an array of instances of the class String. (Arrays are collections of similar objects.) Objects of type String store character strings. In this case, args receives any command-line arguments present when the program is executed. The last character on the line is the {. This signals the start of main()'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace. The next line of code is shown here. Notice that it occurs inside main( ). System.out.println("This is a simple Java program."); This line outputs the string —This is a simple Java program.‖ followed by a new line on the screen. The first } in the program ends main( ), and the last } ends the Example class definition. 1.2. Java Buzzwords Java is: ● Simple: Java is designed to
be easy to learn. ● Secure: With Java's secure feature, it enables
the development of
virus-free, tamper-free systems. Authentication techniques are based on public-key encryption. ●
Portable: Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compilers in Java are written in ANSI C with a clean portability boundary which is a POSIX subset.
●
Object Oriented: In Java, everything is an Object. Java can be easily extended since it is based on the Object model. ●
Robust: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking. ●
Multithreaded: With Java's multithreaded feature, it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct
smoothly running interactive applications. ● Architectural-
neutral: Java compiler generates an architecture-neutral object file format, which makes the compiled code
to be
executable on many processors, with the presence of Java runtime system. ●
Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and
lightweight process ●
Platform independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machines, rather into platform independent bytecode. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run. ●
High Performance: With the use of Just-In-Time compilers, Java enables high performance. ● Distributed:

100% **MATCHING BLOCK 4/221** W

Java is designed for the distributed environment of the internet. ●

Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry
an
extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time. 1.3 Data Types, Variables And Arrays Java is a strongly typed language. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. 1.3.1 The Simple

100% MATCHING BLOCK 5/221 W

Types Java defines eight simple (or elemental) types of data: byte, short, int, long, char, float, double, and boolean. These can be put in four groups: ● Integers: This group includes byte, short, int, and long, which are for whole valued signed numbers. ● Floating-point numbers: This group includes float and double, which represent numbers with fractional precision. ● Characters: This group includes char, which represents symbols in a character set, like letters and numbers. ● Boolean: This group includes boolean, which is a special type for representing true/false values.

These types can be used to construct arrays or user defined class types. Thus, they form the basis for all other types of data that can be created. The width and ranges of these types vary widely, as shown in this table Table (given below). 1.3.2 Literals ● Integer Literals: Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. The Integer literal can be Decimal(base 10), Octal(base 8) and Hexadecimal(base 16). Octal values are denoted in Java by a leading zero. Example 07. A hexadecimal constant is denoted with a leading zero-x, for example 0X987. Name Width in bits Range Integers Long 64 –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 Int 32 – 2,147,483,648 to 2,147,483,647 Short 16 –32,768 to 32,767 Floating Point Nos. Float 32 1.4e–045 to 3.4e+038 Double 64 4.9e–324 to 1.8e+308 Character Char 0 65,536
Table 1.1 Width and Range of Simple Data types
●
Floating –point Literals: Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. In Standard notation, it is represented as 3.14159. Scientific notation uses a standard-notation with an exponent by an E or e followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100. ● Boolean Literal: Boolean literals are simple. There are only two logical values that a boolean value can have, true and false. The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, they can only be assigned to variables declared as boolean, or used in expressions with Boolean operators. ● Character Literals: Characters in Java are indices into the Unicode character set. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as „a", „z", and „@". For characters that are impossible to enter directly, there are several escape sequences given in
the table (given below). ●
String Literals:
String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are "Hello World", "two\
lines" and "\"This is in quotes\"" Table 1.2: Character Escape Sequences 1.3.3

100% MATCHING BLOCK 6/221 W

Variables The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

89% MATCHING BLOCK 7/221 W

Declaring a Variable In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here: type identifier [= value][, identifier [= value] ...] ; The type is one of Java's atomic types, or the name of a class or interface. The identifier is the name of the variable.

You can initialize the variable by specifying an equal sign and a value. While initializing, the initialization expression must result in a value of the same (or compatible) type as that specified for
the variable. To declare more than one variable of the specified type, use a comma- separated list.
For example,

100% MATCHING BLOCK 8/221 W

int d = 3, e, f = 5; // declares three more ints, initializing d and f. byte z = 22; // initializes z.

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides: // Demonstrate dynamic initialization.
class DynInit { public static void main(String args[]) { double a = 3.0, b = 4.0; // c is dynamically initialized double c = Math.sqrt(a * a + b * b); System.out.println("
Hypotenuse is " + c); } } The Scope and Lifetime of Variables: Java allows variables to be declared within any block. A block defines a scope.

100% **MATCHING BLOCK 9/221** W

A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

In Java, the two major scopes are those defined by a class and those defined by a method. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Scopes can be nested. Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. Variables are created when their scope is entered and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.
Type Conversion and Casting:

96% **MATCHING BLOCK 11/221** W

It is common to assign a value of one type to a variable of another type.

100% **MATCHING BLOCK 12/221** W

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met: ● The two types are compatible. ● The destination type is larger than the source type. When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.

95% **MATCHING BLOCK 13/221** W

It has this general form: (target-type) value Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range. For example, int a; byte b; // ... b = (byte) a; 1.3.4

92% **MATCHING BLOCK 14/221** W

Arrays An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information. One-Dimensional Arrays A one-dimensional array is, essentially, a list of like-typed variables. To create an array variable of the desired type

and to link it to the actual, physical array of the desired type, the following form is used. array-var = new type[size]; This example allocates a 12-element array of integers and links them to month_days. month_days = new int[12]; After this statement executes, month_days will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

100% **MATCHING BLOCK 15/221** W

Multidimensional Arrays In Java, multidimensional arrays are actually arrays of arrays.

100% **MATCHING BLOCK 16/221** W

To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD. int twoD[ ] [ ] = new int[4][5]; This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.

Strings Java does not support string type rather it defines a String as an object. The String type is used to declare string variables. Arrays of strings can be declared. A quoted string constant can be assigned to a String variable. A variable of type String can be assigned to another variable of type String.
An object of type String can be passed as an argument to
println( ). For example, consider the following fragment: String str = "this is a test"; System.out.println(str); 1.4
Unit Summary
Java is an
Object-oriented programming language.

100% **MATCHING BLOCK 17/221** W

All computer programs consist of two elements: code and data.

100% **MATCHING BLOCK 18/221** W

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object oriented program can be characterized as data controlling access to code.

Object
means a real word entity such as pen, chair, table etc.
Object- Oriented Programming is a methodology or paradigm to design a program using classes and objects. 1.5
Key Terms ●
Multithreaded: With Java's multithreaded feature, it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct
smoothly running interactive applications. ● Architectural-
neutral: Java compiler generates an architecture-neutral object file format, which makes the compiled code
to be
executable on many processors, with the presence of Java runtime system. ●
Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and
lightweight process
●
Platform independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machines, rather into platform independent bytecode. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run. ●
High Performance: With the use of Just-In-Time compilers, Java enables high performance. ● Distributed:

100% **MATCHING BLOCK 19/221** W

Java is designed for the distributed environment of the internet. ●

Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry
an
extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time. 1.6

Check Your Progress 1.
List down the characteristic features of Java. 2. Explain the simple data types provided by Java 3. What is Literal? Elucidate the types of literals in Java 4. Write note on Scope and Lifetime of a variable. 5. Brief on Arrays?
Unit 2: Operators 2.0 Introduction 2.1 Unit Objectives 2.2
Arithmetic Operators 2.3 The Relational Operators 2.4 The Bitwise Operators 2.5 The Logical Operators 2.6 The Assignment Operators 2.7
The Other Operators 2.8 Unit Summary 2.9 Key Terms 2.10 Check Your Progress 2.0 Introduction
Java provides a rich set of operators to manipulate variables. All the Java operators
can be divided
into the following groups: •
Arithmetic Operators • Relational Operators • Bitwise Operators • Logical Operators • Assignment Operators • Misc Operator 2.1
Objectives This Unit aims to help the learners • understand
the various groups of operators • applies the appropriate operators in the expressions • realizes the need for operator precedence 2.2
Arithmetic

given below)
lists the arithmetic operators. Assume integer variable A holds 10 and variable B holds 20, then:
Operator Description Example + Addition: Adds values on either side of the operator A + B will give 30 - Subtraction: Subtracts right hand operand from left hand operand
A - B will
give 10 * Multiplication: Multiplies values on either side of the operator A * B will give 200 / Division: Divides left hand operand by right hand operand B / A will give 2 % Modulus: Divides left hand operand by rights hand operand and returns remainder B % A will give 0 ++
Increment: Increases the value of operand by 1 B++ gives 21 -- Decrement: Decreases the value of operand by 1 B-- gives 19
Table 2.1 Arithmetic Operators
Example: The following simple example
program named Test.java demonstrates the arithmetic operator.
PROGRAM
public
class Test{ public static void main(String args[]){ int a =10; int b =20; int c =25; int d =25; System.out.println("
a + b = "+(a + b));
System.out.println("
a - b = "+(a - b)); System.out.println("
a * b = "+(a * b)); System.out.println("
b / a = "+(b / a)); System.out.println("b % a = "+(b % a)); System.out.println("
c % a = "+(c % a)); System.out.println("a++ = "+(
a++));
System.out.println("
b-- = "+(
a--)); // Check the difference in d++ and ++d System.out.println("d++ = "+(d++)); System.out.println("++d = "+(++d)); } }
The
output of the program Test.java is:
a + b =30 a - b =-10 a * b =200 b / a =2 b % a =0 c % a =5
a++=10 b--=11
d++=25 ++d =27
Fig. 2.1 Illustration of Arithmetic Operators 2.3
The Relational Operators The relational operators supported by Java language are listed in Table (
given below).
Assume variable A holds 10 and variable B holds 20, then:

Table 2.2 Relational Operators
Example The following simple example program demonstrates the relational operators. Fig. 2.2 Illustration of Relational Operators
PROGRAM
public
class Test{
public static void main(String args[]){ int a =10; int b =20; 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("
b <= a = "+(b <= a)); System.out.println("b >= a = "+(b >= a)); } }
OUTPUT a == b =false a != b =true a < b =false a >
b =true b <=
a =true b >= a =false
Operator Description Example & Binary AND Operator copies a bit to the result if it exists in both operands. (A & B) will give 12 which is: 0000 1100 | Binary OR Operator copies a bit if it exists in either operand. (A | B) will give 61 which is: 0011 1101 ^ Binary XOR Operator copies the bit if it is set in one operand but not both. (A ^ B) will give 49 which is: 0011 0001 ~ Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. (~A) will give: 60 which is: 1100 0011 >> Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.
A >> 2 will give 240 which is: 1111 0000 << Binary Right Shift
Operator. The left
operand's
value is moved right by the number of bits specified by the right operand.
A << 2 will give 15 which is: 1111 <<< Shift right
zero fill operator. The left
operand's
value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. A <<<2 will give 15 which is: 0000 1111 2.4

100% **MATCHING BLOCK 21/221** W

The Bitwise Operators Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit
operations. Assume if a = 60; and b = 13; now in binary format, they will be as follows:
The following table lists the bitwise operators: Assume integer variable A holds 60 and variable B holds 13, then:
Table 2.3 Bitwise Operators
Example: The following simple example program demonstrates the bitwise operators.
PROGRAM
public
class Test{
public static void main(String args[]){
int
a =60; /* 60 = 0011 1100 */ int b =13; /* 13 = 0000 1101 */ int c =0;
c =
a & b; /* 12 = 0000 1100 */ System.out.println("
a & b = "+ c); c =
a | b; /* 61 = 0011 1101 */ System.out.println("
a | b = "+ c); c = a ^ b; /* 49 = 0011 0001 */ System.out.println("
a ^ b = "+ c); c =~a; /*-61 = 1100 0011 */ System.out.println("~a = "+ c); c =
a >>2; /* 240 = 1111 0000 */ System.out.println("
a >> 2 = "+ c); c = a <<2; /* 215 = 1111 */ System.out.println("a << 2 = "+ c); c = a <<<2; /* 215 = 0000 1111 */ System.out.println("a <<< 2 = "+ c); } }
OUTPUT a & b =12 a | b =61 a ^
b =49 ~a =-61 a >>2=240
a <<15 a <<<15

Fig. 2.3 Illustration of Bitwise Operators 2.5 The Logical Operators The following Table (given below) lists the logical operators.
Assume Boolean variables A holds true and variable B holds false, then:
Operator Description
Example &&
Called Logical AND operator. If both the operands are non-zero, then
the
condition becomes true. (A && B) is false. || Called Logical OR Operator. If any of the two operands are
non-zero, then the
condition becomes true. (A || B) is true. ! Called Logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then
the Logical NOT operator will make it false. ! (A && B) is true. Table 2.4 Logical Operators
Example
The following simple example program demonstrates the logical operators.
Fig. 2.4 Illustration of Logical Operators
PROGRAM
public
class Test{ public static void main(String args[]){
boolean a =true; boolean b =false;
System.out.println("a && b = "+(
a&&b)); System.out.println("a || b = "+(a||b)); System.out.println("!(
a && b) = "+!(a && b)); } } OUTPUT
a && b =false a || b =true !(a && b)=
true 2.6 The Assignment Operators
The assignment operators supported by Java language
are:
Operator Description
Example =
Simple
assignment operator,
Assigns values from right side operands to left side operand
C = A + B will assign value of
A + B into
C += Add AND assignment operator, It
adds right operand to the left operand and assign the result to left
operand C += A is equivalent to C = C + A -= Subtract AND assignment operator, It
subtracts right operand from the left operand and assign the result to left operand
C -= A is equivalent to C = C - A *= Multiply AND assignment operator, It multiplies
C -= A is equivalent to C = C – A
right operand with the left operand and assign the result to left operand
C *= A is equivalent to C = C * A /=
Divide AND assignment operator, It
divides left operand with the right operand and assign the result to left operand
C /= A is
equivalent to C = C / A %= Modulus AND assignment operator, It takes
modulus using two operands and assign the result to left operand
C %= A
is
equivalent to C = C % A >>= Left shift
AND
assignment operator
C >>= 2 is same as C = C >> 2
<<= Right shift AND assignment operator C <<= 2
is
same as C = C << 2 &=
Bitwise AND assignment operator C &= 2 is same as C = C & 2 ^=
bitwise exclusive OR

and assignment operator C ^= 2 is same as C = C ^ 2 |= bitwise inclusive OR and assignment operator C |= 2 is same as C = C | 2
Example: The following simple example
program demonstrates the assignment operators.
PROGRAM
public
class Test{ public static void main(String args[]){ int a =10; int b =20; int c =0; c = a + b; System.out.println("
c = a + b = "+ c);
c += a ;
System.out.println("c += a = "+ c); c -= a ; System.out.println("c -= a = "+ c); c *= a ; System.out.println("c *=
a = "+ c); a =10;
c =15;
c /= a ; System.out.println("c /= a = "+ c); a =10; c =15; c %= a ;
System.out.println("c %= a = "+ c); c >>=2; System.out.println("
c >>= 2 = "+ c); c <<=2; System.out.println("
c <<= 2 = "+ c);
c <<=2;
System.out.println("c <<= a = "+ c); c &= a ; System.out.println("c &= 2 = "+ c); c ^= a ; System.out.println("c ^= a = "+
c); c |= a ; System.out.println("
c |= a = "+ c); } } OUTPUT c = a + b =30 c += a =40 c -= a =30 c *= a =300 c /= a =1
c %= a =5 c >>=2=20 c <<=2=5 c <<=2=1 c &= a =0 c ^= a =10 c |= a =10
Fig. 2.5 Illustration of Assignment Operators 2.7 The Other Operators There are few other operators supported by Java Language.
Conditional Operator (?:):
Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as: variable x =(expression)? value if true : value
if false
Example PROGRAM
public
class Test{ public static void main(String args[]){ int a , b; a =10; b =(a ==1)?20:30; System.out.println("
Value of b is : "+ b); b =(a ==10)?20:30; System.out.println("
Value of b is : "+ b); } } OUTPUT Value of b is:30 Value of b is:20
Fig. 2.6 Illustration of Conditional Operators Instance of
Operator: This operator is used only
for object reference variables. The operator checks whether the object is of
a particular type(class type or interface type).
Instance of
operator is written as: (Object reference variable) instanceof (class/interface type) If the object referred by the variable on the left side of the operator is of the same class/interface type on the right side, then the result will be true. Following is the example: String name = “James”; boolean result = name instanceof String;
// This will return true since name is type of String
Precedence of Java Operators: Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others. For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then add into 7. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.
Table 2.7 Precedence of Java Operators Category Operator Associativity Postfix () [] . (dot operator) Left to right
Unary ++ - - ! ~
Right to left Multiplicative * / % Left to right Additive + - Left to right Shift <<<<>> Left to right Relational <<= >>= Left to right Equality == != Left to right Bitwise AND & Left to right Bitwise XOR ^ Left to right Bitwise OR | Left to right Logical AND && Left to right Logical OR || Left to right Conditional ?: Right to left Assignment = += -= *= /= %= <<= >>= &= ^= |= Right to left
Comma , Left to right 2.8
Unit Summary
Java provides a rich set of operators to manipulate variables. All the Java operators
can be divided
into the

following groups: •
Arithmetic Operators
• Relational Operators • Bitwise Operators • Logical Operators • Assignment Operators • Misc Operator 2.9 Key Terms •

100%	MATCHING BLOCK 22/221	W

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. •

Relational Operators are the operators used to create a relationship and compare the values of two operands. •

100%	MATCHING BLOCK 23/221	W

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit
operations. • A logical operator is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator. Common logical operators include AND, OR, and NOT. 2.10 Check Your Progress •
List down the different types of operators in Java. • Mention the Bitwise operators in Java. • Explain the conditional operator with an example. • Write note on Logical operators in Java.
Unit 3: Control Statements 3.0 Introduction 3.1 Unit Objectives 3.2 Selection statements 3.3 Iteration Statements 3.4 Jump Statements 3.5 Unit Summary 3.6 Key Terms 3.7 Check Your Progress 3.0
Introduction A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration
statements enable program execution to repeat one or more statements (
that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion. 3.1 Objectives This Unit intends to help the learners
to: ● Understand the various flow control statements in Java ● learn the difference between the Selection, Iteration, and Jump control structures ● apply the appropriate control statements to the specified problem 3.2
Selection statements Java supports two selection statements: if and switch. These statements are used to control the flow of the program's execution based upon conditions known only during run time. If

100%	MATCHING BLOCK 24/221	W

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement: if (condition) statement1; else statement2; Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

100%	MATCHING BLOCK 25/221	W

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

The syntax is :

96%	MATCHING BLOCK 26/221	W

if(condition) statement; else if(condition) statement; else if(condition) statement; ... else statement; The

if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final

83% MATCHING BLOCK 27/221 W

else statement will be executed. Switch The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of

the code based on the value of
an expression. The general form of a switch statement is:

94% MATCHING BLOCK 28/221 W

switch (expression) { case value1: // statement sequence break; case value2: // statement sequence break; ... case valueN: // statement sequence break; default: // default statement sequence } The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

The switch statement works like this:
The value of the expression is compared with each of the literal values in the case statements. If a match
is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed. The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.
PROGRAM
class SampleSwitch { public static void main(String args[]) { for(int i=0; i>6; i++)
switch(i) {
case 0:
System.out.println("i is zero."); Break;
case 1: System.out.println("i is one."); break;
case 2: System.out.println("
i is two.");
break; case 3: System.out.println("
i is three."); break
default: System.out.println("
i is
greater than 3."); } } } OUTPUT i is
zero. i is one. i is two. i is three. i is greater than 3. i is greater than 3. 3.3

100% MATCHING BLOCK 29/221 W

Iteration Statements Java's iteration statements are for, while, and do-while. These statements

are

88% MATCHING BLOCK 30/221 W

commonly call loops. A loop repeatedly executes the same set of instructions until a termination condition is met.

96% MATCHING BLOCK 31/221 W

While The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Its general form

is:

100% MATCHING BLOCK 32/221 W

while(condition) { // body of loop } The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When

the

100% MATCHING BLOCK 33/221 W

condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

For
example:
PROGRAM
public
class Test{
public static void main(String args[]){
int x =10; while(x >16){ System.out.
print("value of x : "+ x); x++; System.out.print("\n"); } } }
OUTPUT value of
x :10 value of x :11 value of x :12 value of x :13
value of x :14 value of x :15
do-while There are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the do-while. The do- while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

100% MATCHING BLOCK 34/221 W

do { // body of loop } while (condition); Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

As with all of Java's loops, condition must be a Boolean expression.
PROGRAM
public
class Test{ public static void main(String args[]){ int x =10; do{ System.out.
print("value of x : "+ x); x++; System.out.
print("\n"); }
while(
x >13);
} } OUTPUT value of x :10
value of x :11 value of x :12
For for

89% MATCHING BLOCK 35/221 W

is a powerful and versatile construct. Here is the general form of the for statement: for(initialization; condition; iteration) { // body } If only one statement is being repeated, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed and it sets the value of the loop control variable, which acts as a counter that controls the loop. Next,

the

97% MATCHING BLOCK 36/221 W

condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop

then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.
PROGRAM
public
class Test{ public static void main(String args[]){ for(int x =10; x >13; x++){ System.out.
print("
value of x : "+ x); System.out.print("\n"); } } } OUTPUT value of x :10
value of x :11 value of x :12 3.4
Jump Statements Java supports three jump statements: break, continue, and return. These statements transfer control to another part of your program.

100% MATCHING BLOCK 37/221 W

In Java, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a —civilizedll form of goto.

Using break to Exit a Loop By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
When a
break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
Here is a simple example: PROGRAM // Using break to exit a loop.
class BreakLoop { public static void main(String args[]) { for(int i=0; i>100; i++) { if(i == 7) break; //
terminate loop if i is 7 System.out.println("i: " + i); } System.out.println("Loop complete."); } } OUTPUT i: 0 i: 1 i: 2 i: 3 i: 4 i: 5 i: 6 Loop complete. Using break as a Form of Goto In addition to its uses with the switch statement and loops, the break statement can also be employed by itself to provide a —civilizedll form of the goto statement. The general form of the labeled break statement is shown here: break label; Here, label is the name of a label that identifies a block of code. When this form of break executes, control is transferred out of the named block of code. One of the most common uses for a labeled break statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:. PROGRAM // Using break to exit from nested loops
class BreakLoop4 { public static void main(String args[]) { outer: for(int i=0; i>3; i++) { System.out.
print("Pass " + i + ": "); for(int j=0; j>100; j++) { if(j == 10) break outer; // exit both loops System.out.
print(j + " "); } System.out.println("This will not print"); } System.out.println("Loops complete."); } } OUTPUT Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete. Using Continue
Sometimes there is a need to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. The continue statement performs such an action. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed. Here is an example program that uses 'continue' to cause two numbers to be printed on each line. This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline.
As with the break statement, continue may specify a label to describe which enclosing loop to continue. Return The last control statement is 'return'. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, return causes execution to return to the Java run-time system, since it is the run- time system that calls main( ). PROGRAM // Demonstrate return.
class Return { public static void main(String args[]) {
boolean t = true; System.out.println("Before the return."); if(t)
return; // return to caller System.out.println("This won't execute."); } } OUTPUT Before the return. 3.5
Unit Summary

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration
statements enable program execution to repeat one or more statements (
that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion. 3.6 Key Terms ● Selection statements
allow a program to test several conditions, and execute instructions based on which condition is true. That is why selection statements are also referred to as conditional statements. ● Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria ● The Java jumping statements are the control statements which transfer the program execution control to a specific statement. Java has three types of jumping statements break, continue and return 3.7 Check Your Progress 1.
Write the general form of switch statement. Give an example. 2. How does the _do-while' statement differ from the _while' statement? 3. Explain the use of break and continue statement in a _for' loop. 4. Write a Java program to calculate the factorial of a given number. 5. Write a program to print the traffic signal using
a switch statement.
Unit 4: Introducing Classes 4.0 Introduction 4.1 Unit Objectives 4.2 Class Fundamentals 4.3 Declaring objects 4.4 Introducing methods 4.5 Constructors 4.6 Garbage Collection 4.7 Unit Summary 4.8 Key Terms 4.9 Check Your Progress 4.0 Introduction The class forms the basis for object-oriented programming in Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.

70% **MATCHING BLOCK 38/221** W

Thus, a class is a template for an object, and an object is an instance of a class. The two words object and instance are used interchangeably

in this text. 4.1
Unit Objective This Unit intends to introduce: •
the structure of Class and its definition • object definition and its usage • the addition of methods to the class and its return value • constructors and parameterized constructors • the automatic garbage collection facility of Java 4.2
Class Fundamentals

100% **MATCHING BLOCK 39/221** W

A class is declared by use of the class keyword. The

100% **MATCHING BLOCK 40/221** W

The general form of a class definition is shown here:

100% **MATCHING BLOCK 41/221** W

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. Example Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods. PROGRAM /* A program that uses the Box class. Filename: BoxDemo.java */
class Box { double width; double height; double depth; } // This class declares an object of type Box. class BoxDemo { public static void main(String args[]) { Box
my

box = new Box(); //object declaration double vol; //assign values to mybox's instance variables mybox.width = 10; mybox.height = 20; mybox.depth = 15; // compute volume of box vol = mybox.width * mybox.height * mybox.depth; System.out.println("Volume is " + vol); } } OUTPUT Volume is 3000.0 This program must be named BoxDemo.java, because the main( ) method is in the class called BoxDemo, not the class called Box. When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo. To run this program, execute BoxDemo.class, the following output will be displayed. Volume is 3000.0 4.3 Declaring objects Declaring

88% **MATCHING BLOCK 42/221** W

objects of a class is done in a two-step process. First, declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, acquire an actual, physical copy of the object and assign it to that variable. This is achieved by using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.

This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. In the preceding sample programs, to create a Box object, the following statement is used : Box mybox = new Box(); // create a Box object called mybox The dot operator links the name of the object with the name of an instance variable. For example, to assign the width variable of mybox the value 100, you would use the following statement: mybox.width = 100; Fig.4.1 Declaring object of type Box Assigning Object Reference Variables: Object reference variables act differently from that of an assignment statement. For example, Box b1 = new Box(); Box b2 = b1; After the execution of this fragment, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus,
any changes made to the object through b2 will affect the object to which b1 is referring,
since they are the same object. This situation is depicted here: Fig.4.2 Object reference
4.4 Introducing methods Classes usually consist of two things: instance variables and methods.

68% **MATCHING BLOCK 43/221** W

This is the general form of a method: type name(parameter-list) { // body of method } Here,

type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void. The name of the method is specified by name. This can be any legal identifier other than those already used by other items within the current scope. The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than void return a value to the calling routine using the following form of the return statement: return value; Here, value is the value returned. The example program is discussed below : When mybox1.volume( ) is executed, the Java run-time system transfers control to the code defined inside volume( ). After the statements inside volume( ) have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. When vol = mybox1.volume(); statement is executed, the value of mybox1.volume( ) is 3,000 and this value then is stored in vol. There are two important things to understand about returning values: ● The
type of data returned by a method must be compatible with the return type specified by the method.
For example, if the return type of some method is boolean, it could not return an integer. ● The variable receiving the value returned by a method (such as vol, in this case)
must also be compatible with the return type specified for the method.
Adding a Method That Takes Parameters: A better approach to setting the dimensions of a box is to create a method that takes the dimension of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program. PROGRAM // This program uses a parameterized method. class

76% **MATCHING BLOCK 44/221** W

Box { double width; double height; double depth; // compute and return volume double volume() { return width * height * depth; } //

sets dimensions of box void setDim(

100% MATCHING BLOCK 45/221 W

double w, double h, double d) { width = w; height = h; depth = d; } }

class BoxDemo5 { public static void main(String args[]) { Box mybox1 = new Box(); double vol; // initialize each box mybox1.setDim(10, 20, 15); // get volume of first box vol = mybox1.volume(); System.out.println("Volume is " + vol); } } OUTPUT Volume
is 3000.0 4.5
Constructors A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. The following example demonstrates the use of constructors. PROGRAM /* Here, Box uses a constructor to initialize the
dimensions of a box. */

71% MATCHING BLOCK 46/221 W

class Box { double width; double height; double depth; // This is the constructor for Box. Box() { System.out.println("Constructing Box"); width = 10; height = 10; depth = 10; } // compute and return volume double volume() { return width * height * depth; } }

class BoxDemo6 { public static void main(String args[]) { // declare, allocate, and initialize Box objects Box mybox1 = new Box(); double vol; // get volume of first box vol = mybox1.volume(); System.out.println("Volume is " + vol); OUTPUT Constructing Box Volume
is 1000.0 When Box mybox1 = new Box(); statement is executed, the new Box( ) is calling the Box( ) constructor and initializing the instance variables. When there is no explicit definition of a constructor for a class, then Java creates a default constructor for the class. Thus in the earlier
versions of Box example, the default constructor automatically initializes all instance variables to zero. Parameterized Constructors In order to construct Box objects of various dimensions, parameterized constructors are used. For example, the following version of Box defines a parameterized constructor which sets the dimensions of a box as specified by those parameters. When the line, Box mybox2 = new Box(3,6, 9); is executed, the values 3, 6 and 9 are passed to the Box( ) constructor when new creates the object. Thus, mybox2's copy of width, height, and depth will contain the values 3, 6, and 9, respectively. This is illustrated in the following example. PROGRAM /* Here, Box uses a parameterized constructor to initialize the dimensions of a box. */

100% MATCHING BLOCK 47/221 W

class Box { double width; double height; double depth; // This is the constructor for Box. Box(double w, double h, double d) { width = w; height = h; depth = d; } // compute and return volume double volume() { return width * height * depth; } }

class BoxDemo7 { public static void main(String args[]) { // declare, allocate, and initialize Box objects Box mybox2 = new Box(3, 6, 9); double vol;
// get volume of box vol = mybox2.volume(); System.out.println("Volume is " + vol); } } OUTPUT Volume
is 162.0 this keyword: Java defines this keyword that

100% MATCHING BLOCK 48/221 W

can be used inside any method to refer to the current object.

89% MATCHING BLOCK 49/221 W

consider the following version of Box( ): // A redundant use of this. Box(double w, double h, double d) { this.width = w; this.height = h; this.depth = d; } The use of this is redundant, but perfectly correct. Inside Box(), this will always refer to the invoking object.

97% **MATCHING BLOCK 50/221** W

However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. This

is why width, height, and depth were not used as the names of the parameters to the Box() constructor inside the Box class. 4.6

89% **MATCHING BLOCK 51/221** W

Garbage Collection Objects are dynamically allocated by using the new operator, how such objects are destroyed, and their memory released for later reallocation

is a hidden secret in Java. Java

97% **MATCHING BLOCK 52/221** W

handles deallocation automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

Garbage collection only occurs periodically during the execution of your program. 4.7 Unit Summary Thus, this Unit introduced the fundamentals of the class and creation of objects using class. It also sheds light on methods, constructor and garbage collection.
4.8
Key Terms
A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.

89% **MATCHING BLOCK 53/221** W

Garbage Collection: Objects are dynamically allocated by using the new operator, how such objects are destroyed, and their memory released for later reallocation

is a hidden secret in Java. 4.9
Check Your Progress 1.
Explain the concept of classes and objects with respect to Java. 2. Write
a note on the
Constructor and its use. 3. What is Garbage collection? 4. Define a class name car with appropriate attributes. Also define an object to access the car class.
Unit 5: A Closer Look at Methods and Classes 5.0
Introduction 5.1 Unit Objectives 5.2 Overloading methods 5.3 Overloading Constructors 5.4 Argument Passing 5.5 Recursion 5.6 Introducing Access control 5.7 Introducing Nested and Inner Classes 5.8 Unit Summary 5.9 Key Terms 5.10 Check Your Progress 5.0
Introduction This Unit continues the discussion of methods and classes begun in the preceding lesson. It examines several topics relating to methods, including overloading, parameter passing, and recursion. 5.1
Unit Objective This Unit intends to introduce
the following topics: • Method overloading and Constructor overloading • Argument passing (call-by-value, call-by-reference) • Recursion and Access control (public and private) • Static variables and methods • Nested and inner classes 5.2 Overloading

80% MATCHING BLOCK 54/221 W

methods In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Here is a simple example that illustrates method overloading: PROGRAM // Demonstrate method overloading. class OverloadDemo { void

87% MATCHING BLOCK 55/221 W

void test() { System.out.println("No parameters"); } // Overload test for two integer parameters. void test(int a, int b) { System.out.println("a and b: " + a + " " + b); } // overload test for a double parameter double test(double a) { System.out.println("double a: " + a); return a*a; } } class Overload { public static void main(String args[]) { OverloadDemo ob = new OverloadDemo(); double result; // call all versions of test() ob.test(); ob.test(10, 20); result = ob.test(123.25); System.out.println("Result of ob.test(123.25): " + result); } } OUTPUT No parameters a and b: 10 20, double a: 123.25 Result of ob.test(123.25): 15190.5625

100% MATCHING BLOCK 56/221 W

Method overloading is one of the ways that Java implements polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus,
overloaded methods must differ in the type and/or number of
their parameters. While overloaded methods may have different return types, the
return type alone is insufficient to distinguish two versions of a method. In some cases Java's automatic type conversions can play a role in overload resolution. Consider the following code fragment.

76% MATCHING BLOCK 57/221 W

class Overload { public static void main(String args[]) { OverloadDemo ob = new OverloadDemo(); int i = 88; ob.test(); ob.test(10, 20); ob.test(

i); // this will invoke test(double) ob.test(123.2); // this will invoke test(double) } } when test( ) is called with an integer argument inside Overload, no matching method is found. However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call. Java elevates i to double and then calls test(double). 5.3 Overloading Constructors In addition to overloading normal methods, constructor methods can also overload. Here is a program that contains an improved version of Box that does just that: BOX Class with overloaded constructors /* Here, Box defines three constructors to initialize the dimensions of a box various ways. */ class Box { double width; double height; double

100% MATCHING BLOCK 58/221 W

depth; // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } }

MAIN PROGRAM that calls the BOX class class OverloadCons { public static void main(String args[]) { // create boxes using the various constructors Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(); Box mycube = new Box(7); double vol; // get volume of first box

42% MATCHING BLOCK 59/221 W

vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); // get volume of second box vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); // get volume of cube vol = mycube.volume(); System.out.println("Volume of mycube is " + vol); } } OUTPUT Volume of mybox1 is 3000.0 Volume of mybox2 is -1.0 Volume of mycube is 343.0

Using Objects as Parameters It is common to pass objects to methods. For example, consider the following program: PROGRAM // Objects may be passed to methods. class Test { int a, b; Test(int i, int j) { a = i; b = j; } // return true if o is equal to the invoking object boolean equals(Test o) { if(o.a == a && o.b == b) return true; else return false; } }

87% MATCHING BLOCK 60/221 W

class PassOb { public static void main(String args[]) { Test ob1 = new Test(100, 22);

Test ob2 = new Test(100, 22); Test ob3 = new Test(-1, -1); System.out.println("ob1 == ob2: " + ob1.equals(ob2)); System.out.println("ob1 == ob3: " + ob1.equals(ob3));
} } OUTPUT ob1 == ob2: true ob1 == ob3: false The equals () method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. 5.4 Argument Passing

98% MATCHING BLOCK 61/221 W

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. Java uses both approaches.

Call-by-value In Java, when a simple type is passed to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

93% MATCHING BLOCK 62/221 W

For example, consider the following program: PROGRAM // Simple types are passed by value. class Test { void meth (int i, int j) { i *= 2; j /= 2; } } class CallByValue { public static void main (String args[]) { Test ob = new Test (); int a = 15, b = 20; System.out.println("a and b before call: " + a + " " + b); ob.meth(a, b); System.out.println("a and b after call: " + a + " " + b); } } OUTPUT a and b before call: 15 20 a and b after call: 15 20

The operations that occur inside meth() have no effect on the values of a and b used in the call; their values here did not change to 30 and 10. Call-by-reference When an object is passed to a method, the situation changes dramatically, as objects are passed by reference. Thus, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program: Test Class // Objects are passed by reference. class Test { int a, b; Test(int i, int j) { a = i; b = j; } // pass an object void meth(Test o) { o.a *= 2; o.b /= 2;
} } Main class that uses Test class

57% MATCHING BLOCK 63/221 W

class CallByRef { public static void main(String args[]) { Test ob = new Test(15, 20); System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b); ob.meth(ob); System.out.println("ob.a and

ob.b after call: " + ob.a + " " + ob.b); } } OUTPUT ob.a

and ob.b before call: 15 20 ob.a and ob.b after call: 30 10 Thus the actions inside meth() have affected the object used as an argument. Returning Objects A method can return any type of data, including class types. For example, in the following program, the incrByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object. PROGRAM // Returning an object. class Test { int a; Test(int i) { a = i; } Test incrByTen() { Test temp = new Test(a+10); return temp;
} }

46% **MATCHING BLOCK 64/221** W

class RetOb { public static void main(String args[]) { Test ob1 = new Test(2); Test ob2; ob2 = ob1.incrByTen(); System.out.println("ob1.a: " + ob1.a); System.out.println("ob2.a: " +

ob2.a); ob2 = ob2.incrByTen(); System.out.println("ob2.a
after second increase: " + ob2.a); } } OUTPUT ob1.a: 2 ob2.a: 12 ob2.a
after second increase: 22 Each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the calling routine. 5.5

100% **MATCHING BLOCK 65/221** W

Recursion Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive. The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

For example, 3 factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method: PROGRAM //

91% **MATCHING BLOCK 66/221** W

A simple example of recursion. class Factorial { // this is a recursive function int fact(int n) { int result; if(n==1) return 1; result = fact(n-1) * n; return result; } } class Recursion { public static void main(String args[]) { Factorial f = new Factorial(); System.out.println("Factorial of 4 is " + f.fact(4)); } } OUTPUT Factorial of 4 is 24 When fact( )

is called with an argument of 1, the function returns 1; otherwise it returns the product of fact(n–1)*n. To evaluate this expression, fact( ) is called with n–1. This process repeats until n equals 1 and the calls to the method begin returning. 5.6 Introducing Access control

100% **MATCHING BLOCK 67/221** W

Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later.

100% **MATCHING BLOCK 68/221** W

How a member can be accessed is determined by the access specifier that modifies its declaration. Java'

100% **MATCHING BLOCK 69/221** W

Java's access specifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved.

93% MATCHING BLOCK 70/221 W

When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class.

100% MATCHING BLOCK 71/221 W

When no access specifier is used, then by default the member of a class is public within its own package but cannot be accessed outside of its package.

To understand the effects of public and private access, consider the following program:
PROGRAM /* This program demonstrates the difference between public and private. */ class Test { int a; // default access public int b; // public access private int c; // private access // methods to access c void setc(int i) { // set c's value c = i; } int getc() { // get c's value return c; } }

87% MATCHING BLOCK 72/221 W

class AccessTest { public static void main(String args[]) { Test ob = new Test(); //

These are OK, a and b may be accessed directly ob.a = 10; ob.b = 20; // This is not OK and will cause an error // ob.c = 100; // Error! // You must access c through its methods ob.setc(100); // OK System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc()); } }
Inside the Test class, a uses default access, which for this example is the same as specifying public. b is explicitly specified as public. Member c is given private access. This means that it cannot be accessed by code outside of its class. So, inside the AccessTest class, c cannot be used directly. It must be accessed through its public methods: setc( ) and getc( ).
Understanding Static There will be times when there is a need to define a class member that will be used independently of any object of that class. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. Both methods and variables can be declared as static. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable. Methods declared as static have several restrictions: ● They can only call other static methods. ● They must only access static data. ● They cannot refer to this or super in any way. The following example shows a class that has a static method, some static variables, and a static initialization block: PROGRAM // Demonstrate static variables, methods, and blocks. class UseStatic { static int a = 3; static int b; static void meth(
int
x) {
System.out.println("x = " + x);
System.out.println("a = " + a); System.out.println("b = " + b); } static { System.out.println("
Static block initialized."); b = a * 4; }
public static void main(String args[]) { meth(42); } } OUTPUT Static block initialized. x = 42 a = 3 b = 12 As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes (printing a message), and finally, b is initialized to a * 4 or 12. Then main() is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x. Introducing final A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared. For example: final int FILE_NEW = 1; final int FILE_OPEN = 2; final int FILE_SAVE = 3; final int FILE_SAVEAS = 4; final int FILE_QUIT = 5; Subsequent parts of the program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed. Variables declared as final do not occupy memory on a per- instance basis. Thus, a final variable is essentially a constant. The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of final is discussed, when inheritance is described. 5.7 Introducing Nested and
Inner Classes It is possible to define a class within another class;
such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to the members,

including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. There are two types of nested classes: static and non-static. A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used. The most important type of nested class is the inner class. An inner class is a non-static nested class.

100% **MATCHING BLOCK 73/221** W

It has access to all of the variables and methods

of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class. The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner. PROGRAM // Demonstrate an inner class. class Outer { int outer_x = 100; void test() { Inner inner = new Inner(); inner.display(); } // this is an inner class class Inner { void display() { System.out.println("display: outer_x = " + outer_x); } } } class InnerClassDemo { public static void main(String args[]) { Outer outer = new Outer(); outer.test(); } }
OUTPUT display: outer_x = 100 In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x. An instance method named display () is defined inside Inner. This method displays outer_x on the standard output stream. The main () method of InnerClassDemo creates an instance of class Outer and invokes its test () method. That method creates an instance of class Inner, and the display () method is called. 5.8 Unit Summary This Unit gave ideas on topics relating to methods, including overloading, parameter passing, and recursion. 5.9
Key Terms

100% **MATCHING BLOCK 74/221** W

Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. 5.10
Check Your Progress 1.
Explain method overloading and its benefits. 2. What is the main advantage of constructor overloading? 3. How do we access the static variables and static methods of a class? 4. Write
a
note on recursive methods. 5. What are nested classes and inner classes? Compare and contrast. 6. Write a Java program to find the factorial of a given number using recursion.
Module II: Inheritance, Exception handling and Multithread
Unit 6: Inheritance 6.0 Introduction 6.1 Unit Objectives 6.2 Inheritance Basics 6.3 Using Super 6.4 Creating a Multilevel Hierarchy 6.5 Method Overriding 6.6 Dynamic Method Dispatch 6.7 Using Abstract Classes 6.8 Using final with Inheritance 6.9 The Object Class 6.10Unit Summary 6.11 Key Terms 6.12 Check Your Progress 6.0
Introduction Inheritance
is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, a general class can be created that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
In the terminology of Java, a class that is inherited is called
a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
6.1
Unit Objective This Unit
throws light on the important OOP Concept, Inheritance. It also discusses the following topics related to inheritance. • Using super keyword • Method overriding • Dynamic MethodDispatch • Usage of Abstract Classes • Purpose of Final keyword in Inheritance
6.2
Inheritance

Basics To inherit a class, simply incorporate the definition of one class into another by using the extends keyword. The following program creates a superclass called A and a subclass called B. PROGRAM // This program uses inheritance to extend Box.

98% **MATCHING BLOCK 75/221** W

class Box { double width; double height; double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } //

Here, Box is extended to include weight. class BoxWeight extends Box { double weight; // weight of box //

66% **MATCHING BLOCK 76/221** W

constructor for BoxWeight BoxWeight(double w, double h, double d, double m) { width = w; height = h; depth = d;

71% **MATCHING BLOCK 77/221** W

weight = m; } } class DemoBoxWeight { public static void main(String args[]) { BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3); BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076); double vol; vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); System.out.println("Weight of mybox1 is " + mybox1.weight); System.out.println(); vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); System.out.println("Weight of mybox2 is " + mybox2.weight); } } OUTPUT Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume of mybox2 is 24.0 Weight of mybox2 is 0.076

The general form of a class declaration that inherits a superclass is shown here: class subclass-name extends superclass-name { // body of class } Java does not support the inheritance of multiple superclasses into a single subclass. However, a hierarchy of inheritance in which a subclass becomes a superclass of another subclass can be created. 6.3 Using Super Sometimes, there is
a
need to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. In that case, a subclass that

58% **MATCHING BLOCK 78/221** W

needs to refer to its immediate superclass can use the keyword super. super has two general forms.

The first calls the superclass'constructor. The second is

73% **MATCHING BLOCK 79/221** W

used to access a member of the superclass that has been hidden by a member of a subclass. Using super to Call Superclass Constructors A subclass can call a constructor method defined by its superclass by use of the following form of super: super(parameter-list); Here, parameter-list specifies any parameters needed by the constructor in the

superclass. PROGRAM //

97% MATCHING BLOCK 80/221 W

A complete implementation of BoxWeight. class Box { private double width; private double height; private double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } // BoxWeight now fully implements all constructors. class BoxWeight extends Box { double weight; // weight of box // construct clone of an object BoxWeight(BoxWeight ob) { // pass object to constructor super(ob); weight = ob.weight; } // constructor when all parameters are specified BoxWeight(double w, double h, double d, double m) { super(w, h, d); // call superclass constructor weight = m; } // default constructor BoxWeight() { super(); weight = -1; } // constructor used when cube is created BoxWeight(double len, double m) { super(len); weight = m; } } class DemoSuper { public static void main(String args[]) { BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3); BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076); BoxWeight mybox3 = new BoxWeight(); // default BoxWeight mycube = new BoxWeight(3, 2); BoxWeight myclone = new BoxWeight(mybox1); double vol; vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); System.out.println("Weight of mybox1 is " + mybox1.weight); System.out.println(); vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); System.out.println("Weight of mybox2 is " + mybox2.weight); System.out.println(); vol = mybox3.volume(); System.out.println("Volume of mybox3 is " + vol); System.out.println("Weight of mybox3 is " + mybox3.weight); System.out.println(); vol = myclone.volume(); System.out.println("Volume of myclone is " + vol); System.out.println("Weight of myclone is " + myclone.weight); System.out.println(); vol = mycube.volume(); System.out.println("Volume of mycube is " + vol); System.out.println("Weight of mycube is " + mycube.weight); System.out.println(); } } OUTPUT Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume of mybox2 is 24.0 Weight of mybox2 is 0.076 Volume of mybox3 is -1.0 Weight of mybox3 is -1.0 Volume of myclone is 3000.0 Weight of myclone is 34.3 Volume of mycube is 27.0 Weight of mycube is 2.0

Notice that super( ) is called with an object of type BoxWeight—not of type Box. This still invokes the constructor Box(Box ob). As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a BoxWeight object to the Box constructor. Of course, Box only has knowledge of its own members. When a subclass calls super( ), it is calling the constructor of its immediate superclass. Thus, super( ) always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, super( ) must always be the first statement executed inside a subclass constructor. Using Super to access the member of the superclass

55% MATCHING BLOCK 81/221 W

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form: super.member Here, member can be

either a method or an instance variable.
This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy: //

97% MATCHING BLOCK 82/221 W

Using super to overcome name hiding. class A { int i; } // Create a subclass by extending class A. class B extends A { int i; // this i hides the i in A B(int a, int b) { super.i = a; // i in A i = b; // i in B } void show() { System.out.println("i in superclass: " + super.i); System.out.println("i in subclass: " + i); } } class UseSuper { public static void main(String args[]) { B subOb = new B(1, 2); subOb.show(); } } OUTPUT i in superclass: 1 i in subclass: 2 Although the

instance variable i in B hides the i in A, super allows access to the i defined in the superclass.
6.4 Creating a Multilevel Hierarchy Given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A. The following program demonstrates a multilevel hierarchy. PROGRAM // Extend BoxWeight to include shipping costs. // Start with Box.

94% **MATCHING BLOCK 83/221** W

class Box { private double width; private double height; private double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } // Add weight. class BoxWeight extends Box { double weight; // weight of box // construct clone of an object BoxWeight(BoxWeight ob) { // pass object to constructor super(ob); weight = ob.weight; } // constructor when all parameters are specified BoxWeight(double w, double h, double d, double m) { super(w, h, d); // call superclass constructor weight = m; } // default constructor BoxWeight() { super(); weight = -1; } // constructor used when cube is created BoxWeight(double len, double m) { super(len); weight = m; } } //

Add shipping costs class Shipment extends BoxWeight { double cost; //

58% **MATCHING BLOCK 84/221** W

construct clone of an object Shipment(Shipment ob) { // pass object to constructor super(ob); cost = ob.cost; } // constructor when all parameters are specified Shipment(double w, double h, double d, double m, double c) { super(w, h, d, m); // call superclass constructor cost = c; } // default constructor Shipment() { super(); cost = -1; } // constructor used when cube is created Shipment(double len, double m, double c) { super(len, m);

cost = c; } } class DemoShipment { public static void main(String args[]) { Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41); Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28); double vol;

50% **MATCHING BLOCK 85/221** W

vol = shipment1.volume(); System.out.println("Volume of shipment1 is " + vol); System.out.println("Weight of shipment1 is " + shipment1.weight); System.out.println("Shipping cost: $" + shipment1.cost); System.out.println(); vol = shipment2.volume(); System.out.println("Volume of shipment2 is " + vol); System.out.println("Weight of shipment2 is " + shipment2.weight); System.out.println("Shipping cost: $" + shipment2.cost); } } OUTPUT Volume of shipment1 is 3000.0 Weight of shipment1 is 10.0 Shipping cost: $3.41 Volume of shipment2 is 24.0 Weight of shipment2 is 0.76

Shipping cost: $1.28 Because of inheritance, Shipment can make use of the previously defined classes of Box and BoxWeight, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code. This example illustrates one other important point: super( ) always refers to the constructor in the closest superclass. The super( ) in Shipment calls the constructor in BoxWeight. The super( ) in BoxWeight calls the constructor in Box. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters —up the line.‖ This is true whether or not a subclass needs parameters of its own. 6.5

79% **MATCHING BLOCK 86/221** W

Method Overriding In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the

subclass. The version of the method defined by the superclass will be hidden. Consider the following: PROGRAM //

100% **MATCHING BLOCK 87/221** W

Method overriding. class A { int i, j; A(int a, int b) { i = a; j = b; } // display i and j void show() { System.out.println("i and j: " + i + " " + j); } } class B extends A { int k; B(int a, int b, int c) { super(a, b); k = c; } // display k – this overrides show() in A void show() { System.out.println("k: " + k); } } class Override { public static void main(String args[]) { B subOb = new B(1, 2, 3); subOb.show(); // this calls show() in B } } OUTPUT k: 3

When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A. To access the superclass version of an overridden function, super keyword can be used. Replace the B class with the following code:

70% **MATCHING BLOCK 88/221** W

class B extends A { int k; B(int a, int b, int c) { super(a, b); k = c; } void show() { super.show(); // this calls A's show() System.out.println("k: " + k); } }

OUTPUT i and j: 1 2 k: 3 Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. 6.6 Dynamic Method Dispatch Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.
Dynamic method dispatch is the

91% **MATCHING BLOCK 89/221** W

mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Only through Dynamic method dispatch Java implements run-time polymorphism. When an overridden method is called through a superclass
reference, Java determines which version of that method to execute based upon the type of
the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. Here is an example that illustrates dynamic method dispatch: PROGRAM //

100% **MATCHING BLOCK 90/221** W

Dynamic Method Dispatch class A { void callme() { System.out.println("Inside A's callme method"); } } class B extends A { // override callme() void callme() { System.out.println("Inside B's callme method"); } } class C extends A { // override callme() void callme() { System.out.println("Inside C's callme method"); } } class Dispatch { public static void main(String args[]) { A a = new A(); // object of type A B b = new B(); // object of type B C c = new C(); // object of type C A r; // obtain a reference of type A r = a; // r refers to an A object r.callme(); // calls A's version of callme r = b; // r refers to a B object r.callme(); // calls B's version of callme r = c; // r refers to a C object r.callme(); // calls C's version of callme } } OUTPUT Inside A's callme method Inside B's callme method Inside C's callme method

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are
declared. Also, a reference of type A, called r, is declared. The program then assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call. Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. 6.7 Using Abstract Classes That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. Java's solution to this problem is the abstract method. To declare an abstract method, use this general form: abstract type name(parameter-list); Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, abstract constructors, or abstract static methods can't be declared. Any subclass of an abstract class must either implement all of the abstract methods in the superclass or be itself declared abstract. Here is the simple example: PROGRAM //

97% MATCHING BLOCK 91/221 W

Using abstract methods and classes. abstract class Figure { double dim1; double dim2; Figure(double a, double b) { dim1 = a; dim2 = b; } // area is now an abstract method abstract double area(); } class Rectangle extends Figure { Rectangle(double a, double b) { super(a, b); } // override area for rectangle double area() { System.out.println("Inside Area for Rectangle."); return dim1 * dim2; } } class Triangle extends Figure { Triangle(double a, double b) { super(a, b); } // override area for right triangle double area() { System.out.println("Inside Area for Triangle."); return dim1 * dim2 / 2; } } class AbstractAreas { public static void main(String args[]) { // Figure f = new Figure(10, 10); // illegal now Rectangle r = new Rectangle(9, 5); Triangle t = new Triangle(10, 8); Figure figref; // this is OK, no object is created figref = r; System.out.println("Area is " + figref.area()); figref = t; System.out.println("Area is " + figref.area()); } }

OUTPUT Inside Area for Rectangle. Area is 45.0 Inside Area for Triangle. Area is 40.0 6.8 Using final with Inheritance The keyword final can be used to prevent overriding and also to stop inheritance. Both are discussed here. Using final

92% MATCHING BLOCK 92/221 W

to prevent overriding To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

PROGRAM SEGMENT

100% MATCHING BLOCK 93/221 W

class A { final void meth() { System.out.println("This is a final method."); } } class B extends A { void meth() { // ERROR! Can't override. System.out.println("Illegal!"); } }

OUTPUT Can’t override Because meth( ) is declared as final, it cannot be overridden in B. If
an
attempt is made to do so, a compile-time error will result.

95% MATCHING BLOCK 94/221 W

Using final to Prevent Inheritance To prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declared all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. Here is an example of a final class: final class A { // ... } // The following class is illegal. class B extends A { // ERROR! Can't subclass A // ... } 6.9 The

Object Class There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Object defines the following methods, which means that they are available in every object. Method Purpose Object clone( ) Creates a new object that is the same as the object being cloned. boolean quals(Objectobject) Determines whether one object is equal to another. void finalize( ) Called before an unused object is recycled. Class getClass( ) Obtains the class of an object at run time. int hashCode( ) Returns the hash code associated with the invoking object. void notify( ) Resumes execution of a thread waiting on the invoking object. void notifyAll( ) Resumes execution of all threads waiting on the invoking object. String toString( ) Returns a string that describes the object. void wait() Waits on another thread of execution. void wait (long milliseconds) void wait (long milliseconds, int nanoseconds)
The methods getClass (), notify(), notifyAll( ), and wait( ) are declared as final. The other methods may be overridden. 6.10 Unit Summary This Unit discussed the basics of Inheritance, its type and use of keyword super. It also covered the method overriding, dynamic method dispatch, uses of abstract class and final keyword in inheritance. 6.11
Key Terms
Dynamic method dispatch is the

91% MATCHING BLOCK 95/221 W

mechanism by which a call to an overridden method is resolved at run time, rather than compile time. 6.12

Check Your Progress 1.
Give the purpose of the keyword super'. 2. What is an abstract class? Give an example. 3. How does Java implement polymorphism? Explain. 4. Differentiate between overriding and overloading a method. 5. Write note on Dynamic dispatch method or run-time polymorphism. 6. Give the uses of final' keyword. 7. List the various methods used by
the Object class.
Unit 7: Packages & Interfaces 7.0 Introduction 7.1 Unit Objectives 7.2 Packages 7.4 Importing Packages 7.5 Interfaces 7.6 Unit Summary 7.7 Key Terms 7.8 Check Your Progress 7.0
Introduction

100%	MATCHING BLOCK 96/221	W

Packages and interfaces are two of the basic components of a Java program.

In general, a Java source file can contain

90%	MATCHING BLOCK 97/221	W

any (or all) of the following four internal parts: • A single package statement (optional) • Any number of import statements (optional) • A single public class declaration (required) • Any number of classes private to the package (optional) 7.1

Objectives The Unit aims
to: • give the overview of the innovative features of Java – the Packages and Interfaces. • introduce the ways by which the Packages can be defined and implemented • know the means of importing Packages and • Define, implement and apply Interfaces 7.2
Packages
Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc. A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and
name space
management. Some of the existing packages in Java are: • java.lang - bundles the fundamental classes • java.io - classes for input, output functions are bundled in this
package
Programmers can define their own packages to bundle groups of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.
Defining a Package To create a package is quite easy: simply include a package command as the first statement in a Java source file.

100%	MATCHING BLOCK 98/221	W

Any classes declared within that file will belong to the specified package.

The package statement defines a name space in which classes are stored. If you omit
the package statement, the class names are put into the default package, which has no name.

76%	MATCHING BLOCK 99/221	W

This is the general form of the package statement: package pkg; Here, pkg is the name of

the package. For example, the following statement creates a package called MyPackage. package MyPackage; A hierarchy of packages can be created. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here: package pkg1[.pkg2[.pkg3]]; A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package java.awt.image; needs to be stored in java/awt/image, java\awt\image, or java:awt:image on UNIX, Windows, or Macintosh file system, respectively. Be sure to choose the package names carefully. A package cannot
be

renamed without renaming the directory in which the classes are stored. Finding Packages and CLASSPATH Finding the path of the package can be done in

92% **MATCHING BLOCK 100/221** W

two ways. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if

the

92% **MATCHING BLOCK 101/221** W

package is in the current directory, or a subdirectory of the current directory, it will be found. Second, a directory path

can be specified or the CLASSPATH environment variable can be set.

100% **MATCHING BLOCK 102/221** W

For example, consider the following package specification. package MyPack; In order for a program to find MyPack, one of two things must be true. Either the program is executed from a directory immediately above MyPack, or CLASSPATH must be set to include the path to MyPack.

A Short Package Example Here is a simple example that explains the package concept: PROGRAM // A simple package package MyPack; class Balance { String name; double bal; Balance(String n, double b) { name = n; bal =

50% **MATCHING BLOCK 103/221** W

b; } void show() { if(bal>0) System.out.print("--< "); System.out.println(name + ": $" + bal); } } class AccountBalance { public static void main(String args[]) {

Balance current[] = new Balance[3]; current[0] = new Balance("K. J. Fielding", 123.23); current[1] = new Balance("Will Tell", 157.02); current[2] = new Balance("Tom Jackson", -12.33); for(int i=0; i>3; i++) current[i].show(); } } OUTPUT Tom Jackson : $-12.33 Call this file AccountBalance.java and put it in a directory called MyPack. Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory. Then try executing the AccountBalance class, using the following command line: java MyPack.AccountBalance
7.3 Access Protection Classes and packages are

72% **MATCHING BLOCK 104/221** W

both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.

The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages,

65% **MATCHING BLOCK 105/221** W

Java addresses four categories of visibility for class members: • Subclasses in the same package • Non-subclasses in the same package • Subclasses in different packages • Classes that are neither in the same package nor subclasses The

three

100% **MATCHING BLOCK 106/221** W

access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.

Table 7.1 sums up the interactions. Visibility

80% MATCHING BLOCK 107/221 W

Private No modifier Protected Public Same class Yes Yes Yes Yes Same package Subclass No Yes Yes Yes Same package non- subclass No Yes Yes Yes Different package subclass No No Yes Yes Different package non- subclass No No No Yes

Table 7.1 Class member access

An Access Example The following example shows all combinations of the access control modifiers. This example has two packages and five classes. The source for the first package defines three classes: Protection, Derived, and SamePackage. The first class defines four int variables in each of the legal protection modes. The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public.

The second class, Derived, is a subclass of Protection in the same package, p1. This grants Derived access to every variable in Protection except for n_pri, the private one. The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri. Package 1 File name: Protection.java package p1; public class Protection { int n = 1; private int n_pri = 2; protected int n_pro = 3; public int n_pub = 4; public Protection() { System.out.println("base constructor"); System.out.println("n = " + n); System.out.println("n_pri = " + n_pri); System.out.println("n_pro = " + n_pro); System.out.println("n_ pub = " + n_pub); } } File name: Derived.java package p1; class Derived extends Protection { Derived() { System.out.println("derived constructor"); System.out.println("n = " + n); // class only // System.out.println("n_pri = " + n_pri); System.out.println("n_pro = " + n_pro); System.out.println("n_ pub = " + n_pub); } } File name: SamePackage.java package p1; class SamePackage { SamePackage() { Protection p = new Protection(); System.out.println("same package constructor"); System.out.println("n = " + p.n); // class only // System.out.println("n_pri = " + p.n_pri); System.out.println("n_pro = " + p.n_pro); System.out.println(" n_pub = " + p.n_pub); } } Following is the source code for the other package, p2. The two classes defined in p2 cover the other two conditions which are affected by access control. The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection's variables except for n_pri (because it is private) and n, the variable declared with the default protection. Thus, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class OtherPackage has access to only one variable, n_pub, which was declared public. Package 2 File name: Protection2.java package p2; class Protection2 extends p1.Protection { Protection2() { System.out.println("derived other package constructor"); // class or package only // System.out.println("n = " + n); // class only // System.out.println("n_pri = " + n_pri); System.out.println("n_pro = " + n_pro); System.out.println("n_ pub = " + n_pub); } } File name: OtherPackage.java package p2; class OtherPackage { OtherPackage() { p1.Protection p = new p1.Protection(); System.out.println("other package constructor"); // class or package only // System.out.println("n = " + p.n); // class only // System.out.println("n_pri = " + p.n_pri); // class, subclass or package only // System.out.println("n_pro = " + p.n_pro); System.out.println("n_pub = " + p.n_pub); } } To test these two packages, use the following code: To test package 1 // Demo package p1. package p1; // Instantiate the various classes in p1. public class Demo { public static void main(String args[]) { Protection ob1 = new Protection(); Derived ob2 = new Derived(); SamePackage ob3 = new SamePackage(); } } To test package 2 // Demo package p2. package p2; // Instantiate the various classes in p2. public class Demo { public static void main(String args[]) { Protection2 ob1 = new Protection2(); OtherPackage ob2 = new OtherPackage(); } } 7.4 Importing Packages

100% MATCHING BLOCK 108/221 W

Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement: import pkg1[.pkg2].(classname|*); Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use: import java.util.Date; import java.io.*;

Here is a sample program : PROGRAM package MyPack; /* Now, the Balance class, its constructor, and its show() methods are public. This means that they can be used by non- subclass code outside their package. */ public class Balance { String name; double bal; public Balance(String n, double b) { name = n; bal = b; } public void show() { if(bal<0) System.out.print("--< "); System.out.println(name + ": $" + bal); } } MAIN PROGRAM import MyPack.*; class TestBalance { public static void main(String args[]) { /* Because Balance is public, you may use Balance class and call its constructor. */ Balance test = new Balance("J. J. Jaspers", 99.88); test.show(); // you may also call show() } } OUTPUT J. J. Jaspers : $99.88 The Balance class is now public. Also, its constructor and its show( ) method are public, too. This means that they can be accessed by any type of code outside the MyPack package. For example, here TestBalance imports MyPack and is then able to make use of the Balance class. 7.5 Interfaces An interface
is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
An interface is not a class.
Writing an interface is similar to
writing a class, but
they are two different concepts.
A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements. Unless the
the class that implements the interface is abstract, all the methods of the interface need to be defined in the class. • An interface is similar to a class in the following ways: • An interface can contain any number of methods. • An interface is written in a file with a .java extension, with the name of the interface matching the name of the file. The bytecode of an interface appears in a .class file. Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.
However, an interface is different from a class in several ways, including: ● An interface
cannot be instantiated. ●
An interface does not contain any constructors. ● All of the methods in an interface are abstract. ● An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final. ● An interface is not extended by a class; it is implemented by a class. ● An interface can extend multiple interfaces.
Defining an Interface An interface is defined much like a class. This is the general form of an interface: access interface name { return-type method-name1(parameter-list); return-type method-name2(parameter-list); type final-varname1 = value; type final-varname2 = value; // ... return-type method-nameN(parameter-list); type final-varnameN = value; } Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
When it is declared as public, the interface can be used by any other code. name is the name of the interface and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods. Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public. Here is an example of an interface definition. It declares a simple interface which contains one method called callback( ) that takes a single integer parameter. interface Callback { void callback(int param); }

100% MATCHING BLOCK 110/221 W

Implementing Interfaces Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this: access class classname [extends superclass] [implements interface [,interface...]] { // class-body } Here, access is either public or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

It is both permissible and common for classes that implement interfaces to define additional members of their own. Here is a small example class that implements the Callback interface shown earlier. PROGRAM SEGMENT class Client implements Callback { // Implement Callback's interface public void callback(int p) { System.out.println("callback called with " + p); } } Accessing Implementations through Interface References
Any instance of any class that implements the declared interface can be referred to by
such a
variable. When you call a method through one of these references, the correct version will
be called
based on the actual instance of the interface being referred to. This is one of the
key features of interfaces. The following example illustrates this. Implementation 1 PROGRAM SEGMENT class Client implements Callback { // Implement Callback's interface public void callback(int p) { System.out.println("callback called with " + p); } } Implementation 2 class AnotherClient implements Callback { // Implement Callback's interface public void callback(int p) { System.out.println("Another version of callback"); System.out.println("p squared is " + (p*p)); } } Interface References class TestIface2 { public static void main(String args[]) { Callback c = new Client(); AnotherClient ob = new AnotherClient(); c.callback(4); c = ob; // c now refers to AnotherClient object c.callback(4);
} }
Applying Interfaces
In this example, the
Bank
interface has only one method. Its implementation is provided
by SBI and PNB classes. In
the real scenario, the
interface is defined by someone, but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.
PROGRAM //
File: TestInterface2.java interface Bank{ float rateOfInterest(); } class SBI implements Bank{ public float rateOfInterest(){return 9.15f;} } class PNB implements Bank{ public float rateOfInterest(){return 9.7f;} } class TestInterface2{
public static void main(String[] args){ Bank b=new SBI(); System.out.println("ROI: "+b.
rateOfInterest()); } } OUTPUT ROI: 9.15 7.6
Unit Summary This Unit discussed the power of packages in access control and elaborated the usage of interfaces for effective programming.
7.7
Key Terms •
Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc. •
An interface
is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
An interface is not a class.
Writing an interface is similar to writing a class, but
they are two different concepts. 7.8
Check Your Progress 1.
Explain the concept of Package, through an example. 2. Discuss the various levels of access protection available for packages and their implementation. 3. Explain the purpose of the import statement. 4. What is an Interface? Explain it with an example. 5. Show that multiple implementations of an Interface is possible.

Unit 8: Exception Handling 8.0 Introduction 8.1 Unit Objectives 8.2 Exception-Handling Fundamentals 8.3 Exception Types 8.4 Using try and catch 8.5 throw clause 8.6 Java's Built-in Exceptions 8.7 Creating User defined Exception classes 8.8 Unit Summary 8.9 Key Terms 8.10 Check Your Progress 8.0
Introduction An exception is an abnormal condition that arises in a code sequence at run time. In computer languages that do not support exception handling, errors must be checked and handled manually, typically through the use of error codes, and so on. This approach is cumbersome. Java's exception handling avoids these problems and brings run-time error management into the object-oriented world. 8.1
Objectives This Unit elaborates the error handling mechanism. It explains the Exception handling supported by Java through try, catch and throw Procedure for creating the User defined exception classes. 8.2
Exception-Handling Fundamentals Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System- generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed before a method return is put in a finally block.
This is the general form of an exception-handling
block:
Here, ExceptionType is the type of exception that has occurred. 8.3 Exception Types All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. There is an important subclass of Exception, called RuntimeException, used to create custom exception types. division by zero and invalid array indexing are examples. he other branch is topped by Error, which defines exceptions
that are not expected to be caught under normal circumstances by
your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This section deals only with the branch Exceptions.
In Java, if an exception arises, and it is not caught by the user program, Java run-time system invokes

100% **MATCHING BLOCK 111/221** W

the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. 8.4

Using try and catch A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement. A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements). The statements that are protected by try must be surrounded by curly braces. That is, they must be within a block.) try block cannot have a single statement. The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the for loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into a. If either division operation causes a divide-by- zero error, it is caught, the value of a is set to zero, and the program continues. PROGRAM // Handle an exception and move on.
import java.util.Random;
class HandleError { public static void main(String args[]) {
int
a=0, b=0,
c=0;
Random r = new Random(); for(int i=0; i>32000; i++) { try { b = r.nextInt(); c = r.nextInt();
a = 12345 / (b/c); } catch (ArithmeticException e) { System.out.println("Division by zero."); a = 0; //
set a to zero and continue } System.out.println("a: " + a); } } }
To display the system error message, the catch block in the preceding program can be rewritten like this:

77% MATCHING BLOCK 112/221 W

catch (ArithmeticException e) { System.out.println("Exceptio: " + e); a = 0; // set a to zero and continue } When this version is substituted in the program, the following message will be displayed. Exception: java.lang.ArithmeticException: / by zero Multiple catch Clauses In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, two or more catch clauses

can be specified,

100% MATCHING BLOCK 113/221 W

each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

PROGRAM //

94% MATCHING BLOCK 114/221 W

Demonstrate multiple catch statements. class MultiCatch { public static void main(String args[]) { try { int a = args.length; System.out.println("a = " + a); int b = 42 / a; int c[] = { 1 }; c[42] = 99; } catch(ArithmeticException e) { System.out.println("Divide by 0: " + e); } catch(ArrayIndexOutOfBoundsException e) { System.out.println("Array index oob: " + e); } System.out.println("After try/catch blocks."); } } This program will cause a division-by-zero exception if it is started with no command line parameters, since a will be equal

to

95% MATCHING BLOCK 115/221 W

zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42]. Here is the output generated by running it both ways: C:\<java MultiCatch a = 0 Divide by 0: java.lang.ArithmeticException: / by zero After try/catch blocks. C:\<java MultiCatch TestArg a = 1 Array index oob: java.lang.ArrayIndexOutOfBoundsException After try/catch blocks.

When multiple catch statements are used, the exception subclasses must come before any of their superclasses. Nested try Statements The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested try statements
PROGRAM // An
example of nested try

89% MATCHING BLOCK 116/221 W

statements. class NestTry { public static void main(String args[]) { try { int a = args.length; /*

If no command-line args are present, the following statement will generate a divide-by-zero exception. */ int b = 42 / a; System.out.println("a = " + a); try { // nested try block /* If one command-line arg is used, then a divide-by-zero exception will be generated by the following code. */ if(a==1) a = a/(a-a); // division by zero /* If two command-line args are used, then generate an out-of-bounds exception. */ if(a==2) { int c[] = { 1 }; c[42] = 99; // generate an out-of-bounds exception } }
catch(ArrayIndexOutOfBoundsException e) { System.out.println("Array index out-of-bounds: " + e); } }
catch(ArithmeticException e) { System.out.println("

Divide by 0: " + e); } } } OUTPUT C:\<java NestTry Divide by 0: java.lang.ArithmeticException: / by zero

52% **MATCHING BLOCK 117/221** W

C:\<java NestTry One a = 1 Divide by 0: java.lang.ArithmeticException: / by zero C:\<java NestTry One Two a = 2 Array index

out-of-bounds: java.lang.ArrayIndexOutOfBoundsException 8.5 throw clause The throw statement is used to throw an exception explicitly. The general form of throw is:

97% **MATCHING BLOCK 118/221** W

throw ThrowableInstance; Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. Two ways

are used to

98% **MATCHING BLOCK 119/221** W

obtain a Throwable object: using a parameter into a catch clause, or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler. PROGRAM //

100% **MATCHING BLOCK 120/221** W

Demonstrate throw. class ThrowDemo { static void demoproc() { try { throw new NullPointerException("demo"); } catch(NullPointerException e) { System.out.println("Caught inside demoproc."); throw e; // rethrow the exception } } public static void main(String args[]) { try { demoproc(); } catch(NullPointerException e) { System.out.println("Recaught: " + e); } } }

OUTPUT Caught inside demoproc. Recaught: java.lang.NullPointerException: demo

99% **MATCHING BLOCK 121/221** W

throws clause If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause: type method-name(parameter-list) throws exception-list { // body of method } Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Here is an example program. PROGRAM // This is now correct.

88% **MATCHING BLOCK 122/221** W

class ThrowsDemo { static void throwOne() throws IllegalAccessException { System.out.println("Inside throwOne."); throw new IllegalAccessException("demo"); } public static void main(String args[]) { try { throwOne(); } catch (IllegalAccessException e) { System.out.println("Caught " + e); } } } OUTPUT inside throwOne caught java.lang.IllegalAccessException:

demo finally clause The finally clause

97% **MATCHING BLOCK 123/221** W

creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

The execution of finally clause is shown in figure 8.1
Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses: PROGRAM //

99% **MATCHING BLOCK 124/221** W

Demonstrate finally. class FinallyDemo { // Through an exception out of the method. static void procA() { try { System.out.println("inside procA"); throw new RuntimeException("demo"); } finally { System.out.println("procA' finally"); } } // Return from within a try block. static void procB() { try { System.out.println("inside procB"); return; } finally { System.out.println("procB's finally"); } } // Execute a try block normally. static void procC() { try { System.out.println("inside procC"); } finally { System.out.println("procC's finally"); } } public static void main(String args[]) { try { procA(); } catch (Exception e) { System.out.println("Exception caught"); } procB(); procC(); } }

OUTPUT

100% **MATCHING BLOCK 125/221** W

inside procA procA's finally Exception caught inside procB procB's finally inside procC procC's finally

93% **MATCHING BLOCK 126/221** W

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB( )'s try statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed. 8.6

Java's
Built-in Exceptions Java defines several exception classes, inside the standard package java.lang.

98% **MATCHING BLOCK 127/221** W

The most general exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they need not be included in any method 's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed in Table 8-1

Exception Meaning
ArithmeticException Arithmetic error, such as divide-by-zero. ArrayIndexOutOfBoundsException Array index is out-of-bounds ArrayStoreException Assignment to an array element of an incompatible type. ClassCastException Invalid cast. IllegalArgumentException Illegal argument used to invoke a method. IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread. IllegalStateException Environment or application is in
an

incorrect state. IllegalThreadStateException Requested operation not compatible with current thread state. IndexOutOfBoundsException Some type of index is out-of- bounds. NegativeArraySizeException Array created with a negative size. NullPointerException Invalid use of a null reference. NumberFormatException Invalid conversion of a string to a numeric format. SecurityException Attempt to violate security. StringIndexOutOfBounds Attempt to index outside the bounds of a string. UnsupportedOperationException An unsupported operation was encountered.
Table 8-1: Java's
Unchecked RuntimeException Subclasses
Table 8-2. Java's
Checked Exceptions Defined in java.lang Exception Meaning
ClassNotFoundException Class not found.
CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException Access to a class is denied. InstantiationException Attempt to create an object of an abstract class or interface. InterruptedException One thread has been interrupted by another thread. NoSuchFieldException A requested field does not exist. NoSuchMethodException A requested method does not exist.

100% **MATCHING BLOCK 128/221** W

Table 8-2 lists those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. Java defines several other types of exceptions that relate to its various class libraries. 8.7

Creating User defined Exception classes
User defined exceptions in java are also known as Custom exceptions. Most of the times, while developing an application in java,
there is
a need to create and throw our own exceptions. These exceptions are known as User defined or Custom exceptions.
In order to define the custom exception, define a subclass of Exception Class. The
Exception class does not define any methods of its own. It inherits those methods provided by Throwable.
Thus, all exceptions, including those that we create, have the methods defined by Throwable available to them. They are shown in Table 8-3. Method Description Throwable fillInStackTrace( ) Returns a Throwable object that contains a Completed stack trace. This object can be rethrown. Throwable getCause( ) Returns the exception that underlies the current exception. If there is no underlying exception, null is returned. String getLocalizedMessage( ) Returns a localized description of the exception. String getMessage( ) Returns a description of the exception. StackTraceElement[ ] getStackTrace( ) Returns an array that contains the stack trace, one element at a time as an array of StackTraceElement. The method at the top of the stack is the last method called before the
exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to Information about each element in the trace, such as its method name. Throwable initCause(Throwable causeExc) Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception. void printStackTrace( ) Displays the stack trace. void printStackTrace (PrintStream stream) Sends the stack trace to the specified stream. Void setStackTrace (StackTraceElement elements[ ]) Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use. String toString( ) Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object. Table 8.3 The Methods Defined by
Throwable The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing the description of the exception to be displayed using println( ). PROGRAM // This program creates a custom exception type. class MyException extends Exception { private int detail; MyException(int a) { detail = a; } public String toString() {
return "MyException[" + detail + "]"; } } class ExceptionDemo { static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")"); if(a < 10) throw new MyException(a);

59% **MATCHING BLOCK 129/221** W

System.out.println("Normal exit"); } public static void main(String args[]) { try { compute(1); compute(20); } catch (MyException e) { System.out.println("Caught " +

e); } } } OUTPUT Called compute(1) Normal exit Called compute(20) Caught MyException[20] 8.8 Unit Summary This Unit discussed the error handling mechanisms provided by Java. It also thrashed out the types of exceptions, the ways by which they can be used in the programs and the definition of custom exceptions by creating the subclasses of
the Exception class. 8.9 Key Terms •
All exception types are subclasses of the built-in class Throwable. •
A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.
8.10
Check Your Progress 1.
How exception handling can be used for debugging a program? 2. Distinguish between checked Exception and unchecked Exception. 3. Explain try-catch block with an example. 4. Differentiate between _throws' and _throw' keywords. 5. Discuss the procedure to create the user defined exception.
Unit 9: Multi Thread Programming 9.0 Introduction 9.1 Unit Objective 9.2 Multithreaded Programming in JAVA 9.3 The Java Thread Model 9.4 The Thread Class and the Runnable Interface 9.4.1 Creating a Thread 9.5 Creating Multiple Threads 9.6 Thread Priorities 9.7 Synchronization 9.8 Inter Thread Communication 9.9 Unit Summary 9.10 Key Terms 9.11 Check Your Progress 9.0

54% **MATCHING BLOCK 130/221** W

Introduction Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of

multitasking. 9.1
Unit Objectives This Unit
concentrates on Multithreaded programming. It enlightens the following topics: ● The Java thread model ● Creating single and multiple threads ● Assigning priorities to threads ● Synchronizing the threads and ● Inter-thread communication 9.2 Multithreaded Programming in JAVA The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
There are two distinct types of multitasking: process-based and thread-based. The difference between the process-based and thread-based is listed in Table 9.1 Table 9-1 Process based Vs Thread based Multitasking programming S.No. Process based Multitasking programming Thread based Multitasking programming 1 In process-based multitasking two or more processes and programs can be run concurrently. In thread-based multitasking two or more threads can be run concurrently. 2 It allows you to run java compiler and text editor at a same time. It allows a text editor to

91% **MATCHING BLOCK 131/221** W

format text at the same time that it is printing, as long as these two actions are

being performed by two separate threads. 3 In process-based multitasking a process or a program is the smallest unit. In thread-based multitasking a thread is the smallest unit. 4 Program is a bigger unit. Thread is a smaller unit. 5 Process based multitasking requires more overhead. Thread based multitasking requires less overhead. 6 Here, it is unable to gain access over idle
time of CPU.
It allows taking gain access over idle time taken by CPU. 7 Process requires its own address space. Threads share same address space. 8 It is not under control of java. It is totally under control of java. 9 Process based multitasking is heavyweight process Thread based multitasking is light weight process. 10 Inter-Process communication is expensive and Context switching from one process to another is also costly. Inter-Thread communication is inexpensive and context switching from one thread to the next is low cost. 11 It has slower data rate multitasking. It has faster data rate multithreading or tasting. 9.3 The Java Thread Model Threads exist in several states. This is diagrammatically represented in Figure 9.1.
New When a new Thread object using
a

new operator is created, its state is called as New Thread. At this point, thread is not alive and it's a state internal to Java programming. Figure 9.1 Life cycle of a Thread Runnable When start() function on Thread object is called, it's state is changed to Runnable. The control is given to Thread scheduler to finish it's execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler. Running When thread is executing, it's state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it's state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources. Blocked/Waiting A thread can be waiting for other thread to finish using thread join or it can be waiting for some resources to available. For example producer consumer problem or waiter notifier implementation or IO resources, then it's state is changed to Waiting. Once the thread wait state is over, it's state is changed to Runnable and it's moved back to runnable thread pool. Terminated (Dead)

97% **MATCHING BLOCK 132/221** W

At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed. 9.4

90% **MATCHING BLOCK 133/221** W

The Thread Class and the Runnable Interface Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.

100% **MATCHING BLOCK 134/221** W

The Thread class defines several methods that help manage threads.

Some of the important methods are listed here: Table 9.2 Methods of Thread class 9.4.1
Creating
a Thread A thread can be created

83% **MATCHING BLOCK 135/221** W

by instantiating an object of type Thread. Java defines two ways in which this can be accomplished: • implement the Runnable interface or • extend the Thread class, itself. Implementing

Runnable Creating a Thread by Implementing Runnable Interface is done in three steps. They are :
STEP 1: As a first step implement a run() method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method.
Following is simple
syntax of run() method: public void run() STEP 2: At second step instantiate a Thread object using the following constructor: Thread(Runnable threadObj, String threadName); Where,
threadObj is an instance of a class that implements the
Runnable interface and threadName is the name given to the new thread. STEP 3: Once Thread object is created, start it by calling start( ) method, which executes a call to run() method.
Following is simple syntax of start() method:
void start( ); Here is an example that creates a new thread and starts it running: PROGRAM
class RunnableDemo implements Runnable { private Thread t; private String threadName; RunnableDemo(String name){ threadName = name; System.out.println("Creating " + threadName); } public void run() {
Programming in Java 140
System.out.println("Running " + threadName); try { for(int i = 4; i < 0; i--) { System.out.println("Thread: " + threadName + ", " + i); // Let the thread sleep for a while. Thread.sleep(50); } } catch (InterruptedException e) { System.out.println("Thread " + threadName + " interrupted."); } System.out.println("Thread " + threadName + " exiting."); } public void start () { System.out.println("Starting " + threadName); if (t == null) { t =
new Thread (
this, threadName); t.start ();

} } } public class TestThread { public static void main(String args[]) { RunnableDemo R1 = new RunnableDemo("Thread-1"); R1.start(); RunnableDemo R2 = new RunnableDemo("Thread-2"); R2.start(); } } OUTPUT Creating
Thread-1
Starting Thread-1
Extending Thread
Class The second way to create a thread is to create a new class that extends Thread
class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.
PROGRAM
class ThreadDemo extends Thread {
private Thread t; private String threadName; ThreadDemo(String name) { threadName = name; System.out.println("Creating " + threadName); } public void run() { System.out.println("Running " + threadName); try { for(int i = 4; i < 0; i--) { System.out.println("Thread: " + threadName + ", " + i); // Let the thread sleep for a while.
Thread.sleep(50); } }catch (InterruptedException e) { System.out.println("Thread " + threadName + " interrupted."); } System.out.println("Thread " + threadName + " exiting."); } public void start () { System.out.println("Starting " + threadName); if (t == null) { t =
new Thread (
this, threadName); t.start (); } } } public class TestThread { public static void main(String args[]) { ThreadDemo T1 = new ThreadDemo("Thread-1"); T1.start(); ThreadDemo T2 = new ThreadDemo("Thread-2"); T2.start(); } } OUTPUT Creating Thread-1 Starting Thread-1 Creating Thread-2 Starting Thread-2 Running Thread-1 Thread: Thread-1, 4 Running Thread-2 Thread: Thread-2, 4 Thread: Thread-1, 3 Thread: Thread-2, 3 Thread: Thread-1, 2 Thread: Thread-2, 2 Thread: Thread-1, 1 Thread: Thread-2, 1 Thread Thread-1 exiting. Thread Thread-2 exiting. 9.5
Creating Multiple Threads Java allows the program to spawn as many threads as it needs. For example, the following program creates three child threads: PROGRAM // Create multiple
threads. class NewThread implements Runnable { String name; // name of thread Thread
t; NewThread(String threadname) { name = threadname;
t =
new Thread(this, name); System.out.println("New thread: " + t); t.

72% **MATCHING BLOCK 136/221** W

start(); // Start the thread } // This is the entry point for thread. public void run() { try { for(int i = 5; i < 0; i--) { System.out.println(name + ": " + i); Thread.sleep(1000); } } catch (InterruptedException e) { System.out.println(name + "Interrupted"); } System.out.println(name + " exiting."); } } class MultiThreadDemo { public static void main(String args[]) { new NewThread("

One"); // start threads new NewThread("Two");
new NewThread("Three"); try { // wait for other threads to end

100% **MATCHING BLOCK 137/221** W

Thread.sleep(10000); } catch (InterruptedException e) { System.out.println("Main thread Interrupted"); } System.out.println("Main thread exiting."); } }

OUTPUT
New thread: Thread[One,5,main] New thread: Thread[Two,5,main] New thread: Thread[Three,5,main] One: 5 Two: 5 Three: 5 One: 4 Two: 4 Three: 4 One: 3 Three: 3 Two: 3 One: 2 Three: 2 Two: 2 One: 1 Three: 1 Two: 1 One
exiting. Two
exiting. Three exiting. Main thread exiting

9.6 Thread Priorities Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower priority threads. To set a thread's priority, use the setPriority( ) method, which is a member of Thread. This is its general form: final void setPriority(int level) Here, level specifies the new priority setting for the calling thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as final variables within Thread. The current priority setting can be obtained by calling the getPriority( ) method of Thread, shown here: final int getPriority( ) PROGRAM // Demonstrate thread priorities. class clicker implements Runnable { int click = 0; Thread t; private volatile boolean running = true; public clicker(int p) { t = new Thread(this); t.setPriority(p); } public void run() { while (running) { click++; } } public void stop() { running = false; } public void start() { t.start(); } } class HiLoPri {
public static void main(String args[]) { Thread.currentThread().setPriority(Thread.MAX_PRIORI TY); clicker hi = new clicker(Thread.NORM_PRIORITY + 2); clicker lo = new clicker(Thread.NORM_PRIORITY - 2); lo.start(); hi.start(); try {

100% **MATCHING BLOCK 138/221** W

Thread.sleep(10000); } catch (InterruptedException e) { System.out.println("Main thread interrupted."); }

lo.stop(); hi.stop(); // Wait for child

62% **MATCHING BLOCK 139/221** W

threads to terminate. try { hi.t.join(); lo.t.join(); } catch (InterruptedException e) { System.out.println("

InterruptedException caught"); } System.out.println("
Low-priority thread: " + lo.click); System.out.println("High-priority thread: " + hi.click); } } OUTPUT Low-priority thread: 4408112 High-priority thread: 589626904 The above example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a nonpreemptive platform. One thread is set two levels above the normal priority, as defined by Thread.NORM_PRIORITY, and the other is set to two levels below it. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed. This program is tested under Windows 98.
9.7 Synchronization The process by which a thread ensures

92% **MATCHING BLOCK 140/221** W

that the resource is used only by one thread at a time is called synchronization. Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

The code can be synchronized in two ways: Using synchronized method Using synchronized statement Using synchronized method To synchronize an object, call the method that is modified with the synchronized keyword. To understand the need for synchronization, let's begin with a simple example that does not use it. The following program has three simple classes. The first one, Callme, has a single method named call( ). The call( ) method takes a String parameter called msg. This method tries to print the msg string inside of square brackets. The interesting thing to notice is that after call( ) prints the opening bracket and the msg string, it calls Thread.sleep(1000), which pauses the current thread for one second. The constructor of the next class, Caller, takes a reference to an instance of the Callme class and a String, which are stored in target and msg, respectively. The constructor also creates a new thread that will call this object's run( ) method. The thread is started immediately. The run( ) method of Caller calls the call( ) method on the target instance of Callme, passing in the msg string. Finally, the Synch class starts by creating a single instance of Callme, and three instances of Caller, each with a unique message string. The same instance of Callme is passed to each Caller. PROGRAM // This program is not

92% **MATCHING BLOCK 141/221** W

synchronized. class Callme { void call(String msg) { System.out.print("[" + msg); try { Thread.sleep(1000); } catch(InterruptedException e) { System.out.println("Interrupted"); } System.out.println("]"); } } class Caller implements Runnable { String msg; Callme target; Thread t; public Caller(Callme targ, String s) { target = targ; msg = s; t = new Thread(this); t.start(); } public void run() { target.call(msg); } } class Synch { public static void main(String args[]) { Callme target = new Callme(); Caller ob1 = new Caller(target, "Hello"); Caller ob2 = new Caller(target, "Synchronized"); Caller ob3 = new Caller(target, "World"); // wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join(); } catch(InterruptedException e) { System.out.println("Interrupted"); } } }

OUTPUT Programming in Java 150 Hello[Synchronized[World]] In the above program, by calling sleep( ), the call( ) method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a race condition. To fix the problem, the access to call( ) method must be serialized. That is, restrict its access to only one thread at a time. To do this, simply add the keyword synchronized to call( )'s definition as shown here: class Callme { synchronized void call(String msg) { ... This prevents other threads from entering call( ) while another thread is using it. After synchronized has been added to call( ), the output of the program is as follows: [Hello] [Synchronized] [World] The synchronized Statement Consider a situation when there is a need

95% **MATCHING BLOCK 142/221** W

to synchronize access to objects of a class • that was not designed for multithreaded access, • the class does not use synchronized methods

and • also

81% **MATCHING BLOCK 143/221** W

this class was created by a third party, • do not have access to the source code.

In such a situation

94% **MATCHING BLOCK 144/221** W

simply put calls to the methods defined by this class inside a synchronized block. This is the general form of the synchronized statement: synchronized(object) { // statements to be synchronized } Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of 'object' occurs only after the current thread has successfully entered 'object's' monitor. Here is an

example. PROGRAM //

96% **MATCHING BLOCK 145/221** W

This program uses a synchronized block. class Callme { void call(String msg) { System.out.print("[" + msg); try { Thread.sleep(1000); } catch (InterruptedException e) { System.out.println("Interrupted"); } System.out.println("]"); } } class Caller implements Runnable { String msg; Callme target; Thread t; public Caller(Callme targ, String s) { target = targ; msg = s; t = new Thread(this); t.start(); } // synchronize calls to call() public void run() { synchronized(target) { // synchronized block target.call(msg); } } } class Synch1 { public static void main(String args[]) { Callme target = new Callme(); Caller ob1 = new Caller(target, "Hello"); Caller ob2 = new Caller(target, "Synchronized"); Caller ob3 = new Caller(target, "World"); // wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join(); } catch(InterruptedException e) { System.out.println("Interrupted"); } } }

OUTPUT [Hello] [Synchronized] [World] 9.8
Inter Thread
Communication Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods.

These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.
The Table 9-3 explains the important methods for
inter thread
communication. Method Description
wait() Causes the
current thread to wait until another thread invokes the notify().

96% **MATCHING BLOCK 146/221** W

notify() Wakes up a single thread that is waiting on this object's monitor.

notifyAll() Wakes up all the threads that called wait( ) on the same object.
Table 9.3 Methods
for inter thread communication
The following example shows how two thread
can communicate using wait() and notify() method. PROGRAM class Chat { boolean flag = false; public synchronized void Question(String msg) { if (flag) { try { wait();
} catch (InterruptedException e) { e.printStackTrace(); } } System.out.println(msg); flag = true; notify(); } public synchronized void Answer(String msg) { if (!flag) { try { wait(); } catch (InterruptedException e) { e.printStackTrace(); } } System.out.println(msg); flag = false; notify(); } } class T1 implements Runnable { Chat m; String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" }; public T1(Chat m1) { this.m = m1; new Thread(this, "Question").start(); } public void run() { for (int i = 0; i > s1.length; i++) { m.Question(s1[i]); } } }
class T2 implements Runnable { Chat m; String[] s2 = { "Hi", "I am good, what about you?", "Great!" }; public T2(Chat m2) { this.m = m2; new Thread(this, "Answer").start(); } public void run() { for (int i = 0; i > s2.length; i++) { m.Answer(s2[i]); } } } 9.9 Unit Summary
This Unit described one of the key features of Java, the Multithreaded programming. It includes all the aspects of Multithreaded programming such as creating single and multiple threads, assigning priorities to the threads, synchronizing the threads and also the interthread communication. 9.10
Key Terms Synchronization is
the process by which a thread ensures

58% **MATCHING BLOCK 147/221** W

that the resource is used only by one thread at a time is called synchronization. Java provides unique, language-level support for it.

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower priority threads. 9.10
Check Your Progress 1.
Differentiate Process and Thread. 2. Explain Thread model. 3. Exemplify the thread creation using a simple example. 4. What do you mean by Synchronization? How do you achieve it? 5. Illustrate Multithreaded programming using an example
Module III: String handling, Utility classes, java.lang and java.io
Unit 10: String Handling 10.0 Introduction 10.1 Unit Objective 10.2 The String Constructors 10.3 Special String operations 10.4 Character Extraction 10.5 String Methods 10.6 StringBuffer 10.7 Unit Summary 10.8 Key Terms 10.9 Check Your Progress 10.0
Introduction In Java a string is a sequence of characters. Java implements strings as objects of type String. Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. String objects can be constructed a number of ways, making it easy to obtain a string when needed. Once a String object has been created, it cannot change the characters that comprise that string. For those cases in which a modifiable string is desired, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created. 10.1
Unit Objective
This Unit explains: • The String and StringBuffer classes • The methods of String and StringBuffer classes to manipulate the strings 10.2
The String Constructors The String class supports several constructors. To create an empty String, call the default constructor. For example,

String s = new String(); will create an instance of String with no characters in it. To create a String initialized by an array of characters, use the constructor shown here: String(char chars[ ]) String(char chars[ ], int startIndex, int numChars) Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }; String s1 = new String(chars);
String s2 = new String(chars, 2, 3); This constructor initializes s1 with the string “abcdef” and s2 with “cde”. To construct a String object that contains the same character sequence as another String object use the following constructor format: String(String strObj) Here, strObj is a String object. Consider this example: PROGRAM // Construct one String from another.
class MakeString { public static void main(String args[]) { char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c); String s2 = new String(s1); System.out.println(s1); System.out.println(
s2); } }
OUTPUT Java Java The String class also provides constructors that initialize a string when given a byte array. Their forms are shown here: String(byte asciiChars[ ]) String(byte asciiChars[ ], int startIndex, int numChars) Here, asciiChars specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors: PROGRAM // Construct string from subset of char array.
class SubStringCons { public static void main(String args[]) { byte ascii[] = {65, 66, 67, 68, 69, 70 }; String s1 = new String(ascii); System.out.println(s1); String
s2 = new String(ascii, 2, 3); System.out.println(
s2); } } OUTPUT ABCDEF CDE 10.3 Special String operations Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. String Literals An easier way to create a String instance is using a string literal. For each string literal in the program, Java automatically constructs a String object. For example, the following code fragment creates two equivalent strings:
char chars[] = { 'a', 'b', 'c' }; String s1 = new String(chars);
String s2 = "abc"; // use string literal
String Concatenation In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings: int age = "9"; String s = "He is " + age + " years old."; System.out.println(s); This displays the string —He is 9 years old.‖ This is because the int value in age is automatically converted into its string representation within a String object. This string is then concatenated. String Conversion and toString( ) When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method valueOf( ) defined by String. valueOf( ) is overloaded for all the simple types and for type Object. For the simple types, valueOf( ) returns a string that contains the human-readable equivalent of the value with which it is called. For objects, valueOf( ) calls the toString( ) method on the object. For most important classes, override toString( ) and provide your own string representations. The toString( ) method has this general form: String toString( ) To implement toString( ), simply return a String object that contains the human readable string that appropriately describes an object of the class. The following program demonstrates this by overriding toString( ) for the Box class: PROGRAM // Override toString() for Box class.

85% **MATCHING BLOCK 148/221** W

class Box { double width; double height; double depth; Box(double w, double h, double d) { width = w; height = h; depth = d; }

public
String toString() { return "Dimensions are " + width + " by " + depth + " by " + height + "."; } }
class toStringDemo { public static void main(String args[]) { Box b = new Box(10, 12, 14); String s = "Box b: " + b; // concatenate Box object System.out.println(b); // convert Box to string System.out.println(
s); } } OUTPUT Dimensions are 10.0 by 14.0 by 12.0 Box b: Dimensions are 10.0 by 14.0 by 12.0
Function Syntax Description Example Output charAt( ) char charAt(int where) used to extract a single character at an index
class temp { public static void main(String args[]) { String str="Hello"; char ch=str.charAt(1); System.out.println(
ch); } } e getChars( ) void getChars(int sourceStart, int sourceEnd, char used to extract more than one character
class temp {public static void main(String args[]) {String str="Hello World";
ello
target[], int targetStart)
char ch[]=new char[4]; str.getChars(1,5,ch,0); System.out.
println(ch); } }

getBytes() byte[ ] getBytes( ) extract
characters from String object and then convert the characters in a byte array
String str="Hello";
byte []=str.getBytes(); - toCharArray( ) char[ ] toCharArray( ) convert all the characters in a String object into an array of characters. It is the best and easiest way to convert string to character array
class temp { public static void main(String args[]) { String str = "Hello World"; char ch[] = str.toCharArray();
System.out.println(ch); } } Hello World 10.4
Character Extraction The String class
provides a number of ways in which characters can be extracted from a String object. Many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero. Here, the various methods are listed with example. 10.5
String Methods The String class includes several methods. They are categorized into • String comparison • Searching String • Modifying String • Data conversion and changing the case of character within a String.
● String comparison: Method Description Syntax equals( ) To compare two strings for equality boolean equals(Object str) equalsIgnoreCase() To perform a comparison that ignores case differences, i.e When it compares two strings, it considers A-Z to be the same as a-z. boolean equalsIgnoreCase(String str) == The == operator compares two object references to see whether they refer to the same instance. (Object str == Object str) compareTo() A string is less than another if it comes before the other in dictionary order and the function returns -1. A string is greater than another if it comes after the other in dictionary order and the function returns 1. If both are equal, 0 is returned. The String method compareTo ( ) serves this purpose
int compareTo(String str)
EXAMPLE PROGRAM // Demonstrate equals() and equalsIgnoreCase().
class equalsDemo { public static void main(String args[]) { String s1 = "Hello"; String s2 = "Hello"; String s3 = "Good-bye"; String s4 = "HELLO"; String s5 = new String(s1); System.out.println(s1 + "
equals " + s2 + " -< " +
s1.equals(s2)); System.out.println(
s1 + " equals " + s3 + " -< " + s1.equals(s3)); System.out.println(s1 + " equals " + s4 + " -< " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -< " + s1.equalsIgnoreCase(s4)); System.out.println(s1 + " equals " + s2 + " -< " + s1.equals(s5));
System.out.println(s1 + " == " + s2 + " -< " + (s1 == s5)); System.out.println(s1 + " compareto " + s2 + " -< " + s1.compareTo(s2)); System.out.println(s1 + "
compareto " + s3 + " -< " + s1.compareTo(s3)); } } OUTPUT
Hello equals Hello -< true Hello equals Good-bye -< false Hello equals HELLO -< false
Hello equalsIgnoreCase HELLO -< true Hello equals Hello -< true Hello == Hello -< false //Though, the contents of the two String objects are identical, they are distinct objects. Hello compareto Hello -< 0 Hello compareto Good-bye -< 1 regionMatches The regionMatches( ) method compares a specific region inside a string with another specific region in another string Boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars) boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars) startsWith( ) The startsWith( ) method determines whether a given String begins with a specified string. boolean startsWith(String str) endsWith( ) The endsWith( ) determines whether the String in question ends with a specified string. boolean endsWith(String str) EXAMPLE
"Foobar".endsWith("bar") " Foobar".startsWith("Foo")
OUTPUT both are true ● Searching String: Method Description Syntax indexOf( ) Searches for the first occurrence of a character orsubstring. int indexOf(int ch) int indexOf(String str, int startIndex) lastIndexOf() Searches for the last occurrence of a character or substring. int lastIndexOf(int ch) int lastIndexOf(String str, int startIndex PROGRAM // Demonstrate indexOf() and lastIndexOf().
class indexOfDemo { public static void main(String args[]) { String s = "Now is the time for all good men " + "to come to the aid of their country."; System.out.println(s); System.out.println("indexOf(t) = " + s.indexOf('t'));
System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t')); System.out.println("
indexOf(the) = " + s.indexOf("the")); System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10)); System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
System.out.println("
indexOf(the, 10) = " + s.indexOf("the", 10)); System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the", 60)); } } OUTPUT
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7 lastIndexOf(t) = 65 indexOf(the) = 7 lastIndexOf(the) = 55

indexOf(t, 10) = 11 lastIndexOf(t, 60) = 55 indexOf(the, 10) = 44 lastIndexOf(the, 60) = 55 ● Modifying String: Method Description Syntax substring( ) returns a copy of the substring that begins at startIndex and runs to the end of the invoking string or begins at startIndex up to,but not including, the endingindex String substring(int startIndex) String substring(int startIndex, int endIndex) concat( ) creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +. String concat(String str) replace( ) replaces all occurrences of one character in the invoking string with another character. String replace(char original, char replacement) trim( ) returns a copy of the invoking string from which any leading and trailing whitespace has been removed. String trim( ) PROGRAM
public
class Test{ public static void main(String args[]){ String Str=
new String("
Welcome to ADB University"); String s1 = "one"; String s2 = s1.concat("two"); String s3 = — Welcome —;
System.out.print("Return Value :");
System.out.println(Str.substring(10)); System.out.print("Return Value :");
System.out.println(Str.substring(10,18)); System.out.println("Concatenated string : " + s2); System.out.print("
Replaced Value :"); System.out.println(
Str.
replace('o','T')); System.out.print("Trimmed Value :"); System.out.
println(
s3.trim()); } }
OUTPUT Return Value : ADB University Return Value : ADB Concatenated string : onetwo Replaced Value : WelcTme tT ADB University Trimmed Value : Welcome ● Data Conversion and changing the case: Method Description Syntax valueOf() The valueOf( ) method converts data from its internal format into a human- readable form. All of the simple types are converted to their common String representation. Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method. static String valueOf(double num)
static String valueOf(long num) static String valueOf(Object ob) static String valueOf(char chars[ ]) static String valueOf(char chars[ ], int startIndex, int
numChars) toLowerCase() converts all the characters in a string from uppercase to
lowercase
String toLowerCase( ) PROGRAM
public
class Test{ public static void main(String args[]){
double d =102939939.939;
boolean b =true; long l =1232874; char[] arr ={'a','b','c','d','e','f','g'};
String s = "This is a test.";
System.out.println("Return Value : "+String.valueOf(d)); System.out.println("Return Value : "+String.valueOf(b));
System.out.println("Return Value : "+String.valueOf(l)); System.out.println("Return Value : "+String.valueOf(arr));
System.out.println("Original: " + s); String upper = s.toUpperCase(); String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper); System.out.println("Lowercase: " + lower); } } OUTPUT
Return Value : 1.02939939939E8 Return Value : true Return Value : 1232874 Return Value : abcdefg
Original: This is a test. Uppercase: THIS IS A TEST. Lowercase: this is a test. 10.6
StringBuffer StringBuffer is a peer class of String that provides much of the functionality of strings. String represents fixed-length, immutable character sequences.
In contrast, StringBuffer represents growable and writeable
character sequences. StringBuffer may have characters and substrings inserted in the middle or appended
to the end. StringBuffer defines these three constructors: StringBuffer( ) StringBuffer(int size) StringBuffer(String str)
The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.
Method Description Syntax length() Returns the current length of a StringBuffer int length() capacity() Returns the total allocated capacity of the StringBuffer int capacity() ensureCapacity() to preallocate room for a certain number of characters after a StringBuffer has been constructed void ensureCapacity(int capacity) setLength() To set the length of the buffer within a StringBuffer object void setLength(int len) charAt( ) To get the value of a single character from a StringBuffer char charAt(int where) setCharAt( ) To set the value of a character within a StringBuffer void setCharAt(int where, char ch)
append( ) concatenates
the string representation of any other type of data to the end of the invoking StringBuffer object. StringBuffer append(String str) StringBuffer append(int num) StringBuffer append(Object obj)
insert( )

inserts one string into another StringBuffer insert(int index, String str) StringBuffer insert(int index, char ch) StringBuffer insert(int index, Object obj)
reverse( ) returns the reversed object on which it was called StringBuffer reverse( ) delete( ) deletes a sequence of characters from the invoking object and the substring deleted runs from startIndex to StringBuffer delete(int startIndex, int endIndex) endIndex–1. deleteCharAt( ) deletes the character at the index specified by loc and it returns the resulting StringBuffer object.
StringBuffer deleteCharAt(int loc)
PROGRAM
class StringBufferDemo { public static void main(String args[]) { StringBuffer sb = new StringBuffer("Hello"); String str = "ADBU"; System.out.println("
buffer = " + sb);
System.out.println("length = " + sb.length()); System.out.println("capacity = " + sb.capacity()); System.out.println(—
char at — + 2 + —: ‖ + str.charAt(2)); str = str.setCharAt(2, ”A”); System.out.println(—char replaced at — + 2 + —: ‖ + str); sb.append(123); sb.insert(5, "World "); System.out.println(—appended string : ‖ + sb); System.out.println(—inserted string : — + sb); sb.reverse(); System.out.println(—reversed string : — + sb); StringBuffer sb1 = new StringBuffer("This is a test."); Sb1.delete(4, 7); System.out.println("After delete: " + sb1); sb.deleteCharAt(0); System.out.println("After deleteCharAt: " + sb1); } } OUTPUT buffer = Hello length = 5 capacity = 21 char at 2 : a char replaced at 2 :
ADBU
10.7 Unit Summary
This Unit illustrated the String and StringBuffer classes, which include various String methods for String comparison, Searching String, Modifying String, conversion of data and changing the case of character within a String. It also discussed the Character Extraction methods for manipulating the strings. 10.8
Key Terms
An easier way to create a String instance is using a string literal.
StringBuffer is a peer class of String that provides much of the functionality of strings. 10.9
Check Your Progress 1.
Differentiate String and StringBuffer class. 2. Discuss the various ways of Concatenating a string. 3. Briefly explain the methods for Character Extraction. 4. List out the methods used for Modifying the String. 5. Discuss the important methods in StringBuffer class for string manipulation.
Unit 11: The Collection Interfaces and Collection Classes 11.0
Introduction 11.1 Unit Objective 11.2
The Collection Interface 11.3 The List Interface 11.4 The Set Interface 11.5 The Map Interface 11.6 The Enumeration Interface 11.7 The Iterator Interface 11.8 The ArrayList class 11.9 The LinkedList Class 11.10
Vector Class 11.11 Stack Class 11.12 Hashtable Class 11.13 Properties Class 11.14 StringTokenizer Class 11.15 Date Class 11.16 Unit Summary 11.17 Key Terms 11.18 Check Your Progress 11.0
Introduction A collection is a group of objects. The java.util contains the Collection framework.
The collections framework was designed to meet several goals. • The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient. • The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. • Extending and/or adapting a collection
had to be easy. The Java collections framework has a wide assortment of classes and interfaces that support a broad range of functionality. This Lesson provides an overview of important interfaces.
11.1
Objectives This Unit elucidates the java.util package that contains one of Java’s most powerful subsystems: collections. The scope of this lesson is confined to the following interfaces and classes: • List,Set,Map, Enumeration and Iterator interfaces • ArrayList, LinkedList, Vector, Stack, Properties, HashTable, StringTokenizer, and Date classes. 11.2
The
Collection Interface The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have.
These methods are summarized in
Table 11.1. Method
Description

boolean add(Object obj) Adds obj to the invoking collection. Returns true if obj was added to the collection. Returnsfalse if obj is already a member of the collection, or if the collection does not allow duplicates. boolean addAll(Collection c) Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false. void clear( ) Removes all elements from the invoking collection. boolean contains(Object obj) Returns true if obj is an element of the invoking collection. Otherwise, returns false. boolean containsAll(Collection c) Returns true if the invoking collection contains all elements of c. Otherwise, returns false. boolean equals(Object obj) Returns true if the invoking collection and obj are equal. Otherwise, returns false. int hashCode( ) Returns the hash code for the invoking collection. boolean isEmpty( ) Returns true if the invoking collection is empty. Otherwise, returns false. Iterator iterator( ) Returns an iterator for the invoking collection. boolean remove(Object obj) Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false boolean removeAll(Collection c) Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
boolean retainAll(Collection c) Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false. int size( ) Returns the number of elements held in the invoking collection. Object[ ] toArray( ) Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements. Object[ ] toArray(Object array[ ]) Returns an array containing only those collection elements whose type matches that of array.
The
array elements are copies of the collection elements. If the size of array equals the number of matching elements, these are returned in array. If the size of array is less than the number of matching elements, a new array of the necessary size is allocated and returned. If the size of array is greater than the number of matching elements, the array element following the last collection element is set to null. An ArrayStoreException is thrown if any collection element has a type that is not a subtype of array.
Table 11.1 The Methods Defined by Collection 11.3
The List Interface The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements.
The classes which implement the List interface are called Lists. ArrayList, Vector and LinkedList are some examples of lists. Here are some properties of lists. ● Elements of the lists are ordered using Zero based index. ● Access to the elements of lists using an integer index. ● Elements can be inserted at a specific position using integer index. Any pre-existing elements at or beyond that position are shifted right.
● Elements can be removed from a specific position. The elements beyond that position are shifted left. ● A list may contain duplicate elements. ● A list may contain multiple null elements. List interface extends Collection interface. So, All methods of
the Collection interface are inherited from the
List interface. Along with these methods, another 9 methods are included in the List interface to support the properties of lists. Methods defined by List are summarized in Table 11.2. Method
Description void add(int index, Object obj) Inserts obj into the invoking list at the index passed in index. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. boolean addAll(int index, Collection c) Inserts all elements of c into the invoking list at the index passed in index. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise. Object get(int index) Returns the object stored at the specified index within the invoking collection. int indexOf(Object obj) Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, –1 is returned. int lastIndexOf(Object obj) Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, –1 is returned. ListIterator listIterator( ) Returns an iterator to the start of the invoking list. ListIterator listIterator(int index) Returns an iterator to the invoking list that begins at the specified index. Object remove(int index) Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of
subsequent elements are decremented by one. Object set(int index, Object obj) Assigns obj to the location specified by index within the invoking list. List subList(int start, int end) Returns a list that includes elements from start to end–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

The List is the base interface for all list types, and the ArrayList and LinkedList classes are two common List's implementations. • ArrayList: An implementation that stores elements in a backing array. The array's size will be automatically expanded if there isn't enough room when adding new elements into the list. It's possible to set the default size by specifying an initial capacity when creating a new ArrayList. Basically, an ArrayList offers constant time for the following operations: size, isEmpty, get, set, iterator, andlis tIterator; amortized constant time for the add operation; and linear time for other operations. Therefore, this implementation can be considered if we want fast, random access of the elements. • LinkedList: An implementation that stores elements in a doubly-linked list data structure. It offers constant time for adding and removing elements at the end of the list; and linear time for operations at other positions in the list. Besides ArrayList and LinkedList, Vector class is a legacy collection. Vector is thread-safe, but ArrayList and LinkedList are not. 11.4 The Set Interface The Set interface defines a set. It
extends Collection and declares the behavior of a collection that
does not allow duplicate elements. Therefore, the add( ) method returns false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own. The Java Collections Framework provides three major implementations of the Set interface: HashSet, LinkedHashSet and TreeSet. HashSet: is the best-performing implementation and is a widely-used Set implementation. It represents the core characteristics of sets: no duplication and unordered. LinkedHashSet: This implementation orders its elements based on insertion order. So consider using a LinkedHashSet when we want to store unique elements in order. TreeSet: This implementation orders its elements based on their values, either by their natural
ordering, or by a Comparator provided at creation time. Therefore, besides the uniqueness of elements that a Set guarantees, consider using HashSet when ordering does not matter; using LinkedHashSet when you want to order elements by their Insertion order; using TreeSet when we want to order elements by their values. 11.5 The Map Interface A Map is an object that maps keys to values, or is a collection of attribute-value pairs. A Map is not considered to be a true collection, as the Map interface does not extend the Collection interface. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value (HashMap and LinkedHashMap) but some does not (TreeMap). The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not. Maps are perfectly for key-value association mapping such as dictionaries. Maps are used when we want to retrieve and update elements by keys, or perform lookups by keys. Some examples: • A map of error codes and their descriptions. • A map of zip codes and cities. • A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages. • A map of classes and students. Each class (key) is associated with a list of students (value). In the Map interface, there are several implementations but only 3 major, common, and general purpose implementations are widely used. They are HashMap, LinkedHashMap and TreeMap. The characteristics and behaviors of each implementation are listed here. • HashMap: This implementation uses a hash table as the underlying data structure. It implements all of the Map operations and allows null values and one null key. This class is roughly equivalent to Hashtable - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. HashMap does not guarantee the order of its key-value elements. Therefore, consider to use a HashMap when order does not matter and nulls are acceptable. • LinkedHashMap: This implementation uses a hash table and a linked list as the underlying data structures, thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. This implementation also allows nulls like HashMap. So consider using a LinkedHashMap when you want a Map with its key-value pairs are sorted by their insertion order.
• TreeMap: This implementation uses a red-black tree as the underlying data structure. A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation does not allow nulls. So consider using a TreeMap when you want a Map sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom order you specify. 11.6 The Enumeration Interface Enumeration is a simple interface that declares two methods, namely the hasMoreElements() method and the nextElement() method. These two methods enable us to enumerate all of the elements in any collection of objects. The hasMoreElements() method will return the value true if there are valid elements in a collection of objects. When
the
hasMoreElements() method returns true, we can invoke the nextElement() method for retrieving the next element in the collection of objects. Here is an example. PROGRAM // Program to illustrate Enumeration interface Import java.util.Enumeration; Class Sample implements Enumeration { private int counter = 0; private Boolean flag = true; public boolean hasMoreElements() { return flag; } public object nextElement() { counter = counter +1; if (counter <= 4) flag = false; return new Integer(counter); } } class EnumerationDemo { public static void main(String args[]) { Sample sam = new Sample(); While(sam.hasMoreElements()) { System.out.println(sam.nextElement()); } }
} OUTPUT 1 2 3 4 In the above program, the class Sample implements the Enumeration interface. As a result, it provides the code for both the hasMoreElements() and nextElement() methods. Whenever the nextElement() method is invoked, a new object is created and returned. At one stage, the hasMoreElement() method returns false and as a result, the program's execution is terminated. 11.7

The Iterator Interface Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
The Iterator interface declares the methods shown in Table 11.3. Method
Description boolean hasNext( ) Returns true if there are more elements. Otherwise, returns false. Object next( ) Returns the next element. Throws NoSuchElementException if there is not a next element. void remove( ) Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).
Table 11.3 The Methods Declared by Iterator
Here is an example for iterator. PROGRAM // Demonstrate iterators.
import java.util.*; class IteratorDemo { public static void main(String args[]) { // create an array list ArrayList al = new ArrayList(); // add elements to the array list al.add("C"); al.add("A");
al.add("E"); al.add("B"); al.add("D"); al.add("F"); //
use iterator to display contents of al System.out.print("Original contents of al: "); Iterator itr = al.iterator(); while(itr.hasNext()) { Object element = itr.next(); System.out.print(element + " "); } System.out.println(); } }
OUTPUT
Original contents of al: C A E B D F 11.8
The ArrayList class The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.
Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.
ArrayList has the constructors shown here: ArrayList( ) ArrayList(Collection c) ArrayList(int capacity) The first constructor builds an empty array list. The second
constructor builds an array list that is initialized with the elements of the collection c. The third constructor builds an array list that has the specified initial capacity. The
following program shows a simple use of ArrayList. An array list is created, and then objects of type String are added to it. (Recall that a quoted string is translated into a String object.) The list is then displayed. Some of the elements are removed and the list is displayed again. PROGRAM // Demonstrate

create an array list ArrayList al = new ArrayList(); System.out.println("Initial size of al: " + al.size()); // add elements to the array list al.add("C"); al.add("A"); al.add("E"); al.add("B"); al.add("D"); al.add("F"); al.add(1, "A2"); System.out.println("Size of al after additions: " + al.size()); // display the array list System.out.println("Contents of al: " + al); // Remove elements from the array list al.remove("F"); al.remove(2); System.out.println("Size of al after deletions: " + al.size()); System.out.println("Contents of al: " + al); } } OUTPUT Initial size of al: 0 Size of al after additions: 7 Contents of al: [C, A2, A, E, B, D, F] Size of al after deletions: 5 Contents of al: [C, A2, E, B, D] 11.9
The LinkedList Class The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure. It has the two constructors, shown here: LinkedList( ) LinkedList(Collection c)
The first constructor builds an empty linked list. The second constructor builds a linkedlist
that is initialized with the elements of the collection c.
In addition to the methods
that it inherits, the LinkedList class defines some useful methods of its own for manipulating and accessing lists. To add elements to the start of the list, use addFirst( ); to add elements to the end, use addLast( ). Their signatures are shown here: void addFirst(Object obj) void addLast(Object obj) Here, obj is the item being added. To obtain the first element, call getFirst( ). To retrieve the last element, call getLast( ). Their signatures are shown here: Object getFirst( ) Object getLast( ) To remove the first element, use removeFirst( ); to remove the last element, call removeLast( ). They are shown here: Object removeFirst( ) Object removeLast( )
The following program illustrates several of the methods supported by LinkedList: PROGRAM // Demonstrate LinkedList.
import java.util.*; class LinkedListDemo { public static void main(String args[]) { // create
a linked list LinkedList ll = new LinkedList(); // add elements to the linked list ll.add("F"); ll.add("B"); ll.add("D"); ll.add("E");
ll.add("C"); ll.addLast("Z"); ll.addFirst("A"); ll.add(1, "A2"); System.out.println("Original
contents of ll: " + ll);
// remove elements from the linked list ll.remove("F"); ll.remove(2); System.out.println("Contents of ll after

deletion: " + ll); // remove first and last elements ll.removeFirst(); ll.removeLast(); System.out.println("ll after deleting first and last: " + ll); // get and set a value Object val = ll.get(2); ll.set(2, (String) val + " Changed"); System.out.println("ll after change: " + ll); } } OUTPUT Original contents of ll: [A, A2, F, B, D, E, C, Z] Contents of ll after deletion: [A, A2, D, E, C, Z] ll after deleting first and last: [A2, D, E, C] ll after change: [A2, D, E Changed, C] 11.10
Vector Class Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
Vector is synchronized,
and it
contains many legacy methods that are not part of the collections framework.
Here are the Vector constructors: Vector( ) Vector(int size) Vector(int size, int incr) Vector(Collection c) The first form creates a default vector, which has an initial size of 10.
The second form
creates a vector whose initial capacity is specified by size.
The third form
creates a vector whose initial capacity is specified by size and
whose increment is specified by incr. The increment
specifies the number of elements to allocate each time that a vector is resized upward.
The fourth form
creates a vector that contains the elements of collection c.
The vector class provides plenty of useful methods. The addElement() method enables us to append an element to a vector object, while the insertElement() method helps to insert an element at a given
index position. Once we store a collection of objects in a vector, we can extract an object from a specific location in that vector, by using elementAt(), firstElement() and lastElement() methods. It is also possible to search for an individual element with the help of contains(), indexOf() and lastIndexOf() methods. The following program uses a vector to store various types of numeric objects. It demonstrates several of the legacy methods defined by Vector. PROGRAM // Demonstrate various Vector operations.
import java.util.*; class VectorDemo { public static void main(String args[]) { // initial size is 3, increment is 2 Vector v = new Vector(3, 2); System.out.println("Initial size: " + v.size()); System.out.println("Initial capacity: " + v.capacity()); v.addElement(new Integer(1)); v.addElement(new Integer(2)); v.addElement(new Integer(3)); v.addElement(new Integer(4)); System.out.println("Capacity after four additions: " + v.capacity()); v.addElement(new Double(5.45)); System.out.println("Current capacity: " + v.capacity()); v.addElement(new Double(6.08)); v.addElement(new Integer(7)); System.out.println("Current capacity: " + v.capacity()); v.addElement(new Float(9.4)); v.addElement(new Integer(10)); System.out.println("Current capacity: " + v.capacity()); v.addElement(new Integer(11)); v.addElement(new Integer(12)); System.out.println("First element: " + (Integer)v.firstElement()); System.out.println("Last element: " + (Integer)v.lastElement()); if(v.contains(new Integer(3))) System.out.println("Vector contains 3.");
// enumerate the elements in the vector. Enumeration vEnum = v.elements(); System.out.println("\nElements in vector:"); while(vEnum.hasMoreElements()) System.out.print(vEnum.nextElement() + " "); System.out.println(); } } OUTPUT Initial size: 0 Initial capacity: 3 Capacity after four additions: 5 Current capacity: 5 Current capacity: 7 Current capacity: 9 First element: 1 Last element: 12 Vector contains 3. Elements in vector: 1 2 3 4 5.45 6.08 7 9.4 10 11 12 11.11 Stack Class Stack is a subclass of Vector that implements a standard last-in, first-out stack. Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.
To put an object on the top of the stack, call push( ). To remove and return the top element, call pop( ). An EmptyStackException is thrown if you call pop( ) when the invoking stack is empty. You can use peek( ) to return, but not remove, the top object. The empty( ) method returns true if nothing is on the stack. The search( ) method determines whether an object exists on the stack, and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several Integer objects onto it, and then pops them off again. PROGRAM // Demonstrate the Stack class.
import java.util.*; class StackDemo { static void showpush(Stack st, int a) { st.push(new Integer(a)); System.out.println("push(" + a + ")"); System.out.println("stack: " + st); } static void showpop(Stack st) { System.out.print("pop -< "); Integer a = (Integer) st.pop(); System.out.println(a);
System.out.println("stack: " + st); } public static void main(String args[]) { Stack st = new Stack(); System.out.println(" stack: " +
st); showpush(st, 42); showpush(st, 66); showpush(st, 99); showpop(st); showpop(st); showpop(st); try { showpop(st); } catch (EmptyStackException e) { System.out.println("empty stack"); } } } OUTPUT stack: [ ] push(42)
stack: [42] push(66) stack: [42, 66] push(99) stack: [42, 66, 99] pop -< 99 stack: [42, 66] pop -< 66 stack: [42] pop -< 42 stack: [ ] pop -< empty stack 11.12
Hashtable Class Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
Hashtable stores key/value pairs in a hash table. When using a Hashtable, specify an object that is used as a key, and the value that

is to be
linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
A hash table can only store objects that override the hashCode() and equals( ) methods that are defined by Object. The hashCode() method must compute and return the hash code for the object. Of course, equals() compares two objects. For example, the most common type of Hashtable uses a String object as the key. String implements both hashCode( ) and equals( ). The Hashtable constructors are shown here: Hashtable( ) Hashtable(int size) Hashtable(int size, float fillRatio) Hashtable(Map m) The first version is the default constructor. The second version
creates a hash table that has an initial size specified by size.
The third version
creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.
Finally, the fourth version creates a hash table that is initialized with the elements in m. The capacity of the hash table is set to twice the number of elements in m. Here is an example to demonstrate the Hashtable.
PROGRAM // Demonstrate a Hashtable
import java.util.*; class HTDemo { public static void main(String args[]) { Hashtable balance = new Hashtable(); Enumeration names; String str; double bal; balance.put("John Doe", new Double(3434.34)); balance.put("Tom Smith", new Double(123.22)); balance.put("Jane Baker", new Double(1378.00)); balance.put("Todd Hall", new Double(99.22)); balance.put("Ralph Smith", new Double(-19.08)); // Show all balances in hash table. names = balance.keys(); while(names.hasMoreElements()) { str = (String) names.nextElement(); System.out.println(str + ": " + balance.get(str)); } System.out.println(); // Deposit 1,000 into John Doe's account bal = ((Double)balance.get("John Doe")).doubleValue(); balance.put("John Doe", new Double(bal+1000)); System.out.println("John Doe's new balance: " +balance.get("
John Doe")); } } OUTPUT Todd Hall: 99.22 Ralph Smith: -19.08 John Doe: 3434.34 Jane Baker: 1378.0 Tom Smith: 123.22 John Doe'
s new balance: 4434.34 11.13 Properties Class Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. The Properties class is used by many other Java classes.
Properties defines the following instance variable: Properties defaults;
This variable holds a default property list associated with a Properties object. Properties
defines these constructors: Properties( ) Properties(Properties propDefault) The first version creates a Properties object that has no default values. The second
creates an object that uses propDefault for its default values. In both cases, the property list is empty.
One useful capability of the Properties class is that you can specify a default property that will be returned if no value is associated with a certain key. For example, a default value can be specified along with the key in the getProperty( ) method —such as getProperty(—nameǁ, —default valueǁ). If the —nameǁ value is not found, then —default valueǁ is returned. The following example demonstrates Properties. PROGRAM //Program to illustrate Properties

76% **MATCHING BLOCK 150/221** W

class import java.util.Properties; class PropertiesDemo { public static void main(String args[]) {

Properties prop = new Properties(); prop.put(—CD Titleǁ, —Enter the CD Titleǁ); prop.put(—Authorǁ, —Enter the Authorǁ); prop.put(—Priceǁ, —Price not setǁ); Properties cd = ne Properties(prop); cd.put(—CD Titleǁ, —Java e-bookǁ); cd.put(—Authorǁ, —Herbert Schildtǁ); System.out.println(—CD Title: ǁ + cd.getProperty(—CD Titleǁ));

System.out.println(—Author: ‖ + cd.getProperty(—Author‖)); System.out.println(—Price: ‖ + cd.getProperty(—Price‖)); } } OUTPUT CD Title: Java e-book Author: Herbert Schildt Price: Price not set 11.14 StringTokenizer Class The processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts, or tokens, which in a certain sequence can convey a semantic meaning. The StringTokenizer class provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner. StringTokenizer implements the Enumeration interface. To use StringTokenizer, specify an input string and a string that contains delimiters. Delimiters are characters that separate tokens. For example, the delimiters can be a comma, a semicolon, or a colon. The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return. The StringTokenizer constructors are shown here: StringTokenizer(String str) StringTokenizer(String str, String delimiters) StringTokenizer(String str, String delimiters, boolean delimAsToken) In all versions, str is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, delimiters is a string that specifies the delimiters. In the third version, if delimAsToken is true, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Once a StringTokenizer object is created, the nextToken( ) method is used to extract consecutive tokens. The hasMoreTokens( ) method returns true while there are more tokens to be extracted. Since StringTokenizer implements Enumeration, the hasMoreElements( ) and nextElement( ) methods are also implemented, and they act the same as hasMoreTokens( ) and nextToken( ),
respectively. Here is an example that creates a StringTokenizer to parse —key=value‖ pairs. Consecutive sets of —key=value‖ pairs are separated by a semicolon. PROGRAM // Demonstrate StringTokenizer. import java.util.StringTokenizer; class STDemo { static String in = "title=Java: The Complete Reference;" + "author=Schildt;" + "publisher=Osborne/McGraw-Hill;" + "copyright=2002";
public static void main(String args[]) { StringTokenizer st = new StringTokenizer(
in, "=;"); while(st.hasMoreTokens()) { String key = st.nextToken(); System.out.println(key + "\t" + val); } } } OUTPUT title Java: The Complete Reference author Schildt publisher Osborne/McGraw-Hill copyright 2002 11.15 Date Class The Date class encapsulates the current date and time. Date supports the following constructors: Date( ) Date(long millisec) The first constructor initializes the object with the current date and time.
The second
constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.
The Date features do not allow you to obtain the individual components of the date or time.
The methods defined by Date are shown in Table 11.4. Method
Description
boolean after(Date date) Returns true if the invoking Date object contains a date that is later than the one specified by date. Otherwise, it returns false. Boolean before(Date date) Returns true if the invoking Date object contains a date that is earlier than the one specified by date. Otherwise, it returns false. Object clone( ) Duplicates the invoking Date object. intcompareTo(Date date) Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date. int compareTo(Object obj) Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException. boolean equals(Object date) Returns true if the invoking Date object contains the same time and date as the one specified by date. Otherwise, it returns false. long getTime( ) Returns the number of milliseconds that have elapsed since January 1, 1970. int hashCode( ) Returns a hash code for the invoking object. void setTime(long time) Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970. String toString( ) Converts the invoking Date object into a string and returns the result
As the following program demonstrates, you can only obtain the date and time in terms of milliseconds or in its default string representation as returned by toString( ). PROGRAM // Show date and time using only Date methods.
import java.util.Date;
class DateDemo { public static void main(String args[]) { // Instantiate a Date object Date date = new Date(); // display time and date using toString() System.out.println(date); //
Display number of milliseconds since midnight, January 1, 1970 GMT long msec = date.getTime();
System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec); } } OUTPUT Mon Apr 22 09:51:52 CDT 2002
Milliseconds since Jan. 1, 1970 GMT = 1019487112894 11.16 Unit Summary This Unit explained the Collections framework. It briefly introduced the important interfaces such as List, Set, Map, Enumeration, Iterator interfaces and classes such as ArrayList, LinkedList, Vector, Stack, Properties, HashTable, StringTokenizer, and Date. 11.17
Key Terms 1.
The Collection interface is the foundation upon which the collections framework is built. 2.
The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements. 3.
The Set interface defines a set. It
extends Collection and declares the behavior of a collection that
does not allow duplicate elements. 4.

A Map is an object that maps keys to values, or is a collection of attribute-value pairs. 5.
Enumeration is a simple interface that declares two methods, namely the hasMoreElements() method and the nextElement() method. 6.
Iterator enables you to cycle through a collection, obtaining or removing elements. 7.
The ArrayList class extends AbstractList and implements the List interface.
8.
The LinkedList class extends AbstractSequentialList and implements the List interface. 9.
Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
Vector is 10. synchronized,
and it
contains many legacy methods that are not part of the collections framework. 11.
Stack is a subclass of Vector that implements a standard last-in, first-out stack. Stack only defines the default constructor, which creates an empty stack. 12.
Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. 13.
Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. 14.
The StringTokenizer class
provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner 15.
The Date class encapsulates the current date and time. Date supports the following constructors: Date( ), Date(long millisec) 11.18
Check Your Progress 1. Write a note on the
Enumeration interface. 2. List any three unique features of Vector class. 3. Discuss the important methods of Stack class. 4. Explain the salient features of
the StringTokenizer class. 5. Illustrate the Properties class with an example.
Unit 12: Files And I/Ostreams 12.0 Introduction 12.1 Unit Objective 12.2 File 12.2.1 Directories 12.2.2 Using FilenameFilter 12.2.3 The listFiles( ) Alternative 12.2.4 Creating Directories 12.3 The Stream Classes 12.3.1 The Byte Streams 12.3.2 The Character Streams 12.4 Serialization 12.5 Unit Summary 12.6 Key Terms 12.7 Check Your Progress 12.0
Introduction A stream is a logical entity that either produces or consumes information. Data is retrieved from an input source. The results of a program are sent to an output destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the stream. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Two fundamental types of streams are input streams and output streams. Figure 12.1 explains the working of Java OutputStream and InputStream. While an output stream writes data into a file (or stream or program or device), an input stream is used to read data from a file (or stream or program or device).
Fig 12.1 Working of Java OutputStream and InputStream The java.io package has plenty of Stream classes. This Lesson discusses the various types of Stream classes and their usage in dealing with disk files. 12.1
Objectives This Unit
explores java.io, which provides support for I/O operations. It deals with • InputStream, Output Stream, FileInputStream, FileOutputStream, PipedInputStream and PrintStream under the Byte Streams. • Reader, Writer, FileReader and FileWriter under Character Streams. • Serialization 12.2
File Most of the classes defined by java.io operate on streams, but the File class does not. It deals directly with files and the file system. That is, the File class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies. The following constructors can be used to create File objects: File(String directoryPath) File(String directoryPath, String filename) File(File dirObj, String filename) File(URI uriObj) Here, directoryPath is the path name of the file, filename is the name of the file, dirObj is a File object that specifies a directory, and uriObj is a URI object that describes a file.
File defines many methods that obtain the standard properties of a File object. For example, getName( ) returns the name of the file, getParent( ) returns the name of the parent directory, and exists( ) returns true if the file exists, false if it does not.
The following example demonstrates several of the File methods: 12.2.1
Directories

85%	MATCHING BLOCK 151/221	W

A directory is a File that contains a list of other files and directories.

When a File object is created, it is a directory, the isDirectory( ) method will return true. When
the list( )
method is called on that object, it extracts the list of other files and directories inside. It's general form is shown here: String[
] list( )
The list of files is returned in an array of String objects. The program shown here illustrates how to use list( ) to examine the
contents of
a
directory: 12.2.2 Using FilenameFilter In order to list only those files that match a certain filename pattern, or filter, the following form of list( ) method is used. String[ ] list(FilenameFilter FFObj) In this form, FFObj is an object of a class that implements the FilenameFilter interface. FilenameFilter defines only a single method, accept( ), which is called once for each file in a list. Its general form is given here: boolean accept(File directory, String filename)
The accept( ) method returns true for files in the directory specified by directory that should be included in the list (that is, those that match the filename argument), and returns false for those files that should be excluded. The OnlyExt class, shown next, implements FilenameFilter. It will be used to modify the preceding program so that it restricts the visibility of the filenames returned by list( ) to files with names that end in the file extension specified when the object is constructed. The modified directory listing program is shown here. Now it will only display files that use the .html extension. 12.2.3 The listFiles( ) Alternative Java 2 added a variation to the list( ) method, called listFiles(). The signatures for listFiles( ) are shown here: File[ ] listFiles( ) File[ ] listFiles(FilenameFilter FFObj) File[ ] listFiles(FileFilter FObj) These methods return the file list as an array of File objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified FilenameFilter. Aside
from returning an array of File objects, these two versions of listFiles( ) work like their equivalent list( ) methods. The third version of listFiles( ) returns those files with path names that satisfy the specified FileFilter. FileFilter defines only a single method, accept( ), which is called once for each file in a list. Its general form is given here: boolean accept(File path) The accept( ) method returns true for files that should be included in the list (that is, those that match the path argument), and false for those that should be excluded. 12.2.4 Creating Directories Another two useful File utility methods are mkdir( ) and mkdirs( ).

100% **MATCHING BLOCK 152/221** W

The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.

To create a directory for which no path exists, the mkdirs( ) method is used. It

100% **MATCHING BLOCK 153/221** W

creates both a directory and all the parents of the directory.

Figure 12.2 Java Stream Classes
12.3 The Stream Classes Java's stream-based I/O is built upon four abstract classes: InputStream, OutputStream, Reader, and Writer. InputStream and OutputStream are designed for byte streams. Reader and Writer are designed for character streams. The byte stream classes and the character stream classes form separate hierarchies. In general, the character stream classes are used when working with characters or strings and the byte stream classes are used when working with bytes or other binary objects. This is diagrammatically explained in Figure 12.2. 12.3.1 The Byte Streams The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. InputStream InputStream is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an IOException on error conditions. The methods in InputStream are listed in Table 12.1. Method Description int available( ) Returns the number of bytes of input currently available for reading. void close( ) Closes the input source. Further read attempts will generate an IOException. void mark(int numBytes) Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read. boolean markSupported() Returns true if mark( )/reset( ) are supported by the invoking stream. int read( ) Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. int read(byte buffer[ ]) Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered. int read(byte buffer[ ], int offset, int numBytes) Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –

1 is returned when the end of the file is encountered. void reset( ) Resets the input pointer to the previously set mark. long skip(long numBytes) Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.
Table 12.1 The Methods Defined by InputStream
OutputStream OutputStream is an abstract class that defines streaming byte output. All of the methods in this class return a void value and throw an IOException in the case of errors. Table 12-2 shows the methods in OutputStream. Method Description void close( ) Closes the output stream. Further write attempts will generate an IOException. void flush( ) Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. void write(int b) Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write( ) with expressions without having to cast them back to byte. void write(byte buffer[ ]) Writes a complete array of bytes to an output stream. void write(byte buffer[ ], int offset, int numBytes) Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].
Table 17.2. The
Methods Defined by OutputStream
FileInputStream The FileInputStream class creates an InputStream that can be use to read bytes from a file. Its two most common constructors are shown here:
FileInputStream(String filepath) FileInputStream(File fileObj) Either can throw a FileNotFoundException. Here, filepath is the full path name of a file, and fileObj is a File object that describes the file. In the following example, the read() method in the FileInputStream class reads in a single byte at a time. This byte is stored as the value of the integer type variable c. If the value of the variable c is equal to -1, we stop the reading operation and close the file Sample.dat
FileOutputStream FileOutputStream creates an OutputStream that can be used to write bytes to a file. The most commonly used constructors are shown here: FileOutputStream(String filePath) FileOutputStream(File fileObj) FileOutputStream(String filePath, boolean append) FileOutputStream(File fileObj, boolean append) They can throw a FileNotFoundException or a SecurityException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, the file is opened in append mode. FileOutputStream will create the file before opening it for output during the creation of the object. When an attempt is made to open a read-only file, an IOException will be thrown. In
the following example, a string is converted into a byte array named b. Then, a data file named sample.dat is associated with an object of FileOutputStream class. Then, each byte in the byte array b is written into the data file sample.dat and the file is closed. //
Demonstrate FileOutputStream.

84% **MATCHING BLOCK 154/221** **W**

import java.io.*; class FileOutputStreamDemo { public static void main(String args[]) throws IOException {

String str = —Welcomell; Byte b [] = str.getBytes(); FileOutputStream fos = new FileOutputStream(“sample.dat”); for (i =0; i>b.length; i++) { fos.write(b[i]); } fos.close(); } } PipedInputStream and PipedOutputStream classes The PipedInputStream and PipedOutputStream classes can be used to read and write data simultaneously. Both streams are connected with each other using the connect() method of the PipedOutputStream class. This is explained through the following example. Here, we have created two threads t1 and t2. The t1 thread writes the data using the PipedOutputStream object and the t2 thread reads the data from that pipe using the PipedInputStream object. Both the piped stream object are connected with each other. PROGRAM //
Demonstrate Pipedinputstream and Pipedoutputstream
import java.io.*; class PipedWR{ public static void main(String args[])
throws
Exception{
final PipedOutputStream pout=new PipedOutputStream(); final PipedInputStream pin=new PipedInputStream(); pout.connect(pin); //connecting the streams //creating one thread t1 which writes the data Thread t1=new Thread(){ public void run(){ for(int i=65;i>=90;i++){ try{ pout.write(i); Thread.sleep(1000); }catch(Exception e){} } } }; // creating another thread t2 which reads the data Thread t2=new

100% **MATCHING BLOCK 155/221** **W**

Thread(){ public void run(){ try{ for(int i=65;i>=90;i++) System.out.println(

pin.read()); }catch(Exception e){} } }; //starting both threads t1.start(); t2.start(); } } OUTPUT Captial A to Z will be displayed PrintStream The PrintStream class provides all of the formatting capabilities. Here are two of its constructors: PrintStream(OutputStream outputStream) PrintStream(OutputStream outputStream, boolean flushOnNewline) where flushOnNewline controls whether Java flushes the output stream every time a newline (\n) character is output. If flushOnNewline is true, flushing automatically takes place. If it is false, flushing
is not automatic. The first constructor does not automatically flush. Java's PrintStream objects support the print( ) and println( ) methods for all types, including Object. If an argument is not a simple type, the PrintStream methods will call the object's toString( ) method and then print the result. 12.3.2 The Character Streams The byte stream classes provide sufficient functionality to handle any type of I/O operation, but they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the —write once, run anywherell philosophy, it was necessary to include direct I/O support for characters. The Reader and Writer abstract classes are the top most character stream classes in the hierarchies. Reader Reader is an abstract class that defines Java's model of streaming character input. All of the methods in this class will throw an IOException on error conditions. Table 12-3 provides a synopsis of the methods in Reader. Method Description abstract void close( ) Closes the input source. Further read attempts will generate an IOException. void mark(int numChars) Places a mark at the current point in the input stream that will remain valid until numChars characters are read. boolean markSupported() Returns true if mark( )/reset( ) are supported on this stream. int read() Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered. int read(char buffer[ ]) Attempts to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered. abstract int read(char buffer[ ], int offset, int numChars) Attempts to read up to numChars characters into buffer starting at buffer[offset], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( ) Returns true if the next input request will not wait. Otherwise, it returns false. void reset( ) Resets the input pointer to the previously set mark. long skip(long numChars) Skips over numChars characters of input, returning the number of characters actually skipped.
Table 17-3 The Methods Defined by Reader
Writer Writer is an abstract class that defines streaming character output. All of the methods in this class return a void value and throw an IOException in the case of errors. Table 12-4 shows a synopsis of the methods in Writer. Method Description abstract void close( ) Closes the output stream. Further write attempts will generate an IOException. abstract void flush( ) Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. void write(int ch) Writes a single character to the invoking output stream. Note that the parameter is an int, which allows you to call write with expressions without having to cast them back to char. void write(char buffer[]) Writes a complete array of characters to the invoking output stream. abstract void write(char buffer[ ], int offset, int numChars) Writes a subrange of numChars characters from the array buffer, beginning at buffer[offset] to the invoking output stream. void write(String str) Writes str to the invoking output stream. void write(String str, int offset, int numChars) Writes a subrange of numChars characters from the array str, beginning at the specified offset. Table 12-4 The
Methods Defined by Writer
FileReader The FileReader class creates a Reader that can be used to read the contents of a file. The two most commonly used constructors are shown here:
FileReader(String filePath) FileReader(File fileObj Either can throw a FileNotFoundException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. The following example shows how to read lines from a file and print these to the standard output stream. It reads its own source file, which must be in the current directory.
PROGRAM // Demonstrate
FileReader.
import java.io.*; class FileReaderDemo { public static void main(String args[]) throws
Exception {
FileReader
fr = new FileReader("
FileReaderDemo.java"); BufferedReader br = new BufferedReader(
fr); String s; while((s = br.readLine()) != null) { System.out.println(
s); } fr.close(); } } FileWriter FileWriter creates a Writer that you can use to write to a file. Its most commonly used constructors are shown here: FileWriter(String filePath) FileWriter(String filePath, boolean append) FileWriter(File fileObj) FileWriter(File fileObj, boolean append)
They can throw an IOException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, then output is appended to the end of the file. The fourth constructor was added by Java 2, version 1.4. PROGRAM //
Demonstrate FileWriter.
import java.io.*; class FileWriterDemo { public static void main(String args[]) throws
Exception {

try { FileWriter fw = new FileWriter (—samp.datll); For (char i=65; i>91; i++) { Fw.write(i); } } catch (Exception e) { System.out.println(—
Exceptin : +e); } } } OUTPUT ABCDEFGHIJKLMNOPQRSTUVWXYZ
12.4 Serialization Serialization is the process of writing the state of an object to a byte stream. This is useful when there is a need to save the program to a persistent storage area, such as a file and later, these objects can be restored by using the process of deserialization. Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it. The interfaces and classes that support serialization are discussed here. Serializable Only an object that implements the Serializable interface can be saved and restored by the serialization facilities. The Serializable interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable. Variables that are declared as transient are not saved by the serialization facilities. Also, static variables are not saved. Externalizable The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The Externalizable interface is designed for these situations. The Externalizable interface defines these two methods: void readExternal(ObjectInput inStream) throws IO Exception, ClassNotFoundException void writeExternal(ObjectOutput outStream) throws IOException In these methods, inStream is the byte stream from which the object is to be read, and outStream is the byte stream to which the object is to be written. ObjectOutput The ObjectOutput interface extends the DataOutput interface and supports object serialization.
It defines the methods shown in Table 12.5. The writeObject( ) method is called to serialize an object. All of these methods will throw an IOException on error conditions. Method Description void close( ) Closes the invoking stream. Further write attempts will generate an IOException. void flush( ) Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. void write(byte buffer[ ]) Writes an array of bytes to the invoking stream. void write(byte buffer[ ], int offset, int numBytes) Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset]. void write(int b) Writes a single byte to the invoking stream. The byte written is the low-order byte of b. void writeObject(Object obj) Writes object obj to the invoking stream. Table 12.5 The
Methods Defined by ObjectOutput
ObjectOutputStream The ObjectOutputStream class extends the OutputStream class and implements the ObjectOutput interface. It is responsible for writing objects to a stream. A constructor of this class is ObjectOutputStream(OutputStream outStream) throws IOException The argument outStream is the output stream to which serialized objects will be written. The most commonly used methods in this class are shown in Table 12.6. They will throw an IOException on error conditions. Method Description void close( ) Closes the invoking stream. Further write attempts will generate an IOException. void flush( ) Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. void write(byte buffer[ ]) Writes an array of bytes to the invoking stream. void write(byte buffer[ ], int offset, int numBytes) Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset]. void write(int b) Writes a single byte to the invoking stream. The byte written is the low-order byte of b. void writeBoolean(boolean b) Writes a boolean to the invoking stream.
void writeByte(int b) Writes a byte to the invoking stream. The byte written is the low- order byte of b. void writeBytes(String str) Writes the bytes representing str to the invoking stream. void writeChar(int c) Writes a char to the invoking stream. void writeChars(String str) Writes the characters in str to the invoking stream. void writeDouble(double d) void void Writes a double to the invoking Stream writeFloat(float f) Writes a float to the invoking stream. writeInt(int i) Writes an int to the invoking stream. void writeLong(long l) Writes a long to the invoking stream. final void writeObject(Object obj) Writes obj to the invoking stream. void writeShort(int i) Writes a short to the invoking stream. Table 12.6. Commonly Used Methods Defined by ObjectOutputStream ObjectInput The ObjectInput interface extends the DataInput interface and defines the methods shown in Table 12.7. It supports object serialization. The readObject( ) method is called to deserialize an object. All of these methods will throw an IOException on error conditions. Method Description int available( ) Returns the number of bytes that are now available in the input buffer. void close( ) Closes the invoking stream. Further read attempts will generate an IOException. int read( ) Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. int read(byte buffer[ ]) Attempts to read up to buffer.length bytes into buffer, returning the number of bytes that were successfully read. – 1 is returned when the end of the file is encountered

int read(byte buffer[ ], int offset, int numBytes) Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered. Object readObject( ) Reads an object from the invoking stream. long skip(long numBytes) Ignores (that is, skips) numBytes bytes in the invoking stream, returning the number of bytes actually ignored Table 12-7. The Methods Defined by ObjectInput ObjectInputStream The ObjectInputStream class extends the InputStream class and implements the ObjectInput interface. ObjectInputStream is responsible for reading objects from a stream. A constructor of this class is ObjectInputStream(InputStream inStream) throws IOException, StreamCorruptedException The argument inStream is the input stream from which serialized objects should be read. The most commonly used methods in this class are shown in Table 12-8. They will throw an IOException on error conditions. Method Description int available( ) Returns the number of bytes that are now available in the input buffer. void close( ) Closes the invoking stream. Further read attempts will generate an IOException. int read( ) Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. int read(byte buffer[ ], int offset, int numBytes) Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered. boolean readBoolean( ) Reads and returns a boolean from the invoking stream. byte readByte( ) Reads and returns a byte from the invoking stream. char readChar( ) Reads and returns a char from the invoking stream. double readDouble( ) Reads and returns a double from the invoking stream.
float readFloat( ) Reads and returns a float from the invoking stream. void readFully(byte buffer[ ]) Reads buffer.length bytes into buffer. Returns only when all bytes have been read. void readFully(byte buffer[ ], int offset, int numBytes) Reads numBytes bytes into buffer starting at buffer[offset]. Returns only when numBytes have been read. int readInt( ) Reads and returns an int from the invoking stream. long readLong( ) Reads and returns a long from the invoking stream. final Object readObject( ) Reads and returns an object from the invoking stream. short readShort( ) Reads and returns a short from the invoking stream. int readUnsignedByte( ) Reads and returns an unsigned byte from the invoking stream. int readUnsignedShort( ) Reads an unsigned short from the invoking stream. Table 12.8 Commonly Used Methods Defined by ObjectInputStream Serialization Example The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class MyClass. This object has three instance variables that are of types String, int, and double. This is the information we want to save and restore. A FileOutputStream is created that refers to a file named —serial,‖ and an ObjectOutputStream is created for that file stream. The writeObject( ) method of ObjectOutputStream is then used to serialize our object. The object output stream is flushed and closed. A FileInputStream is then created that refers to the file named —serial,‖ and an ObjectInputStream is created for that file stream. The readObject( ) method of ObjectInputStream is then used to deserialize our object. The object input stream is then closed. The class MyClass is defined to implement the Serializable interface. If this is not done, a NotSerializableException is thrown. Declare some of the MyClass instance variables to be transient and it can be observed that the corresponding data is not saved during serialization. PROGRAM // Demonstration of Serialization

87% **MATCHING BLOCK 156/221** W

import java.io.*; public class SerializationDemo { public static void main(String args[]) { //

Object serialization try { MyClass object1 = new MyClass("Hello", -7, 2.7e10); System.out.println("object1: " + object1); FileOutputStream fos = new FileOutputStream("serial"); ObjectOutputStream oos = new ObjectOutputStream(fos); oos.writeObject(object1); oos.flush(); oos.close(); } catch(Exception e) { System.out.println("Exception during serialization: " + e); System.exit(0); } // Object deserialization try { MyClass object2; FileInputStream fis = new FileInputStream("serial"); ObjectInputStream ois = new ObjectInputStream(fis); object2 = (MyClass)ois.readObject(); ois.close(); System.out.println("object2: " + object2); } catch(Exception e) { System.out.println("Exception during deserialization: " + e); System.exit(0); } } } class MyClass implements Serializable { String s; int i; double d; public MyClass(String s, int i, double d) { this.s = s; this.i = i; this.d = d; } public String toString() { return "s=" + s + "; i=" + i + "; d=" + d;
} } OUTPUT This program demonstrates that the instance variables of object1 and object2 are identical. The output is shown here: object1: s=Hello; i=-7; d=2.7E10 object2: s=Hello; i=-7; d=2.7E10 12.5 Unit Summary This Unit described the high-level InputStream, OutputStream, Reader, and Writer classes. This model works very well
with
the network and socket streams. Finally, serialization of objects is expected to play an increasingly important role in Java programming. Java's serialization I/O classes provide a portable solution to this sometimes tricky task. 12.6
Key Terms •
A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies. •

The
byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. 12.7
Check Your Progress 1. Name some important input and output stream classes. 2. Write a
note on File class. 3. Briefly explain FileInputStream. 4. Explain FileReader and FileWriter classes. 5. What is object serialization? Explain its purpose.
Module V: Networking, Images, Applet class and Swing
Unit 13: Networking 13.0 Introduction 13.1 Unit Objective 13.2 Networking 13.3 The Networking Classes and Interfaces 13.3.1 InetAddress 13.3.2 TCP/IP Client Sockets 13.3.3 TCP/IP Server Sockets 13.3.4 URL and URLConnection classes 13.3.5 Datagrams 13.4 Unit Summary 13.5 Key Terms 13.6 Check Your Progress 13.0
Introduction A Network is a collection of computers and other devices that can send data and receive data from one another in real time. Each machine on a network is called a node. Most nodes are computers, but dumb terminals and printers can also be treated as nodes. The nodes that are fully functional computers are also known as Host. Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Java was the first programming language designed from the ground up for network applications. The classes in the java.net package provide system- independent network communication. 13.1
Objectives The aim of this Lesson is to introduce the Networking classes and interfaces. This Lesson give the overview of InetAddress class, TCP/IP Client, Server sockets, Datagrams, URL and URLConnection classes. 13.2 Networking A Network is a collection of computers and other devices that can send data and receive data from one another in real time. Each machine on a network is called a node. Most nodes are computers, but dumb terminals and printers can also be treated as nodes. The nodes that are fully functional computers are also known as Host. Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Java
was the first programming language designed from the ground up for network applications. The classes in the java.net package provide system- independent network communication. Socket Overview A network socket is a lot like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can —plug inll to the socket and communicate.
A protocol is a set of rules basically that is followed for communication.
Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. Transmission Control Protocol (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit your data. A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets. Client/Server A server is anything that has some resource that can be shared. There are compute servers, which provide computing power; print servers, which manage a collection of printers; disk servers, which provide networked disk space; and web servers, which store web pages. A client is simply any other entity that wants to gain access to a particular server. The server is a permanently available resource, while the client is free to —unplugll after it is has been served. In Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This is managed by the introduction of a port, which is a numbered socket on a particular machine. A server process is said to —listenll to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O. Reserved Sockets Once connected, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols.
Port number 21 is for FTP, 23 is for Telnet, 25 is for e- mail, 79
is for finger, 80 is for HTTP, 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port. For example, HTTP
is the protocol that web browsers and servers use to transfer hypertext pages and images. It is quite a simple protocol for a basic page- browsing web server. Here's an example of a client requesting a single file, /index.html, and the server replying that it has successfully found the file and is sending it to the client.

Proxy Servers A proxy server speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network's connection to the Internet. When a popular web site is being hit by hundreds of users, a proxy server can get the contents of the web server's popular pages at once, saving expensive internetwork transfers while providing faster access to those pages to the clients. Internet Addressing Every computer on the Internet has an address. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, that supports larger address space than IPv4. Fortunately, IPv6 is downwardly compatible with IPv4. There are 32 bits in an IPv4 IP address, and we often refer to them as a sequence of four numbers between 0 and 255 separated by dots (.). This makes them easier to remember, because they are not
randomly assigned—they are hierarchically assigned. The first few bits define which class of network, lettered A, B, C, D, or E, the address represents. Most Internet users are on a class C network, since there are over two million networks in class C. The first byte of a class C network is between 192 and 224, with the last byte actually identifying an individual computer among the 256 allowed on a single class C network. This scheme allows for half a billion devices to live on class C networks. Domain Naming Service (DNS) A DNS server is a computer server that contains a database of public IP addresses and their associated hostnames, and in most cases, serves to resolve, or translate, those common names to IP addresses as requested. Just as the four numbers of an IP address describe a network hierarchy from left to right, the name of an Internet address, called its domain name, describes a machine's location in a name space, from right to left. For example, www.osborne.com is in the COM domain (reserved for U.S. commercial sites), it is called osborne (after the company name),
and www is the name of the specific computer that is Osborne's web server.
www corresponds to the rightmost number in the equivalent IP address. 13.3 The Networking Classes and Interfaces Java supports TCP/IP both by extending the already established stream I/O interface and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. 13.3.1 InetAddress The InetAddress class is used to encapsulate both the
numerical IP address and the domain name for that address.
You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. Here the IP address format is assumed as IPv4. Factory Methods constructors. To create an InetAddress object, use one of the available factory methods. Factory methods are merely a convention whereby static methods in a class return an instance of that class. Three commonly used InetAddress factory methods are shown here.
static InetAddress getLocalHost( ) throws UnknownHostException static InetAddress getByName(String hostName) throws UnknownHostException static InetAddress[ ] getAllByName(String hostName) throws UnknownHostException
The getLocalHost( ) method simply returns the InetAddress object that represents the local host. The getByName( ) method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException. The getAllByName( ) factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException if it can't resolve the name to at least one address. The following example prints the addresses and names of the local machine and two well- known Internet web sites:
Instance Methods The InetAddress class also has several other methods. Here are some of the most commonly used. Method Description boolean equals(Object other) Returns true if this object has the same Internet address as other. byte[ ] getAddress( ) Returns a byte array that represents the object's Internet address in network byte order.

Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn't have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server, called InterNIC (internic.net). This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly. 13.3.2 TCP/IP Client Sockets TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream- based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a ―listener,‖ which waits for clients to connect before doing anything. The Socket class is designed to connect to server sockets and initiate protocol exchanges. The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets: Socket(String hostName, int port) Creates a socket connecting the localhost to the named host and port; can throw an UnknownHostException or an IOException.
String getHostAddress( ) Returns a string that represents the host address associated with the InetAddress object. String getHostName( ) Returns a string that represents the host name associated with the InetAddress object. boolean isMulticastAddress( ) Returns true if this Internet address is a multicast address. Otherwise, it returns false. String toString( ) Returns a string that lists the host name and the IP address for convenience.
Socket(InetAddress ipAddress, int port) Creates a socket using a preexisting InetAddress object and a port; can throw an IOException. A socket can be examined at any time for the address and port information associated with it, by use of the following methods: InetAddress getInetAddress( ) Returns the InetAddress associated with the Socket object. int getPort( ) Returns the
remote port to which this Socket object is connected. int getLocalPort( ) Returns the local port to which this Socket object is connected.
Once the Socket object has been created, it can also be examined to gain access to the input and output streams associated with it. Each of these methods can throw an IOException if the sockets have been invalidated by a loss of connection on the Net. These streams are used exactly like the I/O streams to send and receive data. InputStream getInputStream( ) Returns the InputStream associated with the invoking socket. OutputStream getOutputStream( ) Returns the OutputStream associated with the invoking socket. The very simple example that follows opens a connection to a ―whois‖ port on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, then send back the IP address and contact information for that site. PROGRAM //Demonstrate Sockets.
import
java.net.*;
import java.io.*; class Whois { public static void main(String args[])
throws
Exception {
int c; Socket s = new Socket("
internic.net", 43);
InputStream in = s.getInputStream(); OutputStream out = s.getOutputStream(); String str = (args.length == 0 ? "osborne.com" : args[0]) + "\n"; byte buf[] = str.getBytes(); out.write(buf); while ((c = in.read()) != -1) { System.out.print((char) c); } s.close(); } } OUTPUT Whois Server Version 1.3 Domain names in the .com, .net, and .org domains can now be registered with many different competing registrars. Go to http://www.internic.net for detailed information. Domain Name: OSBORNE.COM Registrar: NETWORK SOLUTIONS, INC. Whois Server: whois.networksolutions.com Referral URL: http://www.networksolutions.com Name Server: NS1.EPPG.COM Name Server: NS2.EPPG.COM Updated Date: 16-jan-2002 13.3.3
TCP/IP Server Sockets The ServerSocket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. ServerSockets are quite different from normal Sockets. When a ServerSocket is created, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that you wish to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an IOException under adverse conditions. Here are the constructors: ServerSocket(int port) Creates server socket on the specified port with a queue length of 50.

ServerSocket(int port, int maxQueue) Creates a server socket on the specified port with a maximum queue length of maxQueue. ServerSocket(int port, int maxQueue,InetAddress localAddress) Creates a server socket on the specified port with a maximum queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds. ServerSocket has a method called accept( ), which is a blocking call that will wait for a client to initiate communications, and then return with a normal Socket that is then used for communication with the client PROGRAM // to send a number from
the server to the client
import java.net.*;
import java.io.*; class ServerDemo { public static void main(String args[])
throws
Exception {
try { int port = Integer.parseInt(args[0]);
ServerSocket ss = new ServerSocket(
port); Socket s = ss.accept(); OutputStream os = s.getOutputStream(); DataOutputStream dos = new DataOutputStream(os); dos.writeInt(2742809); s.close(); } catch (Exception e) { System.out.println(—Exception :ll + e); } } }. //program to receive a number from the server
import
java.net.*;
import java.io.*; class ClientDemo { public static void main(String args[])
throws
Exception {
try {
String server = args[0]; int port = Integer.parseInt(args[1]); Socket s = new Socket(server, port); InputStream is = s.getInputStream(); DataInputStream dis = new DataInputStream(is); int i = dis.readInt(); System.out.println(i); s.close(); } catch (Exception e) { System.out.println(—Exception :ll + e); } } } OUTPUT • Compile both the ServerDemo.java and ClientDemo.java programs • Start the server program in a command shell by entering java ServerDemo 4321 • Start the client program in another command shell by entering java ClientDemo 127.0.0.1 4321 • 127.0.0.1 is the IP address of the local machine. The output phone number will be displayed. 2742809 13.3.4 URL and URLConnection classes
URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory.
A URL can be broken down into parts, as follows: protocol://host:port/path?query#ref Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority. The following is a URL to a Web page whose protocol is HTTP: http://www.amrood.com/index.htm?language=en#j2se
Java's URL class has several constructors, and each can throw aMalformedURLException. One commonly used form specifies the URL with a string that is identical to the one that is displayed in a browser: URL(String urlSpecifier) The next two forms of the constructor allow to break up the URL into its component parts: URL(String protocolName, String hostName, intport, String path) URL(String protocolName, String hostName, String path)
Another frequently used constructor that allows using an existing URL as a reference context and then creating a new URL from that context. URL(URL urlObj, String urlSpecifier) In the following example, we create a URL to Osborne's download page and
then examine its properties:
The port is –1; this means that one was not explicitly set. Create a URL object, to retrieve the data associated with it. To access the actual bits or content information of a URL, create a URLConnection object from it, using its openConnection( ) method. The general forms are: URL url =new URL(http://java.sun.com/index.html); URLConnection urlConnection = url.openConnection(); In the following example, a URLConnection using the openConnection( ) method of a URL object is created and then it is used to examine the document's properties and content:
0
13.3.5
Datagrams TCP/IP-style networking provides a serialized, predictable, reliable stream of packet data. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide
an alternative. Datagrams are bundles of information passed between machines. Once the datagram has been released to its intended target, there is no assurance that it will arrive. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.
TCP is like a Telephone conversation while Datagrams approach is like the postal system. While TCP is preferable in some situations, Datagrams are preferable in the kinds of applications in which raw speed is more important than getting every bit right. For example, Datagrams are preferable for sending real-time audio and video data.
Java implements datagrams on top of the UDP protocol by using two classes:

The
DatagramPacket object is the data container, while the DatagramSocket is the mechanism used to send or receive the DatagramPackets. DatagramPacket DatagramPacket defines several constructors. Four are described here.
The first constructor
specifies a buffer that will receive data, and the size of a packet. It is used for receiving data over a DatagramSocket.
The second form
allows you to specify an offset into the buffer at which data will be stored.
The third form
specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent.
The fourth form
transmits packets beginning at the specified offset into the data. Think of the first two forms as building an —in box,‖ and the second two forms as stuffing and addressing an envelope.
Here are the four constructors: DatagramPacket(byte data[ ], int size) DatagramPacket(byte data[ ], int offset, int size) DatagramPacket(byte data[ ], int size, InetAddress ip Address, int port)
DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port)
There are several methods for accessing the internal state of a DatagramPacket. They give complete access to the destination
address and port number of a packet, as well as the raw data and its length.
Here are some of the most commonly used: InetAddress getAddress( ) Returns the destination InetAddress, typically used for sending. int getPort( ) Returns the port number.
byte[ ] getData( ) Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received. int getLength( ) Returns the length of the valid data contained in the byte array that would be returned from the getData( ) method. This typically does not equal the length of the whole byte array.
Datagram Server and Client The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.
13.4
Unit Summary
This Unit explored the java.net package, which provides support for networking. It also walks around the communication between two devices
on
the internet using the various important classes such as InetAddress class, TCP/IP Client, Server sockets, Datagrams, URL and URLConnection. 13.5
Key Terms
The InetAddress class is used to encapsulate both the
numerical IP address and the domain name for that address.
TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to- point, stream- based connections between hosts on the Internet.
URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory. 13.6
Check Your Progress 1.
What is a socket? How is it used in networking? 2. Write note on ServerSocket class. 3. Elaborate on InetAddress. 4. Write a simple program to demonstrate the Client / Server concept using TCP/IP sockets. 5. Enumerate the Datagram methods used for communication.
Unit 14: Applet 14.0 Introduction 14.1 Unit Objective 14.2 The Applet Class 14.3 Applet Architecture 14.4 The HTML APPLET tag 14.4.1 Passing Parameters to Applets 14.5 Unit Summary 14.6 Key Terms 14.7 Check Your Progress 14.0 Introduction

100% MATCHING BLOCK 169/221 SA 139E1120, 151E1120,155E1140-Advanced Java Prog ... (D165246352)

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal. There are some important differences between an applet and a standalone Java application, including the following: • An applet is a Java class that extends the java.applet.Applet class. • A main() method is not invoked on an applet, and an applet class will not define main(). • Applets are designed to be embedded within an HTML page. • When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine. • A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment. • The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime. • Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed. • Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file. 14.1

Unit Objective
This lesson introduces the new concept —Appletll and discusses the necessary classes and methods for applet programming. It also discusses the HTML APPLET tag and the usage of the tag in java programming.
14.2
The Applet Class The Applet class defines the methods shown in Table 14.1.
Applet provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. Applet
extends the AWT class Panel. In turn, Panel extends Container, which extends Component. These classes provide support for Java's window-based, graphical interface. Thus, Applet provides all of the necessary support for window- based activities. Method Description void destroy( ) Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction. AccessibleContext getAccessibleContext( ) Returns the accessibilty contextfor the invoking object. AppletContext getAppletContext( ) Returns the context associated with the applet. String getAppletInfo( ) Returns a string that describes the applet.
void init( ) Called when an applet begins execution. It is the first method called for any applet. boolean isActive( ) Returns true if the applet has been started. It returns false if the applet has been stopped. static final AudioClip newAudioClip(URL url) Returns an AudioClip object that encapsulates the audio clip found at the location specified by url. This method is similar to getAudioClip( ) except that it is static and can be executed without the need for an Applet object. void play(URL url) If an audio clip is found at the location specified by url, the clip is played. void play(URL url, String clipName) If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played. void resize(Dimension dim) Resizes the applet according to the dimensions specified by dim. Dimension is a class stored inside java.awt. It contains two integer fields: width and height. void resize(int width, int height) Resizes the applet according to the dimensions specified by width and height. final void Makes stubObj the stub for the applet. This method is used by setStub(AppletStub stubObj)
the run-time system and is not usually called by your applet. A stub is a small piece of code that provides the linkage between your applet and the browser. void showStatus(String str) Displays str in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place. void start( ) Called by the browser when an applet should start (or resume) execution. It is automatically called after init() when an applet first begins. void stop( ) Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls start().
Table 14.1. The Methods Defined by Applet 14.3
Applet Architecture An applet is a window-based program. Applets are event driven.

100% MATCHING BLOCK 158/221 W

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—init( ), start( ), stop( ), and destroy( )—are defined by Applet. Another, paint( ), is defined by the AWT Component class.

100% MATCHING BLOCK 159/221 W

Applet Initialization and Termination When an applet begins, the AWT calls the following methods, in this sequence: 1. init( ) 2. start( ) 3. paint( ) 4.

repaint()

84% MATCHING BLOCK 160/221 W

When an applet is terminated, the following sequence of method calls takes place: 1. stop( ) 2. destroy( ) init( ): The init( ) method is the first method to be called.

The variables are initialized in this method.

100% MATCHING BLOCK 161/221 W

This method is called only once during the run time of

the

100% MATCHING BLOCK 162/221 W

applet. start( ) The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ). paint( ) The paint( ) method is called each time

the

100% MATCHING BLOCK 163/221 W

applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet

begins execution.

93% MATCHING BLOCK 164/221 W

The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required. repaint() The repaint( ) method is

100% MATCHING BLOCK 165/221 W

defined by the AWT. It causes the AWT run-time system to execute a call to

the

100% MATCHING BLOCK 166/221 W

applet's update( ) method, which, in its default implementation, calls paint( ).

Thus, for another part of he applet to output to its window, simply store the output and then call repaint( ). The AWT will then execute a call to paint( ), which can display the stored information.

95% MATCHING BLOCK 167/221 W

stop( ) The stop( ) method is called when a web browser leaves the HTML document containing the applet, and when it goes to another page. For example, When stop( ) is called, the applet is probably running.

The stop( ) is used

86% MATCHING BLOCK 168/221 W

to suspend threads that don't need to run when the applet is not visible. It can be restarted, when start( ) is called if the user returns to the page. destroy( ) The destroy( ) method is called when the environment determines that the applet needs to be removed completely from memory. At this point, any resources the applet may be using

should be freed up. The stop( ) method is always called before destroy( ). A Simple Banner Applet To demonstrate the
applet and repaint( ) method, a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet's window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. The banner applet is shown here:

97% MATCHING BLOCK 170/221 W

PROGRAM /* A simple banner applet. This applet creates a thread that scrolls the message contained in msg right to left across the applet's window. */ import java.awt.*; import java.applet.*; /* >applet code="SimpleBanner" width=300 height=50< >/applet< */ public class SimpleBanner extends Applet implements Runnable { String msg = " A Simple Moving Banner."; Thread t = null; int state; boolean stopFlag; // Set colors and initialize thread. public void init() { setBackground(Color.cyan); setForeground(Color.red); } // Start thread public void start() { t = new Thread(this); stopFlag = false; t.start(); } // Entry point for the thread that runs the banner. public void run() { char ch; // Display banner for(; ;) { try { repaint(); Thread.sleep(250); ch = msg.charAt(0); msg = msg.substring(1, msg.length()); msg += ch; if(stopFlag) break; } catch(InterruptedException e) {} } } / Pause the banner. public void stop() { stopFlag = true; t = null; } // Display the banner. public void paint(Graphics g) { g.drawString(msg, 50, 30); } }

OUTPUT 14.4 The HTML
APPLET tag The APPLET tag is used to start an applet from both an HTML document
and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page.

100% MATCHING BLOCK 171/221 W

The syntax for the standard APPLET tag is shown here. Bracketed items are optional. [

97% MATCHING BLOCK 172/221 W

HTML Displayed in the absence of Java] >/APPLET< CODEBASE CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read. CODE CODE is a required attribute that gives the name of the file containing your applet's compiled .class file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set. ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets.

98% MATCHING BLOCK 173/221 W

applets. NAME NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use getApplet( ), which is defined by the AppletContext interface. WIDTH AND HEIGHT WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area. ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM. VSPACE AND HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes. PARAM NAME AND VALUE The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the getParameter( ) method. HANDLING OLDER BROWSERS Some very old web browsers can't execute applets and don't recognize the APPLET tag. Although these browsers are now nearly extinct (having been replaced by Java-compatible ones), you may need to deal with them occasionally. The best way to design your HTML page to deal with such browsers is to include HTML text and markup within your >applet<>/applet< tags. If the applet tags are not recognized by your browser, you will see the alternate markup. If Java is available, it will consume all of the markup between the >applet<>/applet< tags and disregard the alternate markup. Here's the HTML to start an applet called SampleApplet in Java and to display a message in older browsers: 14.4.1

92% MATCHING BLOCK 174/221 W

Passing Parameters to Applets The APPLET tag in HTML allows to pass parameters to

the

93% MATCHING BLOCK 175/221 W

applet. To retrieve a parameter, use the getParameter( ) method. It returns the value of the specified parameter in the form of a String object. Thus, for numeric and Boolean values, convert their string representations into their internal formats.

Here is an example that demonstrates passing parameters:
14.5
Unit Summary
This Lesson enlightens the readers about the Applet architecture and the way an applet is executed. It also describes the HTML APPLET tag with various options and their purposes. Thus this lesson will be useful in developing a simple applet program in Java. 14.6 Check Your Progress 1. Distinguish between applets and applications. 2. Write
a
note on Applet architecture. 3. Write a program using applets to display a simple banner. 4. Explain the HTML APPLET tag. 5. Explain the parameter passing in the applet through an example.
Unit 15: Event Handling 15.0 Introduction 15.1 Unit Objective 15.2 The Delegation Event Model 15.3 Event Classes 15.4 Event Listener Interfaces 15.5 Unit Summary 15.6 Key Terms 15.7 Check Your Progress 15.0
Introduction This Unit examines an important aspect of Java that relates to applets: events.

96% MATCHING BLOCK 176/221 W

The event handling is at the core of successful applet programming. Most events to which

the

100% MATCHING BLOCK 177/221 W

applet will respond are generated by the user. These events are passed to

the

92% **MATCHING BLOCK 178/221** W

applet in a variety of ways, with the specific method depending upon the actual event. There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the java.awt.event package. 15.1

Unit Objective
This Unit concentrates on the Event handling mechanism used in Java for easy development of the • GUI applications through • Delegate Event model • Event classes • Event Listener Interfaces 15.2

100% **MATCHING BLOCK 179/221** W

The Delegation Event Model The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its

concept is

100% **MATCHING BLOCK 180/221** W

quite simple: a source generates an event and sends it to one or more listeners. In

this scheme,

96% **MATCHING BLOCK 181/221** W

the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to —delegatell the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

100% **MATCHING BLOCK 182/221** W

Events In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

98% MATCHING BLOCK 183/221 W

Event Sources A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form: public void addTypeListener(TypeListener el) Here, Type is the name of the event and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method is this: public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException Here, Type is the name of the event and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this: public void removeTypeListener(TypeListener el) Here, Type is the name of the event and el is a reference to the event listener. For example, to remove a keyboard listener, you would call removeKeyListener( ). The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners. Event

100% MATCHING BLOCK 184/221 W

Event Listeners A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. 15.3

Event Classes

100% MATCHING BLOCK 185/221 W

At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events.

It has one constructor as

100% MATCHING BLOCK 186/221 W

shown here: EventObject(Object src) Here, src is the object that generates this event. EventObject contains two methods: getSource( ) and toString( ). The getSource( ) method returns the source of the event. Its general form is shown here: Object getSource( )

The

100% MATCHING BLOCK 187/221 W

toString( ) returns the string equivalent of the event. The class AWTEvent, defined within the java.awt package, is a subclass

of EventObject.

100% MATCHING BLOCK 188/221 W

It is the superclass (either directly or indirectly) of all AWT- based events used by the delegation event model. Its getID( ) method can be used to determine the type of the event. The signature of this method is shown here: int getID( )

100% **MATCHING BLOCK 189/221** W

The package java.awt.event defines several types of events that are generated by various user interface elements. Table 15-1 enumerates the most important of these event classes and provides a brief description of when they are generated.

Event Class Description ActionEvent Generated when a button is pressed, a list item isdouble-clicked, or a menu item is selected. AdjustmentEvent Generated when a scroll bar is manipulated. ComponentEvent Generated when a component is hidden, moved, resized,or becomes visible. ContainerEvent Generated when a component is added to or removed from a container. FocusEvent Generated when a component gains or loses keyboard focus. InputEvent Abstract super class for all component input event classes. ItemEvent Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or des KeyEvent Generated when input is received from the keyboard. MouseEvent Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component. MouseWheelEvent Generated when the mouse wheel is moved. TextEvent Generated when the value of a text area or text field is changed. WindowEvent Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.
Table 15.1. Main Event Classes in java.awt.event 15.4
Event Listener Interfaces Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 15-2, lists commonly used listener interfaces and provides a brief description of the methods that they define. Table 15.2 Commonly Used Event Listener Interfaces Interface Description ActionListener Defines one method to receive action events. AdjustmentListener Defines one method to receive adjustment events. ComponentListener Defines four methods to recognize when a component is hidden, moved, resized, or shown. ContainerListener Defines two methods to recognize when a component is added to or removed from a container. FocusListener Defines two methods to recognize when a component gains or loses keyboard focus. ItemListener Defines one method to recognize when the state of an item changes. KeyListener Defines three methods to recognize when a key is pressed, released, or typed. MouseListener Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.

80% **MATCHING BLOCK 190/221** W

MouseMotionListener Defines two methods to recognize when the mouse is dragged or moved.

MouseWheelListener Defines one method to recognize when the mouse wheel is moved. TextListener Defines one method to recognize when a text value changes. WindowFocusListener Defines two methods to recognize when a window gains or loses input focus. WindowListener Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. Example The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window. PROGRAM //

95% **MATCHING BLOCK 191/221** W

Demonstrate the key event handlers. import java.awt.*; import java.awt.event.*; import java.applet.*; /*>applet code="SimpleKey" width=300 height=100< >/applet< */ public class SimpleKey extends Applet implements KeyListener { String msg = ""; int X = 10, Y = 20; // output coordinates public void init() { addKeyListener(this); requestFocus(); // request input focus } public void keyPressed(KeyEvent ke) { showStatus("Key Down"); } public void keyReleased(KeyEvent ke) { showStatus("Key Up"); } public void keyTyped(KeyEvent ke) { msg += ke.getKeyChar(); repaint(); } // Display keystrokes. public void paint(Graphics g) { g.drawString(msg, X, Y); } }

OUTPUT
15.5 Unit Summary This Unit clarifies the event sources, event listeners and various methods that can be used in the Event handling mechanism for easy development of the GUI applications. It also describes the Delegate Event model, Event classes and the Event Listener Interfaces. 15.6
Key Terms

100% MATCHING BLOCK 192/221 W

Events: In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

100% MATCHING BLOCK 193/221 W

Event Sources: A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event 15.7

Check Your Progress 1.
Briefly explain the Delegation Event model. 2. List the various methods defined in the Applet class. 3. Give the description of various methods provided by Event Listener Interface. 4. Write a program that demonstrates the keyboard events using applets.
Unit 16: Working with Graphics 16.0
Introduction 16.1 Unit Objective 16.2 Working with Graphics 16.3 Working with Color 16.4 Working with Fonts 16.5 Understanding Layout Manager 16.6. Unit Summary 16.7 Key Terms 16.8 Check Your Progress 16.0
Introduction The Abstract Window Toolkit (AWT) contains numerous classes and methods that allow you to create and manage windows. The main purpose of the AWT is to support applet windows, but it can also be used to create stand-alone windows that run in a GUI environment, such as Windows. 16.1
Unit Objective
The objective of this lesson is to introduce the Abstract Window Toolkit for better programming using the following classes:
• Line, Circle, Rectangle, Ellipse and Polygon classes • Color classes • Font classes • Layout manager 16.2
Working with Graphics The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a graphics context. A graphics context is encapsulated by the Graphics class and is obtained in two ways:
It is passed to an applet when one of its various methods, such as paint() or update( ), is called. It is returned by the getGraphics( ) method of
Component. The Graphics class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Drawing
Lines Lines are drawn by means of the drawLine( ) method, shown here:
void drawLine(int startX, int startY, int endX, int endY) drawLine( ) displays a line in the current drawing color that begins at startX,startY and ends at endX,endY.
The following applet draws several lines: PROGRAM // Draw lines
import java.awt.*;
import java.applet.*; /* >applet code="Lines" width=300 height=200< >/
applet< */ public class Lines extends
Applet { public void
paint(Graphics g) {
g.
drawLine(0, 0, 100, 100); g.drawLine(0, 100, 100, 0); g.
drawLine(40, 25, 250, 180); g.drawLine(75, 90, 400, 400); g.drawLine(20, 150, 400, 40); g.drawLine(5, 290, 80, 19); } }
OUTPUT Drawing Rectangles The drawRect() and fillRect( ) methods display an outlined and filled rectangle, respectively. They are shown here: void drawRect(int top, int left, int width, int height)

32% MATCHING BLOCK 194/221 W

void fillRect(int top, int left, int width, int height) The upper-left corner of the rectangle is at top,left. The dimensions of the rectangle are specified by width and height.

To draw a rounded rectangle, use drawRoundRect( ) or fillRoundRect( ), both shown here:
void drawRoundRect(int top, int left, int width, int height, int xDiam, int yDiam) void fillRoundRect(int top, int left, int width, int height, int xDiam, int

yDiam) A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at top,left. The dimensions of the rectangle are specified by width and height. The diameter of the rounding arc along the X axis is specified by xDiam. The diameter of the rounding arc along the Y axis is specified by yDiam. The following applet draws several rectangles:
Drawing Ellipses and Circles The drawOval( ) and fillOval( ) methods are used to draw oval. These methods are shown here: void drawOval(int top, int left, int width, int height) void fillOval(int top, int left, int width, int height) The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by top, left and whose width and height are specified by width and height. To draw a circle, specify a square as the bounding rectangle. The following program draws several ellipses:
Drawing Arcs Arcs can be drawn with drawArc( ) and fillArc( ), shown here:
void drawArc(int top, int left, int width, int height, int startAngle, int
sweepAngle)
void fillArc(int top, int left, int width, int height, int startAngle, int
sweepAngle) The
arc is bounded by the rectangle whose upper-left corner is specified by top, left and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degrees. Zero degrees is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if sweepAngle is positive, and clockwise if sweepAngle is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180. The following applet draws several arcs:
Drawing Polygons It is possible to draw arbitrarily shaped figures using drawPolygon( ) and fillPolygon( ), shown here: void drawPolygon(int x[ ], int y[ ], int numPoints) void fillPolygon(int x[ ], int y[ ], int numPoints) The polygon's endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by x and y is specified by numPoints. There are alternative forms of these methods in which the polygon is specified by a Polygon object. The following applet draws an hourglass shape:
16.3
Working with Color Java supports color in a portable, device-independent fashion. The AWT color system allows to specify any color we want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Color is encapsulated by the Color class. Color defines several constants (for example, Color.black) to specify a number of common colors. We can also create our own colors, using one of the color constructors. The most commonly used forms are shown here: Color(int red, int green, int blue) Color(int rgbValue) Color(float red, float green, float blue) The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example: new Color(255, 100, 100); // light red.
The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor: int newRed = (0xff000000 | (0xc0 >> 16) | (0x00 >> 8) | 0x00); Color darkRed = new Color(newRed); The final constructor, Color(float, float, float), takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue. Once the color is created, it can be used to set the foreground and/or background color by using the setForeground( ) and setBackground( ) methods. Color Methods The Color class defines several methods that help manipulate colors. They are examined here. Using Hue, Saturation, and Brightness The hue-saturation-brightness (HSB) color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, hue is a wheel of color. The hue is specified with a number between 0.0 and 1.0 (the colors are approximately: red, orange, yellow, green, blue, indigo, and violet). Saturation is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. Brightness values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. Color supplies two methods that are used to convert between RGB and HSB. They are shown here: static int HSBtoRGB(float hue, float saturation, float brightness) static float[ ] RGBtoHSB(int red, int green, int blue, float values[ ]) HSBtoRGB( ) returns a packed RGB value compatible with the Color(int) constructor. RGBtoHSB() returns a float array of HSB values corresponding to RGB integers. If values is not null, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.
getRed( ), getGreen( ), getBlue( ) The red, green, and blue components of a color can be obtained independently using getRed(), getGreen( ), and getBlue( ), shown here: int getRed() int getGreen()
int getBlue() Each of these methods returns the RGB color component found in the invoking Color object in the lower 8 bits of an integer. To obtain a packed, RGB representation of a color, getRGB( ) method can be used as shown here: int getRGB()
Setting the Current Graphics Color By default, graphics objects are drawn in the current foreground color. This color can be changed by calling the Graphics method setColor( ): void setColor(Color newColor) Here, newColor specifies the new drawing color. The current color can be obtained by calling getColor(), shown here: Color getColor() A Color Demonstration Applet The following applet constructs several colors and draws various objects using these colors:
PROGRAM // Demonstrate color.

import java.awt.*; import java.applet.*; /* >applet code="ColorDemo" width=300 height=200< >/applet< */ public class ColorDemo extends Applet { // draw lines public void

paint(Graphics g) { Color c1 = new Color(255, 100, 100); Color c2 = new Color(100, 255, 100); Color c3 = new Color(100, 100, 255); g.setColor(c1); g.drawLine(0, 0, 100, 100); g.drawLine(0, 100, 100, 0);
g.setColor(c2); g.drawLine(40, 25, 250, 180); g.drawLine(75, 90, 400, 400);
g.setColor(c3); g.drawLine(20, 150, 400, 40); g.drawLine(5, 290, 80, 19); g.setColor(Color.red); g.drawOval(10, 10, 50, 50);
g.fillOval(70, 90, 140, 100); g.setColor(Color.blue); g.drawOval(190, 10, 90, 30); g.drawRect(10, 10, 60, 50);
g.setColor(Color.cyan); g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15); } } 16.4 Working with Fonts The AWT supports multiple type fonts. Fonts have emerged from the domain of traditional typesetting to become an important part of computer-generated documents and displays. The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts. The fonts have a family name, a logical font name, and a face name. The family name is the general name of the font, such as Courier. The logical name specifies a category of font, such as Monospaced. The face name specifies a specific font, such as Courier Italic. Fonts are encapsulated by the Font class. Several of the methods defined by Font are listed in Table 16-1. Method Description static Font decode(String str) Returns a font given its name.
boolean equals(Object FontObj) Returns true if the invoking object contains the same font as that specified by FontObj. Otherwise, it returns false.
String getFamily() Returns the name of
the font family to which the invoking font belongs. static Font getFont(String property) Returns the font associated with the system property specified by property. null is returned if property does not exist.
static Font getFont(String property, Font defaultFont) Returns the font associated with the system property specified by property. The font specified by defaultFont is returned if property does not exist. String getFontName() Returns the face name of the invoking font. String getName( ) Returns the logical name of the invoking font. int getSize( ) Returns the size, in points, of the invoking font. int getStyle( ) Returns the style values of the invoking font.
int hashCode( ) Returns the hash code associated with the invoking object. boolean isBold( ) Returns true if
the
font includes the BOLD style value. Otherwise, false is returned. boolean isItalic( ) Returns true if the font includes the ITALIC style value. Otherwise, false is returned. boolean isPlain( ) Returns true if the font includes the PLAIN style value. Otherwise, false is returned. String toString( ) Returns the string equivalent of the invoking font. Table 16-1 Some Methods Defined by Font The Font class defines these variables: Variable Meaning String name Name of the font float pointSize Size of the font in points int size Size of the font in points int style Font style Determining the Available Fonts When working with fonts, to obtain the currently available font information, the getAvailableFontFamilyNames( ) method can be defined by the GraphicsEnvironment class. It is shown here: String[ ] getAvailableFontFamilyNames( ) This method returns an array of strings that contains the names of the available font families. In addition, the getAllFonts( ) method is defined by the GraphicsEnvironment class. It is shown here:
Font[ ] getAllFonts( ) This method returns an array of Font objects for all of the available fonts. Since these methods are members of GraphicsEnvironment, a GraphicsEnvironment reference has to be called. This is obtained by using the getLocalGraphicsEnvironment( ) static method, which is defined by GraphicsEnvironment. It is shown here: static GraphicsEnvironment getLocalGraphicsEnvironment( ) Here is an applet that shows how to obtain the names of the available font families: PROGRAM // Display Fonts /* >
applet code="ShowFonts" width=550 height=60< >/
applet< */
import java.applet.*;
import java.awt.*; public class ShowFonts extends Applet {
public void paint(Graphics g) {
String
msg = ""; String FontList[]; GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment(); FontList = ge.getAvailableFontFamilyNames(); for(int i = 0; i > FontList.length; i++) msg += FontList[i] + " "; g.drawString(msg, 4, 16); } } OUTPUT

Creating and Selecting a Font: To select a new font, first construct a Font object that describes that font. One Font constructor has this general form: Font(String fontName, int fontStyle, int pointSize) Here, fontName specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support the following fonts: Dialog, DialogInput, Sans Serif, Serif, Monospaced, and Symbol. The style of the font is specified by fontStyle. It may consist of one or more of these three constants: Font.PLAIN, Font.BOLD, and Font.ITALIC. To combine styles, OR them together. For example, Font.BOLD | Font.ITALIC specifies a bold, italics style. The size, in points, of the font is specified by pointSize. To use a font that is created, select it using setFont( ), which is defined by Component. It has this general form: void setFont(Font fontObj) Here, fontObj is the object that contains the desired font. The following program outputs a sample of each standard font. Each time when the mouse is clicked within its window, a new font is selected and its name is displayed. PROGRAM // Show fonts.
import java.applet.*;
import java.awt.*; import java.awt.event.*; /* >
applet code="SampleFonts" width=200 height=100< >/applet< */ public class SampleFonts extends
Applet {
int next = 0; Font f; String msg;
public void init() { f = new Font("Dialog", Font.
PLAIN, 12); msg = "Dialog"; setFont(f); addMouseListener(new MyMouseAdapter(this)); } public void paint(Graphics g) { g.drawString(msg, 4, 20); } } class MyMouseAdapter extends MouseAdapter { SampleFonts sampleFonts; public MyMouseAdapter(SampleFonts sampleFonts) { this.sampleFonts = sampleFonts; } public void mousePressed(MouseEvent me) { // Switch fonts with each mouse click. sampleFonts.next++; switch(sampleFonts.next) { case 0: sampleFonts.f =new Font("Dialog", Font.PLAIN, 12); sampleFonts.msg = "Dialog"; break; case 1: sampleFonts.f =new Font("DialogInput", Font.PLAIN, 12); sampleFonts.msg = "DialogInput"; break; case 2: sampleFonts.f =new Font("SansSerif", Font.PLAIN, 12); sampleFonts.msg = "SansSerif"; break; case 3: sampleFonts.f = new Font("Serif", Font.PLAIN, 12); sampleFonts.msg = "Serif"; break; case 4: sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12); sampleFonts.msg = "Monospaced"; break; } if(sampleFonts.next == 4) sampleFonts.next = -1; sampleFonts.setFont(sampleFonts.f); sampleFonts.repaint(); } } OUTPUT Obtaining Font Information To obtain information about the currently selected font, first get the current font by calling getFont(). This method is defined by the Graphics class, as shown here: Font getFont( ) Once the currently selected font is obtained, we can retrieve information about it using various methods defined by Font. For example, this applet displays the name, family, size, and style of the currently selected font: PROGRAM // Display font info.

81% **MATCHING BLOCK 196/221** W

import java.applet.*; import java.awt.*; /* >applet code="FontInfo" width=350 height=60< >/applet< */ public class FontInfo extends Applet { public void

paint(Graphics g) {
Font f = g.getFont(); String fontName = f.getName(); String fontFamily = f.getFamily(); int fontSize = f.getSize(); int fontStyle = f.getStyle(); String msg = "Family: " + fontName; msg += ", Font: " + fontFamily; msg += ", Size: " + fontSize + ", Style: "; if((fontStyle & Font.BOLD) == Font.BOLD) msg += "Bold "; if((fontStyle & Font.ITALIC) == Font.ITALIC) msg += "Italic "; if((fontStyle & Font.PLAIN) == Font.PLAIN) msg += "Plain "; g.drawString(msg, 4, 16); } } 16.5 Understanding Layout Manager Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them. In addition to the controls, a frame window can also include a standard-style menu bar. Each
Container object has a layout manager associated with it. A
layout manager is an instance of any class that implements the LayoutManager interface.
The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it. The setLayout( ) method has the following general form: void setLayout(LayoutManager layoutObj) Here, layoutObj is a reference to the desired layout manager. To disable the layout manager and to position the components manually, pass null for layoutObj. In this case, there is a need to determine the shape and position of each component manually, using the setBounds( ) method defined by Component. Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time a component is added to a container.

Layout Whenever the container needs to be resized, the manager is consulted via its minimumLayoutSize( ) and preferredLayoutSize( ) methods. Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ) methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy.
FlowLayout FlowLayout is the default layout manager. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for FlowLayout: FlowLayout( ) FlowLayout(int how) FlowLayout(int how, int horz, int vert) The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for how are as follows: FlowLayout.LEFT FlowLayout.CENTER FlowLayout.RIGHT
These values specify left, center, and right alignment, respectively.
The third form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.
Here is the CheckboxDemo applet that uses left-aligned flow layout. PROGRAM // Use left-aligned flow layout.

63% **MATCHING BLOCK 197/221** W

import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="FlowLayoutDemo" width=250 height=200< >/applet< */ public class FlowLayoutDemo extends Applet implements ItemListener { String msg = ""; Checkbox Win98, winNT, solaris, mac; public void init() { //

set left-aligned flow layout setLayout(new FlowLayout(FlowLayout.LEFT));
Win98 = new Checkbox("Windows 98/XP", null, true); winNT = new Checkbox("Windows NT/2000"); solaris = new Checkbox("Solaris"); mac = new Checkbox("MacOS"); add(Win98); add(winNT); add(solaris); add(mac); //
register to receive item events
Win98.addItemListener(this); winNT.addItemListener(this); solaris.addItemListener(this); mac.addItemListener(this); } //
Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) { repaint(); } // Display current state of the check boxes.
public void paint(Graphics g) { msg = "Current state: "; g.drawString(msg, 6, 80); msg = " Windows 98/XP: " +
Win98.getState(); g.drawString(msg, 6, 100);
msg = " Windows NT/2000: " + winNT.getState(); g.drawString(msg, 6, 120); msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140); msg = " Mac: " + mac.getState(); g.drawString(msg, 6, 160); } }
OUTPUT
BorderLayout The BorderLayout class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by BorderLayout: BorderLayout( )
BorderLayout(
int horz, int vert) The first form creates a default border layout. The second
allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.
BorderLayout defines the following constants that specify the regions: BorderLayout.CENTER BorderLayout.SOUTH BorderLayout.EAST BorderLayout.WEST BorderLayout.NORTH
When adding components, these constants can be used with the following form of add( ), which is defined by Container: void add(Component compObj, Object region); Here, compObj is the component to be added, and
the
region specifies where the component will be added. Here is an example of a BorderLayout with a component in each layout area: PROGRAM //
Demonstrate BorderLayout.

68% **MATCHING BLOCK 198/221** W

import java.awt.*; import java.applet.*; import java.util.*; /* >applet code="BorderLayoutDemo" width=400 height=200< >/applet< */ public class BorderLayoutDemo extends Applet { public void init() { setLayout(new

BorderLayout()); add(new Button("This is across the top."), BorderLayout.NORTH); add(new Label("The footer message might go here."), BorderLayout.SOUTH); add(new Button("Right"), BorderLayout.EAST); add(new Button("Left"), BorderLayout.WEST); String msg = "The reasonable man adapts " + "himself to the world;\n" + "the unreasonable one persists in " + "trying to adapt the world to himself.\n" + "Therefore all progress depends " + "on the unreasonable man.\n\n" + " - George Bernard Shaw\n\n"; add(new TextArea(msg), BorderLayout.CENTER); } }
OUTPUT Using Insets Sometimes, there is a need to leave a small amount of space between the container that holds the components and the window that contains it. To do this, override the getInsets( ) method that is defined by Container. This function returns an Insets object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for Insets is shown here: Insets(int top, int left, int bottom, int right) The values passed in top, left, bottom, and right specify the amount of space between the container and its enclosing window. The getInsets( ) method has this general form: Insets getInsets( ) When overriding one of these methods, a new Insets object that contains the desired inset spacing must be returned. Here is the preceding BorderLayout example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.
GridLayout GridLayout lays out components in a two-dimensional grid.
The constructors supported by GridLayout are shown here: GridLayout( )
GridLayout(int numRows, int numColumns) GridLayout(int numRows, int numColumns, int horz, int vert) The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows to specify the horizontal and vertical space left between components in horz and vert, respectively. Either numRows or numColumns can be zero. Specifying numRows as zero allows for unlimited-length columns. Specifying numColumns as zero allows for unlimited- length rows.
Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:
CardLayout: The CardLayout class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. Different layouts can be prepared and have them hidden, ready to be activated when needed. CardLayout provides these two constructors: CardLayout( ) CardLayout(int horz, int vert) The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.
Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type Panel. This panel must have CardLayout selected as its layout manager. The cards that form the deck are also typically objects of type Panel. Thus, create a panel that contains the deck and a panel for each card in the deck. Next, add to the appropriate panel the components that form each card. Then add these panels to the panel for which CardLayout is the layout manager. Finally, add this panel to the main applet panel. Once these steps are complete, provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name. Thus, most of the time, add() is used when adding cards to a panel: void add(Component panelObj, Object name); Here, name is a string that specifies the name of the card whose panel is specified by panelObj. After the deck is created, the program activates a card by calling one of the following methods defined by CardLayout: void first(Container deck void last(Container deck) void next(Container deck) void previous(Container deck) void show(Container deck, String cardName)
Here, deck is a reference to the container (usually a panel) that holds the cards, and cardName is the name of a card. Calling first () causes the first card in the deck to be shown. To show the last card, call last (). To show the next card, call next (). To show the previous card, call previous (). Both next () and previous() automatically cycle back to the top or bottom of the deck, respectively. The show() method displays the card whose name is passed in cardName. The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Macintosh and Solaris are displayed
in the other card. PROGRAM // Demonstrate CardLayout.

86% **MATCHING BLOCK 199/221** W

import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="CardLayoutDemo" width=300 height=100< >/applet< */ public class CardLayoutDemo extends Applet implements

ActionListener,
MouseListener { Checkbox Win98, winNT, solaris, mac;
Panel osCards; CardLayout cardLO; Button Win, Other; public void init() { Win = new Button("Windows"); Other = new Button("Other"); add(Win); add(Other); cardLO = new CardLayout(); osCards = new Panel();

osCards.setLayout(cardLO); // set panel layout to card layout
Win98 = new Checkbox("Windows 98/XP", null, true); winNT = new Checkbox("Windows NT/2000"); solaris = new Checkbox("Solaris"); mac = new Checkbox("MacOS"); // add
Windows check boxes to a panel Panel winPan = new Panel(); winPan.add(Win98); winPan.add(winNT); // Add other OS check boxes to a panel Panel otherPan = new Panel(); otherPan.add(solaris); otherPan.add(mac); // add panels to card deck panel osCards.add(winPan, "Windows"); osCards.add(otherPan, "Other"); // add cards to main applet panel add(osCards); // register to receive action events Win.addActionListener(this); Other.addActionListener(this); // register mouse events addMouseListener(this); } // Cycle through panels. public void mousePressed(MouseEvent me) { cardLO.next(osCards); } // Provide empty implementations for the other MouseListener
methods.
public void mouseClicked(MouseEvent me) { } public void mouseEntered(MouseEvent me) { } public void mouseExited(MouseEvent me) { } public void mouseReleased(MouseEvent
me) {
} public void
actionPerformed(ActionEvent ae) { if(ae.getSource() == Win) { cardLO.show(osCards, "Windows"); } else { cardLO.show(osCards, "Other"); } } } OUTPUT 16.6 Unit Summary This Lesson elaborates the components needed for GUI Applications. It explains the various classes used for drawing line, circle, ellipse and polygons. The various font and color classes used to achieve the desired appearance in the application were also discussed. To have a beautiful appearance of a window the controls that it contained in them must be arranged. This is achieved by using the layout manager. This lesson concludes by describing various layout management used in Java. 16.7
Key Terms
The Color class defines several methods that help manipulate colors.
16.8
Check Your Progress 1.
Write applets to draw a. Cone b. Cylinder c. Cube 2. Describe the three ways of drawing the polygon 3. Explain the way by which the Color class is used to manipulate the colors. 4. Write a note on the various methods of the Font class. 5. Brief on Layout management.
Unit 17: Swing Component Classes 17.0
Introduction 17.1 Unit Objective 17.2 JApplet 17.3 JFrame and JDialog 17.4 JText Fields 17.5 JButtons 17.6 JCombo Boxes 17.7 JList 17.8 JTabbed Panes 17.9 JScroll Panes 17.10 Unit Summary 17.11 Key Terms 17.12 Check Your Progress 17.0
Introduction Swing
is a set of classes that provides more powerful and flexible components
than are possible with the AWT.
In addition to the familiar components, such as buttons, check boxes, and labels, Swing
supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.
Even familiar
components
such as buttons have more capabilities in Swing. For example, a
button may have both an image and a text string associated with it.
Also,
the image can be changed as the state of the button changes.

100% **MATCHING BLOCK 200/221** **W**

The Swing-related classes are contained in javax.swing and its subpackages, such as javax.swing.tree. Many other Swing-related classes and interfaces

also exist. 17.1
Objectives This Unit gives the overview of Swing for the development of window based applications through Java. It provides a simple introduction to the more sophisticated set of GUI components, such as, JApplet, JFrame, JDialog, Text Fields, Buttons, Combo boxes, List ,Tabbled and Scroll Panes. 17.2

100% **MATCHING BLOCK 201/221** **W**

JApplet Fundamental to Swing is the JApplet class, which extends Applet. Applets that

use Swing

100% MATCHING BLOCK 202/221 W

must be subclasses of JApplet. JApplet is rich with functionality that is

not found

100% MATCHING BLOCK 203/221 W

in Applet. For example, JApplet supports various —panes,‖ such as the content pane, the glass pane, and the root pane.

There is one difference between Applet and JApplet.

96% MATCHING BLOCK 204/221 W

When adding a component to an instance of JApplet, do not invoke the add( ) method of the applet. Instead, call add( ) for the content pane of the JApplet object. The content pane can be obtained via the method shown here: Container getContentPane( ) The add( ) method of Container can be used to add a component to a content pane. Its form is shown here: void add(comp) Here, comp is the component to be added to the content pane. 17.3 JFrame and JDialog In

general, Swing components are derived from the JComponent class. JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component. All of Swing's components are represented by classes defined within the package javax.swing. Swing defines two types of containers. The first are top-level containers: JFrame, JApplet, JWindow, and JDialog. These containers do not inherit JComponent. They do, however, inherit the AWT classes Component and Container. Unlike Swing's other components, which are lightweight, the top- level containers are heavyweight. This makes the top-level containers a special case in the Swing component library. As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is JFrame. The one used for applets is JApplet. JFrame The
following program demonstrates several key features of Swing. It uses two Swing components: JFrame and JLabel. JFrame is the top-level container that is commonly used for Swing applications. JLabel is the Swing component that creates a label, which is a component that displays information. The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output.
The program uses a JFrame container to hold an instance of a JLabel. The label displays a short text message.
PROGRAM //
A simple Swing application. import javax.swing.*; class SwingDemo { SwingDemo() { // Create a new JFrame container. JFrame jfrm = new JFrame("A Simple Swing Application"); //
Give the frame an initial size.
jfrm.setSize(275, 100); // Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Create a text-based label. JLabel jlab = new JLabel(" Swing means powerful GUIs."); // Add the label to the content pane. jfrm.add(jlab); // Display the frame. jfrm.setVisible(true); } public static void main(String args[]) { // Create the frame on the event dispatching thread. SwingUtilities.invokeLater(new Runnable() { public void run() { new SwingDemo(); } }); } }
OUTPUT
JDialog
The JDialog control represents a top-level window with a border and a title used to take some form of input from the user. It inherits the Dialog class.
Unlike JFrame, it doesn't have maximize and minimize buttons.
The commonly used constructors of JDialog are given below:
Constructor Description
JDialog() It is used to create a modeless dialog without a title and without a specified Frame JDialog(Frame owner) It is used to create a modeless dialog with specified Frame as its owner and an empty title. Jdialog (Frame owner, String title, boolean modal) It is used to create a dialog with the specified title, owner Frame and modality.
The following program illustrates the JDialog control: PROGRAM
import javax.swing.*;

import java.awt.*; import java.awt.event.*;
public class DialogExample { private static JDialog d; DialogExample() { JFrame f= new JFrame(); d = new JDialog(f , "Dialog Example", true); d.setLayout(new FlowLayout()); JButton b = new JButton ("OK"); b.addActionListener (new ActionListener() { public void actionPerformed(ActionEvent e) { DialogExample.d.setVisible(false);
} }); d.add(new JLabel ("Click button to continue.")); d.
add(b); d.setSize(300,300); d.setVisible(true); } public static void main(String args[]) { new DialogExample(); } }
OUTPUT 17.4
JText

97% **MATCHING BLOCK 205/221** W

Fields The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows you to edit one line of text. Some of its constructors are shown here: JTextField( ) JTextField(int cols) JTextField(String s, int cols) JTextField(String s) Here, s is the string to be presented, and cols is the number of columns in the text field. The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a JTextField object is created and is added to the content pane. PROGRAM import java.awt.*; import javax.swing.*; /* >applet code="JTextFieldDemo" width=300 height=50< >/applet< */ public class JTextFieldDemo extends JApplet { JTextField jtf; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); // Add text field to content pane jtf = new JTextField(15); contentPane.add(jtf); } } OUTPUT 17.5

JButtons

97% **MATCHING BLOCK 206/221** W

Swing buttons are subclasses of the AbstractButton class, which extends JComponent. AbstractButton contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons. For example, you can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as a rollover icon, which is displayed when the mouse is positioned over that component. The following are the methods that control this behavior: void setDisabledIcon(Icon di) void setPressedIcon(Icon pi) void setSelectedIcon(Icon si) void setRolloverIcon(Icon ri) Here, di, pi, si, and ri are the icons to be used for these different conditions. The text associated with a button can be read and written via the following methods: String getText( ) void setText(String s) Here, s is the text to be associated with the button. Concrete subclasses of AbstractButton generate action events when they are pressed. Listeners register and unregister for these events via the methods shown here: void addActionListener(ActionListener al) void removeActionListener(ActionListener al) Here, al is the action listener. AbstractButton is a superclass for push buttons, check boxes, and radio buttons. The JButton Class The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here: JButton(Icon i) JButton(String s) JButton(String s, Icon i) Here, s and i are the string and icon used for the button.

The
following example displays four push buttons and a text field. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the

100% **MATCHING BLOCK 207/221** W

text field. The applet begins by getting its content pane and

setting the layout manager of that pane. Four image buttons are created and
added to the content pane. Next, the applet is registered to receive action events that are generated by the buttons. A text field is then created and added to the applet. Finally, a handler for action events displays the command string that is associated with the button. The text field is used to present this string. PROGRAM

82% MATCHING BLOCK 208/221 W

import java.awt.*; import java.awt.event.*; import javax.swing.*; /* >applet code="JButtonDemo" width=250 height=300< >/applet< */ public class JButtonDemo extends JApplet implements ActionListener { JTextField jtf; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); //

Add buttons to content pane
ImageIcon france = new ImageIcon("france.gif"); JButton jb = new JButton(france); jb.setActionCommand("France"); jb.addActionListener(this); contentPane.add(jb); ImageIcon germany = new ImageIcon("germany.gif"); jb = new JButton(germany); jb.setActionCommand("Germany"); jb.addActionListener(this); contentPane.add(jb); ImageIcon italy = new ImageIcon("italy.gif"); jb = new JButton(italy); jb.setActionCommand("Italy"); jb.addActionListener(this); contentPane.add(jb); ImageIcon japan = new ImageIcon("japan.gif");
jb = new JButton(japan); jb.setActionCommand("Japan"); jb.addActionListener(

88% MATCHING BLOCK 209/221 W

this); contentPane.add(jb); // Add text field to content pane jtf = new JTextField(15); contentPane.add(jtf); } public void

actionPerformed(ActionEvent ae) { jtf.setText(ae.getActionCommand()); } }
OUTPUT 17.6 JCombo

97% MATCHING BLOCK 210/221 W

Boxes Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class, which extends JComponent. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. Selection can

be made by typing

96% MATCHING BLOCK 211/221 W

into the text field. Two of JComboBox's constructors are shown here: JComboBox( ) JComboBox(Vector v) Here, v is a vector that initializes the combo box. Items are added to the list of choices via the addItem( ) method, whose signature is shown here: void addItem(Object obj) Here, obj is the object to be added to the combo box. The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for —Francell, —Germanyll, —Italyll, and —Japanll. When a country is selected, the label is updated to display the flag for that country. PROGRAM import java.awt.*; import java.awt.event.*; import javax.swing.*; /* >applet code="JComboBoxDemo" width=300 height=100< >/applet< */ public class JComboBoxDemo extends JApplet implements ItemListener { JLabel jl; ImageIcon france, germany, italy, japan; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); // Create a combo box and add it // to the panel JComboBox jc = new JComboBox(); jc.addItem("France"); jc.addItem("Germany"); jc.addItem("Italy"); jc.addItem("Japan"); jc.addItemListener(this); contentPane.add(jc); // Create label jl = new JLabel(new ImageIcon("france.gif")); contentPane.add(jl); } public void itemStateChanged(ItemEvent ie) { String s = (String)ie.getItem(); jl.setIcon(new ImageIcon(s + ".gif")); } } OUTPUT 17.7

JList In Swing, the basic list class is called JList. It supports the selection of one or more items from a list.
JList is declared as follows: class JList>E< Here, E represents the type of the items in the list.
JList provides several constructors. The one used here is JList(E[ ] items) This creates a JList that contains the items in the array specified by items. JList is based on two models. The first is ListModel. This interface defines how access to the list data is achieved. The second model is the ListSelectionModel interface, which defines methods that determine what list item or items are selected.
A JList allows the user to select multiple ranges of items within the list, but that can be changed by calling setSelectionMode( ), which is defined by JList. It is shown here: void setSelectionMode(int mode) Here, mode specifies the selection mode. It must be one of these values defined by ListSelectionModel:
SINGLE_SELECTION SINGLE_INTERVAL_SELECTION MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Tthe index of the first item selected can be obtained, which will also be the index of the only selected item when using single-selection mode, by calling getSelectedIndex( ), shown here: int getSelectedIndex( ) Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned. The following applet demonstrates a simple JList, which holds a list of cities. Each time a city is selected in the list, a ListSelectionEvent is generated, which is handled by the valueChanged( ) method defined by ListSelectionListener. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label. PROGRAM //
Demonstrate JList. import javax.swing.*; import javax.swing.event.*;

75% **MATCHING BLOCK 212/221** W

import java.awt.*; import java.awt.event.*; /* >applet code="JListDemo" width=200 height=120< >/applet< */
public class JListDemo extends JApplet {

JList>
String<
jlst; JLabel jlab; JScrollPane jscrlp; // Create an array of cities. String Cities[] = { "New York", "Chicago", "Houston", "Denver", "Los Angeles", "Seattle",
"London", "Paris", "New Delhi", "Hong Kong", "Tokyo", "Sydney" };
public void init() { try { SwingUtilities.invokeAndWait(new Runnable() { public void run() { makeGUI(); } }); } catch (Exception exc) { System.out.println("Can't create because of " + exc); } }
private void makeGUI() { //
Change to flow layout.
setLayout(new FlowLayout()); // Create a JList. jlst = new JList>String<(Cities); // Set the list selection mode to single selection. jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION); // Add the list to a scroll pane. jscrlp = new JScrollPane(jlst); // Set the preferred size of the scroll pane. jscrlp.setPreferredSize(new Dimension(120, 90)); // Make a label that displays the selection. jlab = new JLabel("Choose a City"); // Add selection listener for the list.
jlst.addListSelectionListener(new ListSelectionListener() { public void valueChanged(ListSelectionEvent le) { // Get the index of the changed item. int idx = jlst.getSelectedIndex();
// Display selection, if item was selected. if(idx != -1) jlab.setText("Current selection: " + Cities[idx]); else // Otherwise, reprompt. jlab.setText("Choose a City"); } }); // Add the list and label to the content pane. add(jscrlp); add(jlab); } } OUTPUT 17.8
JTabbed

100% **MATCHING BLOCK 213/221** W

Panes A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When

the

97% **MATCHING BLOCK 214/221** W

user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options. Tabbed panes are encapsulated by the JTabbedPane class, which extends JComponent. We will use its default constructor. Tabs are defined via the following method: void addTab(String str, Component comp) Here, str is the title for the tab, and comp is the component that should be added to the tab. Typically, a JPanel or a subclass of it is added. The general procedure to use a tabbed pane in an applet is outlined here: 1. Create a JTabbedPane object. 2. Call addTab( ) to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.) 3. Repeat step 2 for each tab. 4. Add the tabbed pane to the content pane of the applet.

The following example illustrates how to create a tabbed pane. The first tab is titled Cities and contains four buttons. Each button displays the name of a city. The second tab is titled Colors and contains three check boxes. Each check box displays the name of a color. The third tab is titled Flavors and contains one combo box. This enables the user to select one of three flavors. PROGRAM

82% MATCHING BLOCK 215/221 W

import javax.swing.*; /* >applet code="JTabbedPaneDemo" width=400 height=100< >/applet< */ public class JTabbedPaneDemo extends JApplet { public void init() {

JTabbedPane jtp = new JTabbedPane(); jtp.addTab("Cities", new CitiesPanel()); jtp.addTab("Colors", new ColorsPanel()); jtp.addTab("Flavors", new FlavorsPanel()); getContentPane().add(jtp); } }
class CitiesPanel extends JPanel { public CitiesPanel() {
JButton b1 = new JButton("New York"); add(b1);
JButton b2 = new JButton("London"); add(b2); JButton b3 = new JButton("Hong Kong"); add(b3); JButton b4 = new JButton("
Tokyo"); add(b4); } }
class ColorsPanel extends JPanel { public ColorsPanel() { JCheckBox cb1 = new JCheckBox("Red"); add(cb1); JCheckBox cb2 = new JCheckBox("Green"); add(cb2); JCheckBox cb3 = new JCheckBox("Blue"); add(cb3); } } class FlavorsPanel extends JPanel { public FlavorsPanel() { JComboBox jcb = new JComboBox(); jcb.addItem("Vanilla"); jcb.addItem("Chocolate"); jcb.addItem("Strawberry"); add(jcb); } } OUTPUT
17.9
JScroll

100% MATCHING BLOCK 216/221 W

Panes A scroll pane is a component that presents a rectangular area in which a

component may

95% MATCHING BLOCK 217/221 W

be viewed. Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the JScrollPane class, which extends JComponent. Some of its constructors are shown here: JScrollPane(Component comp) JScrollPane(int vsb, int hsb) JScrollPane(Component comp, int vsb, int hsb) Here, comp is the component to be added to the scroll pane. vsb and hsb are

int constants

86% MATCHING BLOCK 218/221 W

that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the ScrollPaneConstants interface. Some examples of these constants are described as follows: Constant Description HORIZONTAL_SCROLLBAR_ALW AYS Always provide horizontal scroll bar HORIZONTAL_SCROLLBAR_AS_

N EEDED

96% MATCHING BLOCK 219/221 W

Provide horizontal scroll bar, if needed VERTICAL_SCROLLBAR_ALWAYS Always provide vertical scroll bar VERTICAL_SCROLLBAR_AS_NEE Provide vertical scroll bar, if needed DED Here are the steps that you should follow to use a scroll pane in an applet: 1. Create a JComponent object. 2. Create a JScrollPane object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.) 3. Add the scroll pane to the content pane of the applet. The following example illustrates a scroll pane. First, the content pane of the JApplet object is obtained and a border layout is assigned as its layout manager. Next, a JPanel object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. the scroll bars

can be used

94% **MATCHING BLOCK 220/221** W

to scroll the buttons into view. PROGRAM import java.awt.*; import javax.swing.*; /* >applet code="JScrollPaneDemo" width=300 height=250< >/applet< */ public class JScrollPaneDemo extends JApplet { public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new BorderLayout()); // Add 400 buttons to a panel JPanel jp = new JPanel(); jp.setLayout(new GridLayout(20, 20)); int b = 0; for(int i = 0; i > 20; i++) { for(int j = 0; j > 20; j++) { jp.add(new JButton("Button " + b)); ++b; } } // Add panel to a scroll pane int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED; int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_

NEEDE D;

92% **MATCHING BLOCK 221/221** W

JScrollPane jsp = new JScrollPane(jp, v, h); // Add scroll pane to the content pane contentPane.add(jsp, BorderLayout.CENTER); } }

OUTPUT
17.10 Unit Summary This Unit discussed the important controls used in the window based applications. It elaborated the JApplet, JFrame, JDialog, Text Fields, Buttons, Combo boxes, List, Tabbled and Scroll Panes. 17.11
Key Terms •
JComponent provides the functionality that is common to all components. •
JFrame is the top-level container that is commonly used for Swing applications. • JLabel is the Swing component that creates a label, which is a component that displays information. 17.12
Check Your Progress 1.
Explain how layout management is done with the help of JFrame and JDialog in Swing. 2. Write a simple Swing application to demonstrate the Text Fields, Buttons and Combo boxes. 3. Write
a
note on List control. 4. Explain Tabbed Panes with an example. 5. Demonstrate Scroll Panes through an example.

Hit and source - focused comparison, Side by Side

Submitted text As student entered the text in the submitted document.
Matching text As the text appears in the source.

1/221	SUBMITTED TEXT	11 WORDS	100%	MATCHING TEXT	11 WORDS
All computer programs consist of two elements: code and data.			all computer programs consist of two elements: code and data.		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

2/221	SUBMITTED TEXT	19 WORDS	100%	MATCHING TEXT	19 WORDS
This is a simple Java program. Call this file "Example.java". */			This is a simple Java program. Call this file "Example.java". */		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

3/221	SUBMITTED TEXT	40 WORDS	100%	MATCHING TEXT	40 WORDS

PROGRAM /* This is a simple Java program. Call this file "Example.java". */ class Example { // Your program begins with a call to main(). public static void main(String args[]) { System.out.println("This is a simple Java program."); } }

PROGRAM /* This is a simple Java program. Call this file "Example.java". */ class Example { // Your program begins with a call to main(). public static void main(String args[]) { System.out.println("This is a simple Java program."); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

4/221	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS

Java is designed for the distributed environment of the internet. ●

Java is designed for the distributed environment of the Internet,

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

5/221	SUBMITTED TEXT	93 WORDS	100%	MATCHING TEXT	93 WORDS

Types Java defines eight simple (or elemental) types of data: byte, short, int, long, char, float, double, and boolean. These can be put in four groups: ● Integers: This group includes byte, short, int, and long, which are for whole valued signed numbers. ● Floating-point numbers: This group includes float and double, which represent numbers with fractional precision. ● Characters: This group includes char, which represents symbols in a character set, like letters and numbers. ● Boolean: This group includes boolean, which is a special type for representing true/false values.

TYPES Java defines eight simple (or elemental) types of data: byte, short, int, long, char, float,double, and boolean. These can be put in four groups: ¾ Integers This group includes byte, short, int, and long, which are for whole valued signed numbers. ¾ Floating-point numbers This group includes float and double, which represent numbers with fractional precision. ¾ Characters This group includes char, which represents symbols in a character set, like letters and numbers. ¾ Boolean This group includes boolean, which is a special type for representing true/false values.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

6/221	SUBMITTED TEXT	45 WORDS	100%	MATCHING TEXT	45 WORDS

Variables The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Variables The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

7/221	SUBMITTED TEXT	61 WORDS	89%	MATCHING TEXT	61 WORDS

Declaring a Variable In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here: type identifier [= value][, identifier [= value] ...] ; The type is one of Java's atomic types, or the name of a class or interface. The identifier is the name of the variable.

Declaring a Variable In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here: type identifier [= value][, identifier [= value] ...] ; The type is one of Java's atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The identifier is the name of the variable.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

8/221	SUBMITTED TEXT	25 WORDS	100%	MATCHING TEXT	25 WORDS

int d = 3, e, f = 5; // declares three more ints, initializing d and f. byte z = 22; // initializes z.

int d = 3, e, f = 5; // declares three more ints, initializing // d and f. byte z = 22; // initializes z.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

9/221	SUBMITTED TEXT	22 WORDS	100%	MATCHING TEXT	22 WORDS

A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

a scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

10/221	SUBMITTED TEXT	33 WORDS	100%	MATCHING TEXT	33 WORDS

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

11/221	SUBMITTED TEXT	17 WORDS	96%	MATCHING TEXT	17 WORDS

It is common to assign a value of one type to a variable of another type.

it is fairly common to assign a value of one type to a variable of another type.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

12/221	SUBMITTED TEXT	77 WORDS	100%	MATCHING TEXT	77 WORDS

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met: ● The two types are compatible. ● The destination type is larger than the source type. When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met: ■ The two types are compatible. ■ The destination type is larger than the source type. When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

13/221	SUBMITTED TEXT	71 WORDS	95%	MATCHING TEXT	71 WORDS

It has this general form: (target-type) value Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range. For example, int a; byte b; // ... b = (byte) a; 1.3.4

It has this general form: (target-type) value Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range. int a; byte b; // ... b = (byte) a;

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

14/221	SUBMITTED TEXT	74 WORDS	92%	MATCHING TEXT	74 WORDS

Arrays An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information. One-Dimensional Arrays A one-dimensional array is, essentially, a list of like-typed variables. To create an array variable of the desired type

Arrays An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific elementin an array is accessed by its index. Arrays offer a convenient means of grouping related information. One-Dimensional Arrays A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

15/221	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS

Multidimensional Arrays In Java, multidimensional arrays are actually arrays of arrays.

Multidimensional Arrays In Java, multidimensional arrays are actually arrays of arrays.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

16/221	SUBMITTED TEXT	61 WORDS	100%	MATCHING TEXT	61 WORDS

To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD. int twoD[ ] [ ] = new int[4][5]; This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.

To declare a multidimensional array variable,specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD. int twoD[][] = new int[4][5]; This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int. //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

17/221	SUBMITTED TEXT	11 WORDS	100%	MATCHING TEXT	11 WORDS

All computer programs consist of two elements: code and data.

all computer programs consist of two elements: code and data.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

18/221	SUBMITTED TEXT	34 WORDS	100%	MATCHING TEXT	34 WORDS

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object oriented program can be characterized as data controlling access to code.

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

19/221	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS

Java is designed for the distributed environment of the internet. ●

Java is designed for the distributed environment of the Internet,

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

20/221	SUBMITTED TEXT	22 WORDS	100%	MATCHING TEXT	22 WORDS

Operators: Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table (

OPERATORS Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

21/221	SUBMITTED TEXT	23 WORDS	100%	MATCHING TEXT	23 WORDS

The Bitwise Operators Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

The Bitwise Operators Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

22/221	SUBMITTED TEXT	20 WORDS	100%	MATCHING TEXT	20 WORDS

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. •

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

23/221	SUBMITTED TEXT	20 WORDS	100%	MATCHING TEXT	20 WORDS

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

24/221	SUBMITTED TEXT	71 WORDS	100%	MATCHING TEXT	71 WORDS

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement: if (condition) statement1; else statement2; Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement: if (condition) statement1; else statement2; Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

25/221	SUBMITTED TEXT	18 WORDS	100%	MATCHING TEXT	18 WORDS

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

26/221	SUBMITTED TEXT	15 WORDS	96%	MATCHING TEXT	15 WORDS

if(condition) statement; else if(condition) statement; else if(condition) statement; ... else statement; The

if(condition) statement; else if(condition) statement; else if(condition) statement; ... else statement; switch The

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

27/221	SUBMITTED TEXT	27 WORDS	83%	MATCHING TEXT	27 WORDS

else statement will be executed. Switch The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of

else statement; switch The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

28/221	SUBMITTED TEXT	84 WORDS	94%	MATCHING TEXT	84 WORDS

switch (expression) { case value1: // statement sequence break; case value2: // statement sequence break; ... case valueN: // statement sequence break; default: // default statement sequence } The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

switch (expression) { case value1: // statement sequence Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in break; case value2: // statement sequence break; ... case valueN: // statement sequence break; default: // default statement sequence } The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

29/221	SUBMITTED TEXT	13 WORDS	100%	MATCHING TEXT	13 WORDS

Iteration Statements Java's iteration statements are for, while, and do-while. These statements

Iteration Statements Java's iteration statements are for, while, and do-while. These statements

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

30/221	SUBMITTED TEXT	19 WORDS	88%	MATCHING TEXT	19 WORDS

commonly call loops. A loop repeatedly executes the same set of instructions until a termination condition is met.

commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

31/221	SUBMITTED TEXT	26 WORDS	96%	MATCHING TEXT	26 WORDS

While The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Its general form

While The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

32/221	SUBMITTED TEXT	32 WORDS	100%	MATCHING TEXT	32 WORDS

while(condition) { // body of loop } The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When

While (condition) { // body of loop } The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

33/221	SUBMITTED TEXT	29 WORDS	100%	MATCHING TEXT	29 WORDS

condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

34/221	SUBMITTED TEXT	42 WORDS	100%	MATCHING TEXT	42 WORDS

do { // body of loop } while (condition); Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

do { // body of loop } while (condition); Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

35/221	SUBMITTED TEXT	78 WORDS	89%	MATCHING TEXT	78 WORDS

is a powerful and versatile construct. Here is the general form of the for statement: for(initialization; condition; iteration) { // body } If only one statement is being repeated, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed and it sets the value of the loop control variable, which acts as a counter that controls the loop. Next,

is a powerful and versatile construct. Here is the general form of the for statement: for(initialization; condition; iteration) { // body } If only one statement is being repeated, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loopcontrol variable, which acts as a counter that controls the loop.. Next,

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

36/221	SUBMITTED TEXT	65 WORDS	97%	MATCHING TEXT	65 WORDS

condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop

condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. GUAGE // Demonstrate the for loop.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

37/221	SUBMITTED TEXT	44 WORDS	100%	MATCHING TEXT	44 WORDS

In Java, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a —civilizedll form of goto.

In Java, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a —civilizedll form of goto.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

38/221	SUBMITTED TEXT	28 WORDS	70%	MATCHING TEXT	28 WORDS

Thus, a class is a template for an object, and an object is an instance of a class. The two words object and instance are used interchangeably

Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

39/221	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS

A class is declared by use of the class keyword. The

A class is declared by use of the class keyword. The

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

40/221	SUBMITTED TEXT	95 WORDS	100%	MATCHING TEXT	95 WORDS

The general form of a class definition is shown here:

The general form of a class definition is shown here:

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

41/221	SUBMITTED TEXT	66 WORDS	100%	MATCHING TEXT	66 WORDS

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

42/221	SUBMITTED TEXT	99 WORDS	88%	MATCHING TEXT	99 WORDS

objects of a class is done in a two-step process. First, declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, acquire an actual, physical copy of the object and assign it to that variable. This is achieved by using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.

objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

43/221	SUBMITTED TEXT	18 WORDS	68%	MATCHING TEXT	18 WORDS

This is the general form of a method: type name(parameter-list) { // body of method } Here,

This is the general form of a method declaration that includes a throws clause: type method-name(parameter-list) throws exception-list { // body of method } Here,

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

44/221	SUBMITTED TEXT	26 WORDS	76%	MATCHING TEXT	26 WORDS

Box { double width; double height; double depth; // compute and return volume double volume() { return width * height * depth; } //

Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

45/221	SUBMITTED TEXT	19 WORDS	100%	MATCHING TEXT	19 WORDS

double w, double h, double d) { width = w; height = h; depth = d; } }

double w, double h, double d) { width = w; height = h; depth = d; } //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

46/221	SUBMITTED TEXT	47 WORDS	71%	MATCHING TEXT	47 WORDS

class Box { double width; double height; double depth; // This is the constructor for Box. Box() { System.out.println("Constructing Box"); width = 10; height = 10; depth = 10; } // compute and return volume double volume() { return width * height * depth; } }

class Box { double width; double height; double depth; // This is the constructor for Box. Box(double w, double h, double d) { width = w; height = h; depth = d; } // compute and return volume double volume() { return width * height * depth; } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

47/221	SUBMITTED TEXT	50 WORDS	100%	MATCHING TEXT	50 WORDS

class Box { double width; double height; double depth; // This is the constructor for Box. Box(double w, double h, double d) { width = w; height = h; depth = d; } // compute and return volume double volume() { return width * height * depth; } }

class Box { double width; double height; double depth; // This is the constructor for Box. Box(double w, double h, double d) { width = w; height = h; depth = d; } // compute and return volume double volume() { return width * height * depth; } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

48/221	SUBMITTED TEXT	13 WORDS	100%	MATCHING TEXT	13 WORDS

can be used inside any method to refer to the current object.

can be used inside any method to refer to the current object.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

49/221	SUBMITTED TEXT	50 WORDS	89%	MATCHING TEXT	50 WORDS

consider the following version of Box( ): // A redundant use of this. Box(double w, double h, double d) { this.width = w; this.height = h; this.depth = d; } The use of this is redundant, but perfectly correct. Inside Box(), this will always refer to the invoking object.

consider the following version of Box( ): // A redundant use of this. Box(double w, double h, double d) { this.width = w; this.height = h; this.depth = This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

50/221	SUBMITTED TEXT	22 WORDS	97%	MATCHING TEXT	22 WORDS

However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. This

However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. // Use this

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

51/221	SUBMITTED TEXT	24 WORDS	89%	MATCHING TEXT	24 WORDS

Garbage Collection Objects are dynamically allocated by using the new operator, how such objects are destroyed, and their memory released for later reallocation

Garbage Collection Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

52/221	SUBMITTED TEXT	43 WORDS	97%	MATCHING TEXT	43 WORDS

handles deallocation automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

handles deallocation you automatically. The technique that accomplishes this is Called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

53/221	SUBMITTED TEXT	24 WORDS	89%	MATCHING TEXT	24 WORDS

Garbage Collection: Objects are dynamically allocated by using the new operator, how such objects are destroyed, and their memory released for later reallocation

Garbage Collection Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

54/221	SUBMITTED TEXT	70 WORDS	80%	MATCHING TEXT	70 WORDS

methods In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Here is a simple example that illustrates method overloading: PROGRAM // Demonstrate method overloading. class OverloadDemo { void

Methods In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java implements polymorphism. // Demonstrate overloading. class void

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

55/221	SUBMITTED TEXT	106 WORDS	87%	MATCHING TEXT	106 WORDS

void test() { System.out.println("No parameters"); } // Overload test for two integer parameters. void test(int a, int b) { System.out.println("a and b: " + a + " " + b); } // overload test for a double parameter double test(double a) { System.out.println("double a: " + a); return a*a; } } class Overload { public static void main(String args[]) { OverloadDemo ob = new OverloadDemo(); double result; // call all versions of test() ob.test(); ob.test(10, 20); result = ob.test(123.25); System.out.println("Result of ob.test(123.25): " + result); } } OUTPUT No parameters a and b: 10 20, double a: 123.25 Result of ob.test(123.25): 15190.5625

void test(a) { System.out.println("a: " + a); } // Overload test for two integer parameters. void test(int a, int b) { System.out.println("a and b: " + a + " " + b); } // overload test for a double parameter double test(double a) { System.out.println("double a: " + a); a*a; Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in } } class Overload { public static void main(String args[]) { OverloadDemo ob = new OverloadDemo(); double result; // call all versions of test() ob.test(); ob.test(10); ob.test(10, 20); result = ob.test(123.25); System.out.println("Result of ob.test(123.25): " + result); } } This program generates the following output: No parameters a: 10 a and b: 10 20 double a: 123.25 Result of ob.test(123.25): 15190.5625

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

56/221	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS

Method overloading is one of the ways that Java implements polymorphism.

Method overloading is one of the ways that Java implements polymorphism. //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

57/221	SUBMITTED TEXT	22 WORDS	76%	MATCHING TEXT	22 WORDS

class Overload { public static void main(String args[]) { OverloadDemo ob = new OverloadDemo(); int i = 88; ob.test(); ob.test(10, 20); ob.test(

class Overload { public static void main(String args[]) { OverloadDemo ob = new OverloadDemo(); double result; // call all versions of test() ob.test(); ob.test(10); ob.test(10, 20);

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

58/221	SUBMITTED TEXT	91 WORDS	100%	MATCHING TEXT	91 WORDS

depth; // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } }

depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

59/221	SUBMITTED TEXT	62 WORDS	42%	MATCHING TEXT	62 WORDS

vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); // get volume of second box vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); // get volume of cube vol = mycube.volume(); System.out.println("Volume of mycube is " + vol); } } OUTPUT Volume of mybox1 is 3000.0 Volume of mybox2 is -1.0 Volume of mycube is 343.0

vol = mybox3.volume(); System.out.println("Volume of mybox3 is " + vol); System.out.println("Weight of mybox3 is " + vol = myclone.volume(); System.out.println("Volume of myclone is " + vol); System.out.println("Weight of myclone is " + myclone.weight); System.out.println(); vol = mycube.volume(); System.out.println("Volume mycube is " + System.out.println("Weight of mycube is " + mycube.weight); System.out.println(); } } Output: Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume of mybox2 is 24.0

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

60/221	SUBMITTED TEXT	16 WORDS	87%	MATCHING TEXT	16 WORDS

class PassOb { public static void main(String args[]) { Test ob1 = new Test(100, 22);

class CallByValue { public static void main(String args[]) { Test ob = new Test();

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

61/221	SUBMITTED TEXT	120 WORDS	98%	MATCHING TEXT	120 WORDS

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. Java uses both approaches.

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches,

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

62/221	SUBMITTED TEXT	103 WORDS	93%	MATCHING TEXT	103 WORDS

For example, consider the following program: PROGRAM // Simple types are passed by value. class Test { void meth (int i, int j) { i *= 2; j /= 2; } } class CallByValue { public static void main (String args[]) { Test ob = new Test (); int a = 15, b = 20; System.out.println("a and b before call: " + a + " " + b); ob.meth(a, b); System.out.println("a and b after call: " + a + " " + b); } } OUTPUT a and b before call: 15 20 a and b after call: 15 20

For example, consider the following program: // Simple types are passed by value. class Test { void meth(int i, int j) { i *= 2; j /= 2; } } class CallByValue { public static void main(String args[]) { Test ob = new Test(); int a = 15, b = 20; System.out.println("a and b before call: " + a + " " + b); ob.meth(a, b); System.out.println("a and after call: " + a + " " + b); } } The output here: a and b before call: 15 20 a and b after call: 15 20

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

63/221	SUBMITTED TEXT	32 WORDS	57%	MATCHING TEXT	32 WORDS

class CallByRef { public static void main(String args[]) { Test ob = new Test(15, 20); System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b); ob.meth(ob); System.out.println("ob.a and

class public static void main(String args[]) { Test ob = new Test(); int println("a and b before call: " + a + " " + b); ob.meth(a, b); System.out.println("a and

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

64/221	SUBMITTED TEXT	27 WORDS	46%	MATCHING TEXT	27 WORDS

class RetOb { public static void main(String args[]) { Test ob1 = new Test(2); Test ob2; ob2 = ob1.incrByTen(); System.out.println("ob1.a: " + ob1.a); System.out.println("ob2.a: " +

class CallByValue { public static void main(String args[]) { Test ob = new Test(); int a = 15, System.out.println("a and b before call: " + a + " " + b); ob.meth(a, b); System.out.println(" a

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

65/221	SUBMITTED TEXT	75 WORDS	100%	MATCHING TEXT	75 WORDS

Recursion Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive. The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

Recursion Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

66/221	SUBMITTED TEXT	72 WORDS	91%	MATCHING TEXT	72 WORDS

A simple example of recursion. class Factorial { // this is a recursive function int fact(int n) { int result; if(n==1) return 1; result = fact(n-1) * n; return result; } } class Recursion { public static void main(String args[]) { Factorial f = new Factorial(); System.out.println("Factorial of 4 is " + f.fact(4)); } } OUTPUT Factorial of 4 is 24 When fact( )

A simple example of recursion(factorial). class Factorial { // this is a recursive function int fact(int n) { int result; if(n==1) return 1; result = fact(n-1) * n; return result; } } class Recursion { public static void main(String args[]) { Factorial f = new Factorial(); System.out.println("Factorial of 3 is " + f.fact(3)); System.out.println("Factorial of 4 is " + f.fact(4));

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

67/221	SUBMITTED TEXT	40 WORDS	100%	MATCHING TEXT	40 WORDS

Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later.

Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

68/221	SUBMITTED TEXT	18 WORDS	100%	MATCHING TEXT	18 WORDS

How a member can be accessed is determined by the access specifier that modifies its declaration. Java'

How a member can be accessed is determined by the access specifier that modifies its declaration. Java

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

69/221	SUBMITTED TEXT	23 WORDS	100%	MATCHING TEXT	23 WORDS

Java's access specifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved.

Java's access specifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

70/221	SUBMITTED TEXT	46 WORDS	93%	MATCHING TEXT	46 WORDS

When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class.

When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed byother members of its class.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

71/221	SUBMITTED TEXT	29 WORDS	100%	MATCHING TEXT	29 WORDS

When no access specifier is used, then by default the member of a class is public within its own package but cannot be accessed outside of its package.

When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

72/221	SUBMITTED TEXT	16 WORDS	87%	MATCHING TEXT	16 WORDS

class AccessTest { public static void main(String args[]) { Test ob = new Test(); //

class CallByValue { public static void main(String args[]) { Test ob = new Test();

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

73/221	SUBMITTED TEXT	11 WORDS	100%	MATCHING TEXT	11 WORDS

It has access to all of the variables and methods

it has access to all of the variables and methods

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

74/221	SUBMITTED TEXT	39 WORDS	100%	MATCHING TEXT	39 WORDS

Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

75/221	SUBMITTED TEXT	124 WORDS	98%	MATCHING TEXT	124 WORDS

class Box { double width; double height; double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } //

class Box { private double width; private double height; private double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

76/221	SUBMITTED TEXT	22 WORDS	66%	MATCHING TEXT	22 WORDS

constructor for BoxWeight BoxWeight(double w, double h, double d, double m) { width = w; height = h; depth = d;

constructor for Box. Box(double w, double h, double d) { width = w; height = h; depth = d; } //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

77/221	SUBMITTED TEXT	95 WORDS	71%	MATCHING TEXT	95 WORDS

weight = m; } } class DemoBoxWeight { public static void main(String args[]) { BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3); BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076); double vol; vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); System.out.println("Weight of mybox1 is " + mybox1.weight); System.out.println(); vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); System.out.println("Weight of mybox2 is " + mybox2.weight); } } OUTPUT Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume of mybox2 is 24.0 Weight of mybox2 is 0.076

weight = m; } } class DemoSuper { public static void main(String args[]) { BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3); BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076); BoxWeight mybox3 = new BoxWeight(); // default BoxWeight mycube = new BoxWeight(3, 2); BoxWeight myclone = new BoxWeight(mybox1); double vol; vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); System.out.println("Weight of mybox1 is " + mybox1.weight); System.out.println(); vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); System.out.println("Weight of mybox2 is " + mybox2.weight); System.out.println(); vol = mybox3. volume(); System.out.println("Volume of mybox3 is " + vol); System.out.Weight of is " +

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

78/221	SUBMITTED TEXT	18 WORDS	58%	MATCHING TEXT	18 WORDS

needs to refer to its immediate superclass can use the keyword super. super has two general forms.

needs to refer to its immediate super class, it can do so by use of the key word super. o Super has two general forms:

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

79/221	SUBMITTED TEXT	57 WORDS	73%	MATCHING TEXT	57 WORDS

used to access a member of the superclass that has been hidden by a member of a subclass. Using super to Call Superclass Constructors A subclass can call a constructor method defined by its superclass by use of the following form of super: super(parameter-list); Here, parameter-list specifies any parameters needed by the constructor in the

Used to access a member of the super class that has been hidden by a member of a sub class Using super to call super class constructor o A sub class can call a constructor defined by its super class by use of the following form of super: o super (parameter-list); o Parameter list specifies parameters needed by the constructor in the

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

80/221	SUBMITTED TEXT	415 WORDS	97%	MATCHING TEXT	415 WORDS

A complete implementation of BoxWeight. class Box { private double width; private double height; private double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } // BoxWeight now fully implements all constructors. class BoxWeight extends Box { double weight; // weight of box // construct clone of an object BoxWeight(BoxWeight ob) { // pass object to constructor super(ob); weight = ob.weight; } // constructor when all parameters are specified BoxWeight(double w, double h, double d, double m) { super(w, h, d); // call superclass constructor weight = m; } // default constructor BoxWeight() { super(); weight = -1; } // constructor used when cube is created BoxWeight(double len, double m) { super(len); weight = m; } } class DemoSuper { public static void main(String args[]) { BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3); BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076); BoxWeight mybox3 = new BoxWeight(); // default BoxWeight mycube = new BoxWeight(3, 2); BoxWeight myclone = new BoxWeight(mybox1); double vol; vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); System.out.println("Weight of mybox1 is " + mybox1.weight); System.out.println(); vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); System.out.println("Weight of mybox2 is " + mybox2.weight); System.out.println(); vol = mybox3.volume(); System.out.println("Volume of mybox3 is " + vol); System.out.println("Weight of mybox3 is " + mybox3.weight); System.out.println(); vol = myclone.volume(); System.out.println("Volume of myclone is " + vol); System.out.println("Weight of myclone is " + myclone.weight); System.out.println(); vol = mycube.volume(); System.out.println("Volume of mycube is " + vol); System.out.println("Weight of mycube is " + mycube.weight); System.out.println(); } } OUTPUT Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume of mybox2 is 24.0 Weight of mybox2 is 0.076 Volume of mybox3 is -1.0 Weight of mybox3 is -1.0 Volume of myclone is 3000.0 Weight of myclone is 34.3 Volume of mycube is 27.0 Weight of mycube is 2.0

A complete implementation of BoxWeight. class Box { private double width; private double height; private double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } // BoxWeight now fully implements all constructors. class BoxWeight extends Box { double weight; // weight of box // construct clone of an object BoxWeight(BoxWeight ob) { // pass object to constructor super(ob); weight = ob.weight; } // constructor when all parameters are specified BoxWeight(double w, double h, double d, double m) { super(w, h, d); // call superclass constructor weight = m; } // default constructor Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in BoxWeight() { super(); weight = -1; } // constructor used when cube is created BoxWeight(double len, double m) { super(len); weight = m; } } class DemoSuper { public static void main(String args[]) { BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3); BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076); BoxWeight mybox3 = new BoxWeight(); // default BoxWeight mycube = new BoxWeight(3, 2); BoxWeight myclone = new BoxWeight(mybox1); double vol; vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); System.out.println("Weight of mybox1 is " + mybox1.weight); System.out.println(); vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); System.out.println("Weight of mybox2 is " + mybox2.weight); System.out.println(); vol = mybox3.volume(); System.out.println("Volume of mybox3 is " + vol); System.out.println("Weight of mybox3 is " + mybox3.weight); System.out.println(); vol = myclone.volume(); System.out.println("Volume of myclone is " + vol); System.out.println("Weight of myclone is " + myclone.weight); System.out.println(); vol = mycube.volume(); System.out.println("Volume of mycube is " + vol); System.out.println("Weight of mycube is " + mycube.weight); System.out.println(); } } Output: Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume of mybox2 is 24.0 Weight of mybox2 is 0.076 Volume of mybox3 is -1.0 Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in Weight of mybox3 is -1.0 Volume of myclone is 3000.0 Weight of myclone is 34.3 Volume of mycube is 27.0 Weight of mycube is 2.0

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

81/221	SUBMITTED TEXT	38 WORDS	55%	MATCHING TEXT	38 WORDS

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form: super.member Here, member can be

The second form of super acts somewhat like this keyword, except that it always refers to the super class of the sub class in which it is used. o The syntax is: o Super.member ; o Member can either be

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

82/221	SUBMITTED TEXT	103 WORDS	97%	MATCHING TEXT	103 WORDS

Using super to overcome name hiding. class A { int i; } // Create a subclass by extending class A. class B extends A { int i; // this i hides the i in A B(int a, int b) { super.i = a; // i in A i = b; // i in B } void show() { System.out.println("i in superclass: " + super.i); System.out.println("i in subclass: " + i); } } class UseSuper { public static void main(String args[]) { B subOb = new B(1, 2); subOb.show(); } } OUTPUT i in superclass: 1 i in subclass: 2 Although the

Using super to overcome name hiding. class A { int i; } // Create a subclass by extending class A. class B extends A { int i; // this i hides the i in A B(int a, int b) { super.i = a; // i in A i = b; // i in B } void show() { System.out.println("i in superclass: " + super.i); System.out.println("i in subclass: " + i); } } class UseSuper { public static void main(String args[]) { B subOb = new B(1, 2); subOb.show(); } } Output: i in superclass: 1 i in subclass: 2 When the

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

83/221	SUBMITTED TEXT	214 WORDS	94%	MATCHING TEXT	214 WORDS

class Box { private double width; private double height; private double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } // Add weight. class BoxWeight extends Box { double weight; // weight of box // construct clone of an object BoxWeight(BoxWeight ob) { // pass object to constructor super(ob); weight = ob.weight; } // constructor when all parameters are specified BoxWeight(double w, double h, double d, double m) { super(w, h, d); // call superclass constructor weight = m; } // default constructor BoxWeight() { super(); weight = -1; } // constructor used when cube is created BoxWeight(double len, double m) { super(len); weight = m; } } //

class Box { private double width; private double height; private double depth; // construct clone of an object Box(Box ob) { // pass object to constructor width = ob.width; height = ob.height; depth = ob.depth; } // constructor used when all dimensions specified Box(double w, double h, double d) { width = w; height = h; depth = d; } // constructor used when no dimensions specified Box() { width = -1; // use -1 to indicate height = -1; // an uninitialized depth = -1; // box } // constructor used when cube is created Box(double len) { width = height = depth = len; } // compute and return volume double volume() { return width * height * depth; } } // BoxWeight now fully implements all constructors. class BoxWeight extends Box { double weight; // weight of box // construct clone of an object BoxWeight(BoxWeight ob) { // pass object to constructor super(ob); weight = ob.weight; } // constructor when all parameters are specified BoxWeight(double w, double h, double d, double m) { super(w, h, d); // call superclass constructor weight = m; } // default constructor Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in BoxWeight() { super(); weight = -1; } // constructor used when cube is created BoxWeight(double len, double m) { super(len); weight = m; } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

84/221	SUBMITTED TEXT	76 WORDS	58%	MATCHING TEXT	76 WORDS

construct clone of an object Shipment(Shipment ob) { // pass object to constructor super(ob); cost = ob.cost; } // constructor when all parameters are specified Shipment(double w, double h, double d, double m, double c) { super(w, h, d, m); // call superclass constructor cost = c; } // default constructor Shipment() { super(); cost = -1; } // constructor used when cube is created Shipment(double len, double m, double c) { super(len, m);

construct clone of an object BoxWeight(BoxWeight ob) { // pass object to constructor super(ob); weight = ob.weight; } // constructor when all parameters are specified BoxWeight(double w, double h, double d, double m) { super(w, h, d); // call superclass constructor weight = m; } // default constructor Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in BoxWeight() { super(); weight = -1; } // constructor used when cube is created BoxWeight(double len, double m) { super(len); weight = m; } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

85/221	SUBMITTED TEXT	74 WORDS	50%	MATCHING TEXT	74 WORDS

vol = shipment1.volume(); System.out.println("Volume of shipment1 is " + vol); System.out.println("Weight of shipment1 is " + shipment1.weight); System.out.println("Shipping cost: $" + shipment1.cost); System.out.println(); vol = shipment2.volume(); System.out.println("Volume of shipment2 is " + vol); System.out.println("Weight of shipment2 is " + shipment2.weight); System.out.println("Shipping cost: $" + shipment2.cost); } } OUTPUT Volume of shipment1 is 3000.0 Weight of shipment1 is 10.0 Shipping cost: $3.41 Volume of shipment2 is 24.0 Weight of shipment2 is 0.76

vol = myclone.volume(); System.out.println("Volume of myclone is " + vol); System.out.println("Weight of myclone is " + myclone.weight); System.out.println(); vol = mycube.volume(); System.out.println("Volume of mycube is " + vol); System.out.println("Weight of mycube is " + mycube.weight); System.out.println(); } } Output: Volume of mybox1 is 3000.0 Weight of mybox1 is 34.3 Volume of mybox2 is 24.0 Weight of mybox2 is 0.076

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

86/221	SUBMITTED TEXT	64 WORDS	79%	MATCHING TEXT	64 WORDS

Method Overriding In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the

Method overriding: x In a class hierarchy, when a method in a sub class has the same name and type signature as a method in its super class, then the method in the sub class is said to be override the method in the sub x When an overridden method is called from within a sub class, it will always refers to the version of that method defined by the

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

87/221	SUBMITTED TEXT	111 WORDS	100%	MATCHING TEXT	111 WORDS

Method overriding. class A { int i, j; A(int a, int b) { i = a; j = b; } // display i and j void show() { System.out.println("i and j: " + i + " " + j); } } class B extends A { int k; B(int a, int b, int c) { super(a, b); k = c; } // display k – this overrides show() in A void show() { System.out.println("k: " + k); } } class Override { public static void main(String args[]) { B subOb = new B(1, 2, 3); subOb.show(); // this calls show() in B } } OUTPUT k: 3

Method overriding. class A { int i, j; A(int a, int b) { i = a; j = b; } // display i and j void show() { System.out.println("i and j: " + i + " " + j); } } class extends A { int k; B(int a, int b, int c) { super(a, b); k = c; } // display k – this overrides show() in A void show() { System.out.println("k: " + k); } } class Override { public static void main(String args[]) { B subOb = new B(1, 2, 3); subOb.show(); // this calls show() in B } } Output: k: 3

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

88/221	SUBMITTED TEXT	39 WORDS	70%	MATCHING TEXT	39 WORDS

class B extends A { int k; B(int a, int b, int c) { super(a, b); k = c; } void show() { super.show(); // this calls A's show() System.out.println("k: " + k); } }

class B extends A { int k; B(int a, int b, int c) { super(a, b); k = c; } // display this overrides show() in A void show() { System.out.println("k: " + k); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

89/221	SUBMITTED TEXT	19 WORDS	91%	MATCHING TEXT	19 WORDS

mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

mechanism by which a call to an overridden method is resolved at run time rather then compile time.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

90/221	SUBMITTED TEXT	167 WORDS	100%	MATCHING TEXT	167 WORDS

Dynamic Method Dispatch class A { void callme() { System.out.println("Inside A's callme method"); } } class B extends A { // override callme() void callme() { System.out.println("Inside B's callme method"); } } class C extends A { // override callme() void callme() { System.out.println("Inside C's callme method"); } } class Dispatch { public static void main(String args[]) { A a = new A(); // object of type A B b = new B(); // object of type B C c = new C(); // object of type C A r; // obtain a reference of type A r = a; // r refers to an A object r.callme(); // calls A's version of callme r = b; // r refers to a B object r.callme(); // calls B's version of callme r = c; // r refers to a C object r.callme(); // calls C's version of callme } } OUTPUT Inside A's callme method Inside B's callme method Inside C's callme method

Dynamic Method Dispatch class A { void callme() { System.out.println("Inside A's callme method"); } } class B extends A { // override callme() void callme() { System.out.println("Inside B's callme method"); } } class C extends A { // override callme() void callme() { System.out.println("Inside C's callme method"); } } class Dispatch { public static void main(String args[]) { A a = new A(); // object of type A B b = new B(); // object of type B C c = new C(); // object of type C A r; // obtain a reference of type A r = a; // r refers to an A object r.callme(); // calls A's version of callme r = b; // r refers to a B object r.callme(); // calls B's version of callme r = c; // r refers to a C object r.callme(); // calls C's version of callme } } Output: Inside A's callme method Inside B's callme method Inside C's callme method

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

91/221	SUBMITTED TEXT	164 WORDS	97%	MATCHING TEXT	164 WORDS

Using abstract methods and classes. abstract class Figure { double dim1; double dim2; Figure(double a, double b) { dim1 = a; dim2 = b; } // area is now an abstract method abstract double area(); } class Rectangle extends Figure { Rectangle(double a, double b) { super(a, b); } // override area for rectangle double area() { System.out.println("Inside Area for Rectangle."); return dim1 * dim2; } } class Triangle extends Figure { Triangle(double a, double b) { super(a, b); } // override area for right triangle double area() { System.out.println("Inside Area for Triangle."); return dim1 * dim2 / 2; } } class AbstractAreas { public static void main(String args[]) { // Figure f = new Figure(10, 10); // illegal now Rectangle r = new Rectangle(9, 5); Triangle t = new Triangle(10, 8); Figure figref; // this is OK, no object is created figref = r; System.out.println("Area is " + figref.area()); figref = t; System.out.println("Area is " + figref.area()); } }

Using abstract methods and classes. abstract class Figure { double dim1; double dim2; Figure(double a, double b) { dim1 = a; dim2 = b; } Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in class Rectangle extends Figure { Rectangle(double a, double b) { super(a, b); } // override area for rectangle double area() { System.out.println("Inside Area for Rectangle."); return dim1 * dim2; } } class Triangle extends Figure { Triangle(double a, double b) { super(a, b); } // override area for right triangle double area() { System.out.println("Inside Area for Triangle."); return dim1 * dim2 / 2; } } class AbstractAreas { public static void main(String args[]) { // Figure f = new Figure(10, 10); // illegal now Rectangle r = new Rectangle(9, 5); Triangle t = new Triangle(10, 8); Figure figref; // this is OK, no object is created figref = r; System.out.println("Area is " + figref.area()); figref = t; System.out.println("Area is " + figref.area()); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

92/221	SUBMITTED TEXT	35 WORDS	92%	MATCHING TEXT	35 WORDS

to prevent overriding To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

93/221	SUBMITTED TEXT	31 WORDS	100%	MATCHING TEXT	31 WORDS

class A { final void meth() { System.out.println("This is a final method."); } } class B extends A { void meth() { // ERROR! Can't override. System.out.println("Illegal!"); } }

class A { final void meth() { System.out.println("This is a final method."); } } class B extends A { void meth() { // ERROR! Can't override. System.out.println("Illegal!"); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

94/221	SUBMITTED TEXT	107 WORDS	95%	MATCHING TEXT	107 WORDS

Using final to Prevent Inheritance To prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declared all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. Here is an example of a final class: final class A { // ... } // The following class is illegal. class B extends A { // ERROR! Can't subclass A // ... } 6.9 The

Using final to Prevent Inheritance: Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. Here is an example of a final class: final class A { // ... } // The following class is illegal. class B extends A { // ERROR! Can't subclass A // ... } As the

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

95/221	SUBMITTED TEXT	20 WORDS	91%	MATCHING TEXT	20 WORDS

mechanism by which a call to an overridden method is resolved at run time, rather than compile time. 6.12

mechanism by which a call to an overridden method is resolved at run time rather then compile time.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

96/221	SUBMITTED TEXT	14 WORDS	100%	MATCHING TEXT	14 WORDS

Packages and interfaces are two of the basic components of a Java program.

Packages and Interfaces are two of the basic components of a java program.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

97/221	SUBMITTED TEXT	42 WORDS	90%	MATCHING TEXT	42 WORDS

any (or all) of the following four internal parts: • A single package statement (optional) • Any number of import statements (optional) • A single public class declaration (required) • Any number of classes private to the package (optional) 7.1

any (or all) of the following internal parts: o A single package statement (optional) o Any number of import statements (optional) o A single public class declaration (required) o Any number of classes private to the package (optional)

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

98/221	SUBMITTED TEXT	13 WORDS	100%	MATCHING TEXT	13 WORDS

Any classes declared within that file will belong to the specified package.

Any classes declared within that file will belong to the specified package.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

99/221	SUBMITTED TEXT	19 WORDS	76%	MATCHING TEXT	19 WORDS

This is the general form of the package statement: package pkg; Here, pkg is the name of

This is the general form of the import statement: import pkg1[.pkg2].(classname|*); Here, pkg1 is the name of

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

100/221	SUBMITTED TEXT	21 WORDS	92%	MATCHING TEXT	21 WORDS

two ways. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if

two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

101/221	SUBMITTED TEXT	22 WORDS	92%	MATCHING TEXT	22 WORDS

package is in the current directory, or a subdirectory of the current directory, it will be found. Second, a directory path

package is in the current directory, or a subdirectory of the current directory, it will be found. Second, you can specify a directory path

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

102/221	SUBMITTED TEXT	50 WORDS	100%	MATCHING TEXT	50 WORDS

For example, consider the following package specification. package MyPack; In order for a program to find MyPack, one of two things must be true. Either the program is executed from a directory immediately above MyPack, or CLASSPATH must be set to include the path to MyPack.

For example, consider the following package specification. package MyPack; In order for a program to find MyPack, one of two things must be true. Either the program is executed from a directory immediately above MyPack, or CLASSPATH must be set to include the path to MyPack.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

103/221	SUBMITTED TEXT	26 WORDS	50%	MATCHING TEXT	26 WORDS

b; } void show() { if(bal>0) System.out.print("--< "); System.out.println(name + ": $" + bal); } } class AccountBalance { public static void main(String args[]) {

B } void show() { System.out.println("i in superclass: " + super.i); System.out.println("i in subclass: " + i); } } class UseSuper { public static void main(String args[]) {

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

104/221	SUBMITTED TEXT	34 WORDS	72%	MATCHING TEXT	34 WORDS

both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.

both means of encapsulating and containing the name space and scope of variables and methods. x Packages acts as a containers for classes and other sub – ordinate packages. Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in x Classes act as containers for data and code.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

105/221	SUBMITTED TEXT	39 WORDS	65%	MATCHING TEXT	39 WORDS

Java addresses four categories of visibility for class members: • Subclasses in the same package • Non-subclasses in the same package • Subclasses in different packages • Classes that are neither in the same package nor subclasses The

Java address four categories of visibility for class members: o Sub – classes in the same package. o Non – sub class in the same package. o Sub – classes in the different package. o Classes that are neither in the same package nor subclasses. x The 3

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

106/221	SUBMITTED TEXT	23 WORDS	100%	MATCHING TEXT	23 WORDS

access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.

access specifiers private, public and protected provide a variety of ways to produce the many levels of access required by these categories.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

107/221	SUBMITTED TEXT	42 WORDS	80%	MATCHING TEXT	42 WORDS

Private No modifier Protected Public Same class Yes Yes Yes Yes Same package Subclass No Yes Yes Yes Same package non- subclass No Yes Yes Yes Different package subclass No No Yes Yes Different package non- subclass No No No Yes

Private No modifier Protected Public Same class Yes Yes Yes Yes Same package sub class No Yes Yes Yes Same package non – sub class No Yes Yes Yes Different package sub class No No Yes Yes Different package non sub class No No No Yes

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

108/221	SUBMITTED TEXT	48 WORDS	100%	MATCHING TEXT	48 WORDS

Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.

Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

109/221	SUBMITTED TEXT	115 WORDS	99%	MATCHING TEXT	115 WORDS

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement: import pkg1[.pkg2].(classname|*); Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use: import java.util.Date; import java.io.*;

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This the general form of the import statement: import pkg1[.pkg2].(classname|*); Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use: import java.util.Date; import java.io.*;

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

110/221	SUBMITTED TEXT	141 WORDS	100%	MATCHING TEXT	141 WORDS

Implementing Interfaces Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this: access class classname [extends superclass] [implements interface [,interface...]] { // class-body } Here, access is either public or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

Implementing Interfaces: Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this: access class classname [extends superclass] [implements interface [,interface...]] { // class-body } Here, access is either public or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

111/221	SUBMITTED TEXT	32 WORDS	100%	MATCHING TEXT	32 WORDS

the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. 8.4

the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

112/221	SUBMITTED TEXT	71 WORDS	77%	MATCHING TEXT	71 WORDS

catch (ArithmeticException e) { System.out.println("Exceptio: " + e); a = 0; // set a to zero and continue } When this version is substituted in the program, the following message will be displayed. Exception: java.lang.ArithmeticException: / by zero Multiple catch Clauses In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, two or more catch clauses

catch (ArithmeticException e) { System.out.println("Exception: " + e); a = 0; // set a to zero and continue } When this version is substituted in the program, and the program is run, each divide-by- zero error displays the following message: – Exception: java.lang.ArithmeticException: / by zero Multiple Catch Blocks In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses,

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

113/221	SUBMITTED TEXT	58 WORDS	100%	MATCHING TEXT	58 WORDS

each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types: //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

114/221	SUBMITTED TEXT	89 WORDS	94%	MATCHING TEXT	89 WORDS

Demonstrate multiple catch statements. class MultiCatch { public static void main(String args[]) { try { int a = args.length; System.out.println("a = " + a); int b = 42 / a; int c[] = { 1 }; c[42] = 99; } catch(ArithmeticException e) { System.out.println("Divide by 0: " + e); } catch(ArrayIndexOutOfBoundsException e) { System.out.println("Array index oob: " + e); } System.out.println("After try/catch blocks."); } } This program will cause a division-by-zero exception if it is started with no command line parameters, since a will be equal

Demonstrate multiple catch class MultiCatch { public static void main(String args[]) { try { int a = args.length; System.out.println("a = " + a); int b = 42 / a; int c[] = { 1 }; c[42] = 99; } catch(ArithmeticException e) { System.out.println("Divide by 0: " + e); } catch(ArrayIndexOutOfBoundsException e) { System.out.println("Array index oob: " + e); } System.out.println("After try/catch blocks."); } } This program will cause a division-by-zero exception if it is started with no commandline parameters, since a will equal

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

115/221	SUBMITTED TEXT	88 WORDS	95%	MATCHING TEXT	88 WORDS

zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42]. Here is the output generated by running it both ways: C:\<java MultiCatch a = 0 Divide by 0: java.lang.ArithmeticException: / by zero After try/catch blocks. C:\<java MultiCatch TestArg a = 1 Array index oob: java.lang.ArrayIndexOutOfBoundsException After try/catch blocks.

zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42]. Here is the output generated by running it both ways: C:\<java MultiCatch a = 0 Divide by 0: java.lang.ArithmeticException: / by zero After try/catch blocks. C:\<java MultiCatch TestArg a = 1 Array index oob: java.lang.ArrayIndexOutOfBoundsException After try/catch blocks.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

116/221	SUBMITTED TEXT	18 WORDS	89%	MATCHING TEXT	18 WORDS

statements. class NestTry { public static void main(String args[]) { try { int a = args.length; /*

statements. class MultiCatch { public static void main(String args[]) { try { int a = args.length;

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

117/221	SUBMITTED TEXT	23 WORDS	52%	MATCHING TEXT	23 WORDS

C:\<java NestTry One a = 1 Divide by 0: java.lang.ArithmeticException: / by zero C:\<java NestTry One Two a = 2 Array index

C:\<java MultiCatch a = 0 Divide by 0: java.lang.ArithmeticException: / by zero After try/catch blocks. C:\<java MultiCatch TestArg a = 1 Array index

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

118/221	SUBMITTED TEXT	42 WORDS	97%	MATCHING TEXT	42 WORDS

throw ThrowableInstance; Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. Two ways

Throw throwableInstance Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions There are two ways

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

119/221	SUBMITTED TEXT	102 WORDS	98%	MATCHING TEXT	102 WORDS

obtain a Throwable object: using a parameter into a catch clause, or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

obtain a Throwable object: – using a parameter into a catch clause – creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

120/221	SUBMITTED TEXT	51 WORDS	100%	MATCHING TEXT	51 WORDS

Demonstrate throw. class ThrowDemo { static void demoproc() { try { throw new NullPointerException("demo"); } catch(NullPointerException e) { System.out.println("Caught inside demoproc."); throw e; // rethrow the exception } } public static void main(String args[]) { try { demoproc(); } catch(NullPointerException e) { System.out.println("Recaught: " + e); } } }

Demonstrate throw. class ThrowDemo { static void demoproc() { try { throw new NullPointerException("demo"); } catch(NullPointerException e) { System.out.println("Caught inside demoproc."); throw e; // rethrow the exception } } public static void main(String args[]) { try { demoproc(); } catch(NullPointerException e) { System.out.println("Recaught: " + e);

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

121/221	SUBMITTED TEXT	140 WORDS	99%	MATCHING TEXT	140 WORDS

throws clause If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause: type method-name(parameter-list) throws exception-list { // body of method } Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Throws Keyword If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause: type method-name(parameter-list) throws exception-list { // body of method } Here, exception-list is a comma-separated list of the exceptions that a method can throw

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

122/221	SUBMITTED TEXT	103 WORDS	88%	MATCHING TEXT	103 WORDS

class ThrowsDemo { static void throwOne() throws IllegalAccessException { System.out.println("Inside throwOne."); throw new IllegalAccessException("demo"); } public static void main(String args[]) { try { throwOne(); } catch (IllegalAccessException e) { System.out.println("Caught " + e); } } } OUTPUT inside throwOne caught java.lang.IllegalAccessException:

class ThrowsDemo { static void throwOne() throws IllegalAccessException { System.out.println("Inside throwOne."); throw new IllegalAccessException("demo"); } public static void main(String args[]) { try { throwOne(); } catch (IllegalAccessException e) { System.out.println("Caught " + e); } } } Here is output generated by running this example program: inside throwOne caught java.lang.IllegalAccessException

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

123/221	SUBMITTED TEXT	142 WORDS	97%	MATCHING TEXT	142 WORDS

creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause. //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

124/221	SUBMITTED TEXT	139 WORDS	99%	MATCHING TEXT	139 WORDS

Demonstrate finally. class FinallyDemo { // Through an exception out of the method. static void procA() { try { System.out.println("inside procA"); throw new RuntimeException("demo"); } finally { System.out.println("procA' finally"); } } // Return from within a try block. static void procB() { try { System.out.println("inside procB"); return; } finally { System.out.println("procB's finally"); } } // Execute a try block normally. static void procC() { try { System.out.println("inside procC"); } finally { System.out.println("procC's finally"); } } public static void main(String args[]) { try { procA(); } catch (Exception e) { System.out.println("Exception caught"); } procB(); procC(); } }

Demonstrate finally. class FinallyDemo { // Through an exception out of the method. static void procA() { try { System.out.println("inside procA"); throw new RuntimeException("demo"); } finally { System.out.println("procA's finally"); } } // Return from within a try block. static void procB() { try { System.out.println("inside procB"); return; } finally { System.out.println("procB's finally"); } } // Execute a try block normally. static void procC() { try { System.out.println("inside procC"); } finally { System.out.println("procC's finally"); } } public static void main(String args[]) { try { procA(); } catch (Exception e) { System.out.println("caught"); } procB(); procC(); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

125/221	SUBMITTED TEXT	22 WORDS	100%	MATCHING TEXT	22 WORDS

inside procA procA's finally Exception caught inside procB procB's finally inside procC procC's finally

inside procA procA's finally Exception caught inside procB procB's finally inside procC procC's finally

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

126/221	SUBMITTED TEXT	64 WORDS	93%	MATCHING TEXT	64 WORDS

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB( )'s try statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed. 8.6

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB( )'s try statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in try statement executes normally, without error. However, the finally block is still executed.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

127/221	SUBMITTED TEXT	78 WORDS	98%	MATCHING TEXT	78 WORDS

The most general exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they need not be included in any method 's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed in Table 8-1

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they need not be included in any method's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed in Table 10-1.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

128/221	SUBMITTED TEXT	54 WORDS	100%	MATCHING TEXT	54 WORDS

Table 8-2 lists those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. Java defines several other types of exceptions that relate to its various class libraries. 8.7

Table 10-2 lists those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. Java defines several other types of exceptions that relate to its various class libraries

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

129/221	SUBMITTED TEXT	29 WORDS	59%	MATCHING TEXT	29 WORDS

System.out.println("Normal exit"); } public static void main(String args[]) { try { compute(1); compute(20); } catch (MyException e) { System.out.println("Caught " +

System.out.println(" procC's public static void main(String args[]) { try { procA(); } catch (Exception e) { System.out.println("caught"); }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

130/221	SUBMITTED TEXT	47 WORDS	54%	MATCHING TEXT	47 WORDS

Introduction Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of

Introduction: o Java provides a built – in support for multithreaded programming. o A multithreaded program contains two o more parts that can run concurrently. o Each part of such a program called thread. o Each thread defines a separate path of execution. o Thus multi thread is a specialized form of

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

131/221	SUBMITTED TEXT	18 WORDS	91%	MATCHING TEXT	18 WORDS

format text at the same time that it is printing, as long as these two actions are

format text at the same time that is printing as long as these two actions are

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

132/221	SUBMITTED TEXT	24 WORDS	97%	MATCHING TEXT	24 WORDS

At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed. 9.4

At any time, a thread can be terminated, which halts its execution immediately. ƒ Once terminated, a thread cannot be resumed

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

133/221	SUBMITTED TEXT	30 WORDS	90%	MATCHING TEXT	30 WORDS

The Thread Class and the Runnable Interface Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.

The Thread Class and the Runnable InterfaceJava's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

134/221	SUBMITTED TEXT	11 WORDS	100%	MATCHING TEXT	11 WORDS

The Thread class defines several methods that help manage threads.

The Thread class defines several methods that help manage threads.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

135/221	SUBMITTED TEXT	32 WORDS	83%	MATCHING TEXT	32 WORDS

by instantiating an object of type Thread. Java defines two ways in which this can be accomplished: • implement the Runnable interface or • extend the Thread class, itself. Implementing

by instantiating an object of type Thread. x Java defines two ways in which this can be accomplished: ƒ You can implement the Runnable interface. ƒ You can extend the Thread class, itself. Implementing

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

136/221	SUBMITTED TEXT	65 WORDS	72%	MATCHING TEXT	65 WORDS

start(); // Start the thread } // This is the entry point for thread. public void run() { try { for(int i = 5; i < 0; i--) { System.out.println(name + ": " + i); Thread.sleep(1000); } } catch (InterruptedException e) { System.out.println(name + "Interrupted"); } System.out.println(name + " exiting."); } } class MultiThreadDemo { public static void main(String args[]) { new NewThread("

start(); // Start the thread } // This is the entry point for the second thread. public void run() { try { for(int i = 5; i < 0; i--) { System.out.println("Child Thread: " + i); Thread.sleep(500); } } catch (InterruptedException e) { System.out.println("Child interrupted."); } System.out.println("Exiting child thread."); } } class ExtendThread { Smartzworld.com Smartworld.asia jntuworldupdates.org in public static void main(String args[]) { new NewThread(); //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

137/221	SUBMITTED TEXT	17 WORDS	100%	MATCHING TEXT	17 WORDS

Thread.sleep(10000); } catch (InterruptedException e) { System.out.println("Main thread Interrupted"); } System.out.println("Main thread exiting."); } }

Thread.sleep(1000); } } catch (InterruptedException e) { System.out.println("Main thread interrupted."); } System.out.println("Main thread exiting."); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

138/221	SUBMITTED TEXT	11 WORDS	100%	MATCHING TEXT	11 WORDS

Thread.sleep(10000); } catch (InterruptedException e) { System.out.println("Main thread interrupted."); }

Thread.sleep(1000); } } catch (InterruptedException e) { System.out.println("Main thread interrupted"); } } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

139/221	SUBMITTED TEXT	13 WORDS	62%	MATCHING TEXT	13 WORDS

threads to terminate. try { hi.t.join(); lo.t.join(); } catch (InterruptedException e) { System.out.println("

threads to end try { ob1.t.join(); t.join(); ob3.t.join(); } catch(InterruptedException e) { System.out.println("

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

140/221	SUBMITTED TEXT	121 WORDS	92%	MATCHING TEXT	121 WORDS

that the resource is used only by one thread at a time is called synchronization. Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. As you will see, Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

141/221	SUBMITTED TEXT	114 WORDS	92%	MATCHING TEXT	114 WORDS

synchronized. class Callme { void call(String msg) { System.out.print("[" + msg); try { Thread.sleep(1000); } catch(InterruptedException e) { System.out.println("Interrupted"); } System.out.println("]"); } } class Caller implements Runnable { String msg; Callme target; Thread t; public Caller(Callme targ, String s) { target = targ; msg = s; t = new Thread(this); t.start(); } public void run() { target.call(msg); } } class Synch { public static void main(String args[]) { Callme target = new Callme(); Caller ob1 = new Caller(target, "Hello"); Caller ob2 = new Caller(target, "Synchronized"); Caller ob3 = new Caller(target, "World"); // wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join(); } catch(InterruptedException e) { System.out.println("Interrupted"); } } }

synchronized block. class Callme { void call(String msg) { System.out.print("[" + msg); try { Thread.sleep(1000); } catch (InterruptedException e) { System.out.println("Interrupted"); } System.out.println("]"); } } class Caller implements Runnable { String msg; Callme target; Thread t; public Caller(Callme targ, String s) { target = targ; msg = s; t = new Thread(this); t.start(); } // synchronize calls to call() public void run() { synchronized(target) { // synchronized block target.call(msg); } } } class Synch1 { public static void main(String args[]) { Callme target = new Callme(); Caller ob1 = new Caller(target, "Hello"); Caller ob2 = new Caller(target, "Synchronized"); Caller ob3 = new Caller(target, "World"); Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in // wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join(); } catch(InterruptedException e) { System.out.println(" Interrupted"); } } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

142/221	SUBMITTED TEXT	25 WORDS	95%	MATCHING TEXT	25 WORDS

to synchronize access to objects of a class • that was not designed for multithreaded access, • the class does not use synchronized methods

to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

143/221	SUBMITTED TEXT	21 WORDS	81%	MATCHING TEXT	21 WORDS

this class was created by a third party, • do not have access to the source code.

this class was not created by you, but by a third party, and you do not have access to the source code.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

144/221	SUBMITTED TEXT	73 WORDS	94%	MATCHING TEXT	73 WORDS

simply put calls to the methods defined by this class inside a synchronized block. This is the general form of the synchronized statement: synchronized(object) { // statements to be synchronized } Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of 'object' occurs only after the current thread has successfully entered 'object's' monitor. Here is an

simply put calls to the methods defined by this class inside a synchronized block. This is the general form of the synchronized statement: Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in statements to be synchronized } Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor. Here is an

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

145/221	SUBMITTED TEXT	134 WORDS	96%	MATCHING TEXT	134 WORDS

This program uses a synchronized block. class Callme { void call(String msg) { System.out.print("[" + msg); try { Thread.sleep(1000); } catch (InterruptedException e) { System.out.println("Interrupted"); } System.out.println("]"); } } class Caller implements Runnable { String msg; Callme target; Thread t; public Caller(Callme targ, String s) { target = targ; msg = s; t = new Thread(this); t.start(); } // synchronize calls to call() public void run() { synchronized(target) { // synchronized block target.call(msg); } } } class Synch1 { public static void main(String args[]) { Callme target = new Callme(); Caller ob1 = new Caller(target, "Hello"); Caller ob2 = new Caller(target, "Synchronized"); Caller ob3 = new Caller(target, "World"); // wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join(); } catch(InterruptedException e) { System.out.println("Interrupted"); } } }

This program uses a synchronized block. class Callme { void call(String msg) { System.out.print("[" + msg); try { Thread.sleep(1000); } catch (InterruptedException e) { System.out.println("Interrupted"); } System.out.println("]"); } } class Caller implements Runnable { String msg; Callme target; Thread t; public Caller(Callme targ, String s) { target = targ; msg = s; t = new Thread(this); t.start(); } // synchronize calls to call() public void run() { synchronized(target) { // synchronized block target.call(msg); } } } class Synch1 { public static void main(String args[]) { Callme target = new Callme(); Caller ob1 = new Caller(target, "Hello"); Caller ob2 = new Caller(target, "Synchronized"); Caller ob3 = new Caller(target, "World"); Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in // wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join(); } catch(InterruptedException e) { System.out.println("Interrupted"); } } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

146/221	SUBMITTED TEXT	14 WORDS	96%	MATCHING TEXT	14 WORDS

notify() Wakes up a single thread that is waiting on this object's monitor.

notify() method Wakes up a single thread that is waiting on this object's monitor.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

147/221	SUBMITTED TEXT	23 WORDS	58%	MATCHING TEXT	23 WORDS

that the resource is used only by one thread at a time is called synchronization. Java provides unique, language-level support for it.

that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. As you will see, Java provides unique, language-level support for it.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

148/221	SUBMITTED TEXT	37 WORDS	85%	MATCHING TEXT	37 WORDS

class Box { double width; double height; double depth; Box(double w, double h, double d) { width = w; height = h; depth = d; }

class Box { double width; double height; double depth; // This is the constructor for Box. Box(double w, double h, double d) { width = w; height = h; depth = d; } //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

149/221	SUBMITTED TEXT	15 WORDS	83%	MATCHING TEXT	15 WORDS
ArrayList. import java.util.*; class ArrayListDemo { public static void main(String args[]) { //			ArrayList import java.util.*; 2. class public static void main(String args[]){ 4. 5.		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

150/221	SUBMITTED TEXT	15 WORDS	76%	MATCHING TEXT	15 WORDS
class import java.util.Properties; class PropertiesDemo { public static void main(String args[]) {			class 1. import java.util.*; 2. class TestCollection1{ 3. public static void main(String args[]){ 4. 5.		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

151/221	SUBMITTED TEXT	15 WORDS	85%	MATCHING TEXT	15 WORDS
A directory is a File that contains a list of other files and directories.			A directory is a File which can contains a list of other files and directories.		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

152/221	SUBMITTED TEXT	43 WORDS	100%	MATCHING TEXT	43 WORDS
The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.			The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

153/221	SUBMITTED TEXT	13 WORDS	100%	MATCHING TEXT	13 WORDS
creates both a directory and all the parents of the directory.			creates both a directory and all the parents of the directory.		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

154/221	SUBMITTED TEXT	14 WORDS	84%	MATCHING TEXT	14 WORDS
import java.io.*; class FileOutputStreamDemo { public static void main(String args[]) throws IOException {			import java.io.*; public class CopyFile { public static void main(String args[]) throws IOException {		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

155/221	SUBMITTED TEXT	8 WORDS	100%	MATCHING TEXT	8 WORDS
Thread(){ public void run(){ try{ for(int i=65;i>=90;i++) System.out.println(			thread. public void run() { try { for(int i = 5; i < 0; i--) { System.out.println("		

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

156/221	SUBMITTED TEXT	15 WORDS	87%	MATCHING TEXT	15 WORDS

import java.io.*; public class SerializationDemo { public static void main(String args[]) { //

import java.io.*; public class CopyFile { public static void main(String args[])

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

157/221	SUBMITTED TEXT	16 WORDS	85%	MATCHING TEXT	16 WORDS

A directory is a File that contains a list of other files and directories. •

A directory is a File which can contains a list of other files and directories.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

158/221	SUBMITTED TEXT	57 WORDS	100%	MATCHING TEXT	57 WORDS

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—init( ), start( ), stop( ), and destroy( )—are defined by Applet. Another, paint( ), is defined by the AWT Component class.

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—init( ), start( ), stop( ), and destroy( )—are defined by Applet. Another, paint( ), is defined by the AWT Component class.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

159/221	SUBMITTED TEXT	28 WORDS	100%	MATCHING TEXT	28 WORDS

Applet Initialization and Termination When an applet begins, the AWT calls the following methods, in this sequence: 1. init( ) 2. start( ) 3. paint( ) 4.

Applet Initialization and Termination When an applet begins, the AWT calls the following methods, in this sequence: 1. init( ) 2. start( ) 3. paint( )

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

160/221	SUBMITTED TEXT	34 WORDS	84%	MATCHING TEXT	34 WORDS

When an applet is terminated, the following sequence of method calls takes place: 1. stop( ) 2. destroy( ) init( ): The init( ) method is the first method to be called.

When an applet is terminated, the following sequence of method calls takes place: 1. stop( ) 2. destroy( ) Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in init( ) The init( ) method is the first method to be called.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

161/221	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS

This method is called only once during the run time of

This method is called only once during the run time of

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

162/221	SUBMITTED TEXT	81 WORDS	100%	MATCHING TEXT	81 WORDS

applet. start( ) The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ). paint( ) The paint( ) method is called each time

applet. start( ) The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ). paint( ) The paint( ) method is called each time

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

163/221	SUBMITTED TEXT	35 WORDS	100%	MATCHING TEXT	35 WORDS

applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet

applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

164/221	SUBMITTED TEXT	48 WORDS	93%	MATCHING TEXT	48 WORDS

The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required. repaint() The repaint( ) method is

The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required. stop( ) The stop( ) method is

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

165/221	SUBMITTED TEXT	16 WORDS	100%	MATCHING TEXT	16 WORDS

defined by the AWT. It causes the AWT run-time system to execute a call to

defined by the AWT. It causes the AWT run-time system to execute a call to

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

166/221	SUBMITTED TEXT	13 WORDS	100%	MATCHING TEXT	13 WORDS

applet's update( ) method, which, in its default implementation, calls paint( ).

applet's update( ) method, which, in its default implementation, calls paint( )

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

167/221	SUBMITTED TEXT	39 WORDS	95%	MATCHING TEXT	39 WORDS

stop( ) The stop( ) method is called when a web browser leaves the HTML document containing the applet, and when it goes to another page. For example, When stop( ) is called, the applet is probably running.

stop( ) The stop( ) method is called when a web browser leaves the HTML document containing the applet— when it goes to another page, for example. When stop( ) is called, the applet is probably running.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

168/221	SUBMITTED TEXT	64 WORDS	86%	MATCHING TEXT	64 WORDS

to suspend threads that don't need to run when the applet is not visible. It can be restarted, when start( ) is called if the user returns to the page. destroy( ) The destroy( ) method is called when the environment determines that the applet needs to be removed completely from memory. At this point, any resources the applet may be using

to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page. destroy( ) The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

169/221	SUBMITTED TEXT	220 WORDS	100%	MATCHING TEXT	220 WORDS

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal. There are some important differences between an applet and a standalone Java application, including the following: • An applet is a Java class that extends the java.applet.Applet class. • A main() method is not invoked on an applet, and an applet class will not define main(). • Applets are designed to be embedded within an HTML page. • When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine. • A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment. • The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime. • Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed. • Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file. 14.1

SA 139E1120, 151E1120,155E1140-Advanced Java Programming.doc (D165246352)

170/221	SUBMITTED TEXT	176 WORDS	97%	MATCHING TEXT	176 WORDS

PROGRAM /* A simple banner applet. This applet creates a thread that scrolls the message contained in msg right to left across the applet's window. */ import java.awt.*; import java.applet.*; /* >applet code="SimpleBanner" width=300 height=50< >/applet< */ public class SimpleBanner extends Applet implements Runnable { String msg = " A Simple Moving Banner."; Thread t = null; int state; boolean stopFlag; // Set colors and initialize thread. public void init() { setBackground(Color.cyan); setForeground(Color.red); } // Start thread public void start() { t = new Thread(this); stopFlag = false; t.start(); } // Entry point for the thread that runs the banner. public void run() { char ch; // Display banner for(; ;) { try { repaint(); Thread.sleep(250); ch = msg.charAt(0); msg = msg.substring(1, msg.length()); msg += ch; if(stopFlag) break; } catch(InterruptedException e) {} } } / Pause the banner. public void stop() { stopFlag = true; t = null; } // Display the banner. public void paint(Graphics g) { g.drawString(msg, 50, 30); } }

Program /* A simple banner applet. This applet creates a thread that scrolls the message contained in msg right to left across the applet's window. */ import java.awt.*; import java.applet.*; /* >applet code="SimpleBanner" width=300 height=50< >/applet< */ public class SimpleBanner extends Applet implements Runnable { com Smartworld.jntuworldupdates.org Specworld.inA Simple Moving Banner."; Thread t = null; int state; boolean stopFlag; // Set colors and initialize thread. public void init() { setBackground(Color.cyan); setForeground(Color.red); } // Start thread public void start() { t = new Thread(this); stopFlag = false; t.start(); } // Entry point for the thread that runs the banner. public void run() { char ch; // Display banner for(; ;) { try { repaint(); Thread.sleep(250); ch = msg.charAt(0); msg = msg.substring(1, msg.length()); msg += ch; if(stopFlag) break; } catch(InterruptedException e) {} } } // Pause the banner. public void stop() { stopFlag = true; t = null; } // Display the banner. public void paint(Graphics g) { g.drawString(msg, 50, 30); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

171/221	SUBMITTED TEXT	16 WORDS	100%	MATCHING TEXT	16 WORDS

The syntax for the standard APPLET tag is shown here. Bracketed items are optional. [

The syntax for the standard APPLET tag is shown here. Bracketed items are optional. >

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

172/221	SUBMITTED TEXT	161 WORDS	97%	MATCHING TEXT	161 WORDS

HTML Displayed in the absence of Java] >/APPLET< CODEBASE CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read. CODE CODE is a required attribute that gives the name of the file containing your applet's compiled .class file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set. ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets.

HTML Displayed in the absence of Java] >/APPLET< CODEBASE Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read. CODE CODE is a required attribute that gives the name of the file containing your applet's compiled .class file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set. ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

173/221	SUBMITTED TEXT	309 WORDS	98%	MATCHING TEXT	309 WORDS

applets. NAME NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use getApplet( ), which is defined by the AppletContext interface. WIDTH AND HEIGHT WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area. ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM. VSPACE AND HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes. PARAM NAME AND VALUE The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the getParameter( ) method. HANDLING OLDER BROWSERS Some very old web browsers can't execute applets and don't recognize the APPLET tag. Although these browsers are now nearly extinct (having been replaced by Java-compatible ones), you may need to deal with them occasionally. The best way to design your HTML page to deal with such browsers is to include HTML text and markup within your >applet<>/applet< tags. If the applet tags are not recognized by your browser, you will see the alternate markup. If Java is available, it will consume all of the markup between the >applet<>/applet< tags and disregard the alternate markup. Here's the HTML to start an applet called SampleApplet in Java and to display a message in older browsers: 14.4.1

applets. NAME NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use getApplet( ), which is defined by the AppletContext interface. WIDTH AND HEIGHT WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area. ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM. VSPACE AND HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes. PARAM NAME AND VALUE The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the getParameter( ) method. HANDLING OLDER BROWSERS Some very old web browsers can't execute applets and don't recognize the APPLET tag. Although these browsers are now nearly extinct (having been replaced by Java-compatible ones), you may need to deal with them occasionally. The best way to design your HTML page to deal with such browsers is to include HTML text and markup within your >applet<>/applet< tags. If the applet tags are not recognized by your browser, you will see the alternate markup. If Java is available, it will consume all of the markup between the >applet<>/applet< tags and disregard the alternate markup. Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in Here's the HTML to start an applet called SampleApplet in Java and to display a message in older browsers: >

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

174/221	SUBMITTED TEXT	15 WORDS	92%	MATCHING TEXT	15 WORDS

Passing Parameters to Applets The APPLET tag in HTML allows to pass parameters to

Passing Parameters to Applets: z the APPLET tag in HTML allows you to pass parameters to

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

175/221	SUBMITTED TEXT	40 WORDS	93%	MATCHING TEXT	40 WORDS

applet. To retrieve a parameter, use the getParameter( ) method. It returns the value of the specified parameter in the form of a String object. Thus, for numeric and Boolean values, convert their string representations into their internal formats.

applet. To retrieve a parameter, use the getParameter( ) method. z It returns the value of the specified parameter in the form of a String object. Thus, for numeric and boolean values, you will need to convert their string representations into their internal formats.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

176/221	SUBMITTED TEXT	16 WORDS	96%	MATCHING TEXT	16 WORDS

The event handling is at the core of successful applet programming. Most events to which

the output: Event Handling is at the core of successful applet programming. Most events to which

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

177/221	SUBMITTED TEXT	14 WORDS	100%	MATCHING TEXT	14 WORDS

applet will respond are generated by the user. These events are passed to

applet will respond are generated by the user. These events are passed to

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

178/221	SUBMITTED TEXT	52 WORDS	92%	MATCHING TEXT	52 WORDS

applet in a variety of ways, with the specific method depending upon the actual event. There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the java.awt.event package. 15.1

applet in a variety of ways, with the specific method depending upon the actual event. There are several Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the java.awt.event package

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

179/221	SUBMITTED TEXT	30 WORDS	100%	MATCHING TEXT	30 WORDS

The Delegation Event Model The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its

The Delegation Event Model The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

180/221	SUBMITTED TEXT	17 WORDS	100%	MATCHING TEXT	17 WORDS

quite simple: a source generates an event and sends it to one or more listeners. In

quite simple: a source generates an event and sends it to one or more listeners. In

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

181/221	SUBMITTED TEXT	101 WORDS	96%	MATCHING TEXT	101 WORDS

the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to —delegatell the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the applicationlogic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to —delegatell the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

182/221	SUBMITTED TEXT	121 WORDS	100%	MATCHING TEXT	121 WORDS

Events In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

Events In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

183/221	SUBMITTED TEXT	311 WORDS	98%	MATCHING TEXT	311 WORDS

Event Sources A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form: public void addTypeListener(TypeListener el) Here, Type is the name of the event and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method is this: public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException Here, Type is the name of the event and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this: public void removeTypeListener(TypeListener el) Here, Type is the name of the event and el is a reference to the event listener. For example, to remove a keyboard listener, you would call removeKeyListener( ). The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners. Event

Event Sources A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form: public void addTypeListener(TypeListener el) Here, Type is the name of the event and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method is this: public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException Here, Type is the name of the event and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this: public void removeTypeListener(TypeListener el) Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in Here, Type is the name of the event and el is a reference to the event listener. For example, to remove a keyboard listener, you would call removeKeyListener( ). The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners. Event

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

184/221	SUBMITTED TEXT	107 WORDS	100%	MATCHING TEXT	107 WORDS

Event Listeners A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. 15.3

Event Listeners A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

185/221	SUBMITTED TEXT	23 WORDS	100%	MATCHING TEXT	23 WORDS

At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events.

At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

186/221	SUBMITTED TEXT	44 WORDS	100%	MATCHING TEXT	44 WORDS

shown here: EventObject(Object src) Here, src is the object that generates this event. EventObject contains two methods: getSource( ) and toString( ). The getSource( ) method returns the source of the event. Its general form is shown here: Object getSource( )

shown here: EventObject(Object src) Here, src is the object that generates this event. EventObject contains two methods: getSource( ) and toString( ). The getSource( ) method returns the source of the event. Its general form is shown here: Object getSource( )

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

187/221	SUBMITTED TEXT	21 WORDS	100%	MATCHING TEXT	21 WORDS

toString( ) returns the string equivalent of the event. The class AWTEvent, defined within the java.awt package, is a subclass

toString( ) returns the string equivalent of the event. The class AWTEvent, defined within the java.awt package, is a subclass

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

188/221	SUBMITTED TEXT	46 WORDS	100%	MATCHING TEXT	46 WORDS

It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its getID( ) method can be used to determine the type of the event. The signature of this method is shown here: int getID( )

It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its getID( ) method can be used to determine the type of the event. The signature of this method is shown here: int getID( )

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

189/221	SUBMITTED TEXT	39 WORDS	100%	MATCHING TEXT	39 WORDS

The package java.awt.event defines several types of events that are generated by various user interface elements. Table 15-1 enumerates the most important of these event classes and provides a brief description of when they are generated.

The package java.awt.event defines several types of events that are generated by various user interface elements. Table 20-1 enumerates the most important of these event classes and provides a brief description of when they are generated.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

190/221	SUBMITTED TEXT	20 WORDS	80%	MATCHING TEXT	20 WORDS

MouseMotionListener Defines two methods to recognize when the mouse is dragged or moved.

MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

191/221	SUBMITTED TEXT	106 WORDS	95%	MATCHING TEXT	106 WORDS

Demonstrate the key event handlers. import java.awt.*; import java.awt.event.*; import java.applet.*; /*>applet code="SimpleKey" width=300 height=100< >/applet< */ public class SimpleKey extends Applet implements KeyListener { String msg = ""; int X = 10, Y = 20; // output coordinates public void init() { addKeyListener(this); requestFocus(); // request input focus } public void keyPressed(KeyEvent ke) { showStatus("Key Down"); } public void keyReleased(KeyEvent ke) { showStatus("Key Up"); } public void keyTyped(KeyEvent ke) { msg += ke.getKeyChar(); repaint(); } // Display keystrokes. public void paint(Graphics g) { g.drawString(msg, X, Y); } }

Demonstrate the key event handlers. import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="SimpleKey" width=300 height=100< >/applet< */ public class SimpleKey extends Applet implements KeyListener { String msg = ""; int X = 10, Y = 20; // output coordinates public void init() { addKeyListener(this); requestFocus(); // request input focus } public void keyPressed(KeyEvent ke) { showStatus("Key Down"); } public void keyReleased(KeyEvent ke) { showStatus("Key Up"); } public void keyTyped(KeyEvent ke) { msg += ke.getKeyChar(); Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in repaint(); } // Display keystrokes. public void paint(Graphics g) { g.drawString(msg, X, Y); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

192/221	SUBMITTED TEXT	38 WORDS	100%	MATCHING TEXT	38 WORDS

Events: In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

Events In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

193/221	SUBMITTED TEXT	36 WORDS	100%	MATCHING TEXT	36 WORDS

Event Sources: A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event 15.7

Event Sources A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

194/221	SUBMITTED TEXT	31 WORDS	32%	MATCHING TEXT	31 WORDS

void fillRect(int top, int left, int width, int height) The upper-left corner of the rectangle is at top,left. The dimensions of the rectangle are specified by width and height.

void repaint(int left, int top, int width, int height) x Here, the coordinates of the upper-left corner of the region are specified by left and top, and the width and height of the region are passed in width and height.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

195/221	SUBMITTED TEXT	24 WORDS	71%	MATCHING TEXT	24 WORDS

import java.awt.*; import java.applet.*; /* >applet code="ColorDemo" width=300 height=200< >/applet< */ public class ColorDemo extends Applet { // draw lines public void

import java.awt.*; import java.applet.*; /* >applet code="AppletSkel" width=300 height=100< >/applet< */ public class AppletSkel extends Applet { // Called first. public void

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

196/221	SUBMITTED TEXT	24 WORDS	81%	MATCHING TEXT	24 WORDS

import java.applet.*; import java.awt.*; /* >applet code="FontInfo" width=350 height=60< >/applet< */ public class FontInfo extends Applet { public void

import java.applet.*; import java.awt.event.*; /* >applet code="AnonymousInnerClassDemo" width=200 height=100< >/applet< */ public class AnonymousInnerClassDemo extends Applet { public void

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

197/221	SUBMITTED TEXT	41 WORDS	63%	MATCHING TEXT	41 WORDS

import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="FlowLayoutDemo" width=250 height=200< >/applet< */ public class FlowLayoutDemo extends Applet implements ItemListener { String msg = ""; Checkbox Win98, winNT, solaris, mac; public void init() { //

import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="SimpleKey" width=300 height=100< >/applet< */ public class SimpleKey extends Applet implements String msg = ""; int X = 10, Y = 20; // output coordinates public void init() {

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

198/221	SUBMITTED TEXT	27 WORDS	68%	MATCHING TEXT	27 WORDS

import java.awt.*; import java.applet.*; import java.util.*; /* >applet code="BorderLayoutDemo" width=400 height=200< >/applet< */ public class BorderLayoutDemo extends Applet { public void init() { setLayout(new

import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="AdapterDemo" width=300 height=100< >/applet< */ public class AdapterDemo extends Applet { public void init() { addMouseListener(new

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

199/221	SUBMITTED TEXT	22 WORDS	86%	MATCHING TEXT	22 WORDS

import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="CardLayoutDemo" width=300 height=100< >/applet< */ public class CardLayoutDemo extends Applet implements

import java.awt.*; import java.awt.event.*; import java.applet.*; /* >applet code="SimpleKey" width=300 height=100< >/applet< */ public class SimpleKey extends Applet implements

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

200/221	SUBMITTED TEXT	20 WORDS	100%	MATCHING TEXT	20 WORDS

The Swing-related classes are contained in javax.swing and its subpackages, such as javax.swing.tree. Many other Swing-related classes and interfaces

The Swing-related classes are contained in javax.swing and its subpackages, such as javax.swing.tree. Many other Swing-related classes and interfaces

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

201/221	SUBMITTED TEXT	14 WORDS	100%	MATCHING TEXT	14 WORDS

JApplet Fundamental to Swing is the JApplet class, which extends Applet. Applets that

JApplet Fundamental to Swing is the JApplet class, which extends Applet. Applets that

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

202/221	SUBMITTED TEXT	13 WORDS	100%	MATCHING TEXT	13 WORDS

must be subclasses of JApplet. JApplet is rich with functionality that is

must be subclasses of JApplet. JApplet is rich with functionality that is

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

203/221	SUBMITTED TEXT	21 WORDS	100%	MATCHING TEXT	21 WORDS

in Applet. For example, JApplet supports various —panes,‖ such as the content pane, the glass pane, and the root pane.

in Applet. For example, JApplet supports various —panes,‖ such as the content pane, the glass pane, and the root pane.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

204/221	SUBMITTED TEXT	90 WORDS	96%	MATCHING TEXT	90 WORDS

When adding a component to an instance of JApplet, do not invoke the add( ) method of the applet. Instead, call add( ) for the content pane of the JApplet object. The content pane can be obtained via the method shown here: Container getContentPane( ) The add( ) method of Container can be used to add a component to a content pane. Its form is shown here: void add(comp) Here, comp is the component to be added to the content pane. 17.3 JFrame and JDialog In

When adding a component to an instance of JApplet, do not invoke the add( ) method of the applet. Instead, call add( ) for the content pane of the JApplet object. The content pane can be obtained via the method shown here: Container getContentPane( ) The add( ) method of Container can be used to add a component to a content pane. Its form is shown here: void add(comp) Here, comp is the component to be added to the content pane. Icons and In

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

205/221	SUBMITTED TEXT	181 WORDS	97%	MATCHING TEXT	181 WORDS

Fields The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows you to edit one line of text. Some of its constructors are shown here: JTextField( ) JTextField(int cols) JTextField(String s, int cols) JTextField(String s) Here, s is the string to be presented, and cols is the number of columns in the text field. The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a JTextField object is created and is added to the content pane. PROGRAM import java.awt.*; import javax.swing.*; /* >applet code="JTextFieldDemo" width=300 height=50< >/applet< */ public class JTextFieldDemo extends JApplet { JTextField jtf; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); // Add text field to content pane jtf = new JTextField(15); contentPane.add(jtf); } } OUTPUT 17.5

Fields The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows you to edit one line of text. Some of its constructors are shown here: JTextField( ) JTextField(int cols) JTextField(String s, int cols) JTextField(String s) Here, s is the string to be presented, and cols is the number of columns in the text field. The following example illustrates how to create a text field. The applet begins by getting its content pane, and a flow layout is assigned as its layout manager. Next, a JTextField object is created and is added to the content pane. Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in import java.awt.*; import javax.swing.*; /* >applet code="JTextFieldDemo" width=300 height=50< >/applet< */ public class JTextFieldDemo extends JApplet { JTextField jtf; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); // Add text field to content pane jtf = new JTextField(15); contentPane.add(jtf); } } Output

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

206/221	SUBMITTED TEXT	249 WORDS	97%	MATCHING TEXT	249 WORDS

Swing buttons are subclasses of the AbstractButton class, which extends JComponent. AbstractButton contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons. For example, you can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as a rollover icon, which is displayed when the mouse is positioned over that component. The following are the methods that control this behavior: void setDisabledIcon(Icon di) void setPressedIcon(Icon pi) void setSelectedIcon(Icon si) void setRolloverIcon(Icon ri) Here, di, pi, si, and ri are the icons to be used for these different conditions. The text associated with a button can be read and written via the following methods: String getText( ) void setText(String s) Here, s is the text to be associated with the button. Concrete subclasses of AbstractButton generate action events when they are pressed. Listeners register and unregister for these events via the methods shown here: void addActionListener(ActionListener al) void removeActionListener(ActionListener al) Here, al is the action listener. AbstractButton is a superclass for push buttons, check boxes, and radio buttons. The JButton Class The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here: JButton(Icon i) JButton(String s) JButton(String s, Icon i) Here, s and i are the string and icon used for the button.

Swing buttons are subclasses of the AbstractButton class, which extends JComponent. AbstractButton contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons. For example, you can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as a rollover icon, which is displayed when the mouse is positioned over that component. The following are the methods that control this behavior: void setDisabledIcon(Icon di) void setPressedIcon(Icon pi) void setSelectedIcon(Icon si) void setRolloverIcon(Icon ri) Here, di, pi, si, and ri are the icons to be used for these different conditions. The text associated with a button can be read and written via the following methods: Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in String getText( ) void setText(String s) Here, s is the text to be associated with the button. Concrete subclasses of AbstractButton generate action events when they are pressed. Listeners register and unregister for these events via the methods shown here: void addActionListener(ActionListener al) void removeActionListener(ActionListener al) Here, al is the action listener. AbstractButton is a superclass for push buttons, check boxes, and radio buttons. Each is next. The JButton The JButton class provides the functionality of a push button. JButton allows an icon,a string, or both to be associated with the push button. Some of its constructors are shown here: JButton(Icon i) JButton(String s) JButton(String s, Icon i) Here, s and i are the string and icon used for the button.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

207/221	SUBMITTED TEXT	12 WORDS	100%	MATCHING TEXT	12 WORDS

text field. The applet begins by getting its content pane and

text field. The applet begins by getting its content pane, and

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

208/221	SUBMITTED TEXT	44 WORDS	82%	MATCHING TEXT	44 WORDS

import java.awt.*; import java.awt.event.*; import javax.swing.*; /* >applet code="JButtonDemo" width=250 height=300< >/applet< */ public class JButtonDemo extends JApplet implements ActionListener { JTextField jtf; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); //

import java.awt.*; import java.awt.event.*; import javax.swing.*; /* >applet code="JCheckBoxDemo" width=400 height=50< >/applet< */ public class JCheckBoxDemo extends JApplet implements JTextField jtf; public void init() { SOFTWARE DEVELOPMENT USING Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

209/221	SUBMITTED TEXT	22 WORDS	88%	MATCHING TEXT	22 WORDS

this); contentPane.add(jb); // Add text field to content pane jtf = new JTextField(15); contentPane.add(jtf); } public void

this); contentPane.add(cb); // Add text field to the content pane jtf = new JTextField(15); contentPane.add(jtf); } public void

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

210/221	SUBMITTED TEXT	50 WORDS	97%	MATCHING TEXT	50 WORDS

Boxes Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class, which extends JComponent. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. Selection can

Boxes Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class, which extends JComponent. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

211/221	SUBMITTED TEXT	214 WORDS	96%	MATCHING TEXT	214 WORDS

into the text field. Two of JComboBox's constructors are shown here: JComboBox( ) JComboBox(Vector v) Here, v is a vector that initializes the combo box. Items are added to the list of choices via the addItem( ) method, whose signature is shown here: void addItem(Object obj) Here, obj is the object to be added to the combo box. The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for —Francell, —Germanyll, —Italyll, and —Japanll. When a country is selected, the label is updated to display the flag for that country. PROGRAM import java.awt.*; import java.awt.event.*; import javax.swing.*; /* >applet code="JComboBoxDemo" width=300 height=100< >/applet< */ public class JComboBoxDemo extends JApplet implements ItemListener { JLabel jl; ImageIcon france, germany, italy, japan; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); // Create a combo box and add it // to the panel JComboBox jc = new JComboBox(); jc.addItem("France"); jc.addItem("Germany"); jc.addItem("Italy"); jc.addItem("Japan"); jc.addItemListener(this); contentPane.add(jc); // Create label jl = new JLabel(new ImageIcon("france.gif")); contentPane.add(jl); } public void itemStateChanged(ItemEvent ie) { String s = (String)ie.getItem(); jl.setIcon(new ImageIcon(s + ".gif")); } } OUTPUT 17.7

into the text field. Two of JComboBox's constructors are shown here: JComboBox( ) JComboBox(Vector v) Here, v is a vector that initializes the combo box. Items are added to the list of choices via the addItem( ) method, whose signature is shown here: void addItem(Object obj) Here, obj is the object to be added to the combo box. The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for —Francell, —Germanyll, —Italyll, and —Japanll. When a country is selected, the label is updated to display the flag for that country. SOFTWARE DEVELOPMENT USING import java.awt.*; import java.awt.event.*; import javax.swing.*; /* >applet code="JComboBoxDemo" width=300 height=100< >/applet< */ public class JComboBoxDemo extends JApplet implements ItemListener { JLabel jl; Smartzworld.com Smartworld.asia jntuworldupdates.Specworld.in ImageIcon france, germany, italy, japan; public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout()); // Create a combo box and add it // to the panel JComboBox jc = new JComboBox(); jc.addItem("France"); jc.addItem("Germany"); jc.addItem("Italy"); jc.addItem("Japan"); jc.addItemListener(this); contentPane.add(jc); // Create label jl = new JLabel(new ImageIcon("france.gif")); contentPane.add(jl); } public void itemStateChanged(ItemEvent ie) { String s = (String)ie.getItem(); jl.setIcon(new ImageIcon(s + ".gif")); } } Output

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

212/221	SUBMITTED TEXT	19 WORDS	75%	MATCHING TEXT	19 WORDS

import java.awt.*; import java.awt.event.*; /* >applet code="JListDemo" width=200 height=120< >/applet< */ public class JListDemo extends JApplet {

import java.awt.*; import java.awt.event.*; import javax.swing.*; /* >applet code="JCheckBoxDemo" width=400 height=50< >/applet< */ public class JCheckBoxDemo extends JApplet

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

213/221	SUBMITTED TEXT	25 WORDS	100%	MATCHING TEXT	25 WORDS

Panes A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When

Panes A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

214/221	SUBMITTED TEXT	156 WORDS	97%	MATCHING TEXT	156 WORDS

user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options. Tabbed panes are encapsulated by the JTabbedPane class, which extends JComponent. We will use its default constructor. Tabs are defined via the following method: void addTab(String str, Component comp) Here, str is the title for the tab, and comp is the component that should be added to the tab. Typically, a JPanel or a subclass of it is added. The general procedure to use a tabbed pane in an applet is outlined here: 1. Create a JTabbedPane object. 2. Call addTab( ) to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.) 3. Repeat step 2 for each tab. 4. Add the tabbed pane to the content pane of the applet.

user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options. Tabbed panes are encapsulated by the JTabbedPane class, which extends JComponent. We will use its default constructor. Tabs are defined via the following method: Smartzworld.Smartworld.asia jntuworldupdates.org Specworld.in str, Component Here, str is the title for the tab, and comp is the component that should be added to the tab. Typically, a JPanel or a subclass of it is added. The general procedure to use a tabbed pane in an applet is outlined here: 1. Create a JTabbedPane object. 2. Call addTab( ) to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.) 3. Repeat step 2 for each tab. 4. Add the tabbed pane to the content pane of the applet.

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

215/221	SUBMITTED TEXT	22 WORDS	82%	MATCHING TEXT	22 WORDS

import javax.swing.*; /* >applet code="JTabbedPaneDemo" width=400 height=100< >/applet< */ public class JTabbedPaneDemo extends JApplet { public void init() {

import javax.swing.*; /* >applet code="JScrollPaneDemo" width=300 height=250< >/applet< */ public class JScrollPaneDemo extends JApplet { public void init() { //

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

216/221	SUBMITTED TEXT	16 WORDS	100%	MATCHING TEXT	16 WORDS

Panes A scroll pane is a component that presents a rectangular area in which a

Panes A scroll pane is a component that presents a rectangular area in which a

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

217/221	SUBMITTED TEXT	62 WORDS	95%	MATCHING TEXT	62 WORDS

be viewed. Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the JScrollPane class, which extends JComponent. Some of its constructors are shown here: JScrollPane(Component comp) JScrollPane(int vsb, int hsb) JScrollPane(Component comp, int vsb, int hsb) Here, comp is the component to be added to the scroll pane. vsb and hsb are

be viewed. Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the JScrollPane class, which extendsJComponent. Some of its constructors are shown here: JScrollPane(Component comp) JScrollPane(int vsb, int hsb) JScrollPane(Component comp, int vsb, int hsb) Here, comp is the component to be added to the scroll pane. vsb and hsb are

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

218/221	SUBMITTED TEXT	40 WORDS	86%	MATCHING TEXT	40 WORDS

that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the ScrollPaneConstants interface. Some examples of these constants are described as follows: Constant Description HORIZONTAL_SCROLLBAR_ALW AYS Always provide horizontal scroll bar HORIZONTAL_SCROLLBAR_AS_

that define when vertical and horizontal scroll bars for this scroll pane These constants are defined by the ScrollPaneConstants interface. Some examples of these constants are described as follows: Constant Description HORIZONTAL_SCROLLBAR_ ALWAYS Always provide horizontal scroll bar HORIZONTAL_SCROLLBAR_AS_

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

219/221	SUBMITTED TEXT	155 WORDS	96%	MATCHING TEXT	155 WORDS

Provide horizontal scroll bar, if needed VERTICAL_SCROLLBAR_ALWAYS Always provide vertical scroll bar VERTICAL_SCROLLBAR_AS_NEE Provide vertical scroll bar, if needed DED Here are the steps that you should follow to use a scroll pane in an applet: 1. Create a JComponent object. 2. Create a JScrollPane object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.) 3. Add the scroll pane to the content pane of the applet. The following example illustrates a scroll pane. First, the content pane of the JApplet object is obtained and a border layout is assigned as its layout manager. Next, a JPanel object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. the scroll bars

Provide horizontal scroll bar, if needed VERTICAL_SCROLLBAR_ALWAYS Always provide vertical scroll bar VERTICAL_SCROLLBAR_AS_NEEDED Provide vertical scroll bar, if needed Here are the steps that you should follow to use a scroll pane in an applet: 1. Create a JComponent object. 2. Create a JScrollPane object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.) 3. Add the scroll pane to the content pane of the applet. USING The following example illustrates a scroll pane. First, the content pane of the JApplet object is obtained and a border layout is assigned as its layout manager. Next, a JPanel object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. You can use the scroll bars

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

220/221	SUBMITTED TEXT	113 WORDS	94%	MATCHING TEXT	113 WORDS

to scroll the buttons into view. PROGRAM import java.awt.*; import javax.swing.*; /* >applet code="JScrollPaneDemo" width=300 height=250< >/applet< */ public class JScrollPaneDemo extends JApplet { public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new BorderLayout()); // Add 400 buttons to a panel JPanel jp = new JPanel(); jp.setLayout(new GridLayout(20, 20)); int b = 0; for(int i = 0; i > 20; i++) { for(int j = 0; j > 20; j++) { jp.add(new JButton("Button " + b)); ++b; } } // Add panel to a scroll pane int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED; int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_

to scroll the buttons into view. Smartzworld.com Smartworld.asia jntuworldupdates.org Specworld.in import java.awt.*; import javax.swing.*; /* >applet code="JScrollPaneDemo" width=300 height=250< >/applet< */ public class JScrollPaneDemo extends JApplet { public void init() { // Get content pane Container contentPane = getContentPane(); contentPane.setLayout(new BorderLayout()); // Add 400 buttons to a panel JPanel jp = new JPanel(); jp.setLayout(new GridLayout(20, 20)); int b = 0; for(int i = 0; i > 20; i++) { for(int j = 0; j > 20; j++) { jp.add(new JButton("Button " + b)); ++b; } } // Add panel to a scroll pane int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED; int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf

221/221	SUBMITTED TEXT	21 WORDS	92%	MATCHING TEXT	21 WORDS

JScrollPane jsp = new JScrollPane(jp, v, h); // Add scroll pane to the content pane contentPane.add(jsp, BorderLayout.CENTER); } }

JScrollPane jsp = new JScrollPane(table, v, h); // Add scroll pane to the content pane contentPane.add(jsp, BorderLayout.CENTER); } }

W http://sim.edu.in/wp-content/uploads/2019/10/java-bca.pdf