Biyani's Think Tank

Concept based notes

Programming in Java

MCA

Mr Dhanesh
Revised by: Mr Shiv Kishore Sharma
Deptt. of IT
Biyani Girls College, Jaipur

Published by :
Think Tanks
Biyani Group of Colleges

Sector-3, Vidhyadhar Nagar,
Jaipur-302 023 (Rajasthan)
Ph : 0141-2338371, 2338591-95 • Fax : 0141-2338007
E-mail : acad@biyanicolleges.org
Website :www.gurukpo.com; www.biyanicolleges.org

Edition : 2012

Leaser Type Setted by :
Biyani College Printing Department

Preface

I am glad to present this book, especially designed to serve the needs of the students. The book has been written keeping in mind the general weakness in understanding the fundamental concepts of the topics. The book is self-explanatory and adopts the "Teach Yourself" style. It is based on question-answer pattern. The language of book is quite easy and understandable based on scientific approach.

Any further improvement in the contents of the book by making corrections, omission and inclusion is keen to be achieved based on suggestions from the readers for which the author shall be obliged.

I acknowledge special thanks to Mr. Rajeev Biyani, *Chairman &* Dr. Sanjay Biyani, *Director* (*Acad.*) Biyani Group of Colleges, who are the backbones and main concept provider and also have been constant source of motivation throughout this endeavour. They played an active role in coordinating the various stages of this endeavour and spearheaded the publishing work.

I look forward to receiving valuable suggestions from professors of various educational institutions, other faculty members and students for improvement of the quality of the book. The reader may feel free to send in their comments and suggestions to the under mentioned address.

Author

Q.1 Write the Object Oriented Concepts in Java.

Ans. Object Oriented is a technique used to develop programs revolving around the real world entities. In OOPs programming model, programs are developed around data rather than actions and logics. In OOPs, every real life object has properties and behavior which is achieved through the class and object creation.

There are four main pillars of an Object Oriented Programming Language :

- **Inheritance:**
  Inheritance provide the facility to drive one class by another using simple syntax. You can say that it is a process of creating new class and use the behavior of the existing class by extending them for reuse the existing code and adding the additional features as you need. It also use to manage and make well structured software.
- **Encapsulation:**
  Encapsulation is the ability to bundle the property and method of the object and also operate them. It is the mechanism of combining the information and providing the abstraction as well.
- **Polymorphism:**
  As the name suggest one name multiple form, Polymorphism is the way that provide the different functionality by the functions having the same name based on the signatures of the methods. There are two type of polymorphism first is run-time polymorphism and second is compile-time polymorphism.
- **Dynamic binding:**
  It is the way that provide the maximum functionality to a program for a specific type at runtime. There are two type of binding first is dynamic binding and second is static binding.

  Java is a fully Object Oriented language because object is at the outer most level of data structure in java. No stand alone methods, constants, and variables are there in java. Everything in java is object even the primitive data types can also be converted into object by using the wrapper class.

Q.2 Why Java?

Ans. The programs that we are writing are very similar to their counterparts in several other languages, so our choice of language is not crucial. We use Java because it is widely available, embraces a full set of modern

abstractions, and has a variety of automatic checks for mistakes in programs, so it is suitable for learning to program.

Q.3 Write difference between Java Applets and Applications

Ans. We used Java to create two types of programs: applications and applets.

An application is a program that runs on our computer, under the operating system of that computer. Application created by Java is like one created using C or C++. When used to create applications, Java is not much different from any other computer language.

An applet is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser. An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip. In other words, an applet is a program that can react to user input and dynamically change not just run the same animation or sound over and over.

Q.4 Explain the Java features.

Ans.

SECURITY

Java does not use memory pointers explicitly. All the programs in java are run under an area known as the sand box. Security manager determines the accessibility options of a class like reading and writing a file to the local disk. Java uses the public key encryption system to allow the java applications to transmit over the internet in the secure encrypted form. The bytecode Verifier checks the classes after loading.

PORTABILITY

The feature Write-once-run-anywhere makes the java language portable provided that the system must have interpreter for the JVM. Java also have the standard data size irrespective of operating system or the processor. These features makes the java as a portable language.

THE BYTECODE

The output of a Java compiler is not executable code rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). That is, in its standard form, the JVM is an interpreter for bytecode. The use of bytecode enables the Java run-time system to execute programs much faster than you might expect.

OBJECT ORIENTED

To be an Object Oriented language, any language must follow at least the four characteristics.

Inheritance : It is the process of creating the new classes and using the behavior of the existing classes by extending them just to reuse the existing code and adding the additional features as needed.

Encapsulation: It is the mechanism of combining the information and providing the abstraction.

Polymorphism: As the name suggest one name multiple form, Polymorphism is the way of providing the different functionality by the functions having the same name based on the signatures of the methods.

Dynamic binding :Sometimes we don't have the knowledge of objects about their specific types while writing our code. It is the way of providing the maximum functionality to a program about the specific type at runtime.

ROBUST

Java has the strong memory allocation and automatic garbage collection mechanism. It provides the powerful exception handling and type checking mechanism as compare to other programming languages. Compiler checks the program whether there any error and interpreter checks any run time error and makes the system secure from crash. All of the above features make the java language robust.

MULTITHREADED

Java is a Multithreaded programming language. Multithreading means a single program having different threads executing independently at the same time. Multiple threads execute instructions according to the program code in a process or a program. Multithreading works the similar way as multiple processes run on one computer. Multithreading programming is a very interesting concept in Java. In multithreaded programs not even a single thread disturbs the execution of other thread. Threads are obtained from the pool of available ready to run threads and they run on the system CPUs.

ARCHITECTURE NEUTRAL

The term architectural neutral seems to be weird, but yes Java is an architectural neutral language as well. The growing popularity of networks makes developers think distributed. In the world of network it is essential that the applications must be able to migrate easily to different computer systems. Not only to computer systems but to a wide variety of hardware architecture and Operating system architectures as well. The Java compiler does this by generating byte code instructions, to be easily interpreted on any machine and to be easily translated into native machine code on the fly. The compiler generates an architecture-neutral object file format to enable a Java application to execute anywhere on the network and then the compiled code is executed on many processors, given the presence of the Java runtime system. Hence Java was designed to support applications on network. This feature of Java has thrived the programming language.

DISTRIBUTED

The widely used protocols like HTTP and FTP are developed in java. Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing codes on their local system.

While executing the java program the user can get the required files dynamically from a local drive or from a computer thousands of miles away from the user just by connecting with the Internet.

DYNAMIC

While executing the java program the user can get the required files dynamically from a local drive or from a computer thousands of miles away from the user just by connecting with the Internet.

Q.5 What are the Data types in Java.

Ans. The eight primitive data types supported by the Java programming language are:

- **byte**: The byte data type is an 8-bit data type. It has a minimum value of -128 and a maximum value of 127 . The default value is 0.
- **short**: The short data type is a 16-bit data type. It has a minimum value of -32,768 and a maximum value of 32,767 . The default value is 0.
- **int**: The int data type is a 32 bit data type. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). This data type will most likely be large enough for the numbers our

program will use, but if we need a wider range of values, use long instead. The default value is 0.

- **long**: The long data type is a 64-bit data type. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 . Use this data type when we need a range of values wider than those provided by int. The default value is 0L.
- **float**: The float data type is a single-precision 32-bit data type. As with the recommendations for byte and short, use a float (instead of double) if we need to save memory in large arrays of floating point numbers .The default value is 0.0f.
- **double**: The double data type is a double-precision 64-bit data type., For decimal values, this data type is generally the default choice.
- **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions.
- **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of u0000' (or 0) and a maximum value of 'uffff' (or 65535).

Q.6 What are the different operators in Java.

Ans. Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

THE SIMPLE ASSIGNMENT OPERATOR

One of the most common operators is the simple assignment operator "=". it assigns the value on its right to the operand on its left:

```
int abc = 0;
```

THE ARITHMETIC OPERATORS

The Java programming language provides operators that perform addition, subtraction, multiplication, division and Remainder.

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

Eg

```
class abc {
  public static void main (String[] args){
            // result is now 3
int result = 1 + 2;
```

```
System.out.println(result);
            // result is now 2
 result = result - 1;
System.out.println(result);
            // result is now 4
result = result * 2;
System.out.println(result);
            // result is now 2
result = result / 2;
System.out.println(result);
            // result is now 10
 result = result + 8;
            // result is now 3
result = result % 7;
System.out.println(result);
            }
}
```

We can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*. For example, x+=1; and x=x+1; both increment the value of x by 1.
The + operator can also be used for concatenating two strings together,

THE UNARY OPERATORS

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

THE EQUALITY AND RELATIONAL OPERATORS

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==", not "=", when testing if two primitive values are equal.

==	equal to
!=	not equal to

> greater than
>= greater than or equal to
< less than
<= less than or equal to

MCQ.

1. which of the following are primitive types?

(a) byte
(b) string
(c) integer
(d) float

2. what is the range of char type?

a) 0 to 216
b) 0 to 215
c) 0 to 216-1
d) 0 to 215-1

3. converting of primitive types to objects can be exciplicty?

a) true
b) flase

4. what is the value of 111%13?

a) 3
b) 5
c) 7
d) 9

5.what is the result of expression 5.45+"3.2"?

a) the double value 8.6
b) the string ""8.6"
c) the long value 8
d) the string "5.453.2"

1. (a)	2. (d)	3. (b)	4. (c)	5. (d)

THE CONDITIONAL OPERATORS

The && and || operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

Eg.

```
class Conditional {
   public static void main(String[] args){
 int value1 = 1;
 int value2 = 2;
 if((value1 == 1) && (value2 == 2))
  System.out.println("value1 is 1 AND value2 is 2");
if((value1 == 1) || (value2 == 1))
  System.out.println("value1 is 1 OR value2 is 1");
          }
}
```

TERNARY OPERATOR

? This operator is also known as the ternary operator because it uses three operands. It is shorthand for an if-then-else statement In the following example, this operator should be read as: "If someCondition is true, assign the value of value1 to result. Otherwise, assign the value of value2 to result."

THE TYPE COMPARISON OPERATOR INSTANCEOF

The instanceof operator compares an object to a specified type.We can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

BITWISE AND BIT SHIFT OPERATORS

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators are less commonly used.

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand and the number of positions

to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.
The bitwise & operator performs a bitwise AND operation.
The bitwise ^ operator performs a bitwise exclusive OR operation.
The bitwise | operator performs a bitwise inclusive OR operation.
The following program, Bit Demo, uses the bitwise AND operator to print the number "2" to standard output.

Q.7 What are the control statements in java

Ans. Control Statements

The control statement are used to control the flow of execution of the program . This execution order depends on the supplied data values and the conditional logic. Java contains the following types of control statements:

1- Selection Statements
2- Repetition Statements
3- Branching Statements

SELECTION STATEMENTS:

1. **If Statement:**

This is a control statement to execute a single statement or a block of code, when the given condition is true and if it is false then it skips **if** block and rest code of program is executed .

Syntax:

```
if(conditional_expression){
<statements>;
...;
...;
}
```

2. **If-else Statement:**

The **"if-else"** statement is an extension of if statement that provides another option when 'if' statement evaluates to "false" i.e. else block is executed if **"if"** statement is false.

Syntax:

```
if(conditional_expression){
<statements>;
...;
...;
}
else{
<statements>;
....;
....;
}
```

3. **Switch Statement:**

This is an easier implementation to the if-else statements. The keyword **"switch"** is followed by an expression that should evaluates to byte, short, char or int primitive data types ,only. In a switch block there can be one or more labeled cases. The expression that creates labels for the case must be unique. The switch expression is matched with each case label. Only the matched case is executed ,if no case matches then the default statement (if present) is executed.

Syntax:

```
switch(control_expression){
case expression 1:
<statement>;
case expression 2:
<statement>;
...
...
case expression n:
<statement>;
default:
<statement>;
}//end switch
```

REPETITION STATEMENTS:

1. **while loop statements:**

This is a looping or repeating statement. It executes a block of code or statements till the given condition is true. The expression must be evaluated to a boolean value. It continues testing the condition and executes the block of code. When the expression results to false control comes out of loop.

Syntax:

```
 while(expression){
<statement>;
...;
...;
}
```

2. **do-while loop statements:**

This is another looping statement that tests the given condition past so you can say that the do-while looping statement is a past-test loop statement. First the **do** block statements are executed then the condition given in **while** statement is checked. So in this case, even the condition is false in the first attempt, do block of code is executed at least once.

Syntax:

```
do{
<statement>;
...;
...;
}while (expression);
```

3. **for loop statements:**

This is also a loop statement that provides a compact way to iterate over a range of values. From a user point of view, this is reliable because it executes the statements within this block repeatedly till the specified conditions is true .

Syntax:

```
  for (initialization; condition; increment or decrement){
 <statement>;
 ...;
 ...;
 }
```

initialization: The loop is started with the value specified.
condition: It evaluates to either 'true' or 'false'. If it is false then the loop is terminated.
increment or decrement: After each iteration, value increments or decrements.

BRANCHING STATEMENTS:

1. **Break statements:**

The break statement is a branching statement that contains two forms: labeled and unlabeled. The break statement is used for breaking the execution of a loop (while, do-while and for) . It also terminates the switch statements.

Syntax:

```
 break;  // breaks the innermost loop or switch statement.
 break label;   // breaks the outermost loop in a series of nested loops.
```

2. **Continue statements:**

This is a branching statement that are used in the looping statements (while, do-while and for) to skip the current iteration of the loop and resume the next iteration .

Syntax:
continue;

3. **Return statements:**

It is a special branching statement that transfers the control to the caller of the method. This statement is used to return a value to the caller method and terminates execution of method. This has two forms: one that returns a value and the other that can not return. the returned value type must match the return type of method.

Syntax:
return;
return values;

return; //This returns nothing. So this can be used when method is declared with void return type.
return expression; //It returns the value evaluated from the expression.

1. Which four options describe the correct default values for array elements of the types indicated?

   1. int -> 0
   2. String -> "null"
   3. Dog -> null
   4. char -> '\u0000'
   5. float -> 0.0f
   6. boolean -> true

A.1, 2, 3, 4 B.1, 3, 4, 5
C.2, 4, 5, 6 D.3, 4, 5, 6

2. Which one of these lists contains only Java programming language keywords?
A.class, if, void, long, Int, continue

B. goto, instanceof, native, finally, default, throws
C. try, virtual, throw, final, volatile, transient
D. strictfp, constant, super, implements, do
E. byte, break, assert, switch, include

3. Which will legally declare, construct, and initialize an array?
A. int [] myList = {"1", "2", "3"};
B. int [] myList = (5, 8, 2);
C. int myList [] [] = {4,9,7,0};
D. int myList [] = {4, 3, 7};

4. Which is a valid keyword in java?
A. interface
B. string
C. Float
D. unsigned

5. Which three are legal array declarations?

1. int [] myScores [];
2. char [] myChars;
3. int [6] myScores;
4. Dog myDogs [];
5. Dog myDogs [7];

A. 1, 2, 4
B. 2, 4, 5
C. 2, 3, 4
D. All are correct.

.

6.
```
public void foo( boolean a, boolean b)
{
    if( a )
    {
        System.out.println("A"); /* Line 5 */
    }
    else if(a && b) /* Line 7 */
    {
        System.out.println( "A && B");
    }
```

```
        else /* Line 11 */
        {
            if ( !b )
            {
                System.out.println( "notB") ;
            }
            else
            {
                System.out.println( "ELSE" ) ;
            }
        }
    }
```

A.If *a* is true and *b* is true then the output is "*A && B*"
B.If *a* is true and *b* is false then the output is "*notB*"
C.If *a* is false and *b* is true then the output is "*ELSE*"
D.If *a* is false and *b* is false then the output is "*ELSE*"

.7.
```
switch(x)
{
    default:
        System.out.println("Hello");
}
```

Which two are acceptable types for x?

1. byte
2. long
3. char
4. float
5. Short
6. Long

A.1 and 3 B.2 and 4
C.3 and 5 D.4 and 6

8.
```
public void test(int x)
{
    int odd = 1;
    if(odd) /* Line 4 */
    {
        System.out.println("odd");
    }
    else
    {
        System.out.println("even");
    }
}
```

9.Which statement is true?
A.Compilation fails.
B."odd" will always be output.

C."even" will always be output.
D."odd" will be output for odd values of x, and "even" for even values.

10.
```
public class While
{
    public void loop()
    {
        int x= 0;
        while ( 1 ) /* Line 6 */
        {
            System.out.print("x plus one is " + (x + 1)); /* Line
8 */
        }
    }
}
```

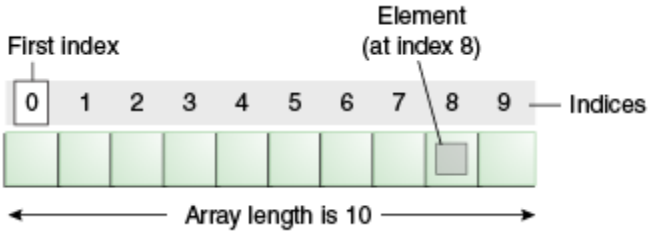
Which statement is true?
A.There is a syntax error on line 1.
B.There are syntax errors on lines 1 and 6.
C.There are syntax errors on lines 1, 6, and 8.
D.There is a syntax error on line 6.

1.B	2.B	3.D	4.B	5.A
6.A	7.C	8.A	9.A	10.D

ARRAYS

Q.8 How ARRAYS is define in Java.

Ans. An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You've seen an example of arrays already, in the main method of the "Hello World!" application. This section discusses arrays in greater detail.



An array of ten elements

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.
The following program, creates an array of integers, puts some values in it, and prints each value to standard output.

```
class ArrayDemo {
        public static void main(String[] args) {
 // declares an array of integers
        int[] anArray;
// allocates memory for 10 integers
        anArray = new int[10];
        // initialize first element
         anArray[0] = 100;
// initialize second element
 anArray[1] = 200;
// etc.
anArray[2] = 300;
anArray[3] = 400;
 anArray[4] = 500;
 anArray[5] = 600;
anArray[6] = 700;
 anArray[7] = 800;
 anArray[8] = 900;
 anArray[9] = 1000;
System.out.println("Element at index 0: "  + anArray[0]);
System.out.println("Element at index 1: " + anArray[1]);
System.out.println("Element at index 2: "  + anArray[2]);
System.out.println("Element at index 3: "   + anArray[3]);
 System.out.println("Element at index 4: "  + anArray[4]);
```

```
System.out.println("Element at index 5: "  + anArray[5]);
System.out.println("Element at index 6: "    + anArray[6]);
System.out.println("Element at index 7: "  + anArray[7]);
System.out.println("Element at index 8: " + anArray[8]);
System.out.println("Element at index 9: "  + anArray[9]);
        }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

Declaring a Variable to Refer to an Array

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

```
// declares an array of integers

int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as *type*[], where *type* is the data type of the contained elements; the square brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty).

You can also place the square brackets after the array's name:

```
float anArrayOfFloats[];
```

CREATING, INITIALIZING, AND ACCESSING AN ARRAY

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable.

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

If this statement were missing, the compiler would print an error like the following, and compilation would fail:
ArrayDemo.java:4: Variable anArray may not have been initialized.
The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
100, 200, 300,
400, 500, 600,
700, 800, 900, 1000
};
```

Here the length of the array is determined by the number of values provided between *{* and *}*.
We can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of square brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.
In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArray program:

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

```
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

Finally, you can use the built-in length property to determine the size of any array. The code

```
System.out.println(anArray.length);
```

will print the array's size to standard output.

Q.9 How to copy an array to another array.

Ans. COPYING ARRAYS

The System class has an arraycopy method that we can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

The two Object arguments specify the array to copy *from* and the array to copy *to*. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

CLASS & METHODS

Q.1 What is CLASS?

Ans. classes defined in the following way:

class *MyClass* {
// field, constructor, and
// method declarations
}

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. We can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

class *MyClass extends MySuperClass implements YourInterface* {
// field, constructor, and
// method declarations
}

It means that MyClass is a subclass of MySuperClass and that it implements the YourInterface interface.

We can also add modifiers like *public* or *private* which determine what other classes can access MyClass,

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*,
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, {}.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:

```
public int cadence;
public int gear;
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only public and private. Other access modifiers will be discussed later.

- Public modifier—the field is accessible from all classes.
- Private modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

```
public class Bicycle {
private int cadence;
private int gear;
 private int speed;

public Bicycle(int startCadence, int startSpeed, int startGear) {
gear = startGear;
 cadence = startCadence;
speed = startSpeed;
}

public int getCadence() {
return cadence;
```

```
        }

public void setCadence(int newValue) {
cadence = newValue;
}

public int getGear() {
return gear;
 }

public void setGear(int newValue) {
gear = newValue;
}

public int getSpeed() {
return speed;
}

public void applyBrake(int decrement) {
speed -= decrement;
}

public void speedUp(int increment) {
speed += increment;
}
}
```

Defining Methods

A typical method declaration:

```
public double calculate Answer (double wing Span, int number Of Engines,
            double length, double grossTons) {
  //do the calculation here
  }
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private.

2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Definition:

Two of the components of a method declaration comprise the method signature—the method's name and the parameter types.

The signature of the method declared above is:
calculateAnswer(double, int, double, double)

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to method overloading.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists .

Suppose we have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, we can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
 ...
 public void draw(String s) {
...
 }
public void draw(int i) {
...
}
public void draw(double f) {
 ...
}
public void draw(int i, double f) {
...
}
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Providing Constructors for our Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear)
```

```
{
gear = startGear;
cadence = startCadence;
speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
gear = 1;
cadence = 10;
speed = 0;
  }
```

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which *does* have a no-argument constructor.

.

INHERITANCE

Q.1 Explain INHERITANCE.

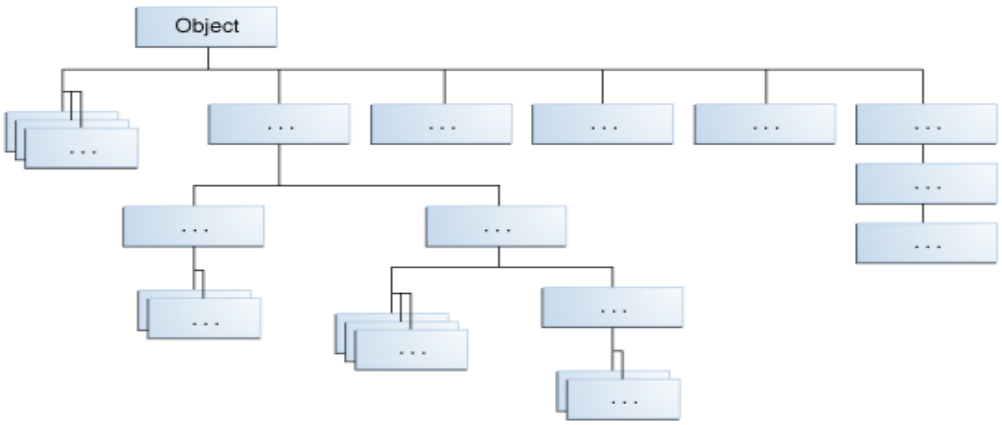
Ans. **Definitions:** A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). The class from which the subclass is derived is called a superclass (also a base class or a parent class).

Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to Object.

The Java Platform Class Hierarchy

The Object class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that we write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

An Example of Inheritance

Code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:

```
public class Bicycle {
 // the Bicycle class has
 // three fields
 public int cadence;
public int gear;
 public int speed;
// the Bicycle class has
 // one constructor
public Bicycle(int startCadence, int startSpeed, int startGear) {
gear = startGear;
cadence = startCadence;
speed = startSpeed;
}

 // the Bicycle class has
 // four methods
 public void setCadence(int newValue) {
cadence = newValue;
}

 public void setGear(int newValue) {
gear = newValue;
 }

 public void applyBrake(int decrement) {
speed -= decrement;
}

public void speedUp(int increment) {
 speed += increment;
 }

}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {

// the MountainBike subclass adds one field
 public int seatHeight;
```

```
// the MountainBike subclass has one constructor
public MountainBike(int startHeight,int startCadence,int
startSpeed,int startGear)
{
super(startCadence, startSpeed, startGear);
seatHeight = startHeight;
}

// the MountainBike subclass adds one method
public void setHeight(int newValue) {
seatHeight = newValue;
}
}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the Bicycle class were complex and had taken substantial time to debug.

Q.2 What we Can Do in a Subclass

Ans. A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. We can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Private Members in a Super class

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

INTERFACES

Q.1 What are Interfaces?

Ans. Interfaces in Java

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated they can only be *implemented* by classes or *extended* by other interfaces.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {
 // constant declarations, if any
// method signatures
// An enum with values RIGHT, LEFT
int turn(Direction direction,
double radius,
double startSpeed,
double endSpeed);
int changeLanes(Direction direction, double startSpeed, double endSpeed);
int signalTurn(Direction direction,boolean signalOn);
int getRadarFront(double distanceToCar,double speedOfCar);
int getRadarRear(double distanceToCar, double speedOfCar);
  .....
 // more method signatures
}
```

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

```
public class OperateBMW760i implements OperateCar
{
// the OperateCar method signatures, with implementation --
// for example:
 int signalTurn(Direction direction, boolean signalOn)
{
 // code to turn BMW's LEFT turn indicator lights on
```

```
 // code to turn BMW's LEFT turn indicator lights off
// code to turn BMW's RIGHT turn indicator lights on
// code to turn BMW's RIGHT turn indicator lights off
 }
// other members, as needed -- for example, helper classes not
// visible to clients of the interface
}
```

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

Interfaces as APIs

The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

Interfaces and Multiple Inheritance

Interfaces have another very important role in the Java programming language. Interfaces are not part of the class hierarchy, although they work in combination with classes. The Java programming language does not permit multiple inheritance (inheritance is discussed later in this lesson), but interfaces provide an alternative.

In Java, a class can inherit from only one class but it can implement more than one interface. Therefore, objects can have multiple types: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface. This is discussed later in this lesson, in the section titled "Using an Interface as a Type."

Defining an Interface

An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
// constant declarations
// base of natural logarithms
double E = 2.718282;
// method signatures
 void doSomething (int i, double x);
 int doSomethingElse(String s);
}
```

The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

The Interface Body

The interface body contains method declarations for all the methods included in the interface. A method declaration within an interface is followed by a semicolon, but no braces, because an interface does not provide implementations for the methods declared within it. All methods declared in an interface are implicitly public, so the public modifier can be omitted.

An interface can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, these modifiers can be omitted.

PACKAGE

Q.1 Explain Java Package.

Ans. JAVA PACKAGE

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 group of classes/interfaces etc. It is a good practice to group related classes implemented by you so that a programmers can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classed.

Creating a package:

When creating a package, we should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Example:

Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

Put an interface in the package *animals*:

```
/* File name : Animal.java */
package animals;
```

```
interface Animal {
  public void eat();
  public void travel();
}
```

Now put an implementation in the same package *animals*:

```
package animals;

/* File name : MammalInt.java */
public class MammalInt implements Animal{

  public void eat(){
    System.out.println("Mammal eats");
  }

  public void travel(){
    System.out.println("Mammal travels");
  }

  public int noOfLegs(){
    return 0;
  }

  public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

Now we compile these two files and put them in a sub-directory called **animals** and try to run as follows:

```
$ mkdir animals
$ cp Animal.class  MammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travels
```

MCQ.

1. A top level class may have only the following access modifier.

a) package
b) friendly
c) private
d) protected
e) public

2. A class is a template for multiple objects with similar features

a) true
b) false

3. which of the following feature are common to both java and c++?

a) the class declaratin
b) the access modifier
c) the encapsulation of data and method with in object
d) all of the above

4. because finalize() belongs to the java.lang. object class, it is present in all____

a) Object
b) Classes
c) Methods
d) None of above

5.An interface contains _____ methods

a) Non abstract
b) Implemented
c) Unimplemented
d) Extends

6.by default,all program import the java.lang package

a) True
b) False

7.java compiler stores the .class files in the path specified in CLASSPATH environmental variable

a) True
b) False

8.user defined package can also be imported just like the standard packages

a) True
b) False

9.can u achieve multiple interface through interface?

a) True
b) False

10. interfaces can't be executed.
a) True
b) False

1. (e)	2. (a)	3. (d)	4.(b)	5. (c)
6. (a)	7. (b)	8. (a)	9. (a)	10. (b)

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding byte code resides.

Q.2 What do you mean by exceptional handling?

Ans. Exceptions:

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

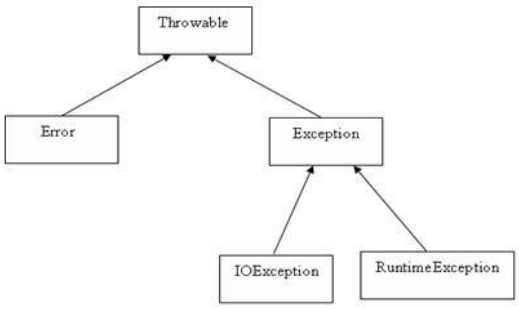
- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compliation.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses : IOException class and RuntimeException Class.



Exceptions Methods:
Following is the list of important medthods available in the Throwable class.

SN	Methods with Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage()
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  //Protected code
}catch(ExceptionName
e1)
{
  //Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

  public static void main(String args[]){
    try{
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This would produce following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
```

Out of the block

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}
```

The previous statements demonstrate three catch blocks, but we can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

```
try
{
  file = new FileInputStream(fileName);
  x = (byte) file.read();
}catch(IOException i)
{
  i.printStackTrace();
  return -1;
```

```
}catch(FileNotFoundException f) //Not valid!
{
  f.printStackTrace();
  return -1;
}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.
The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
  public void deposit(double amount) throws RemoteException
  {
    // Method implementation
    throw new RemoteException();
  }
  //Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
import java.io.*;
public class className
{
  public void withdraw(double amount) throws RemoteException,
                    InsufficientFundsException
  {
     // Method implementation
  }
//Remainder of class definition  }
```

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}finally
{
  //The finally block always executes.
}
```

Q.3 Explain Multithreading in Java.

Ans. 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.

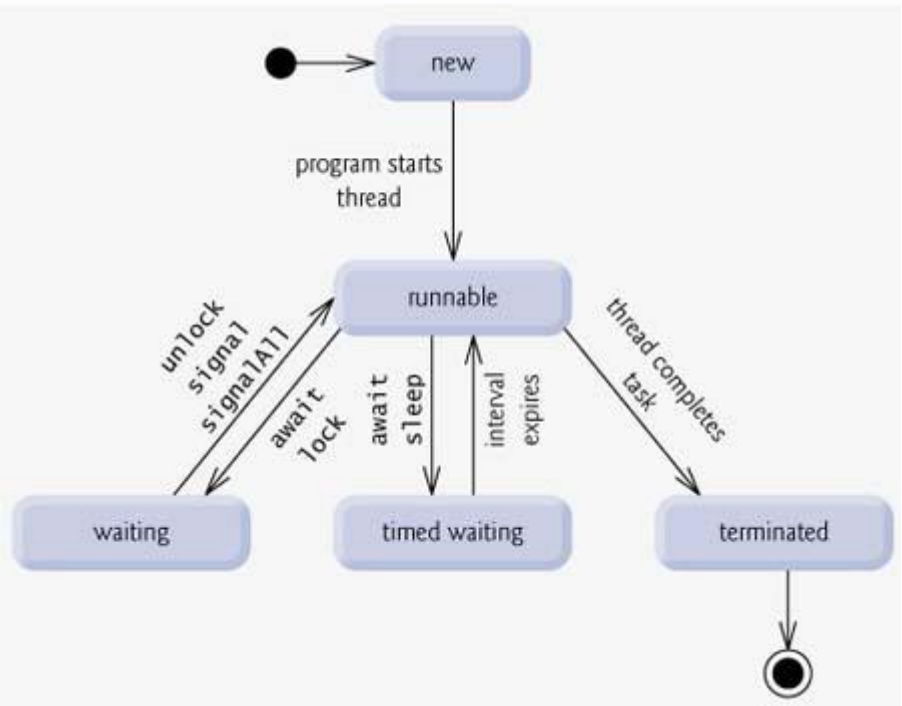
A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

Another term related to threads: **process:** A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Creating a Thread:

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable, a class need only implement a single method called **run ( )**, which is declared like this:

public void run( )

We will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.
After we create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName);

Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.
After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread. The start( ) method is shown here:

void start( );

Example:
Here is an example that creates a new thread and starts it running:

```
// Create a new thread.
class NewThread implements Runnable {
  Thread t;
  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.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);
        // Let the thread sleep for a while.
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
```

```
    }
    System.out.println("Exiting child thread.");
  }
}

class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

This would produce following result:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Create Thread by Extending Thread:

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.

The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

Example:

Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    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);
                    // Let the thread sleep for a while.
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
```

```
}
```

This would produce following result:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Q.4 Explain Synchronization and Messaging.

Ans. Because multithreading introduces an asynchronous behavior to programs, there must be a way for us to enforce synchronicity when need it. For example, if we want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That s, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of inter process synchronization: the *monitor.* The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Messaging

After divide our program into separate threads, you need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a

synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

Q.5 I/O Basics

Ans. aside from print( ) and println( ),none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are graphically oriented applets that rely upon Java's Abstract Window Toolkit (AWT) for interaction with the user. Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not very important to Java programming. The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master.

MCQ.

1. the finally block is executed when an exception is thrown.even if no catch matches it

a) true
b) false

2. the subclass exception should precede the base class exception when used within the class the catch clause

a) true
b) false

3. exception can be caught or re thrown to a calling method

a) true
b) false

4. the statement following
the throw

keyword in a program are not executed

a) true
b) false

5. thr STRING() method in the user-defined exception class is overridden

a) true
b) false

6.the word synchronized can be used with only a method

a) true
b) false

7.the suspend() method is used to terminate a thread

a) true
b) false

8. the run() method should necessary exists in classes created as subclass of thread

a) true
b) false

9. when two thread are waiting on each other and cant proceed the program is said to be in a deadlock?

a) true
b) false

10.which of the following is true?

1. wait(),notify(), notifyall() are defined as final & can be called only from with a synchronized method
2.among wait(),notify(),notifyall(),the wait() method only throw IOException
3. wait(),notify(),notifyall() & sleep() are method of object class

a) 1
b) 2

c) 3
d) 1&2
e) 1,2&3

1.(A)	2.(A)	3.(A)	4. (A)	5. (A)
6. (B)	7. (B)	8. (A)	9.(A)	10.(D)

APPLET

Q.1 Explain Applet in Java.

Ans. 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.

Life Cycle of an Applet:

Four methods in the Applet class give you the framework on which you build any serious applet:

- **In it:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **Start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **Stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **Destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you

should not normally leave resources behind after a user leaves the page that contains the applet.

- **Paint:** Invoked immediately after the start () method, and also any time the applet needs to repaint itself in the browser. The paint () method is actually inherited from the java.awt.

A "Hello, World" Applet:

The following is a simple applet named HelloWorldApplet.java:

```
Import java. Applet.*;
Import java.awt.*;

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

These import statements bring the classes into the scope of our applet class:

- java.applet.Applet.
- java.awt.Graphics.

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet CLASS:

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following:

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may:

- request information about the author, version and copyright of the applet
- request a description of the parameters the applet recognizes
- initialize the applet
- destroy the applet
- start the applet's execution
- stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.
The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet:

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.
The <applet> tag is the basis for embedding an applet in an HTML file. example that invokes the "Hello, World" applet:

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="HelloWorldApplet.class" width="320" height="120">
If your browser was Java-enabled, a "Hello, World"
message would appear here.
</applet>
<hr>
</html>
```

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with a </applet> tag.
If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.
The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown:

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class" width="320" height="120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example:

```
<applet code="mypackage.subpackage.TestApplet.class"
  width="320" height="120">
```

Getting Applet Parameters:
The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color.
The second color and the size of each square may be specified as parameters to the applet within the document.
CheckerApplet gets its parameters in the init() method. It may also get its parameters in the paint() method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.
The applet viewer or browser calls the init() method of each applet it runs. The viewer calls init() once, immediately after loading the applet. (Applet.init() is implemented to do nothing.) Override the default implementation to insert custom initialization code.
The Applet.getParameter() method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.
The following is a skeleton of CheckerApplet.java:

```
import java.applet.*;
import java.awt.*;
public class CheckerApplet extends Applet
{
  int squareSize = 50;// initialized to default size
  public void init () {}
```

```
  private void parseSquareSize (String param) {}
  private Color parseColor (String param) {}
  public void paint (Graphics g) {}
}
```

Here are CheckerApplet's init() and private parseSquareSize() methods:

```
public void init ()
{
  String squareSizeParam = getParameter ("squareSize");
  parseSquareSize (squareSizeParam);
  String colorParam = getParameter ("color");
  Color fg = parseColor (colorParam);
  setBackground (Color.black);
  setForeground (fg);
}
private void parseSquareSize (String param)
{
  if (param == null) return;
  try {
    squareSize = Integer.parseInt (param);
  }
  catch (Exception e) {
   // Let default value remain
  }
}
```

The applet calls parseSquareSize() to parse the squareSize parameter. parseSquareSize() calls the library method Integer.parseInt(), which parses a string and returns an integer. Integer.parseInt() throws an exception whenever its argument is invalid.
Therefore, parseSquareSize() catches exceptions, rather than allowing the applet to fail on bad input.
The applet calls parseColor() to parse the color parameter into a Color value. parseColor() does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet works.

Specifying Applet Parameters:

The following is an example of an HTML file with a CheckerApplet embedded in it. The HTML file specifies both parameters to the applet by means of the <param> tag.

```
<html>
<title>Checkerboard Applet</title>
<hr>
<applet code="CheckerApplet.class" width="480" height="320">
<param name="color" value="blue">
<param name="squaresize" value="30">
</applet>
<hr>
</html>
```

Note: Parameter names are not case sensitive.

Application Conversion to Applets:

It is easy to convert a graphical Java application (that is, an application that uses the AWT and that you can start with the java program launcher) into an applet that we can embed in a web page.

Here are the specific steps for converting an application to an applet.

1. Make an HTML page with the appropriate tag to load the applet code.
2. Supply a subclass of the JApplet class. Make this class public. Otherwise, the applet cannot be loaded.
3. Eliminate the main method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.
4. Move any initialization code from the frame window constructor to the init method of the applet. You don't need to explicitly construct the applet object.the browser instantiates it for you and calls the init method.
5. Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.
6. Remove the call to setDefaultCloseOperation. An applet cannot be closed; it terminates when the browser exits.
7. If the application calls setTitle, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML title tag.)
8. Don't call setVisible(true). The applet is displayed automatically.

Q.2 Explain Event Handling in Java

Ans. Applets inherit a group of event-handling methods from the Container class. The Container class defines several methods, such as processKeyEvent and processMouseEvent, for handling particular types of events, and then one catch-all method called processEvent.
Inorder to react an event, an applet must override the appropriate event-specific method.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet
                    implements MouseListener {

  StringBuffer strBuffer;

  public void init() {
 addMouseListener(this);
 strBuffer = new StringBuffer();
    addItem("initializing the apple ");
  }

  public void start() {
    addItem("starting the applet ");
  }

  public void stop() {
    addItem("stopping the applet ");
  }

  public void destroy() {
    addItem("unloading the applet");
  }

  void addItem(String word) {
    System.out.println(word);
    strBuffer.append(word);
    repaint();
```

```
    }

    public void paint(Graphics g) {
    //Draw a Rectangle around the applet's display area.
       g.drawRect(0, 0,
                  getWidth() - 1,
                  getHeight() - 1);

    //display the string inside the rectangle.
       g.drawString(strBuffer.toString(), 10, 20);
    }


    public void mouseEntered(MouseEvent event) {
    }
    public void mouseExited(MouseEvent event) {
    }
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }

    public void mouseClicked(MouseEvent event) {
   addItem("mouse clicked! ");
    }
}
```

Now let us call this applet as follows:

```
<html>
<title>Event Handling</title>
<hr>
<applet code="ExampleEventHandling.class"
width="300" height="300">
</applet>
<hr>
</html>
```

Initially the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle "mouse clicked" will be displayed as well.
Based on the above examples, here is the live applet example: Applet Example.

Displaying Images:
An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within the applet, you use the drawImage() method found in the java.awt.Graphics class.
Following is the example showing all the steps to show images:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class ImageDemo extends Applet
{
  private Image image;
  private AppletContext context;
  public void init()
  {
     context = this.getAppletContext();
     String imageURL = this.getParameter("image");
     if(imageURL == null)
     {
        imageURL = "java.jpg";
     }
     try
     {
        URL url = new URL(this.getDocumentBase(), imageURL);
        image = context.getImage(url);
     }catch(MalformedURLException e)
     {
        e.printStackTrace();
        // Display in browser status bar
        context.showStatus("Could not load image!");
     }
  }
  public void paint(Graphics g)
  {
     context.showStatus("Displaying image");
```

```
    g.drawImage(image, 0, 0, 200, 84, null);
    g.drawString("www.javalicense.com", 35, 100);
  }
}
```

Now let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="300" height="200">
<param name="image" value="java.jpg">
</applet>
<hr>
</html>
```

Playing Audio:

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including:

- **public void play():** Plays the audio clip one time, from the beginning.
- **public void loop():** Causes the audio clip to replay continually.
- **public void stop():** Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is the example showing all the steps to play an audio:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet
{
  private AudioClip clip;
  private AppletContext context;
  public void init()
  {
    context = this.getAppletContext();
    String audioURL = this.getParameter("audio");
```

```
    if(audioURL == null)
    {
      audioURL = "default.au";
    }
    try
    {
      URL url = new URL(this.getDocumentBase(), audioURL);
      clip = context.getAudioClip(url);
    }catch(MalformedURLException e)
    {
      e.printStackTrace();
      context.showStatus("Could not load audio file!");
    }
  }
  public void start()
  {
    if(clip != null)
    {
      clip.loop();
    }
  }
  public void stop()
  {
    if(clip != null)
    {
      clip.stop();
    }  }  }
```

Now let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="0" height="0">
<param name="audio" value="test.wav">
</applet>
<hr>
```


STRING

Q.1 What is a String?

Ans. In java the string data type is now an object type. The class name is starts with an uppercase. A string is anything and everything grouped together between 2 double quotes. For example, "Hello world" is one string while "sdad anesd ccn " is another.

Using Strings

Strings can be manipulated in numerous ways. The first is with some member functions of the string class. Let's see an example first before we continue.

Different String functions

```
public class StringExample{

  public static void main(String args[]){

 String word;

  //assign the string to the variable:
 word = "Alexander";

 //preform some actions on the string:

 //1. retrieve the length by calling the
 //length method:
int length = word.length();
 System.out.println("Length: " + length);

//2. use the case functions:
 System.out.println("toUpperCase: " + word.toUpperCase());
 System.out.println("toLowerCase: " + word.toLowerCase());

//3. use the trim function to eliminate leading
 //or trailing white spaces:
 word = word.trim();
System.out.println("trim: " + word);

  //4. check for a certain character using indexOf()
  System.out.println("indexOf('s'): " + word.indexOf('s'));


//5. print out the beginning character using charAt()
```

```
System.out.println("first character: " + word.charAt(0));

//6. make the string shorter
 word = word.substring(0, 4);
System.out.println("shorter string: " + word);
  }
  }
```

When WE run the above program, here is the output that you will get:

```
Length: 9
toUpperCase: ALEXANDER
toLowerCase: alexander
trim: Alexander
indexOf('s'): -1
first character: A
shorter string: Alex
```

A lot certainly has happened in this program. Let's observe some of the most used functions in the String class.

int length()

The length function will simply return the length of a string as an integer.

String toUpperCase()

The toUpperCase function will return the uppercase version of a string. Say you have a string "Welcome". This function will return "WELCOME".

String toLowerCase()

The toLowerCase function will return the lowercase version of a string. Say you have a string "Welcome TO Earth". This function will return "welcome to earth".

String trim()

The trim function will return the string without leading or trailing white space characters. Say that a string was " hello ". There are 4 spaces in the front and 3 spaces at the end. The trim function would make this "hello".

int indexOf(int ch)
int indexOf(int ch, int begin)
int indexOf(String ch)
int indexOf(String ch, int begin)

Notice that there are 4 different functions listed here. All of them preform the same overall action of returning an integer, which represents the FIRST OCCURRANCE of a character or String contained in that string. So say that we have the string "Hello" and we say

indexOf('l'). This function will use the first function above and return 2. Notice it doesn't reutrn 3.

We can also say, using the same string "Hello", indexOf("He"). It will return 0 as the string that you are searching for starts at index 0.

By default, if a string or character is not found in the string, the any of the functions will return -1.

char charAt(int index)

This function will look for a character at a specific index. It will return that character if it is found. Say we have the string "Hello" and we say charAt(4). It will return 'o' as that character is at index 4.

String substring(int begin)

String substring(int begin, int end)

Either of these functions will shorten a string when called. If no ending index is specified, it will return the rest of that string. Say we have the string "Hello there" and we say substring(4), the function will return "o there". Let's use substring(2,5), the function will return "llo".

MCQ.

1. which of the following is not a wrapper class?

a) **string**
b) **integer**
c) **boolean**
d) **character**

2.which of the following method are methods of string class?

a) **delete()**
b) **append()**
c) **reverse()**
d) **replace()**

3.
which of the following methods cause string object referenced by s to be changed?

a) **s.concat()**
b) **s.toUpperCase()**
c) **s.replace()**
d) **s.valueOf()**

4. string is a wrapper class

a) true
b) false

5.all applet are subclasses of applet
a) true
b) false

6.applet's getParameter() method can be used to get parameter values

a) true
b) false

7.what tags are mandatory when creating html to display an applet
a) name,height,width
b) code,name
c) codebase,height,width
d) code,height,width

8.all applet must import java.applet and java.awt

a) true
b) false

9.every color is created from RGB value

a) true
b) false

10. what is the default layout manager for panels and applets
a) FlowLayout
b) GridLayout
c) BorderLayout
d) CardLayout

1(a)	2(c)	3(d)	4(b)	5 (a)
6.(a)	7.(d)	8.(a)	9 (a)	10 (a)

JAVA BEANS

Q.1 What is Java Beans?

Ans. "A Java Bean is a reusable software component that can be manipulated visually in a builder tool".

Software components are self-contained software units developed according to the motto "Developed them once, run and reused them everywhere".

Builder tools allow a developer to work with JavaBeans in a convenient way. By ex- amining a JavaBean by a process known as Introspection, a builder tool exposes the discovered features of the JavaBean for visual manipulation. A builder tool maintains a list of all JavaBeans available. It allows us to compose the Bean into applets, application, servlets and composite components (e.g. a JFrame), customize its behavior and appearance by modifying its properties and connect other components to the event of the Bean or vice versa.

JavaBeans can appear in two forms: visual and non visual. Majority of JavaBeans are extensions of AWT or Swing components. Some Beans may be simple GUI elements such as a specialized button or slider. Others may be sophisticated visual software components such as a diagram component offering different ways to present the bounded data.

Q.2 What are the Advantages of Java Beans ?

Ans. Advantages of Java Beans

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.

Q.3 What are the basic rules of Java Beans?

Ans. Java Beans basic rules

A Java Bean should:
Be public
Implement the Serializable interface
Have a no-arg constructor
Be derived from javax.swing.J Component or java.awt.Component if it is visual

Q.4 What is Bean Developer Kit (BDK)?

Ans. The Bean Developer Kit (BDK) is a simple tool that enables us to create, configure, and connect a set of Beans. There is also a set of sample Beans with their source code. This section provides step-by-step instructions for installing and using this tool.

Q.5 What are JAR file?

Ans. A JAR file allows us to efficiently deploy a set of classes and their associated resources. For example, a developer may build a multimedia application that uses various sound and image files. A set of Beans can control how and when this information is presented. All of these pieces can be placed into one JAR file.JAR technology makes it much easier to deliver and install software.

Q.6 What is Instrospection ?

Ans. We know builder tools provide a property sheet where we can conveniently set the properties of a JavaBean component or connect event listeners to an exposed event. In order to provide this service a builder tool needs to examine the component for its features like properties, events and methods. This process is referred to as "introspection". To obtain information about a specific JavaBean we can use the static getBeanInfo() method of the Introspector class. This method returns an instance of the BeanInfo class, which describes all features a JavaBean exposes.

Q.7 What are the Properties of Java Beans?

Ans. Properties are named public attributes which determine the internal state of a JavaBean and thus its behavior and appearance. For example, a GUI textfield might have a property named 'maxlength' which restricts the number of characters one can insert into the texfield. All types of property have in common that they are characterized by a pair of set/get methods. A getter method is used to read the value of a readable property. To update the

property's value a setter method has to be called. different kinds of properties a JavaBean can expose:

Simple Properties

As the name simple properties are the simplest among the properties. Eg. For our FontSelector we would like to have a default font size, which will be used as initial font size at runtime.

The following code hows how simple property is defined:

```
private String name = " FontSelector ";
public void setName ( String name ){
this . name = name ;
}

public String getName (){
 return name ;
}
```

Indexed Properties

If a simple property can hold an array of value they are no longer called simple but instead indexed properties. It can be identified by the following design patterns, where N is the name of the property and T is its type:

```
public T getN(int index);
public void setN(int index, T value);
public T[ ] getN( );
```

Q.8 What are the sreps to develop simple bean using BDK

Ans. Create a New Bean

the steps to create a new Bean:

1. Create a directory for the new Bean.
2. Create the Java source file(s).
3. Compile the source file(s).
4. Create a manifest file.
5. Generate a JAR file.
6. Start the BDK.
7. Test.

Q.9 Bound Properties and Constrained Properties

Ans. **Bound Properties**

A Bean that has a bound property generates an event when the property is changed.The event is of type PropertyChangeEvent and is sent to objects that previously registered an interest in receiving such notifications. Bound properties notify other components of their changed value. JavaBeans follow the Observer pattern to do this notification.

Constrained Properties

A Bean that has a constrained property generates an event when an attempt is made to change its value. The event is of type PropertyChangeEvent. It is sent to objects that previously registered an interest in receiving such notifications. Those other objects have the ability to veto the proposed change. This capability allows a Bean to operate differently according to its run-time environment.

Q.10 What is Persistence ?

Ans. "Persistence is the ability of an object to store its state." JavaBeans use the Java Serialization API to gain this ability. The simplest way to enable serialization of a Bean is by implementing the Serializable interface. The points we have to consider when declaring a Bean to be serializable are the same as for any other serializable object (e.g. how to deal with transient and static variables, whether a validation is required upon deserialization etc.). The greatest strength of persistence regarding JavaBeans lies in the ability to create prototypes a new JavaBean can be instantiated from. For example,

A Java application using our FontSelector Bean can serialize the Bean on a Microsoft Windows machine, the serialized file can be sent to a Linux machine, where another Java application can instantiate a new Bean with the exact state (same fonts, fontsize etc.) its counterpart had on the Windows machine. Although serialization is a convenient way to persist a JavaBean this method has a downside. Serialization is intended to persist an object for a short time, for example to transfer the object to a remote machine through Java RMI (Remote Machine Invocation). But JavaBeans are typically persisted to serve as a prototype that requires a mechanism for long-term persistence.

Q.11 What is Customizers

Ans. The Properties window of the BDK allows a developer to modify the properties of a Bean. However, this may not be the best user interface for a complex component with many interrelated properties. Therefore, a Bean developer can provide a customizer that helps another developer configure the bean. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the market place.

SERVLETS

Q.1 What do you mean by servlets?

Ans. Servlets are small programs that execute on the serverside of a Web connection. Just as applets dynamically extend the functionality of a Web browser, servlets dynamically extend the functionality of a Web server.

Q.2 What are the advantages of servlets ?

Ans. Servlets offer several advantages in comparison with CGI.

- It performance is significantly better. Servlets execute within the address space of a Web server. It is not necessary to create a separate process to handle each client request.
- Servlets are platform-independent because they are written in Java. A number of Web servers from different vendors offer the Servlet API. Programs developed for this API can be moved to any of these environments without recompilation.
- The Java security manager on the server enforces a set of restrictions to protect the resources on a server 950 Java™2:The Complete Reference machine. We see that some servlets are trusted and others are untrusted. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Q.3 Explain the life cycle of servlets.

Ans. A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet

- The servlet is initialized by calling the **init ()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

The init() method :

The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.
When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.
The init method definition looks like this:

```
public void init() throws ServletException {
// Initialization code...
  }
```

The service () method:

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.
Here is the signature of this method:

```
public void service(ServletRequest request,
    ServletResponse response)
throws ServletException, IOException{
}
```

The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.
The doGet() and doPost() are most frequently used methods with in each service request. Here are the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request,
```

```
                HttpServletResponse response)
        throws ServletException, IOException {
        // Servlet code
    }
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

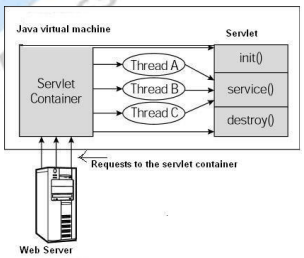
```
public void doPost(HttpServletRequest request,
    HttpServletResponse response)
throws ServletException, IOException {
// Servlet code
}
```

The destroy() method :

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
public void destroy() {
// Finalization code...
}
```



Q.4 What is Apache Tomcat ?

Ans. Apache Tomcat is an open source software implementation of the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed under the Java Community Process.

Apache Tomcat is developed in an open and participatory environment and released under the Apache License version 2. Apache

Tomcat is intended to be a collaboration of the best-of-breed developers from around the world. The last release of version 7.0.29 of Apache Tomcat.

Q.5 What are Servlet API

Ans. The Java Servlet API defines the interface between servlets and servers. This API is packaged as a standard extension to the JDK under javax:

Package javax.servlet
Package javax.servlet.http

The API provides support in four categories:

Servlet life cycle management
Access to servlet context
Utility classes
HTTP-specific support classes

Q.6 What are Package javax.servlet and Package javax.servlet.http.

Ans. The javax.servlet contains a no of interface and classes that establish the framework in which servlets operate.

Package javax.servlet

Interface

RequestDispatcher : Defines a request dispatcher object that receives request from the client and sends them to any resource (such as a servlet, CGI script, HTML file, or JSP file) available on the server.

Servlet : A Servlet is a small program that runs inside a web server.

ServletConfig: Defines an object that a servlet engine generates to pass configuration information to a servlet when such servlet is initialized.

ServletContext: A servlet engine generated object that gives servlets information about their environment.

ServletRequest: Defines a servlet engine generated object that enables a servlet to get information about a client request.

ServletResponse :Interface for sending MIME data from the servlet's service method to the client.

SingleThread: Model Defines a "single" thread model for servlet execution.

Class

GenericServlet :The GenericServlet class implements the Servlet interface and, for convenience, the ServletConfig interface.

ServletInputStream: An input stream for reading servlet requests, it provides an efficient readLine method.

ServletOutput : Stream An output stream for writing servlet responses.

Package javax.servlet.http

The javax.servlet.http contains a no of interface and classes that are commonly used servlet developers.

Interface

HttpServletRequest : An HTTP servlet request.

HttpServletResponse: An HTTP servlet response.

HttpSession: The HttpSession interface is implemented by services to provide an association between an HTTP client and HTTP server.

HttpSessionBindingListener : Objects implement this interface so that they can be notified when they are being bound or unbound from a HttpSession.

HttpSessionContext: Deprecated. The HttpSessionContext class has been deprecated for security reasons.

Class

Cookie :This class represents a "Cookie", as used for session management with HTTP and HTTPS protocols.

HttpServlet :An abstract class that simplifies writing HTTP servlets.

HttpSessionBindingEvent :This event is communicated to a HttpSessionBindingListener whenever the listener is bound to or unbound from a HttpSession value.

HttpUtils: A collection of static utility methods useful to HTTP servlets.

MCQ.

Q1.Which of the following are interface?
1.ServletContext
2.Servlet
3.GenericServlet
4.HttpServlet
A.1,2,3,4
B,1,2
C.1,3,4
D.1,4

Q2 1. Servlet is a Java technology based Web component.
2. Servlet servlets are platform-independent
3. Servlet has run on Web server which has a containers
4. Servlets interact with Web clients via a request/response using HTTP protocol.

A.1,2,3,4

B.1,2,3
C.1,3,4
D.None

Q3. Which of the following are class?
1.ServletContext
2.Servlet
3.GenericServlet
4.HttpServlet
A.1,2,3,4
B,1,2
C.3,4
D.1,4

Q4. Which of the following methods are main methods in life cycle of servlet?
1.init()
2 .service()
3.destroy()
4.srop()
5.wait()
A.1,2,3,4,5
B,1,2,3
C.3,4,5
D.1,4,5

Q5. init(),service() and destroy()methods are define in

1.javax.servlet.Servlet interface
2.javax.servlet.ServletHttp class
3.javax.servlet.ServletRequest interface
4.javax.servlet.ServletResponse interface

A.1,2,3,4,5
B,1
C.3,4,5
D.1,4,5

Q6. Int init() ,.service() and destroy() methods which methods are call only once at life cycle of servlets
1.init()
2.service()
3.destroy()
A.1,2,3
B,1,3

C.2
D.None

Q7. During initialization of servlet a servlet instance can throw

1.An UnavailableException
2 A ServletException
3.Both
4.None

Q8. Is Servlet thread Safe?
1.Yes
2.No
3.None

Q9. ServletContext is

1. an Interface
2.A container which is used to store an object so that it is available for whole application
3.A container which is used to store an object so that it is available for session only.
4.A container which is used to store an object so that it is available for request only.
A.1,2,3,4
B.1,2
C.1,3,4
D.4

Q10. A servlet can access the headers of an HTTP request through which following methods of the HttpServletRequest interface:
1.getHeader()
2 getHeaders()
3.getHeaderNames()
4.All
5.None

1. (b)	2. (a)	3. (c)	4. (b)	5. (b)
6. (b)	7. (c)	8. (b)	9. (b)	10. (d)

JDBC

Q.1 What is JDBC?

Ans. JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

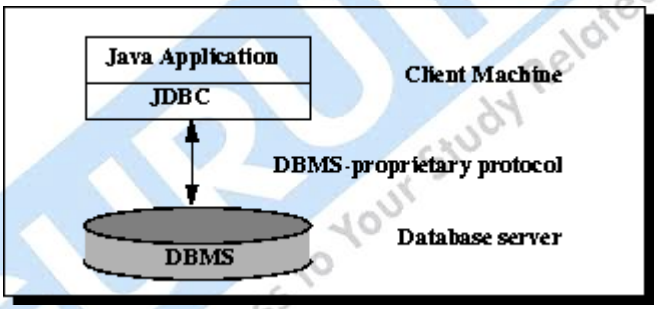
The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing that SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

JDBC Architecture:

The JDBC API supports both two-tier and three-tier processing models for database access.

Figure 1: Two-tier Architecture for Data Access.

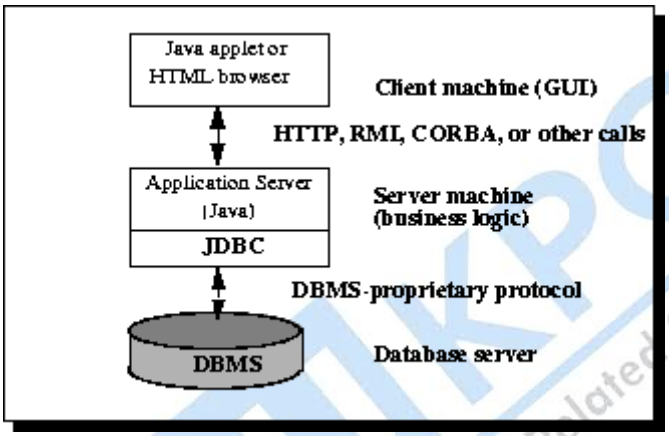


In the two-tier model, a Java application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the

middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 2: Three-tier Architecture for Data Access.



The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

Q.2 What are the JDBC API and JDBC Driver.

Ans. **JDBC API:** JDBC is a SQL-level API-one that allows us to execute SQL statements and retrieve the results, if any. The API itself is a set of interfaces and classes designed to perform actions against any database .This provides the application-to-JDBC Manager connection.

JDBC Driver: The JDBC API defines the Java interfaces and classes that programmers use to connect to databases and send queries. A JDBC driver implements these interfaces and classes for a particular DBMS vendor.

A Java program that uses the JDBC API loads the specified driver for a particular DBMS before it actually connects to a database. The JDBC DriverManager class then sends all JDBC API calls to the loaded driver.

The four types of JDBC drivers are:

- JDBC-ODBC bridge plus ODBC driver, also called Type 1.

Translates JDBC API calls into Microsoft Open Database Connectivity (ODBC) calls that are then passed to the ODBC driver.

The ODBC binary code must be loaded on every client computer that uses this type of driver.

- Native-API, partly Java driver, also called Type 2.
  Converts JDBC API calls into DBMS-specific client API calls. Like the bridge driver, this type of driver requires that some binary code be loaded on each client computer.
- JDBC-Net, pure Java driver, also called Type 3.
  Sends JDBC API calls to a middle-tier net server that translates the calls into the DBMS-specific network protocol. The translated calls are then sent to a particular DBMS.
- Native-protocol, pure Java driver, also called Type 4

Q.3 What are the JDBC Components?

Ans. Common JDBC Components:

The JDBC API provides the following interfaces and classes:

- **DriverManager:** This interface manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication subprotocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects
- **Connection :** Interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement :** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Q.4 What do you mean by Result Sets and Result Sets Metadata.

Ans. **Result Sets and Cursors**

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface DatabaseMetaData, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata.

1. How many JDBC driver types does Sun define?
   A.One B.Two
   C.Three D.Four

2. Where is metadata stored in MySQL?
   A.In the MySQL database *metadata*
   B.In the MySQL database *metasql*
   C.In the MySQL database *mysql*
   D.None of the above is correct.

3. Who invented Java?
   A.Netscape
   B.Microsoft

C.Sun
D.None of the above is correct.

4. To run a compiled Java program, the machine must have what loaded and running?
A.Java virtual machine
B.Java compiler
C.Java bytecode
D.A Web browser

5. Which JDBC driver Type(s) can be used in either applet or servlet code?
A.Both Type 1 and Type 2
B.Both Type 1 and Type 3
C.Both Type 3 and Type 4
D.Type 4 only
6. _____ is an open source DBMS product that runs on UNIX, Linux and Windows.
A.MySQL
B.JSP/SQL
C.JDBC/SQL
D.Sun ACCESS

7. What is sent to the user via HTTP, invoked using the HTTP protocol on the user's computer, and run on the user's computer as an application?
A.A Java application
B.A Java applet
C.A Java servlet
D.None of the above is correct.

8. What MySQL property is used to create a surrogate key in MySQL?
A.UNIQUE
B.SEQUENCE
C.AUTO_INCREMENT
D.None of the above -- Surrogate keys are not implemented in MySQL.

9. What is **not** true of a Java bean?
A.There are no public instance variables.
B.All persistent values are accessed using get*xxx* and set*xxx* methods.
C.It may have many constructors as necessary.
D.All of the above are true of a Java bean.

10. A JSP is transformed into a(n):
A.Java applet.
B.Java servlet.
C.Either 1 or 2 above.
D.Neither 1 nor 2 above.

D	C	C	A	C
A	B	C	C	B

Glossary of Java

Abstract Window Toolkit (AWT)

A collection of graphical user interface (GUI) components that were implemented using native-platform versions of the components. These components provide that subset of functionality which is common to all native platforms.

abstract class

A class that contains one or more *abstract methods*, and therefore can never be instantiated. Abstract classes are defined so that other classes can extend them and make them concrete by implementing the abstract methods.

actual parameter list

The arguments specified in a particular method call.

applet

A program written in the Java(TM) programming language to run within a web browser compatible with the Java platform, such as HotJava(TM) or Netscape Navigator(TM).

argument

A data item specified in a method call. An argument can be a literal value, a variable, or an expression.

array

A collection of data items, all of the same type, in which each item's position is uniquely designated by an integer.

Bean

A reusable software component. Beans can be combined to create an application.

break

A Java(TM) programming language keyword used to resume program execution at the statement immediately following the current statement. If followed by a label, the program resumes execution at the labeled statement.

bytecode

Machine-independent code generated by the Java(TM) compiler and executed by the Java interpreter.

catch

A Java(TM) programming language keyword used to declare a block of statements to be executed in the event that a Java exception, or run time error, occurs in a preceding "try" block.

class

In the Java(TM) programming language, a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicitly be Object.

client

In the client/server model of communcations, the client is a process that remotely accesses resources of a compute server, such as compute power and large memory capacity.

core class

A public class (or interface) that is a standard member of the Java(TM) Platform. The intent is that the core classes for the Java platform, at minimum, are available on all operating systems where the Java platform runs. A program written entirely in the Java programming language relies only on core classes, meaning it can run anywhere.

declaration

A statement that establishes an identifier and associates attributes with it, without necessarily reserving its storage (for data) or providing the implementation (for methods).

distributed

Running in more than one address space.

EmbeddedJava(TM) Technology

The availability of Sun's Java 2 Platform, Micro Edition technology under a restrictive license agreement that allows a licensee to leverage certain Java technologies to create and deploy a closed-box application that exposes no APIs.

exception handler

A block of code that reacts to a specific type of *exception*. If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed.

ormal parameter list

The parameters specified in the definition of a particular method

garbage collection

The automatic detection and freeing of memory that is no longer in use. The Java(TM) runtime system performs garbage collection so that programmers never explicitly free objects.

GUI

Graphical User Interface. Refers to the techniques involved in using graphics, along with a keyboard and a mouse, to provide an easy-to-use interface to some program.

hierarchy

A classification of relationships in which each item except the top one (known as the root) is a specialized form of the item above it. Each item can have one or more items below it in the hierarchy. In the Java(TM) class hierarchy, the root is the Object class.

import

A Java(TM) programming language keyword used at the beginning of a source file that can specify classes or entire packages to be referred to later without including their package names in the reference.

instance

An object of a particular class. In programs written in the Java(TM) programming language, an instance of a class is created using the new operator followed by the class name.

JAR file format

JAR (Java Archive) is a platform-independent file format that aggregates many files into one. Multiple applets written in the Java(TM) programming language, and their requisite components (.class files, images, sounds and other resource files) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction. It also supports file compression and digital signatures.

Java Application Environment (JAE)

The source code release of the Java Development Kit (JDK(TM)) software.

Java Database Connectivity (JDBC(TM))

An industry standard for database-independent connectivity between the Java(TM) platform and a wide range of databases. The JDBC(TM) provides a call-level API for SQL-based database access.

Java Development Kit (JDK(TM))

A software development environment for writing applets and applications in the Java programming language.

Java(TM) Platform

Consists of a language for writing programs ("the Java(TM) programming language"); a set of APIs, class libraries, and other programs used in developing, compiling, and error-checking programs; and a virtual machine which loads and executes the class files.

In addition, the Java platform is subject to a set of compatibility requirements to ensure consistent and compatible implementations. Implementations that meet the compatibility requirements may qualify for Sun's targeted compatibility brands.

The Java(TM) 2 platform is the current generation of the Java platform.

Java(TM) Runtime Environment (JRE)

A subset of the Java Development Kit (JDK(TM)) for end-users and developers who want to redistribute the runtime environment alone. The Java runtime environment consists of the Java virtual machine*, the Java core classes, and supporting files.

Java(TM) virtual machine (JVM)*

Sun's specification for or implementation of a software "execution engine" that safely and compatibly executes the byte codes in Java class files on a microprocessor (whether in a computer or in another electronic device).

multithreaded

Describes a program that is designed to have parts of its code execute concurrently.

object-oriented design

A software design method that models the characteristics of abstract or real objects using classes and objects.

servlet

A server-side program that gives Java(TM) technology-enabled servers additional functionality.

Swing Set

The code name for a collection of graphical user interface (GUI) components that runs uniformly on any native platform which supports the Java(TM) virtual machine*. Because they are written entirely in the Java programming language, these components may provide functionality above and beyond that provided by native-platform equivalents.

virtual machine

An abstract specification for a computing device that can be implemented in different ways, in software or hardware. You compile to the instruction set of a virtual machine much like you'd compile to the instruction set of a microprocessor. The Java(TM) virtual machine* consists of a bytecode instruction set, a set of registers, a stack, a garbage-collected heap, and an area for storing methods.

wrapper

An object that encapsulates and delegates to another object to alter its interface or behavior in some way.

LAST YEAR PAPERS
MCA
Programming in Java (2012)

Time-3 Hrs. **T.M 80**

Q1 Explain in one line :
(a) What is applets ?
(b) What is inner class ?
(c) What is byte code ?
(d) What is thread ?
(e) What is interface ?
(f) What is JAR file ?
(g) What is package ?
(h) What is JVM ?
(i) What is this key word ?
(j) What is AWT ? 1x10=10

Q2 Explain in 50 words :
(a) Describe the main features of JAVA programming language .
(b) Explain frame windows in java .
(c) Explain multi threading .
(d) What are wrapper classes ? explain .
(e) Three tier client server model . 3x5=15

Q3 Explain in 100 words :
(a) What is package ? explain the steps for creating user defined package .
(b) Write a program in java to display Fibonacci series using suitable recursive function.
(c) What is an exception ? describe exception handling mechanism with example ?
(d) Explain the life cycle of Servlet.
(e) Explain Java Bean. How you prepare Java bean class. 4x5=20

Q4
(a) What is JDBC? Write the basic steps required to connect and process the data from a database using JDBC.
(b) Define Java beans.List out the advantages of java beans. List out the steps involved creating a nw bean . Explain each steps in detail.

Q5

(a) Write a Servlet for email registration with user Id , password, name , dob , address fields.Variable user id if any user exixts with same user id give proper message otherwise welcome user with message containing his/her user id and password.

OR

(b) Write short notes on:

(i) Event handling
(ii) AWT
(iii) Interface
(iv) String functions
(v) Controlling of thread. 3x5=15

MCA
Programming in Java (2010)

Time-3 Hrs. **T.M 80**

Q1 a. Explain the concept of Data abstraction in brief. 4

b. How Java ensures the concept of platform neutrality? 4

c. Explain in brief. What is meant by final class, method and variables? 4

d. What is the use of synchronization? 4

Q2 a. What is applet? How an applet is different than stand-alone Java program? 5

b. What do you mean by exceptional Handling? Explain it in brief. 6

c. Explain the use of packages and Interface. 5

OR

Write short notes on any three.

a. Event Handling. 5

b. Java security Features. 5

c. Inter Class. 5

d. Buffered Reader and Input Stream Reader class. 6

Q3 a. Explain the principles of OOPs. 5

b. Why do we need Java Beans? What are the mandatory features that a Java bean must Process? 6

c. What is JSP? Describe how it is different from servlet? 5

Q4 a. What do you mean by Access specifiers? How Java uses different various Specifiers to specify the data members of a class? Explain with suitable examples. 2+6

b. Differentiate between:

i) Java and C++ 2

ii) Array and Vector. 2

iii) Final, finally and finalize. 2

iv) JDK and BDK. 2

Q5

a. What is servlet? Explain how it is secure? How are HTTP requests and responses handled? 2+3+5

b. What is AWT? What are its components? Describe the working with graphics in AWT with the help of suitable examples. 2+3+5

MCA

Programming in Java (2009)

Time-3 Hrs. **T.M 80**

Q1 (a) What are unique features of an object- oriented programming paradigm? (4)

(b) Explain, "Write once and run anywhere" nature of Java. (4)

(c) How is Java strongly associated with the internet? (4)

(d) How is Java more secure than other languages? (4)

Q2 (a) What is the need of looping? Explain Entry Control and Exit Control loop in Java. (5)

(b) What is synchronization? When do we use it? (5)

(c) Write a Java program to create a thread that displays odd numbers starting from 1 to 100. (6)

OR

(a) What is an interface? Differentiate between Abstract class and an Interface. (5)

(b) What is synchronization? When do we use it? (5)

(c) What is an applet? How do applets differ from application programs? (6)

Q3 (a) What is an Exception? Is it essential to catch all types of exceptions? Explain. (5)

(b) What is Runnable interface? Explain with a suitable example? (5)

(c) What is AWT? What are its components? (6)

Q4 (a) What is a Java Beam? Explain its advantages also. Explain constrained properties and persistence. (8)

(b) What is a servlet? Describe its life cycle. How are HTTP requests and responses handled? (8)

Q5 (a) What is JDBC? What are its drivers? Explain the procedure for setting up a connection to database using JDBC. (8)

(b) Explain the two tier and three tier client server model. (8)

MCA
Programming in Java (2008)

Time-3 Hrs. **T.M 80**

Q1 Answer the following questions in one line:

(a) What are the three basic characteristics of an object oriented programming language.

(b) What is an applet?

(c) Define bytecode.

(d) Define the dynamic method dispatch concept.

(e) What are the packages?

(f) List the steps to create a new bean.

(g) What is persistence?

(h) What is a wiapper?

(i) What is the role of streams in the java I/O system?

(j) Assume the class rect as

Class rect (int length, int width)

What does the following code fragment do?

Rect my rect = rect ( ); 10x1=10

Q2. Answer the following question in not more than 50 words:

(a) Why is java important to the internet?

(b) What is the role of finalize( ) method in java?

(c) Differentiate between process based and thread based multitasking?

(d) What is a collection? Explain the java collection framework.

(e) How does java facilitate networking in programs? 5x3=15

Q3 Answer the following question in not more than 150 words:

(a) Differentiate between the throw's statement and the throws'clause with example.

(b) Write an applet to display the national flag of India.

(c) Differentiate between two tier and three tier client server model.

(d) What is a servlet? Describe the life cycle of a servlet.

(e) What is a java bean? What are its advantage ? 5x4=20

Q4

(a) Write a program to calculate the area and perimeter of a rectangle. The rectangle class includes length and width and the relevant constructors and member functions.

[Area of a rectangle = length x breadth , Perimeter = Sum of the four sides]

(b) How is the connection to a database established? Trace the basic steps of JDBC with example . 10

Q5 How are events handled in Java? Describe the Delegation Event Model in Detail. 15

OR

Write a program to display the initials from a full name e.g If the full name is Sachin Ramesh Tendulkar , the output should be SRT.

15

A Java Bibliography

Ken Arnold and James Gosling, *The Java Programming Language*, second ed., Addison-Wesley,

Gary Cornell and Cay S. Horstmann, *Core Java*, second ed., SunSoft Press,

Robert Englander, *Developing Java Beans*, O'Reilly,

David Flanagan, *Java Foundation Classes in a Nutshell*, O'Reilly,

http://java.sun.com/.

A Java applet index and repository is available at http://www.gamelan.com/.

A tutorial on Java is available at http://java.sun.com/tutorial