Module 2 Questions Solution

1. List and explain the salient/characteristic features of java language. OR
   List and explain the Java buzzwords. OR
   Briefly explain six key considerations used for designing JAVA language.

   1. Simple
   2. Object-oriented
   3. Portable/Architecture-neutral/Platform independent
   4. Robust
   5. Secure
   6. Multithreaded
   7. Interpreted and High performance
   8. Distributed
   9. Dynamic

Simple

- If you already understand the basic concepts of object-oriented programming, learning Java will be even easier
- Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java
- Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have *surprising* features

Object oriented

- Java was not designed to be source-code compatible with any other language
- The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance non-objects

Portable/Architecture-Neutral/Platform independent

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction
- Java designers made several hard decisions in Java language and Java Virtual Machine in an attempt to alter this situation
- Their goal was "write once; run anywhere, any time, forever."
- Java Virtual Machine provides a **platform independent environment** for the execution of Java bytecode
- Same application runs on all platforms
- The sizes of the primitive data types are always the same
- The libraries define portable interfaces

Robust

- Ability to create robust programs was given a high priority in the design of Java
- To better understand how Java is robust, consider two of the main reasons for program failure: ***memory management mistakes*** and ***mishandled exceptional conditions*** (that is, run-time errors)
- Memory management can be a difficult, tedious task in traditional programming environments
- For example, in C/C++, the programmer must manually allocate and free all dynamic memory
- Programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using
- Java virtually eliminates these problems by managing memory allocation and deallocation for you
- Java provides object-oriented exception handling

Secure

- Usage in networked environment requires more security. Code security is attained in Java through the implementation of its Java Runtime Environment (JRE)
- Programs are confined to the JRE and cannot access other parts of the computer

- The memory layout of the executable is determined during run time. This adds protection against unauthorized access to restricted areas of the code

Multithreaded

- Java supports multithreaded programming, which allows you to write programs that do many things simultaneously i.e. perform concurrent computations
- Multiple concurrent threads of executions can run simultaneously
- The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems
- Utilizes a sophisticated set of synchronization primitives (based on monitors and condition variables paradigm) to achieve this.

Interpreted and High Performance

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called **Java bytecode**
- This code can be **interpreted** on any system that provides a Java Virtual Machine (JVM)
- Bytecode can be also translated into the native machine code for better efficiency
- The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very **high performance by using a just-in-time compiler**

Distributed

- Java is designed for the distributed environment of the Internet, because it **handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.**
- Java is extensively used for **Network applications** and **WEB projects**
- Java can open and access "objects" across the Net via URLs eg. "http//:gamut.neiu.edu/~ylei/home.html", with the same ease as when accessing a local file system
- **Remote Method Invocation (RMI)** feature of Java brings an unparalleled level of abstraction to client/server programming

Dynamic

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
- Makes it possible to dynamically link code in a safe and expedient manner.
- Small fragments of bytecode may be dynamically updated on a running system
- Can check the class type of an object at runtime
- Libraries can freely add new methods and instance variables without any effect on their clients

2. Explain any five object oriented features supported by java, with examples. OR
   Discuss the OOP principles.
3. What is polymorphism? Explain with an example.

- Object-oriented programming is at the core of Java.
- ***Object-oriented programming*** organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data, Characterized as *data controlling access to the code*
-

The Three OOP Principles:

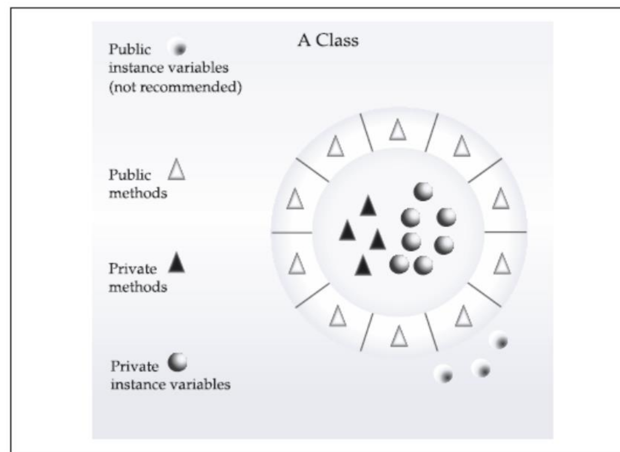
Java being an Object Oriented language works on the principles of classes and objects. A class is a template that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. A Java class uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing

actions, such as initializing the data fields of objects. Objects are made up of attributes and methods. Attributes are the characteristics that define an object; the values contained in attributes differentiate objects of the same class from one another.

There are three main features of OOPS. i) Encapsulation ii) Inheritance iii) Polymorphism

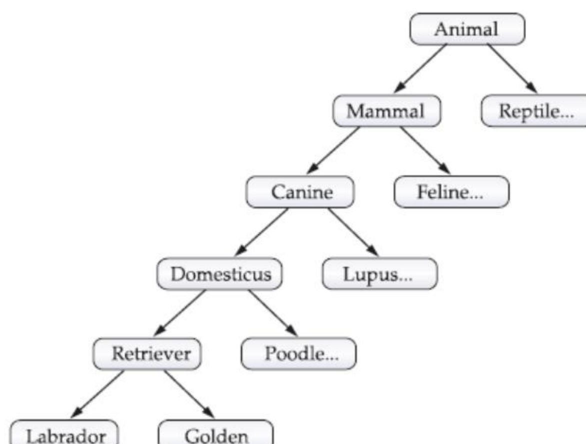
- ***Encapsulation*** - is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse – class, member variables and methods. Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a "black box," or a device that you can use without regard to the internal mechanisms. A programmer can access and use the methods and data contained in the black box but cannot change them.

  Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class.



  Above figure shows that using encapsulation the public methods can be used to protect the private data.

- **Inheritance**: An important feature of object-oriented programs is inheritance—the ability to create classes that share the attributes and methods of existing classes, but with more specific features. Inheritance is mainly used for code reusability. Inheritance - the process by which one object acquires the properties of another object. So you are making use of already written the classes and further extending on that. That why we discussed the code reusability the concept. In general one line definition, we can tell that deriving a new class from existing class, it's called as Inheritance. Following figure shows the inheritance hierarchy.



Polymorphism: In Core, Java Polymorphism is one of easy concept to understand. Polymorphism definition is that Poly means many and morphos means forms. It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. There are two types of Polymorphism available in Java. Object-oriented programs are written so that the methods having the same name works differently in different context. Java provides two ways to implement polymorphism.

- **Static Polymorphism (compile time polymorphism/ Method overloading):** The ability to execute different method implementations by altering the argument used with the method name is known as method overloading.
- **Dynamic Polymorphism (run time polymorphism/ Method Overriding)**: When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members. At run time depending object of the class appropriate methods are called.

4. Explain how JAVA is robust and architecture neutral. OR
   Why is java considered to be a robust programming language?

   Robust: The ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

   To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

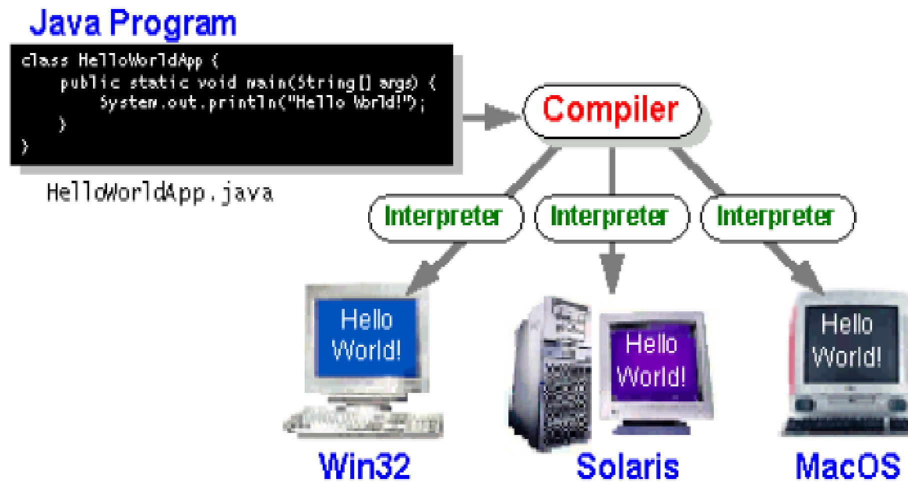
   Architecture Neutral/Platform Independent/Portable:

   A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished.

5. Define byte code. How it help java program (s) achieve portability.
6. How compile once and run anywhere is implemented in JAVA language? Explain.

   Java's Magic: 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).
   - Bytecode is a special machine language that can be understood by the Java Virtual Machine (JVM). It is Independent of any particular computer hardware, so any computer with a Java interpreter can execute the compiled Java program, no matter what type of computer, the program was compiled on
   - Bytecode generated after compiling in Mac, Windows, Linux or Unix will be same which makes Bytecode platform independent. So, Bytecode compiled in one platform can be executed into another platform.
   - Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform.

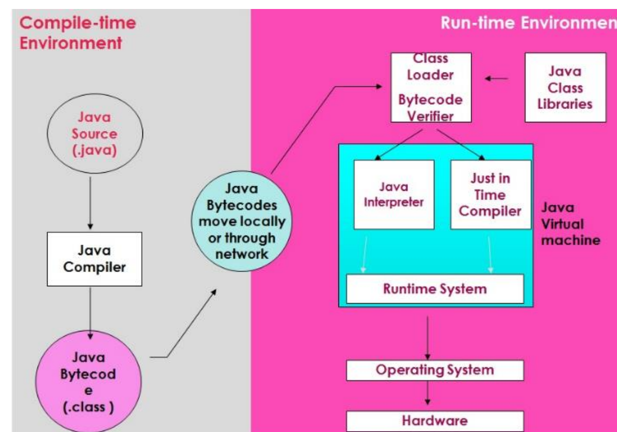


Above figures illustrates the JAVA programs platform independence. Therefore Java program can be compiled once and can be run anywhere on various platforms.

7. Write a note on JAVA environment.

Java Runtime Environment (JRE)

- Java programs cannot be executed on machine without the JVM installed on machine.
- Java Runtime Environment (JRE) is a software which we can download and install on the any operating system like Windows, Mac or Linux.
- JRE is combination of JVM and Java Application Programming Interface (Java API). Java API are set of tools and libraries that is required by the JVM to execute the java programs.
- Java Runtime Environment provides an environment to execute java programs on the computer.
  JRE = JVM + Java API's (like util, math, lang, awt, swing etc) + Runtime libraries.
- Java Runtime Environment (JRE) does NOT contain any development tools such as compiler, debugger, etc. and it is NOT for development purpose.

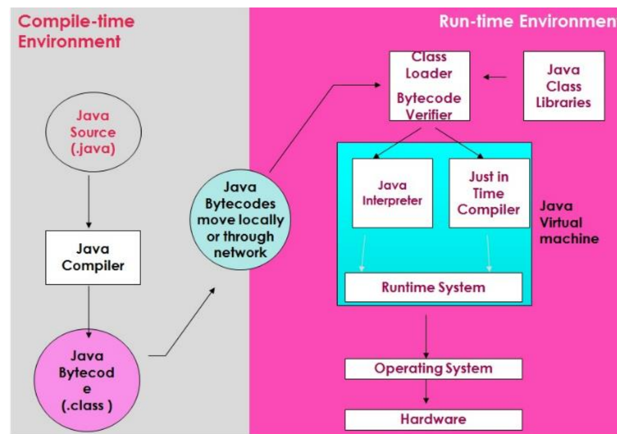


8. Explain the Java Virtual Machine (JVM)

JVM:

- Java Virtual Machine (JVM) is An imaginary machine that is implemented by emulating software on a real machine
- It provides the hardware platform specifications to which you compile all Java technology code
- Java Virtual Machine **is an interpreter for bytecode**
- To run a program in a wide variety of environments, only the JVM needs to be implemented for each platform
- When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code
- The use of bytecode enables the Java run-time system to execute programs much faster than you might expect by making use of **Just In Time compiler**
- When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. The JIT compiler compiles code as it is needed, during execution.

Java Runtime Environment (JRE) is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library. The Java Runtime environment software includes the Java Virtual Machine called .

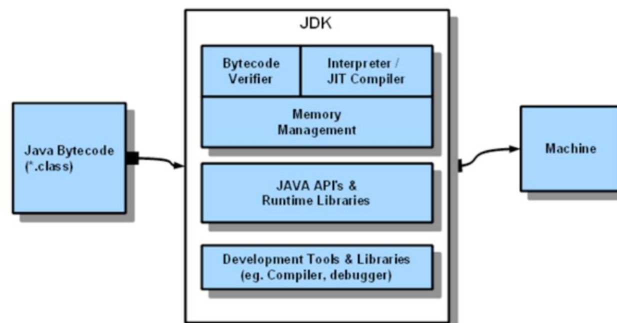


9. Briefly discuss about the java development tool kit.

Java Development Kit (JDK)

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.

- **JDK** consists of the Java Compiler and related development tools which enable users to create applications in Java.
- It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), a debugger (jdb), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.
- **JRE** is the Java Runtime Environment. i.e., the JRE is an implementation of the Java Virtual Machine which actually executes Java programs.
- JDK = JRE + Java Development Tools + Libraries



Some of the tools provided by JDK are:
javac the Java compiler

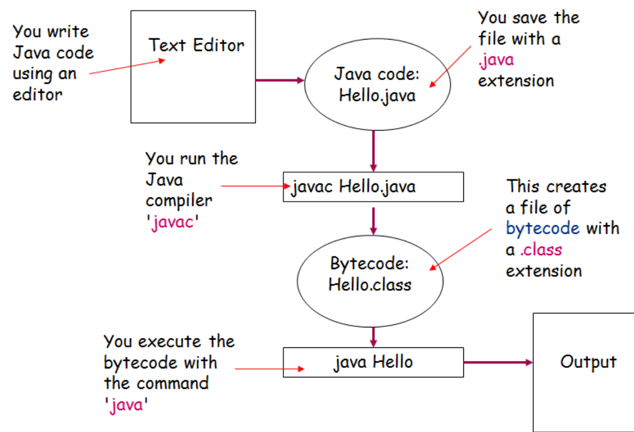
java VM for running stand-alone Java applications
appletviewer a utility to view applets
javadoc HTML generator from Java source code
jdb a rudimentary Java debugger
javah Header file generator for interlanguage linking
javap A disassembler

10. Explain/Describe the process of building and running java application program. OR
Explain the process of compiling and running the JAVA application. with the help of "Hello world" program.

Consider the following Java Hello World program.

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

Java programs are both compiled and interpreted. Following figure shows the steps involved in Building and running a Java Hello World Programs.



i) You write your Java Hello World code using an editor. Save you code in Hello.java file.
ii) To compile the program call java compiler **javac**
javac Hello.java

The javac compiler creates a file called **Hello.class** that contains the bytecode version of the program. The output of **javac** is not code that can be directly executed.

iii) To actually run the program, you must use the Java interpreter (provided by JVM) called **java**
java Hello
It produces the output
Hello World!

Compilation happens once. Interpretation occurs each time the program is executed.

11. Explain the following.
    i. JVM
    ii. Type casting.
12. Define typecasting. Explain with an example.

Type Casting Incompatible Types

It is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use **type casting**, which performs an explicit conversion between incompatible types.

For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.

It has this general form: **(*target-type*) *value***
Here, *target-type* specifies the desired type to convert the specified value to.

Examples:

1) integer value will be reduced to module byte's range:
int i;
byte b = (byte) i;

2) floating-point value will be truncated to integer value:
float f;
int i = (int) f;

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
public static void main(String args[]) {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);
        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);
        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
}
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

13. List out various operators available in JAVA language.

Java operators are used to build value expressions. Java provides a rich set of operators:

1. assignment
2. arithmetic
3. relational
4. logical
5. bitwise
6. other

- Assignment Operator: It assigns the value of the expression to the variable as shown below. The types of the variable and expression must be compatible.

  variable = expression;

- Arithmetic Operators: Java supports arithmetic operators for integer and floating-point numbers.

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2
-	op1 - op2	Subtracts op2 from op1

- Increment and decrement operators: increase and decrease a value stored in a number variable by 1.

Operator	Use	Description
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

- Relational operators: determine the relationship that one operand has to the other operand. The outcome is always a value of type boolean i.e. true or false. They are most often used in branching and loop control statements.

Operator	Use	Description
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

- Logical operators act upon boolean operands only. The outcome is always a value of type boolean. In particular,

<table>
<tr><th>Operator</th><th>Use</th><th>Description</th></tr>
<tr><td>&</td><td>op1 & op2</td><td>Logical AND</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>Logical OR</td></tr>
<tr><td>&&</td><td>op1 && op2</td><td>Short-circuit AND</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>Short-circuit OR</td></tr>
<tr><td>!</td><td>! op</td><td>Logical NOT</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>Logical XOR</td></tr>
<tr><td>&</td><td>op1 & op2</td><td>Logical AND</td></tr>
</table>

- AND and OR logical operators occur in two forms:

- **Full** - **op1 & op2** and **op1 | op2** where both op1 and op2 are evaluated
- **Short-circuit** - **op1 && op2** and **op1 || op2** where op2 is only evaluated if the value of op1 is insufficient to determine the final outcome

- **Bitwise operators**: apply to integer types only. They act on individual bits of their operands.

<table>
<tr><th>Operator</th><th>Use</th><th>Description</th></tr>
<tr><td>~</td><td>~ op</td><td>inverts all bits of its operand</td></tr>
<tr><td>&</td><td>op1 & op2</td><td>produces 1 bit if both operands are 1</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>produces 1 bit if either operand is 1</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>produces 1 bit if exactly one operand is 1</td></tr>
<tr><td>>></td><td>op1 >> op2</td><td>shifts all bits in op1 right by the value of op2</td></tr>
<tr><td><<</td><td>op1 << op2</td><td>shifts all bits in op1 left by the value of op2</td></tr>
<tr><td>>>></td><td>op1 >>> op2</td><td>shifts op1 right by op2 value, write zero on the left</td></tr>
</table>

- Other operators:

?:	shortcut if-else statement / conditional operator
[ ]	used to declare arrays, create arrays, access array elements
.	used to form qualified names
(params)	delimits a comma-separated list of parameters
(type)	casts a value to the specified type
new	creates a new object or a new array
instanceof	determines if its first operand is an instance of the second

14. Explain arrays in JAVA with examples. OR
How arrays are defined and used in Java? Explain with an example.

Arrays in Java:

- An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index. Arrays can have one or several dimensions.
- Arrays are:
  1. declared
  2. created
  3. initialized
  4. used
- **Array declaration** involves: declaring an array identifier, declaring the number of dimensions and declaring the data type of the array elements. There are two styles of array declaration:

```
type array-variable[];  or
type [] array-variable;
```

- **Array Creation:** After declaration, no array actually exists. In order to create an array, we use the new operator:

```
type array-variable[];
array-variable = new type[size];
```

This creates a new array to hold size elements of type type, which reference will be kept in the variable array-variable.

- **Array Initialization**: Arrays can be initialized when they are declared:

```
int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31}
```

- **Array Use**: Elements of array can be used through their indexes. The array index always starts with zero. The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

```
class Array {
   public static void main(String args[]) {
    int monthDays[];
    monthDays = new int[12];
    monthDays[0] = 31;
    monthDays[1] = 28;
    monthDays[2] = 31;
```

```
        …
        monthDays[12] = 31;
        System.out.print("April has ");
        System.out.println(monthDays[3] +" days.");
       }
  }
```

Multidimensional Arrays: Multidimensional arrays are arrays of arrays:

1. declaration
   ```
   int array[][];
   ```
2. Creation
   ```
   int array = new int[2][3];
   ```
3. initialization
   ```
   int array[][] = { {1, 2, 3}, {4, 5, 6} };
   ```

Example: Multidimensional Arrays

```
class Array {
    public static void main(String args[]) {
            int array[][] = { {1, 2, 3}, {4, 5, 6} };
            int i, j, k = 0;
            for(i=0; i<2; i++) {
                    for(j=0; j<3; j++)
                       System.out.print(array[i][j] + " ");
                    System.out.println();
            }
    }
}
```

15. Write a Java program to sum only the first five elements of the array {1, 2, 3, 4, 5, 6, 7, 8.9, 10}, using "for-each" version of the for loop.

This program sums only the first five elements of nums:

```
// Use break with a for-each style for.
class ForEach2 {
public static void main(String args[]) {
int sum = 0;
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// use for to display and sum the values
for(int x : nums) {
System.out.println("Value is: " + x);
sum += x;
if(x == 5) break; // stop the loop when 5 is obtained
}
System.out.println("Summation of first 5 elements: " + sum);
}
}
```

This is the output produced:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15
```

16. Discuss the labeled break and continue statements.

What is jump statement'? Explain with examples.

Java Jump Statements: Java jump statements enable transfer of control to other parts of program. Java provides three jump statements:

1) **break**
2) **continue**
3) **return**

In addition, Java supports exception handling that can also alter the control flow of a program.

break Statements: The break statement has three uses:

1. to terminate a case inside the switch statement
2. to exit an iterative statement
3. to transfer control to another statement

- (1) has been described.
- We continue with (2) and (3).

Loop Exit with break: When break is used inside a loop, the loop terminates and control is transferred to the following instruction

```
class BreakLoop {
	public static void main(String args[]) {
		for (int i=0; i<100; i++) {
			if (i == 10) break;
			System.out.println("i: " + i);
		}
		System.out.println("Loop complete");
	}
}
```

break in Nested Loop: Used inside nested loops, break will only terminate the innermost loop

```
class NestedLoopBreak {
	public static void main(String args[]) {
		for (int i=0; i<3; i++) {
			System.out.print("Pass " + i + ": ");
			for (int j=0; j<100; j++) {
				if (j == 10) break;
				System.out.print(j + " ");
			}
			System.out.println();
		}
		System.out.println("Loops complete.");
	}
}
```

Control Transfer with break: Java does not have an unrestricted "goto" statement, which tends to produce code that is hard to understand and maintain. However, in some places, the use of gotos is well justified. In particular, when breaking out from the deeply nested blocks of code.

break occurs in two versions:
1) unlabelled
2) labeled
The labeled break statement is a "civilized" replacement for goto

Labeled break: General form:

```
	break label;
```

where label is the name of a label that identifies a block of code:

```
	label: { … }
```

The effect of executing break label; is to transfer control immediately after the block of code identified by label.

```
class Break {
	public static void main(String args[]) {
		boolean t = true;
		first: {
			second: {
				third: {
					System.out.println("Before the break.");
					if (t) break second;
					System.out.println("This won't execute");
				}
				System.out.println("This won't execute");
			}
			System.out.println("After second block.");
		}
	}
 }
```

continue Statement: The continue statement forces the early termination of the current iteration to begin immediately the next iteration. Like break, continue has two versions:

1. unlabelled – continue with the next iteration of the current loop
2. labeled – specifies which enclosing loop to continue

Example: Unlabeled continue

```
class Continue {
	public static void main(String args[]) {
		for (int i=0; i<10; i++) {
			System.out.print(i + " ");
```

```
                    if (i%2 == 0) continue;
                    System.out.println("");
            }
        }
}
```

Example: Labeled continue:

```
class LabeledContinue {
        public static void main(String args[]) {
                outer: for (int i=0; i<10; i++) {
                        for (int j=0; j<10; j++) {
                                if (j > i) {
                                        System.out.println();
                                        continue outer;
                                }
                                System.out.print(" " + (i * j));
                        }
                }
                System.out.println();
        }
}
```

return Statement: The return statement is used to return from the current method: it causes program control to transfer back to the caller of the method. Two forms of break statement:

1) return without value

```
return;
```

2) return with value

```
return expression;
```

Example: return

```
class Return {
        public static void main(String args[]) {
                boolean t = true;
                System.out.println("Before the return.");
                if (t) return; // return to caller
                System.out.println("This won't execute.");
        }
}
```

17. What is command-line arguments? Write a program to demonstrate command-line arguments.

Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to **main( )**. A command-line argument is the information that directly follows the program's name on the command line when it is executed. In Java programs the command-line arguments are stored as strings in a **String** array passed to the **args** parameter of **main( )**. The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on. For example, following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
        public static void main(String args[]) {
                for(int i=0; i<args.length; i++)
                        System.out.println("args[" + i + "]: " + args[i]);
        }
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

REMEMBER All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually.

18. Explain i) >>> ii) short circuit logical operators iii) for each
19. With example, explain the working of >> and >>>.

Short-Circuit Logical Operators

Java provides secondary versions of the Boolean AND and OR operators, and are known as *short-circuit* logical operators.
|| Short-circuit OR
&& Short-circuit AND

In **A** OR **B,** the OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, in **A** AND **B,** the AND operator results in **false** when **A** is **false**, no matter what **B** is. If you use the || and **&&** forms, rather than the | and **&** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```

Since the short-circuit form of AND (**&&**) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single **&** version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & e++ < 100) d = 100;
```

Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

The Left Shift << Operator
The left shift operator, **<<**, shifts all of the bits in a value to the left a specified number of times. It has this general form:
value << num
Here, *num* specifies the number of positions to left-shift the value in *value.* That is, the **<<** moves all of the bits in the specified value to the left by the number of bit positions specified by *num.* For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

The Right Shift >> Operator
The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:
value >> num

Here, *num* specifies the number of positions to right-shift the value in *value.* That is, the **>>** moves all of the bits in the specified value to the right the number of bit positions specified by *num.* The following code fragment shifts the value 32 to the right by two positions, resulting in **a** being set to 8:

```
int a = 32;
a = a >> 2; // a now contains 8
```

When a value has bits that are "shifted off," those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low-order bits to be lost, resulting again in **a** being set to 8.

```
int a = 35;
a = a >> 2; // a still contains 8
```

Looking at the same operation in binary shows more clearly how this happens:

```
00100011 35
>> 2
00001000 8
```

The Unsigned Right Shift >>> Operator
As you have just seen, the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift.* To accomplish this, you will use Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit. The following code fragment demonstrates the >>>.

Here, **a** is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:
11111111 11111111 11111111 11111111 –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int

The For-Each Version of the for Loop

Java adds the for-each capability by enhancing the **for** statement. The for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-each version of the **for** is shown here:
for(*type itr-var : collection*) *statement-block*

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained. Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the base type of the array. To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace. The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

20. What is the default package and default class in Java?

What is the default package?
The *default package* is an *unnamed package*. The *unnamed package* contains java classes whose source files did not contain a package declaration. The purpose of *default package* is for convenience when developing small or temporary applications or when just beginning development. The compiled class files will be in the current working directory.
A compilation unit that has no package declaration is part of an unnamed package. Note that an unnamed package cannot have subpackages, since the syntax of a package declaration always includes a reference to a named top level package. An implementation of the Java platform must support at least one unnamed package; it may support more than one unnamed package but is not required to do so. Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development.

21. Write a program to calculate the average among the elements {4, 5, 7, 8} using for each in Java. How for each is different from for loop?

```
// Java program to calculate the average among the elements {4, 5, 7, 8} using for each
class Average {
    public static void main(String args[]) {
        int nums[] = { 4,  5, 7, 8};
        int sum = 0;
        double avg = 0.0;
        // use for-each style for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
```

```
            }
            avg = sum /  4;
            System.out.println("Average: " + avg);
        }
    }
```

Output:

```
Value is: 4
Value is: 5
Value is: 7
Value is: 8
Average: 6.0
```

How for each is different from for loop?

Java Tradition for loop	Java for-each loop

General form of the traditional **for** statement:
for(*initialization*; *condition*; *iteration*) {
// body
}
	The general form of the for-each version of the **for**:
	for(*type itr-var : collection*) *statement-block*

Example: for(int i=0; i < 10; i++) sum += nums[i];	Example: for(int x: nums) sum += x;
To compute the sum, each element in **nums** is read, in order, from start to finish. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable.	It eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end.
Allows to modify the array elements	Does not allow to modify the array elements
you know the index • can be used to refer to another list of the same size • can be used to refer to previous/next element	you don't know the index of the element • you cannot refer to previous/next element
Error prone, a lot of things that can go wrong, e.g.: • i = 0; instead of int i = 0; - will refer to variable declared before the loop, possible side effects outside of the loop • > instead of < - loop will not execute • j++ instead of i++ - infinite loop .get(j) instead of .get(i) - will always get the same element	With each pass through the loop, x is automatically given a value equal to the next element in nums. Not only is the syntax streamlined, but it also prevents boundary errors. more robust (less code, fewer special characters)

22. Explain the syntax of for-each loop. Write a JAVA program to search a key element by using for-each loop.

The general form of the for-each version of the **for** is shown here:

```
for(type itr-var : collection) statement-block
```

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained. Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

```
// Java program to search a key element by using for-each loop
import java.util.Scanner;
class Search {
        public static void main(String args[]) {
                int nums[] = { 4,  5, 7, 8};
                boolean found = false;

                Scanner s = new Scanner(System.in);
                System.out.print("Enter the key element to be searched: ");
                int key = s.nextInt();

                // use for-each style for to search the key value
                for(int x : nums) {
                        if( x == key ) {
```

```
                                found = true;
                                break;
                        }
                }
                if ( found == true )
                        System.out.println("Key " + key + " found");
                else
                        System.out.println("Key " + key + " not found");
        }
}
```

23. Write and demonstrate a JAVA program to initialize and display different types of integers and floating point varibles.

```
public class IntFloat {
    public static void main(String[] args) {
    byte b =100;
    short s =123;
    int v = 123543;
    int calc = -9876345;
    long amountVal = 1234567891;
    float intrestRate = 12.25f;
    double sineVal = 12345.234d;

    System.out.println("byte Value = "+ b);
    System.out.println("short Value = "+ s);
    System.out.println("int Value = "+ v);
    System.out.println("int second Value = "+ calc);
    System.out.println("long Value = "+ amountVal);
    System.out.println("float Value = "+ intrestRate);
    System.out.println("double Value = "+ sineVal);
    }
}

Output
byte Value = 100
short Value = 123
int Value = 123543
int second Value = -9876345
long Value = 1234567891
float Value = 12.25
double Value = 12345.234
```

24. What is the output of the code below? If you insert another 'int b' outside the for loop what is the output?

```
class Example  {
  public static void main (String s[]) {
      int a;
      for (a=0; a 3< ; a++)  {
             int b = -1;
             system.out.println(" " + b);
             b = 50;
             system.out.println(" " + b);
      }
  }
}

It gives following compilation error
Example.java:5: error: ';' expected
                for (a=0; a 3<; a++)  {
```

Output Note: After correcting **a 3<** to **a < 3**

```
-1
 50
 -1
 50
 -1
 50
```

If you insert another 'int b' outside the for loop what is the output?

```
It gives following compilation error

MyExample.java:6: error: variable b is already defined in method main(String[])
                        int b = -1;
                            ^
1 error
```

25. What is the output of the above code? If you insert another 'int b' outside the for loop what is the output?

26. Correct the errors from the following code and explain:

```
byte b = 50:
b = b * 2;
```

For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if,as in this particular case, the value being assigned would still fit in the target type. In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

27. Write the output of the following code:

```
byte b;
int i = 257;
double d = 323.142;
b = (byte) i;
systemout.println(b):
b = (byte) d;
systern.out.println(b);

Output
1
67
```

28. Compare and explain the above two snippets.

```
i)   int num, den;
     if(den ! 0 && num | den > 2) {}
ii)  int num, den ;
     if(den ! 0 && num | den == 2) {}
```