

## Contents

2006 .....	3
Q1. What is Package? Explain any one package in Java in Detail? (2004) .....	3
Q2. A) What is meant by platform independence? How is it achieved in Java? (10 marks) .....	4
Q2.B. Which are the different primitive datatypes in Java? Where are they used? What is the difference between the primitive and object data types (10 marks) .....	5
Q3.A. Explain general coding structure of a class with coding example. List a few coding conventions followed in the language? (10 marks).....	11
Q3. B) How are methods defined in Java? Explain the usage of most common keywords used in method definition? (10 marks) .....	12
Q4. What is Object Oriented Programming? Enumerate and explain various features in OOP with suitable code examples? (20 marks).....	14
Q5. Write Short notes on: .....	15
Q5.1. String class .....	15
Q5.2) Constructors .....	16
Q 5.3) Java Collections .....	18
Q5.4) Wrapper classes .....	20
Q6. What is Method Overloading? Where is it used? How is it different than Method Overriding? ..	22
Q7. Explain salient features, advantages and disadvantages of Java as programming language. How does this language compare with other programming languages? What are the practical applications of java? Explain a few.....	25
Q8. A) What is container software?.....	28
Q8.B) what are the different types of EJB's? Where are EJB's used? .....	28
2009 .....	31
Q. What do you mean by Declaration, definition and Usage. Illustrate with example of each. ....	31
Q. "Byte Code" is the key that allows Java to solve both the security and portability problems. ....	35
Q. What do you mean by "Structure" in structured languages? Discuss. ....	37
Q. What are the identifying characteristics of a object oriented programming paradigm. ....	38
Q. Briefly describe history and evolution of java .....	44
Q. "Java is a strongly typed language". Discuss .....	46
Q. "It is simple to manage stack memory than heap memory". Discuss? .....	48
2010 .....	51
Q1: Write answer in complete one sentence (20 marks) .....	51
Q2a) What is a structure? Explain with example. How structure is different from Array? Distinguish between Structure and Class (8 marks) [Not Applicable] .....	51

Q2b) Describe all the various features of Java (8 marks) .....	51
Q2c) What is pointer variable. How it differs from reference variable? (4 marks) [Not Applicable] .....	53
Q3a) What is the meaning of abstract method? What is the advantage of declaring class as abstract? What is the difference between abstract and final class? (8 marks).....	53
Q3b) What are the different types of function declaration? How will you declare a function outside and inside the class? (8 marks) [Not Applicable] .....	56
Q3c) Compare class and object with suitable example (4 marks).....	56
Q4a) What is object oriented paradigm? Explain the various features of object oriented programming with example (8 marks) [Repeat in 2004].....	57
Q4b) Explain practical usage of interface with example Describe how it differs from class (8 marks)..	60
Q4c) What do you mean by Exception Handling? What are the types of exception? Compare them (4 marks) .....	60
Q5a) Compare and contrast C++ and Java (8 marks).....	62
Q5b) Write brief note on .....	63
1. JVM.....	63
2. Development tools for JDK .....	63
Q5c) What are the uses of Keyword final in Java? Give example (4 marks).....	64
Q6a) What is array? Write example for each type of array. Describe 2 ways to declare array in Java. How do you get size of an array? Is index checking supported by Java? (8 marks) .....	65
Q6b) What is friend function? Describe their benefits and limitations. Give suitable example (6 marks) [Not applicable].....	68
Q6c) What is the need of dynamic method dispatch? Explain with Example (6 marks) .....	68
2004 .....	70
Q2) What are constructors? How are objects created and destroyed in JVM? Explain briefly the functioning of Garbage Collection in JVM. ....	70
Q3) What is method/constructor overloading? How is it used in Programming? .....	71
Q6) What is Threading? Explain the applications and common problems in threading. ....	73
Q7) What is loop (control) Structure. Explain the 3 types of loop with small code snippets.....	76
Q8) What is JDBC? What are the different types of JDBC drivers? .....	78

## 2006


### Q1. What is Package? Explain any one package in Java in Detail? (2004)

Ans. A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a String object contains state and behavior for character strings; a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

#### The Java I/O Package

The Java I/O package, a.k.a. java.io, provides a set of input streams and a set of output streams used to read and write data to files or other input and output sources. There are three categories of classes in java.io: input streams, output streams and everything else.

The pages of this lesson provide overviews of Java's I/O classes. They give information about what each class does and how you can use them. These pages do not provide any practical examples or details of each class. For more practical information regarding reading and writing data using these classes, see [Input and Output Streams](#) .

#### Input Streams

Input streams read data from an input source. An input source can be a file, a string, or memory--anything that can contain data. All input streams inherit from InputStream--an abstract class that defines the programming interface for all input streams.

The InputStream class defines a programming interface for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes that are available for reading, and resetting the current position within the stream. An input stream is automatically opened when you create it. You can explicitly close a stream with the close() method, or let it be closed implicitly when the object is garbage collected.

#### Output Streams

Output streams write data to an output source. Similar to input sources, an output source can be anything that can contain data: a file, a string, or memory.

The OutputStream class is a sibling to InputStream and is used to write data that can then be read by an input stream. The OutputStream class defines a programming interface for writing bytes or arrays of bytes to the stream and flushing the stream. Like an input stream, an output stream is automatically

opened when you create it. You can explicitly close an output stream with the `close()` method, or let it be closed implicitly when the object is garbage collected.

### ***Everything Else***

In addition to the streams classes, `java.io` contains a few other classes:

### **File**

Represents a file on the host system.

### **RandomAccessFile**

Represents a random access file.

### **StreamTokenizer**

Tokenizes the contents of a stream.

## **Q2. A) What is meant by platform independence? How is it achieved in Java? (10 marks)**

**Ans.** A Platform-Independent Model (PIM) in software engineering is a model of a software system or business system, that is independent of the specific technological platform used to implement it.

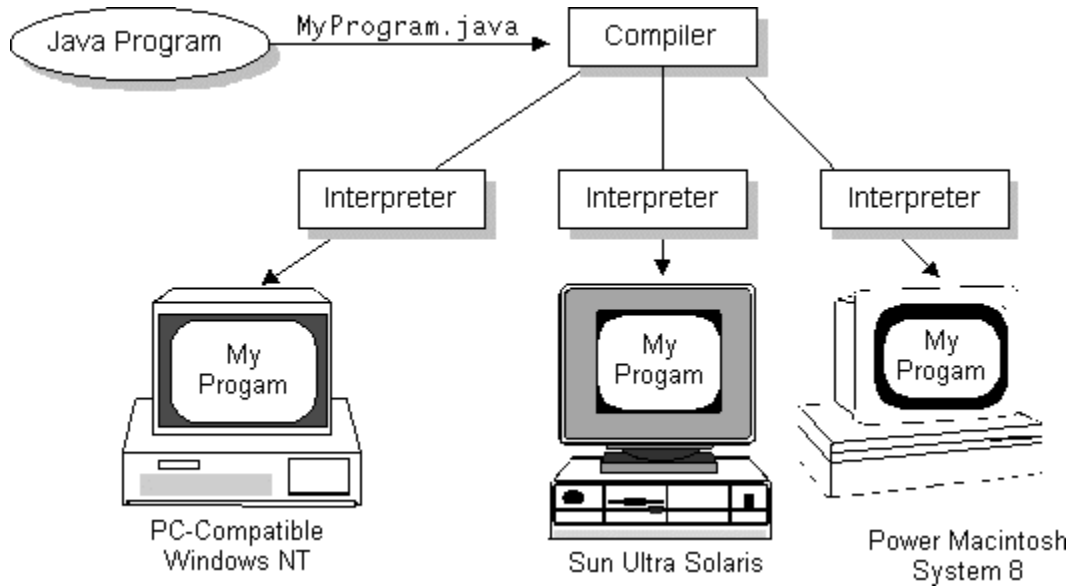
The term platform-independent model is most frequently used in the context of the model-driven architecture approach. Platform independent is program running on different processors like intel, AMD, Sun Micro Systems etc.; This model-driven architecture approach corresponds the Object Management Group vision of Model Driven Engineering.

Java solves the problem of platform-independence by using byte code. The Java compiler does not produce native executable code for a particular machine like a C compiler would. Instead it produces a special format called byte code. Java byte code written in hexadecimal, byte by byte, looks like this:

```
CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08
```

This looks a lot like machine language, but unlike machine language Java byte code is exactly the same on every platform. This byte code fragment means the same thing on a Solaris workstation as it does on a Macintosh PowerBook. Java programs that have been compiled into byte code still need an interpreter to execute them on any given platform. The interpreter reads the byte code and translates it into the native language of the host machine on the fly. The most common such interpreter is Sun's program `java` (with a little `j`). Since the byte code is completely platform independent, only the interpreter and a few native libraries need to be ported to get Java to run on a new computer or operating system. The rest of the runtime environment including the compiler and most of the class libraries are written in Java.

All these pieces, the `javac` compiler, the `java` interpreter, the Java programming language, and more are collectively referred to as Java.



**Q2.B. Which are the different primitive datatypes in Java? Where are they used? What is the difference between the primitive and object data types (10 marks)**

Ans. The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int gear = 1;
```

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other *primitive data types*. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte:** The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large [arrays](#), where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short:** The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
- **int:** The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.

- **long**: The long data type is a 64-bit signed two's complement integer. 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 (inclusive). Use this data type when you need a range of values wider than those provided by int.
- **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](#) section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the [java.math.BigDecimal](#) class instead. [Numbers and Strings](#) covers BigDecimal and other useful classes provided by the Java platform.
- **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](#) section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **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. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **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 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the [java.lang.String](#) class. Enclosing your character string within double quotes will automatically create a new String object; for example, String s = "this is a string";. String objects are *immutable*, which means that once created, their values cannot be changed. The String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. You'll learn more about the String class in [Simple Data Objects](#)

**Default Values**

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following chart summarizes the default values for the above data types.

Data Type	Default Value (for fields)
byte	0

short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	Null
boolean	False

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

### Literals

You may have noticed that the `new` keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

### Integer Literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise it is of type `int`. It is recommended that you use the upper case letter `L` because the lower case letter `l` is hard to distinguish from the digit `1`.

Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Values of type `long` that exceed the range of `int` can be created from `long` literals. Integer literals can be expressed by these number systems:

- Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
- Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F

- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

### **Floating-Point Literals**

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

### **Character and String Literals**

Literals of types char and String may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as '\u0108' (capital C with circumflex), or "\u00ED Se\u00F1or" (Sí Señor in Spanish). Always use 'single quotes' for char literals and "double quotes" for String literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r(carriage return), \" (double quote), \' (single quote), and \\ (backslash).

There's also a special null literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types. There's little you can do with a null value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending ".class"; for example, String.class. This refers to the object (of type Class) that represents the type itself.

### **Using Underscore Characters in Numeric Literals**



In Java SE 7 and later, any number of underscore characters (`_`) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// This is an identifier, not
// a numeric literal
int x1 = _52;
// OK (decimal literal)
int x2 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x3 = 52_;
// OK (decimal literal)
int x4 = 5_____2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x5 = 0_x52;
```

```
// Invalid: cannot put underscores
// at the beginning of a number
int x6 = 0x_52;
// OK (hexadecimal literal)
int x7 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x8 = 0x52_;
```

## Object Data Type

Like variables are instances of primitive data types, Object Data Type essentially means class that is used to create object instances. In other words, objects which are instances of classes are referred to be of type “Object Data Type”. The class could be user-defined class or the one provided through Java Library. For example, String which happens to be most frequently used class for creating objects such as “studentName”. The most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class.

When you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object.

Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let’s look at the details of this procedure.

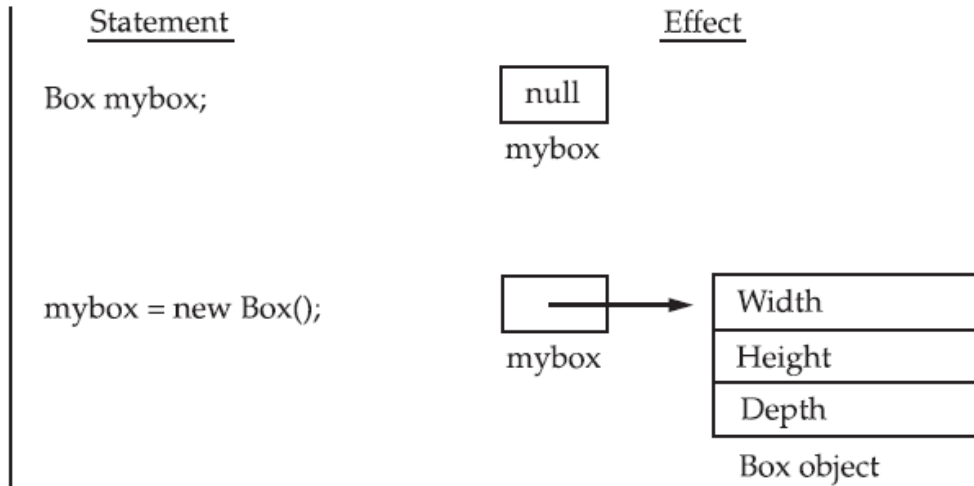
In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

```
Box mybox;

mybox = new Box();
```

The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object. Any attempt to use mybox at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds the memory address of the actual Box object. The effect of these two lines of code is depicted

**FIGURE 6-1**  
 Declaring an object  
 of type **Box**



Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. Objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently.

**Q3.A. Explain general coding structure of a class with coding example. List a few coding conventions followed in the language? (10 marks)**

Ans.

```

package firstproject;

public class FirstProject {

    public static void main(String[] args) {

    }

}
    
```

You can see we have the package name first. Notice how the line ends with a semicolon. If you miss the semicolon out, the programme won't compile:

```
package firstproject;
```

The class name comes next:

```
public class FirstProject {  
  
}
```

You can think of a class as a code segment. But you have to tell Java where code segments start and end. You do this with curly brackets. The start of a code segment is done with a left curly bracket { and is ended with a right curly bracket }. Anything inside of the left and right curly brackets belong to that code segment.

What's inside of the left and right curly brackets for the class is another code segment. This one:

```
public static void main( String[ ] args ) {  
  
}
```

The word "main" is the important part here. Whenever a Java programme starts, it looks for a method called main. (A method is just a chunk of code. You'll learn more about these later.) It then executes any code within the curly brackets for main. You'll get error messages if you don't have a main method in your Java programmes. But as its name suggest, it is the main entry point for your programmes.

The blue parts before the word "main" can be ignored for now.

(If you're curious, however, public means that the method can be seen outside of this class; static means that you don't have to create a new object; and void means it doesn't return a value - it just gets on with it. The parts between the round brackets of main are something called command line arguments. Don't worry if you don't understand any of that, though.)

The important point to remember is that we have a class called FirstProject. This class contains a method called main. The two have their own sets of curly brackets. But the main chunk of code belongs to the class FirstProject.

### Q3. B) How are methods defined in Java? Explain the usage of most common keywords used in method definition? (10 marks)

Ans.

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, and others you will learn about later.
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—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,  
                             double length, double grossTons) {  
    //do the calculation 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 (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you 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, you 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.

**Note:** Overloaded methods should be used sparingly, as they can make code much less readable.

#### Q4. What is Object Oriented Programming? Enumerate and explain various features in OOP with suitable code examples? (20 marks)

Ans. **Object-oriented programming (OOP)** is a [programming paradigm](#) that represents concepts as "[objects](#)" that have [data fields](#) (attributes that describe the object) and associated procedures known as [methods](#). Objects, which are usually [instances of classes](#), are used to interact with one another to design applications and computer programs.

**In programming languages an object is the composition of nouns (like data, such as numbers, strings, or variables) and verbs (like actions, such as functions).**

An object-oriented program may be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks ([subroutines](#)) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. Actions (or "[methods](#)") on these objects are closely associated with the object. For example, OOP [data structures](#) tend to "carry their own operators around with them" (or at least "[inherit](#)" them from a similar object or class)—except when they must be serialized.

#### Encapsulation Enforces Modularity

Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called "classes," and one instance of a class is an "object." For example, in a payroll system, a class could be Manager, and Pat and Jan could be two instances (two objects) of the Manager class. Encapsulation ensures good code modularity, which keeps routines separate and less prone to conflict with each other.

### **Inheritance Passes "Knowledge" Down**

Classes are created in hierarchies, and inheritance lets the structure and methods in one class pass down the hierarchy. That means less programming is required when adding functions to complex systems. If a step is added at the bottom of a hierarchy, only the processing and data associated with that unique step must be added. Everything else about that step is inherited. The ability to reuse existing objects is considered a major advantage of object technology.

### **Polymorphism Takes any Shape**

Object-oriented programming lets programmers create procedures for objects whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement can be written for "cursor," and polymorphism lets that cursor take whatever shape it requires at runtime. It also lets new shapes be easily integrated.

### **OOP Languages**

Used for simulating system behavior in the late 1960s, SIMULA was the first object-oriented language. In the 1970s, Xerox's Smalltalk was the first object-oriented programming language and was used to create the graphical user interface (GUI). Today, C++ and Java are the major OOP languages, while C#, Visual Basic.NET, Python and JavaScript are also popular. ACTOR and Eiffel were earlier OOP languages.

## **Q5. Write Short notes on:**

### **Q5.1. String class**

The `String` class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};  
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");  
String cde = "cde";  
System.out.println("abc" + cde);  
String c = "abc".substring(2,3);  
String d = cde.substring(1, 2);
```

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the [Character](#) class.

The Java language provides special support for the string concatenation operator ( `+` ), and for conversion of other objects to strings. String concatenation is implemented through the `StringBuilder` (or `StringBuffer`) class and its `append` method. String conversions are implemented through the method `toString`, defined by `Object` and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a `null` argument to a constructor or method in this class will cause a [NullPointerException](#) to be thrown.

A `String` represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section [Unicode Character Representations](#) in the `Character` class for more information). Index values refer to `char` code units, so a supplementary character uses two positions in a `String`.

The `String` class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., `char` values).

## Q5.2) Constructors

A **constructor** is a special method that is used to **initialize a newly created object** and is called just after the memory is allocated for the object. It can be used to initialize the objects, to **required**, or **default values** at the



time of object creation It is **not mandatory** for the coder to write a constructor for the class

If no user defined constructor is provided for a class, compiler initializes member variables to its default values.

- numeric data types are set to 0
- char data types are set to null character(‘\0’)
- reference variables are set to null

In order to create a Constructor observe the following rules

1. It has the **same name** as the class
2. It should not return a value not even **void**

**Assignment 1:** Create your First Constructor

Step 1: Type following code in your editor

```
1 class Demo{
2
3     int value1;
4
5     int value2;
6
7     Demo () {
8
9         value1 = 10;
10
11        value2 = 20;
12
13        System.out.println("Inside Constructor");
14
15    }
16
17    public void display() {
18
```

```
19     System.out.println("Value1 === "+value1);
20
21     System.out.println("Value2 === "+value2);
22
23     }
24
25     public static void main(String args[]){
26
27         Demo d1 = new Demo();
28
29         d1.display();
30     }
31
32 }
33
```

Step 2) Save , Run & Compile the code. Observe the output.

## constructor overloading

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type

### Q 5.3) Java Collections

Ans. A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

## What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

## Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own

data structures, you'll have more time to devote to improving programs' quality and performance.

- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

#### Q5.4) Wrapper classes

Ans. A **primitive wrapper class** in the [Java](#) and [ActionScript](#) programming languages is one of eight classes provided in the [java.lang](#) package to provide [object methods](#) for the eight [primitive types](#). All of the primitive wrapper classes in Java are [immutable](#). [J2SE 5.0](#) introduced [autoboxing](#) of primitive types into their wrapper object, and automatic unboxing of the wrapper objects into their primitive value—the implicit conversion between the wrapper objects and primitive values.

[Wrapper classes](#) are used to represent primitive values when an [Object](#) is required. The wrapper classes are used extensively with [Collection](#) classes in the [java.util](#) package and with the classes in the [java.lang.reflect](#) [reflection](#) package.

The primitive wrapper classes and their corresponding primitive types are:

Primitive type	Wrapper class	Constructor Arguments
byte	<a href="#">Byte</a>	byte or String

short	<a href="#">Short</a>	short or String
int	<a href="#">Integer</a>	int or String
long	<a href="#">Long</a>	long or String
float	<a href="#">Float</a>	float, double or String
double	<a href="#">Double</a>	double or String
char	<a href="#">Character</a>	char
boolean	<a href="#">Boolean</a>	boolean or String

The `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` wrapper classes are all [subclasses](#) of the `Number` class.

The wrapper classes `BigDecimal` and `BigInteger` are not one of the primitive wrapper classes and are mutable.<sup>[1]</sup>

## void

Although it is not a wrapper class, the `Void` class is similar in that it provides an object representation of the `void` return. "The `Void` class is an uninstantiable placeholder class used by the `java.lang.reflect` [API](#) to hold a reference to the `Class` object representing the [Java keyword](#) `void`."[\(Javadoc for Void\)](#)

## Atomic wrapper classes

With `Java 5.0`, additional wrapper classes were introduced in the `java.util.concurrent.atomic` package. These classes are mutable and cannot be used as a replacement for the regular wrapper classes. Instead, they provide [atomic operations](#) for addition, increment and assignment.

The atomic wrapper classes and their corresponding types are:

Primitive type	Wrapper class
----------------	---------------

int	<u>AtomicInteger</u>
long	<u>AtomicLong</u>
boolean	<u>AtomicBoolean</u>
V	<u>AtomicReference&lt;V&gt;</u>

The `AtomicInteger` and `AtomicLong` classes are subclasses of the `Number` class. The `AtomicReference` class accepts the type parameter `V` that specifies the type of the object reference. (See "[Generics in Java](#)" for a description of type parameters in Java).

### Q6. What is Method Overloading? Where is it used? How is it different than Method Overriding?

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

If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call. Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Overload test for one integer parameter.
    void test(int a) {
```

```
System.out.println("a: " + a);  
}  
  
// Overload test for two integer parameters.  
void test(int a, int b) {  
System.out.println("a and b: " + a + " " + b);  
}  
  
// overload test for a double parameter  
double test(double a) {  
System.out.println("double a: " + a);  
return a*a;  
}  
}  
}  
  
class Overload {  
public static void main(String args[]) {  
OverloadDemo ob = new OverloadDemo();  
double result;  
// call all versions of test()  
ob.test();  
ob.test(10);  
ob.test(10, 20);  
result = ob.test(123.2);  
System.out.println("Result of ob.test(123.2): " + result);
```

```
}
}
```

This program generates the following output:

No	parameters
a:	10
a                      and                      b:	10                      20
double                      a:	123.2
Result of ob.test(123.2): 15178.24	

As you can see, **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test( )** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters

### Difference between Method Overloading and Overriding

The difference between overriding and overloading in Java is a common source of confusion – but it is fairly simple to understand. Let's start the discussion by talking more about method overloading. Overloading in Java *can* occur when two or more methods in the same class share the same name or even if a child class shares a method with the same name as one of its parent classes. But, in order to *actually* have overloaded methods, the methods not only have to have the same name, but there are other conditions that must be satisfied – read below to see what those conditions are.

Suppose we have a class called TestClass which has 2 methods, and both methods have the *same* name – let's say that name is "someMethod" – this would be considered to be method overloading if **at least one of these 2 things is true:**

- 1.) The number of parameters is different for the methods
- 2.) The parameter types are different.

### How to NOT overload methods:

It's important to understand that method overloading is NOT something you can accomplish by doing these 2 things:



- 
1. Changing the return type of the method
  2. Changing the name of the method parameters, but not changing the parameter types.
- 

**Q7. Explain salient features, advantages and disadvantages of Java as programming language. How does this language compare with other programming languages? What are the practical applications of java? Explain a few.**

Ans. Java was developed by taking the best points from other programming languages, primarily C and C++. Java therefore utilises algorithms and methodologies that are already proven. Error prone tasks such as pointers and memory management have either been eliminated or are handled by the Java environment automatically rather than by the programmer. Since Java is primarily a derivative of C++ which most programmers are conversant with, it implies that Java has a familiar feel rendering it easy to use.

---

### **JAVA IS OBJECT-ORIENTED**

---

Even though Java has the look and feel of C++, it is a wholly independent language which has been designed to be object-oriented from the ground up. In object-oriented programming (OOP), data is treated as objects to which methods are applied. Java's basic execution unit is the *class*. Advantages of OOP include: reusability of code, extensibility and dynamic applications.

---

### **JAVA IS DISTRIBUTED**

---

Commonly used Internet protocols such as HTTP and FTP as well as calls for network access are built into Java. Internet programmers can call on the functions through the supplied libraries and be able to access files on the Internet as easily as writing to a local file system.

---

### **JAVA IS INTERPRETED**

---

When Java code is compiled, the compiler outputs the Java Bytecode which is an executable for the Java Virtual Machine. The Java Virtual Machine does not exist physically but is the specification for a hypothetical processor that can run Java code. The bytecode is then run through a Java interpreter on any given platform that has the interpreter ported to it. The interpreter converts the code to the target hardware and executes it.

---

### **JAVA IS ROBUST**

---

Java compels the programmer to be thorough. It carries out type checking at both compile and runtime making sure that every data structure has been clearly defined and typed. Java manages memory automatically by using an automatic garbage collector. The garbage collector runs as a low priority thread in the background keeping track of all objects and references to those objects in a Java program. When an object has no more references, the garbage collector tags it for removal and removes the object either when there is an immediate need for more memory or when the demand on processor cycles by the program is low.

---

## JAVA IS SECURE

---

The Java language has built-in capabilities to ensure that violations of security do not occur. Consider a Java program running on a workstation on a local area network which in turn is connected to the Internet. Being a dynamic and distributed computing environment, the Java program can, at runtime, dynamically bring in the classes it needs to run either from the workstation's hard drive, other computers on the local area network or a computer thousands of miles away somewhere on the Internet. This ability of classes or applets to come from unknown locations and execute automatically on a local computer sounds like every system administrator's nightmare considering that there could be lurking out there on one of the millions of computers on the Internet, some viruses, Trojan horses or worms which can invade the local computer system and wreak havoc on it. Java goes to great lengths to address these security issues by putting in place a very rigorous multilevel system of security:

- First and foremost, at compile time, pointers and memory allocation are removed thereby eliminating the tools that a system breaker could use to gain access to system resources. Memory allocation is deferred until runtime.
- Even though the Java compiler produces only correct Java code, there is still the possibility of the code being tampered with between compilation and runtime. Java guards against this by using the bytecode verifier to check the bytecode for language compliance when the code first enters the interpreter, before it ever even gets the chance to run.

The bytecode verifier ensures that the code does **not** do any of the following:

- Forge pointers
- Violate access restrictions
- Incorrectly access classes
- Overflow or underflow operand stack
- Use incorrect parameters of bytecode instructions
- Use illegal data conversions
- At runtime, the Java interpreter further ensures that classes loaded do not access the file system except in the manner permitted by the client or the user.

Sun Microsystems will soon be adding yet another dimension to the security of Java. They are currently working on a public-key encryption system to allow Java applications to be stored and transmitted over the

Internet in a secure encrypted form.

### JAVA IS ARCHITECTURALLY NEUTRAL

The Java compiler compiles source code to a stage which is intermediate between source and native machine code. This intermediate stage is known as the bytecode, which is neutral. The bytecode conforms to the specification of a hypothetical machine called the Java Virtual Machine and can be efficiently converted into native code for a particular processor.

### JAVA IS PORTABLE

By porting an interpreter for the Java Virtual Machine to any computer hardware/operating system, one is assured that all code compiled for it will run on that system. This forms the basis for Java's portability. Another feature which Java employs in order to guarantee portability is by creating a single standard for data sizes irrespective of processor or operating system platforms.

### JAVA IS HIGH-PERFORMANCE

The Java language supports many high-performance features such as **multithreading**, **just-in-time compiling**, and **native code usage**.

- Java has employed **multithreading** to help overcome the performance problems suffered by interpreted code as compared to native code. Since an executing program hardly ever uses CPU cycles 100 % of the time, Java uses the idle time to perform the necessary garbage cleanup and general system maintenance that renders traditional interpreters slow in executing applications. [NB: Multithreading is the ability of an application to execute more than one task (thread) at the same time e.g. a word processor can be carrying out spell check in one document and printing a second document at the same time.]
- Since the bytecode produced by the Java compiler from the corresponding source code is very close to machine code, it can be interpreted very efficiently on any platform. In cases where even greater performance is necessary than the interpreter can provide, **just-in-time compilation** can be employed whereby the code is compiled at run-time to native code before execution.
- An alternative to just-in-time compilation is to link in **native C code**. This yields even greater performance but is more burdensome on the programmer and reduces the portability of the code.

### JAVA IS DYNAMIC

By connecting to the Internet, a user immediately has access to thousands of programs and other computers. During the execution of a program, Java can dynamically load classes that it requires either from the local hard drive, from another computer on the local area network or from a computer somewhere on the Internet.

### Q8. A) What is container software?

Ans. Container, in the context of Java development, refers to a part of the server that is responsible for managing the lifecycle of Web applications. The Web applications specify the required lifecycle management with the help of a contract presented in XML format. The Web container cannot be accessed directly by a client. Rather, the server manages the Web container, which in turn manages the Web application code.

The container is an important component of Web applications in Java-based J2EE technology. It is responsible for maintaining the individual components on the server side, which include Java servlets, Java server pages and Java server faces. How the services will be provided and accessed is determined by a contract, which is an agreement between the Web application and the container. This provides a considerable amount of security in the J2EE framework because the client applications are unaware of the existence of the container and therefore it cannot be accessed directly. Thus, the Web container is responsible for initializing Web application components and invoking client requests on the components.

### Q8.B) what are the different types of EJB's? Where are EJB's used?

#### Types of EJB

In most texts on this subject you will see pictures of a 3-tier system containing boxes labeled "EJB." It is actually more important to identify what application functionality that should go into an EJB.

At the start of application development, regardless of the precise development process used there is generally some analysis that delivers a model, or set of classes and packages, that represent single or grouped business concepts.

Two types of functionality are generally discovered during analysis—data manipulation and business process flow. The application model will usually contain data-based classes such as `Customer` or `Product`. These classes will be manipulated by other classes or roles that represent business processes, such as `Purchaser` or `CustomerManager`. There are different types of EJB that can be applied to these different requirements:

- **Session EJB**—A Session EJB is useful for mapping business process flow (or equivalent application concepts). There are two sub-types of Session EJB—stateless and stateful—that are discussed in more detail on Day 5. Session EJBs commonly represent "pure" functionality and are created as needed.
- **Entity EJB**—An Entity EJB maps a combination of data (or equivalent application concept) and associated functionality. Entity EJBs are usually based on an underlying data store and will be created on the data within that store.
- **Message-Driven EJB**—A Message-driven EJB is very similar in concept to a Session EJB, but is only activated when an asynchronous message arrives.

As an application designer, you should choose the most appropriate type of EJB based on the task to be accomplished.

## Common Uses of EJBs

So, given all of this, where would you commonly encounter EJBs and in what roles? Well, the following are some examples:

- In a Web-centric application, the EJBs will provide the business logic that sits behind the Web-oriented components, such as servlets and JSPs. If a Web-oriented application requires a high level of scalability or maintainability, use of EJBs can help to deliver this.
- Thick client applications, such as Swing applications, will use EJBs in a similar way to Web-centric applications. To share business logic in a natural way between different types of client applications, EJBs can be used to house that business logic.
- Business-to-business (B2B) e-commerce applications can also take advantage of EJBs. Because B2B e-commerce frequently revolves around the integration of business processes, EJBs provide an ideal place to house the business process logic. They can also provide a link between the Web technologies often used to deliver B2B and the business systems behind.
- Enterprise Application Integration (EAI) applications can incorporate EJBs to house processing and mapping between different applications. Again, this is an encapsulation of the business logic that is needed when transferring data between applications (in this case, in-house applications).

These are all high-level views on how EJBs are applied. There are various other EJB-specific patterns and idioms that can be applied when implementing EJB-based solutions. These are discussed more on Day 18, "Patterns."

## Why Use EJBs?

Despite the recommendations of the J2EE Blueprints, the use of EJBs is not mandatory. You can build very successful applications using servlets, JSPs or standalone Java applications.

As a general rule of thumb, if an application is small in scope and is not required to be highly scalable, you can use J2EE components, such as servlets, together with direct JDBC connectivity to build it. However, as the application complexity grows or the number of concurrent users increases, the use of EJBs makes it much easier to partition and scale the application. In this case, using EJBs gives you some significant advantages.

The main advantage of using EJBs in your application is the framework provided by the EJB container. The container provides various services for the EJB to relieve the developer from having to implement such services, namely

- Distribution via proxies—The container generates a client-side stub and server-side skeleton for the EJB. The stub and skeleton use RMI over IIOP to communicate.
- Lifecycle management—Bean initialization, state management, and destruction is driven by the container, all the developer has to do is implement the appropriate methods.
- Naming and registration—The EJB container and server provide the EJB with access to naming services. These services are used by local and remote clients to look up the EJB and by the EJB itself to look up resources it may need.

- Transaction management—Declarative transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.
- Security and access control—Again, declarative security provides a means for the developer to easily delegate the enforcement of security to the container.
- Persistence (if required)—Using the Entity EJB's Container-Managed Persistence mechanism (CMP), state can be saved and restored without having to write a single line of code.

## 2009

### Q. What do you mean by Declaration, definition and Usage. Illustrate with example of each.

Soln. Declaration, Definition and Usage

**Declaration:** It's a methodology used for familiarizing the compiler with the contents/components of the code. Declaration uses the predefined keywords to declare the various elements that are involved in the program. Declaration can be for classes, interfaces, functions, variables etc.

In Java programming, a class is defined by a class declaration, which is a piece of code that follows this basic form:

```
[public] class ClassName {class-body}
```

The public keyword indicates that this class is available for use by other classes. Although it's optional, you usually include it in your class declarations so that other classes can create objects from the class you're defining.

The ClassName provides the name for the class. You can use any identifier you want to name a class, but the following three guidelines can simplify your life:

- Begin the class name with a capital letter. If the class name consists of more than one word, capitalize each word: for example, Ball, RetailCustomer, and GuessingGame.
- Whenever possible, use nouns for your class names. Classes create objects, and nouns are the words you use to identify objects. Thus, most class names should be nouns.
- Avoid using the name of a Java API class. No rule says that you absolutely have to, but if you create a class that has the same name as a Java API class, you have to use fully qualified names (such as java.util.Scanner) to tell your class apart from the API class with the same name.

The class body of a class is everything that goes within the braces at the end of the class declaration, which can contain the following elements:

- Fields: Variable declarations define the public or private fields of a class.
- Methods: Method declarations define the methods of a class.
- Constructors: A constructor is a block of code that's similar to a method but is run to initialize an object when an instance is created. A constructor must have the same name as the class itself, and although it resembles a method, it doesn't have a return type.
- Initializers: These stand-alone blocks of code are run only once, when the class is initialized. The two types are static initializers and instance initializers.

- Other classes: A class can include another class, which is then called an inner class or a nested class.

A public class must be written in a source file that has the same name as the class, with the extension .java. A public class named Greeter, for example, must be placed in a file named Greeter.java.

You can't place two public classes in the same file. For example, you can't have a source file that looks like this:

```
public class Class1
{
    // class body for Class1 goes here
}
public class Class2
{
    // class body for Class2 goes here
}
```

The compiler will generate an error message indicating that Class2 is a public class and must be declared in a file named Class2.java. In other words, Class1 and Class2 should be defined in separate files.

Method declaration:

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                             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, and others you will learn about later.
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—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

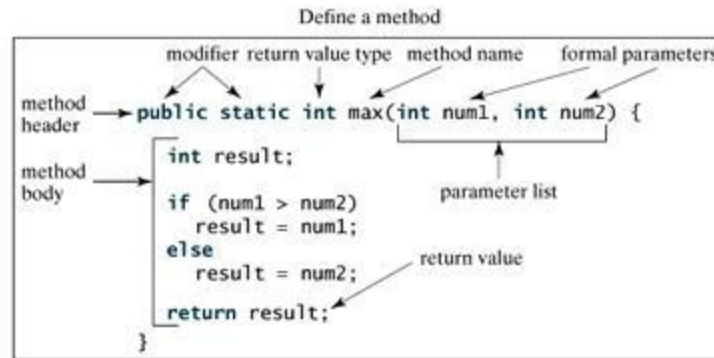
The signature of the method declared above is:

calculateAnswer(double, int, double, double)

**Definition**

The declaration part familiarizes the elements to the compiler in contrary definition provides information of what action has to be done as part of a particular program. The definition is generally provided as part of method or method body.

Method Body: The method body contains a collection of statements that define what the method does.



**Note:** In certain other languages, methods are referred to as procedures and functions. A method with a nonvoid return value type is called a function; a method with a void return value type is called a procedure.

**Example:**

Here is the source code of the above defined method called max(). This method takes two parameters num1 and num2 and returns the maximum between the two:

```

/** Return the max between two numbers */
public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
    
```

### Usage of definition and declaration

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

If the method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(30, 40);
```

If the method returns void, a call to the method must be a statement. For example, the method `println` returns void. The following call is a statement:

```
System.out.println("Welcome to Java!");
```

### Example:

Following is the example to demonstrate how to define a method and how to call it:

```
public class TestMax {
    /** Main method */
    public static void main(String[] args) {
        int i = 5;
        int j = 2;
        int k = max(i, j);
        System.out.println("The maximum between " + i +
            " and " + j + " is " + k);
    }

    /** Return the max between two numbers */
    public static int max(int num1, int num2) {
        int result;
        if (num1 > num2)
            result = num1;
        else
            result = num2;

        return result;
    }
}
```

This would produce following result:

```
The maximum between 5 and 2 is 5
```

This program contains the main method and the max method. The main method is just like any other method except that it is invoked by the JVM.

The main method's header is always the same, like the one in this example, with the modifiers public and static, return value type void, method name main, and a parameter of the String[] type. String[] indicates that the parameter is an array of String.

### Q. "Byte Code" is the key that allows Java to solve both the security and portability problems.

**Soln.** Java bytecode is the form of instructions that the Java virtual machine executes. Bytecode is computer object code that is processed by a program, usually referred to as a virtual machine, rather than by the "real" computer machine, the hardware processor. The virtual machine converts each generalized machine instruction into a specific machine instruction or instructions that this computer's processor will understand. Bytecode is the result of compiling source code written in a language that supports this approach. Most computer languages, such as C and C++, require a separate compiler for each computer platform - that is, for each computer operating system and the hardware set of instructions that it is built on. Windows and the Intel line of microprocessor architectures are one platform; Apple and the PowerPC processors are another. Using a language that comes with a virtual machine for each platform, your source language statements need to be compiled only once and will then run on any platform. The best-known language today that uses the bytecode and virtual machine approach is Java. This is how bytecode resolves the portability problem.

Rather than being interpreted one instruction at a time, Java bytecode can be recompiled at each particular system platform by a just-in-time compiler. Usually, this will enable the Java program to run faster. In Java, bytecode is contained in a binary file with a .CLASS suffix.

Security is the practice by which individuals and organizations protect their physical and intellectual property from all forms of attack and pillage. In order to combat potential security threats, users need programs they can rely on. Moreover, developers are looking for a development platform that has been designed with built-in security capabilities. This is where the Java platform comes in. As a matter of fact, Java is designed from the ground up for network-based computing, and security measures are an integral part of Java's design. Compilers and a bytecode verifier ensure that only legitimate Java code is executed. The bytecode verifier, together with the Java virtual machine, guarantees language type safety at run time.

Java source code → Java compiler → trusted byte code → Byte code verifier

Java Bytecode Verifier : Java compiler compiles source programs into bytecodes, and a trustworthy compiler ensures that Java source code does not violate the safety rules. At runtime, a compiled code fragment can come from anywhere on the net, and it is unknown if the code fragment comes from a trustworthy compiler or not. So, practically the Java runtime simply does not trust the incoming code, and instead subjects it to a series of tests by bytecode verifier.

The bytecode verifier is a mini theorem prover, which verifies that the language ground rules are respected. It checks the code to ensure that [5]:

- Compiled code is formatted correctly.

- Internal stacks will not overflow or underflow.
- No "illegal" data conversions will occur (i.e., the verifier will not allow integers to serve as pointers).

This ensures that variables will not be granted access to restricted memory areas.

- Byte-code instructions will have appropriately-typed parameters (for the same reason as described in the previous bullet).
- All class member accesses are "legal". For instance, an object's private data must always remain private.

The bytecode verifier ensures that the code passed to the Java interpreter is in a fit state to be executed and can run without fear of breaking the Java interpreter

Q. Compare "Passing parameter by value" with "Passing parameter by reference". Illustrate with an example.

Soln. "Passing parameter by value" with "passing parameter by reference"

**Call by Value:**

When values of built in types are passed as arguments to a function. It is known as Call by value. A copy of the argument is passed as parameter. So the caller method and the called method are working on different sets of data. The changes made to formal parameters in the called function are not reflected in the actual arguments class.

Pass By Value: It refers to pass the variables or a constant that holds the primitive data type to a method.

Example:

```
public class PassByValue{
public static void display(int a){
a=5;
System.out.println(a);
}
public static void main(String[]args){
int i=20;
System.out.println(i);
display(i);
```

```
}
}
```

**Call by Reference:**

In this approach, an object is passed as an argument, which is assigned to an object reference are directly reflected to actual argument. The address of the object on the heap, w.r.t java, is passed as the parameter. So modifying one will have an effect on the other.

Pass By Reference: It refers to pass an object variable to a method.

Example:

```
public class PassByRefernce{
public static void display(StringBuffer sb){
sb=sb.insert(8,"to");
System.out.println(sb);
}
```

```
public static void main(String[]args){
StringBuffer sb1=new StringBuffer("Welcome RoseIndia");
System.out.println(sb1);
display(sb1);
}
}
```

**Q. What do you mean by “Structure” in structured languages? Discuss.**

**Soln Structure in structured languages**

Structure is often composed of simple, hierarchical program flows. These are sequence, selection, and repetition:

- "Sequence" refers to an ordered execution of statements.
- In "selection" one of a number of statements is executed depending on the state of the program. This is usually expressed with keywords such

`asif..then..else..endif`, `switch`, or `case`. In some languages keywords cannot be written verbatim, but must be stopped.

- In "repetition" a statement is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as `while`, `repeat`, `for` or `do..until`. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).

A set of quality standards that make programs more verbose but more readable, reliable, and easily maintained. The goal of structured programming is to avoid spaghetti code caused by overreliance on GOTO statements, a problem often found in BASIC and FORTRAN programs. Structured programming-such as that promoted by C, Pascal Modula-2, and the dBASE software command language - insists that the overall program structure reflect what the program is supposed to do, beginning with the first task and proceeding logically. Indentations help make the logic clear, and the programmer is encouraged to use loops and branch control structures and named procedures rather than GOTO statements.

## Q. What are the identifying characteristics of a object oriented programming paradigm.

Soln. characteristics of a object oriented programming

**Object-oriented programming (OOP)** is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

Object oriented characteristics:

Abstraction

"Eliminate the Irrelevant, Amplify the Essential"

Abstraction is to focus on the essential and discard the irrelevant. Abstraction in Java is achieved by using interface and abstract class in Java. An interface or abstract class is something which is not concrete , something which is incomplete. In order to use interface or abstract class we need to extend and implement abstract method with concrete behavior. One example of Abstraction is creating interface to denote common behavior without specifying any details about how that behavior works e.g. You create an interface called `Server` which as `start()` and `stop()` method. This is called abstraction of `Server` because every server should have way to start and stop and details may differ.

Encapsulation

"Hiding the Unnecessary"

**Encapsulation in Java** or object oriented programming language is a concept which enforce protecting variables, functions

from outside of class, in order to better manage that piece of code and having least impact or no impact on other parts of program due to change in protected code. *Encapsulation in Java* is visible at different places and Java language itself provide many construct to encapsulate members. You can completely encapsulate a member be it a variable or method in Java by using `private` keyword and you can even achieve a lesser degree of encapsulation in Java by using other access modifier like `protected` or `public`.

Encapsulation is hiding the unnecessary. Meaning things that you do not want other people/classes to know are to be hidden

In a way encapsulation protect the system, since it hides stuff that can be vulnerable to malicious attack.

### Advantage of Encapsulation in Java and OOPS

Here are few advantages of using **Encapsulation** while writing code Java or any Object oriented programming language:

1. Encapsulated Code is more flexible and easy to change with new requirements.
2. Encapsulation in Java makes unit testing easy.
3. Encapsulation in Java allows you to control who can access what.
4. Encapsulation also helps to write immutable class in Java which are a good choice in multi-threading environment.
5. Encapsulation reduce coupling of modules and increase cohesion inside a module because all piece of one thing are encapsulated in one place.
6. Encapsulation allows you to change one part of code without affecting other part of code.

### Inheritance

#### “Modeling the Similarity”

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order. When we talk about inheritance the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.

#### IS-A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal{  
}  
  
public class Mammal extends Animal{
```

```
}  
  
public class Reptile extends Animal{  
}  
  
public class Dog extends Mammal{  
}
```

Now based on the above example, In Object Oriented terms following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are sub classes of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now if we consider the IS-A relationship we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

#### Example:

```
public class Dog extends Mammal{  
  
    public static void main(String args[]){  
  
        Animal a = new Animal();  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal);  
        System.out.println(d instanceof Mammal);  
        System.out.println(d instanceof Animal);  
    }  
}
```

This would produce following result:



```
true
true
```

```
true
```

Since we have a good understanding of the **extends** keyword let us look into how the **implements** keyword is used to get the IS-A relationship.

The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

**Example:**

```
public interface Animal {}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

**The instanceof Keyword:**

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal

```
interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{
    public static void main(String args[]){

        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}
```

This would produce following result:

```
true
true
true
```

**HAS-A relationship:**

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets us look into an example:

```
public class Vehicle{}
public class Speed{}
```

```
public class Van extends Vehicle{
    private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

In Object Oriented feature the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. SO basically what happens is the users would ask the Van class to do a certain action and the Vann class will either do the work by itself or ask another class to perform the action.

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

```
public class extends Animal, Mammal{}
```

However a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance

## Polymorphism

"Same Function different behavior"

**Polymorphism** is an OOPS concept which advice use of common interface *instead of concrete implementation* while writing code. When we program for interface our code is capable of handling any new requirement or enhancement arise in near future due to new implementation of our common interface. If we don't use common interface and rely on concrete implementation, we always need to change and duplicate most of our code to support new implementation. Its not only Java but other object oriented language like C++ also supports polymorphism and it comes as fundamental along with other OOPS concepts like Encapsulation, Abstraction and Inheritance.

Java has excellent support of polymorphism in terms of Inheritance, method overloading and method overriding. Method overriding allows Java to invoke method based on a particular object at run-time instead of declared type while coding. To get hold of concept let's see an **example of polymorphism in Java**:

```
public class TradingSystem{
    public String getDescription(){
        return "electronic trading system";
    }
}

public class DirectMarketAccessSystem extends TradingSystem{
```

```
public String getDescription(){
    return "direct market access system";
}

}

public class CommodityTradingSystem extends TradingSystem{
    public String getDescription(){
        return "Futures trading system";
    }
}
```

Here we have a super class called `TradingSystem` and there two implementation `DirectMarketAccessSystem` and `CommodityTradingSystem` and here we will write code which is flexible enough to work with any future implementation of `TradingSystem` we can achieve this by using Polymorphism in Java which we will see in further example.

Where you can use *Polymorphism in Java* while [writing code](#).

#### 1) Method argument:

Always use super type in method argument that will give you leverage to pass any implementation while invoking method. For example:

```
public void showDescription(TradingSystem tradingSystem){
    tradingSystem.description();
}
```

If you have used concrete implementation e.g. `CommodityTradingSystem` or `DMATradingSystem` then that code will require frequent changes whenever you add new Trading system.

#### 2) Variable names:

Always use Super type while you are storing reference returned from any [Factory method in Java](#), This gives you flexibility to accommodate any new implementation from Factory. Here is an example of polymorphism while writing Java code which you can use retrieving reference from Factory:

```
String systemName = Configuration.getSystemName();
```

```
TradingSystem system = TradingSystemFactory.getSystem(systemName);
```

### 3) Return type of method

Return type of any method is another place where you should be using interface to take advantage of Polymorphism in Java. In fact this is a requirement of [Factory design pattern in Java](#) to use interface as return type for factory method.

```
public TradingSystem getSystem(String name){  
    //code to return appropriate implementation  
}
```

**Q. What are the characteristics of a objected oriented programming paradigm.**  
soln. Characteristic of structured programming language.

#### Why To Use Structured Programming?

In computer programming, the spaghetti code (unstructured code) confuses the program flow. The use of GOTO or jump statements in programming kills the natural flavour of programming. Hence Structured programming (Modular Programming) is used for well-organized programs that are easier to –

- Design
- Read and understand
- Modify
- Test and debug
- Compare with other programs
- Properly documented
- Efficient

#### Characteristics of Structured Programming

1. A structured program is based on top down approach. In other words, the problem is broken down in to major components, each of which is further broken down if necessary. Therefore the process involves working from the most general down to the most specific.
2. Each module has one entry and one exit point.
3. Use of GOTO or jump statements is avoided.
4. A rule of thumb is that the modules should not be more than a half page long. If they are longer than this, they should preferably be split into two or more sub modules.
5. Two way decisions are based on IF...THEN....ELSE, and nested IF statements.
6. Repetition processes are implemented through various loops available.
7. It is much easier for the programmers to debug the structured programs.

### Q. Briefly describe history and evolution of java

Soln. Java is an object-oriented programming language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s. Unlike conventional languages which are

generally designed either to be compiled to native (machine) code, or to be interpreted from source code at runtime, Java is intended to be compiled to a bytecode, which is then run (generally using JIT compilation) by a Java Virtual Machine.

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled by applets, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the Thread class in which the methods `suspend()`, `resume()`, and `stop()` were deprecated.

The next major release of Java was J2SE 1.3. This version of Java was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword `assert`, chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions

The release of J2SE 5 was a profoundly significant event in the life cycle of Java. Unlike most of the previous Java upgrades, which offered important, but incremental improvements,

J2SE 5 fundamentally expands the scope, power, and range of the language. Not since its original launch nearly a decade ago has a release of Java been so important, or so eagerly awaited. To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

- Generics
- Metadata
- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style forloop
- Variable-length arguments (varargs)
- Static import
- Formatted I/O
- Concurrency utilities
- Upgrades to the API

This is not a list of minor tweaks or incremental upgrades. Each item in the list represents a

significant addition to the Java language. Some, such as generics, the enhancedfor, and varargs, introduce new syntax elements. Others, such as autoboxing and auto-unboxing, alter the semantics of the language. Metadata adds an entirely new dimension to programming. In all cases, the impact of these additions goes beyond their direct effects. They change the very character of Java, itself. Thus, the current product is called J2SE 5, and the developer's kit is called JDK 5. However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number. Thus, "5" is the external version number, and "1.5" is the internal version number.

### Q. "Java is a strongly typed language". Discuss

Soln. Java is a strongly typed programming language because every variable must be declared with a data type. A variable cannot start off life without knowing the range of values it can hold, and once it is declared, the data type of the variable cannot change.

We call **C/C++** and **Java** are **strongly typed languages** and **JavaScript** and **PERL** are **loosely typed languages**.

```
int marks = 20;
String str = "way2java.com";
boolean raining = true;
```

The variable must be declared along with the data type. Similarly, an array object must be instantiated with the size. This is a feature of strongly typed languages.

Just reverse with loosely typed languages. Observe the variable declaration in JavaScript, an object-based language.

```
var marks = 20.  
var str = "way2java.com";  
var raining = true;
```

The string and data types int and boolean are replaced by var that stands for variable. Depending on the value assigned, var becomes a number or a string or boolean. This is done implicitly by the JavaScript interpreter. It looks strange and unbelievable for a C or Java programmer. A JavaScript programmer need not remember the list of data types (actually, they exist), JavaScript supports.

It looks nice and easy with loosely typed languages. But it raises a lot of problems in coding. Observe the following.

Say in Java language:

```
int cost = 10;  
int cost = 20;
```

It is compilation error as two times cost is initialized. You must declare only ones and can be reassigned number of times as follows.

```
int cost = 10;  
cost = 20;
```

Same thing in JavaScript or PERL is not error. Observe the code.

```
var cost = 10;  
var cost = 20;
```

The earlier cost value is overridden with 20, but not error; earlier value is lost. This is a very dangerous situation which leads problems. If an article costs Rs.10 and the programmer declared correctly earlier. In the later part of the code, he forgot that he declared and re-declares with a different value (which may be wrong). For this reason, the object-based languages like JavaScript, PERL and Ruby etc. have limited scope in programming and are not used as full-fledged languages to develop software like banking, insurance etc.

A strongly typed language compiler enforces strict rules over the operations, what programmer can do, on data types and also passing parameters and return type to a method. An advantage of strongly typed language is it gives consistency over the results (by declaring the variables with data types). Specific operations are allowed on certain types. For example boolean cannot be used in addition, but an int can be used in addition. Boolean is used in control structures. This type of accidental wrong coding raises error by a strongly typed language.

### **Q. “It is simple to manage stack memory than heap memory”. Discuss?**

Heap: When program allocate memory at runtime using calloc and malloc function, then memory gets allocated in heap. when some more memory need to be allocated using calloc and malloc function, heap grows upward as shown in above diagram.

- Stack: Stack is used to store your local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. When a new stack frame needs to be added (as a result of a newly called function), the stack grows downward.

The stack and heap are traditionally located at opposite ends of the process’s virtual address space. The stack grows automatically when accessed, up to a size set by the kernel (which can be adjusted with `setrlimit(RLIMIT_STACK, ...)`). The heap grows when the memory allocator invokes the `brk()` or `sbrk()` system call, mapping more pages of physical memory into the process’s virtual address space. Implementation of both the stack and heap is usually down to the runtime/OS. Often games and other applications that are performance critical create their own memory solutions that grab a large chunk of memory from the heap and then dish it out internally to avoid relying on the OS for memory.

Stacks in computing architectures are regions of memory where data is added or removed in a last-in-first-out manner. Because the data is added and removed in a last-in-first-out manner, stack allocation is very simple and typically faster than heap-based memory allocation (also



known as dynamic memory allocation). Another feature is that memory on the stack is automatically, and very efficiently, reclaimed when the function exits, which can be convenient for the programmer if the data is no longer required. If however, the data needs to be kept in some form, then it must be copied from the stack before the function exits. Therefore, stack based allocation is suitable for temporary data or data which is no longer required after the creating function exits. Stores local data, return addresses, used for parameter passing.

Deallocating the stack is pretty simple because you always deallocate in the reverse order in which you allocate. Stack stuff is added as you enter functions, the corresponding data is removed as you exit them. This means that you tend to stay within a small region of the stack unless you call lots of functions that call lots of other functions

The heap contains a linked list of used and free blocks. New allocations on the heap (by new or malloc) are satisfied by creating a suitable block from one of the free blocks. This requires updating list of blocks on the heap. This meta information about the blocks on the heap is also stored on the heap often in a small area just in front of every block.

• The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system) (see Footnote 6).

- Stored in computer RAM like the stack.
- Variables on the heap must be destroyed manually and never fall out of scope. The data is freed with delete, delete[] or free
- Slower to allocate in comparison to variables on the stack.
- Used on demand to allocate a block of data for use by the program
- Can have fragmentation when there are a lot of allocations and deallocations
- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
- Responsible for memory leaks

Another factor to consider is that there will be some overhead in the management of the heap.

•The heap manager will need to keep track of the amount of heap used or remaining, the size of the block being allocated and will usually contain a pointer to the next available memory location available.

•One more factor is that the tools may reserve a larger block of memory than requested to accommodate the memory architecture. For instance, EWARM compiler always allocates blocks of memory in multiples of 8 bytes to maintain stack alignment

It is difficult to estimate how much heap space you will need without some sort of tool to help analyze your dynamic memory needs

- There exist such tools for desktop Java (HAT, Heap Analysis Tool)
- No such tool as yet for Embedded C/C++

- As embedded systems have limited resources, dynamic memory should be used sparingly due to overhead and the possibility of heap fragmentation

Unlike a heap, a stack will never become fragmented or suffer from memory leaks

## 2010

### Q1: Write answer in complete one sentence

(20 marks)

- 1) A
- 2) D
- 3) B
- 4) B
- 5) B
- 6) C
- 7) C
- 8) H (C)
- 9) C
- 10) C
- 11) B
- 12) B
- 13) B
- 14) A
- 15) C
- 16) B
- 17) B
- 18) A
- 19) A
- 20) A

**Q2a) What is a structure? Explain with example. How structure is different from Array? Distinguish between Structure and Class (8 marks) [Not Applicable]**

**Q2b) Describe all the various features of Java (8 marks)**

Ans) [Ref: Java Complete Reference – Page 9, 10, 11]

The features of Java are explained below:

#### **Object-Oriented**

Java is Object Oriented Programming Language and is strongly typed in nature. Java balances between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

#### **Robust**

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. 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.

### **Security**

Java Programs are more secure than the downloaded “normal” program written in other language. The other binary code have risk of containing a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack. Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

### **Portability**

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The same code must work on all computers. Therefore, some means of generating portable executable code was needed. The key that allows Java to solve both the security and the portability problems just described is that 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).

### **Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things 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. Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

### **Architecture-Neutral**

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. Java is “write once; run anywhere, any time, forever” language.

### **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 executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, 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. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

### **Distributed**

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

### **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. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

**Q2c) What is pointer variable. How it differs from reference variable? (4 marks) [Not Applicable]**

**Q3a) What is the meaning of abstract method? What is the advantage of declaring class as abstract? What is the difference between abstract and final class? (8 marks)**

There are situations in which one wants to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes one will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class Figure used in the preceding example. The definition of `area()` is simply a placeholder. It will not compute and display the area of any type of object. One may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class Triangle. It has no meaning if `area()` is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. The solution to this requirement is the abstract method. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

`abstract type name(parameter-list);`

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract. Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();
    // concrete methods are still allowed in abstract classes
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}
class B extends A {
    void callme() {
        System.out.println("B's implementation of callme.");
    }
}
class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();
        b.callme();
        b.callmetoo();
    }
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class A implements a concrete method called callmetoo(). This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example. Using an abstract class, you can improve the Figure class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares area() as abstract inside Figure. This, of course, means that all classes derived from Figure must override area().

```
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }
    // area is now an abstract method
    abstract double area();
}
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }
    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

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

figref = t;

System.out.println("Area is " + figref.area());

}

}
```

As the comment inside main( ) indicates, it is no longer possible to declare objects of type Figure, since it is now abstract. And, all subclasses of Figure must override area( ). To prove this to yourself, try creating a subclass that does not override area( ). You will receive a compile-time error.

Although it is not possible to create an object of type Figure, you can create a reference variable of type Figure. The variable figref is declared as a reference to Figure, which means that it can be used to refer to an object of any class derived from Figure. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

#### Difference between Final and Abstract Class

##### Abstract Class:

- Abstract class shall have one or more methods defined to be abstract
- Subclass inheriting from Abstract Class should either implement the abstract method or else it will be treated as Abstract class itself
- Abstract class cannot be instantiated however reference variable can be created for Abstract Class

##### Final Class:

- Final Class shall have specifier set to Final
- Final class cannot be inherited further
- Final Class can be instantiated.

### Q3b) What are the different types of function declaration? How will you declare a function outside and inside the class? (8 marks) [Not Applicable]

### Q3c) Compare class and object with suitable example (4 marks)

Class defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
  
    // ...  
    type instance-variableN;  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```

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

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

#### Declaring Object

Obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator.

The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure. In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

```
Box mybox = new Box();
```



This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
```

```
mybox = new Box(); // allocate a Box object
```

The first line declares `mybox` as a reference to an object of type `Box`. After this line executes, `mybox` contains the value `null`, which indicates that it does not yet point to an actual object. Any attempt to use `mybox` at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to `mybox`. After the second line executes, you can use `mybox` as if it were a `Box` object. But in reality, `mybox` simply holds the memory address of the actual `Box` object.

### Q4a) What is object oriented paradigm? Explain the various features of object oriented programming with example (8 marks) [Repeat in 2004]

Features of Object Oriented Programming

#### Abstraction

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to do something. This is the essence of object-oriented programming.

Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

#### Encapsulation

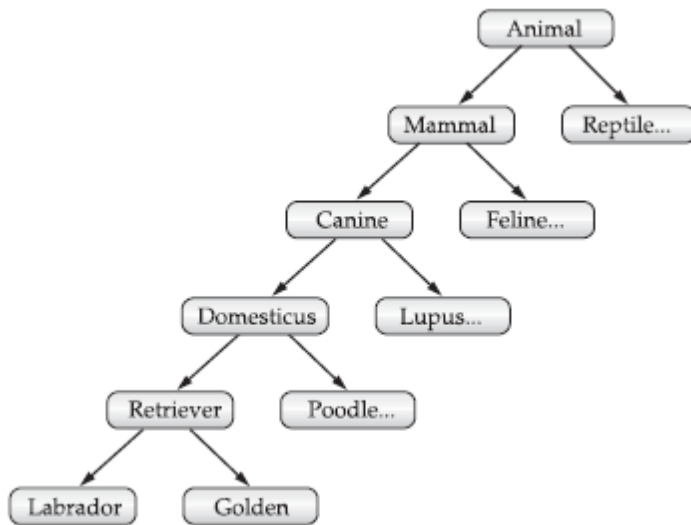
Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects. In Java, the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods. (If you are familiar with C/C++, it may help to know that what a Java programmer calls a method, a C/C++ programmer calls a function.) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data. 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.

## **Inheritance**

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the class definition for animals. If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a subclass of animals, where animals are referred to as mammals' superclass. Since mammals are simply more precisely specified animals, they inherit all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.



Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization (see Figure 2-2). This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

**Polymorphism**

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler’s job to select the specific action (that is, method) as it applies to each situation. You, the

programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose!

#### **Q4b) Explain practical usage of interface with example Describe how it differs from class (8 marks)**

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

An interface is defined much like a class. This is the general form of an interface:

#### **Q4c) What do you mean by Exception Handling? What are the types of exception? Compare them (4 marks)**

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically

thrown by the Java run-time system. To manually throw an exception, use the keyword `throw`. Any exception that is thrown out of a method must be specified as such by a `throws` clause. Any code that absolutely must be executed after a `try` block completes is put in a `finally` block.

This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed after try block ends  
}
```

### Types of Exception

All exception types are subclasses of the built-in class `Throwable`. Thus, `Throwable` is at the top of the exception class hierarchy. Immediately below `Throwable` are two subclasses that partition exceptions into two distinct branches. One branch is headed by `Exception`. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of `Exception`, called `RuntimeException`. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by `Error`, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type `Error` are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Inside the standard package `java.lang`, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type `RuntimeException`. These exceptions need not be included in any method's `throws` list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in `java.lang` are listed below

Exception	Meaning
<code>ArithmeticException</code>	Arithmetic error, such as divide-by-zero.
<code>ArrayIndexOutOfBoundsException</code>	Array index is out-of-bounds.
<code>ArrayStoreException</code>	Assignment to an array element of an incompatible type.
<code>ClassCastException</code>	Invalid cast.

The following lists those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions.

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the <b>Cloneable</b> interface.
IllegalAccessException	Access to a class is denied.

### Q5a) Compare and contrast C++ and Java (8 marks)

[Self written. Answers were not available from Complete Reference book. Readers may kindly verify the validity or refer other writer's content]

Both C++ and Java are object oriented programming language with following differences

- 1) C++ was created as extension to C. Hence a code written in C syntax can be compiled by any ANSI C++ compliant compiler. Java was created from scratch and is not an extension of any existing language.
- 2) Both Java and C++ support following primitive types: int, char, float, double
- 3) Java provides with an additional access specified (package) in addition to the three access specifiers which are also supported in C++ viz., Public, Protected, Private
- 4) Java compiler produces byte-code which is then interpreted by JVM. C++ compiler compiles to native code (object code).
- 5) C++ supports pointers which enables programmers to write unsafe code. Java does not have support for pointers or any other means to access memory locations.
- 6) C++ requires compiling and linking with libraries to create executable code. The linking can be static linking or dynamic linking to create static libraries or shared objects/libraries in C++. Java only allows for creation of .class files or executable jars which essentially is an archive package created with all libraries contained in it.
- 7) C++, alike C, follows convention of header files (extension ".h") carrying declarations and source files (extension ".c" or ".cpp") which carry definitions. C++ therefore is vulnerable to cyclic dependency problems where in A and B both could be indirectly dependent on each other. Java follows convention of single source code file with extension ".java".
- 8) C++ supports multiple inheritance and is amenable to Diamond-of-death problem. Java does not support multiple inheritance at class level, however permits multiple inheritance at Interface level.
- 9) Contract establishment in Java is through creation of Interfaces while in C++ is through Abstract Classes with pure virtual functions. While C++ does not have support for interfaces, Java does not have support for "struct" (structures).
- 10) C++ compilers exists for various architecture which requires code to be written once and then ported + compiled on the specific architecture (quite often through cross-compiling). Native objects produced by C++ compilers are not executable across varying architectures. In contrast, Java provides with JVM on different architectures. Presence of JVM enables portability of bytecode. The limitation of how much the bytecode would work on the given architecture will depend upon the JVM capabilities.

- 11) Modern day Java support generic programming like C++. JDK prior to 5 did not have support for Generic Programming.

## Q5b) Write brief note on

### 1. JVM

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). In essence, the original JVM was designed as an interpreter for bytecode. Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs. The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, Sun began supplying its HotSpot technology not long after Java's initial release. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

### 2. Development tools for JDK

[Self written. Reader may take necessary caution...]

There are different development tools that can be used with JDK. While most developers are comfortable with use of IDE such as Eclipse or Netbeans (or Developer Studio) which provides with following facilities

- a) Auto-alignment or beautification of code to improve readability of source code.
- b) Use a disciplined framework for creation of binaries (.class files and/or .jar libraries) in specific directories instead of placing them with source code itself.
- c) Linkage with compiler wherein compilation and debugging can be invoked from within IDE itself.
- d) Plugins for facilitating development of different types of applications viz., Rich UI applications, JEE applications, Struts development, Springs framework etc.
- e) Linkage with Application or Web Servers such as Oracle Application Server, JBoss Application Server, Glassfish Application Server, Tomcat etc
- f) Creation of API documentations through doxygen etc

Source code editors:

Apart from such IDEs, one may use standard editors such as wordpad/notepad (or emacs/vim on Linux) to create source code files.

Compilers:

There are different flavors of Java Compilers available in addition to the Oracle's JDK viz., OpenJDK, Icedtea, Gnu Java, JRockit etc

Documentation generators:

Doxygen is the most popularly used documentation generator which parses source code for comment blocks starting with "///" to locate developer's intentional comments/instructions in the documentation.

### Q5c) What are the uses of Keyword final in Java? Give example (4 marks)

#### Final Variable

A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared. For example:

```
final int FILE_NEW = 1;
```

```
final int FILE_OPEN = 2;
```

```
final int FILE_SAVE = 3;
```

```
final int FILE_SAVEAS = 4;
```

```
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use FILE\_OPEN, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant.

#### Final Method

Using final to Prevent Overriding While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

```
class A {  
  
    final void meth() {  
  
        System.out.println("This is a final method.");  
  
    }  
}
```



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

Because `meth()` is declared as `final`, it cannot be overridden in `B`. If you attempt to do so, a compile-time error will result. Methods declared as `final` can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small `final` method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with `final` methods. Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since `final` methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

### Using `final` to Prevent Inheritance

`Final` is also used to prevent a class from being inherited. To do this, precede the class declaration with `final`. Declaring a class as `final` implicitly declares all of its methods as `final`, too. As you might expect, it is illegal to declare a class as both `abstract` and `final` since an `abstract` class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a `final` class:

```
final class A {  
// ...  
}  
// The following class is illegal.  
class B extends A { // ERROR! Can't subclass A  
// ...  
}
```

As the comments imply, it is illegal for `B` to inherit `A` since `A` is declared as `final`.

### Q6a) What is array? Write example for each type of array. Describe 2 ways to declare array in Java. How do you get size of an array? Is index checking supported by Java? (8 marks)

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

## One dimensional Array

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

```
type var-name[ ];
```

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named month\_days with the type "array of int":

```
int month_days[];
```

Although this declaration establishes the fact that month\_days is an array variable, no array actually exists. In fact, the value of month\_days is set to null, which represents an array with no value. To link month\_days with an actual, physical array of integers, you must allocate one using new and assign it to month\_days. new is a special operator that allocates memory. You need to use it now to allocate memory for arrays. The general form of new as it applies to one-dimensional arrays appears as follows:

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

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by new will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to month\_days.

```
month_days = new int[12];
```

After this statement executes, month\_days will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero. It is also possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

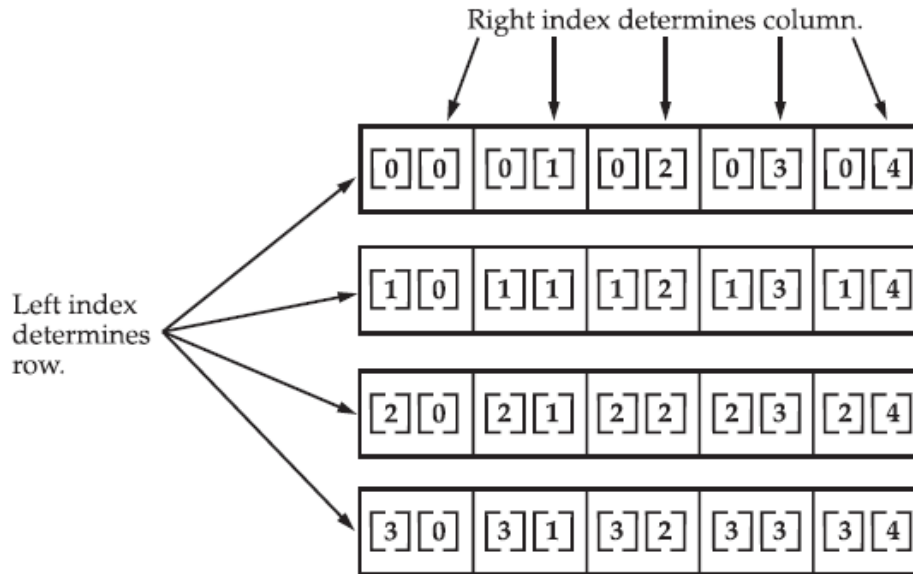
```
int month_days[] = new int[12];
```

## Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.



Given: `int twoD [ ] [ ] = new int [4] [5];`

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of `twoD` when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

While there is no advantage to individually allocating the second dimension arrays in this example, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.

### Alternative array declaration

There is a second form that may be used to declare an array:

```
type[ ] var-name;
```

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int al[] = new int[3];
```

```
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
```

```
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
```

creates three array variables of type int. It is the same as writing

```
int nums[], nums2[], nums3[]; // create three arrays
```

The alternative declaration form is also useful when specifying an array as a return type for a method.

**Q6b) What is friend function? Describe their benefits and limitations. Give suitable example (6 marks) [Not applicable]**

**Q6c) What is the need of dynamic method dispatch? Explain with Example (6 marks)**

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

It is based on important principle wherein a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
```

```
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A

r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme( ) method.

## 2004

### Q2) What are constructors? How are objects created and destroyed in JVM? Explain briefly the functioning of Garbage Collection in JVM.

Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor. A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes. Constructors they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

#### Default Constructor

When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

#### Parameterized Constructor

One can add parameters to Constructor in order to initialize the member variables.

#### Object Creation (New)

The new operator dynamically allocates memory for an object. It has this general form:

```
class-var = new classname( );
```

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created. If no explicit constructor is specified, then Java will automatically supply a default constructor.

#### Object Destruction (Finalize)

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize( ) method on the object. The finalize( ) method has this general form:

```
protected void finalize( )
```

```
{  
// finalization code here  
}
```

It is important to understand that `finalize()` is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—`finalize()` will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on `finalize()` for normal program operation.

### Garbage Collection

Java handles deallocation automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection,

### Q3) What is method/constructor overloading? How is it used in Programming?

It is possible to define two or more methods (or Constructors) within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods (or Constructors) are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.  
  
class OverloadDemo {  
  
void test() {  
System.out.println("No parameters");  
}  
  
// Overload test for one integer parameter.  
  
void test(int a) {
```

```
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

No parameters

a: 10



a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

In this example, test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution. When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

## Q6) What is Threading? Explain the applications and common problems in threading.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

### The Java Thread model

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a singled-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Threads exist in several states. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

### Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch. The rules that determine when a context switch takes place are simple:

- A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

### **Synchronization**

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you 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 is, 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 interprocess 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.

Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

### **Messaging**

After you divide your 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.

## The Thread Class and the Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it. To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads list of which is provided below

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

## The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

## Common Problems

Some common problems pertaining to multi-threading are

- Designing parallel execution is not intuitive: Humans are trained to perform in sequential mode and hence it is not very forthcoming to think of writing parallel code. A poorly designed multithreaded code can degrade the efficiency and performance of the program instead of improving it.
- Synchronizing concurrent access: If two or more threads are operating upon same shared data then data may get corrupted due to nonconsistent access and updates made by concurrent threads. Possible solution is to enforce synchronized access which protects shared data from becoming inconsistent but at the cost of performance overhead as well as risk of entering deadlock.
- Deadlock: This condition occurs when two or more threads are waiting indefinitely for each other to release the lock. Consider this example wherein there are two resources (say R1 and R2) and two threads (say T1 and T2). Assuming that synchronized access is enforced on both resources thereby enabling the first thread to get access to resource holds lock on it. If T1 gets lock on R1 and is waiting for access to R2 which is locked by T2 which in turn is waiting for

access to R1. In this situation neither T1 and T2 are releasing lock and are waiting for other thread to release the lock.

### Q7) What is loop (control) Structure. Explain the 3 types of loop with small code snippets.

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. In other words, loop structures are employed when repetitive tasks are to be performed in some predetermined count or till some condition is met.

#### while

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

```
while(condition) {  
    // body of loop  
}
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to `println( )` is never executed:

```
int a = 10, b = 20;  
  
while(a > b)  
  
System.out.println("This will not be displayed");
```

The body of the while (or any other of Java's loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.

#### do-while

Sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {  
  
    // body of loop  
  
} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, condition must be a Boolean expression.

```
// Demonstrate the do-while loop.
class DoWhile {
    public static void main(String args[]) {
        int n = 10;
        do {
            System.out.println("tick " + n);
            n--;
        } while(n > 0);
    }
}
```

### For loop

Beginning with JDK 5, there are two forms of the for loop. The first is the traditional form that has been in use since the original version of Java. The second is the new "for-each" form.

#### Traditional form

Here is the general form of the traditional for statement:

```
for(initialization; condition; iteration) {
    // body
}
```

If only one statement is being repeated, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.

Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

```
// Demonstrate the for loop.  
  
class ForTick {  
  
public static void main(String args[]) {  
  
int n;  
  
for(n=10; n>0; n--)  
  
System.out.println("tick " + n);  
  
}  
  
}
```

### The For-Each Version of the for Loop

Beginning with JDK 5, a second form of for was defined that implements a “for-each” style loop. A foreach style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. 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. The collection being cycled through is specified by collection. 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.

## Q8) What is JDBC? What are the different types of JDBC drivers?

[Ref: [http://en.wikipedia.org/wiki/Java\\_Database\\_Connectivity](http://en.wikipedia.org/wiki/Java_Database_Connectivity)]

JDBC stands for Java Database Connectivity. This technology is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases.

The API provides a mechanism for dynamically loading the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is used as a connection factory for creating JDBC connections. JDBC connections support creating and executing statements. These may be update statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Additionally, stored procedures may be invoked through a JDBC connection. JDBC represents statements using one of the following classes:

- Statement – the statement is sent to the database server each and every time.

- PreparedStatement – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.
- CallableStatement – used for executing stored procedures on the database.

Update statements such as INSERT, UPDATE and DELETE return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to walk over the result set. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. The row result set has metadata that describes the names of the columns and their types.

There are 4 types of JDBC drivers viz.,

1. Type 1 that calls native code of the locally available ODBC driver.
2. Type 2 that calls database vendor native library on a client side. This code then talks to database over network.
3. Type 3, the pure-java driver that talks with the server-side middleware that then talks to database.
4. Type 4, the pure-java driver that uses database native protocol.