❖ Introduction to Java.

Java is a general purpose object oriented language developed by SUN Microsystems of USA in 1991. originally called **OAK**. James gosling, one of the inventors of the language, but was renamed "Java" in 1995.

Java was designed for development of software for consumer electronics devices like TVs, VCRs and Toaster and such other electronic machines.

The goal had a strong impact on the development team to make the language simple, portable and highly reliable.

The team members of java are James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sherdin.

All of the above developers thought that the existing languages like C & C++ had limitation in terms of both reliability and portability.

However, they modelled their new language Java on C and C++ but removed a number of features of C & C++ that were considered as sources of problems and thus made Java really simple, reliable, portable and powerful language.

❖ History of Java or Milestone of Java.

Following table lists some important milestones in the development of Java.

YEAR	DEVELOPMENT STAGE
1990	Sun Microsystem decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems Programmers headed by James Gosling was formed to undertake this task.
1991	After exploring the possibility of using the most popular object-oriented language **C++**, the team announced a new language named **"Oak".**
1992	The team, known as **Green Project team** by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.
1993	The **World Wide Web (WWW)** appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The **Green Project team** developed a new tiny program known as **applet** that could run on all types of computers connected to the Internet.
1994	The team developed a web browser called **"HotJava"** to locate and run applet programs on Internet.

1995	**Oak** was renamed **"Java"**, due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced to their support to Java.
1996	Java established itself not only as a leader for Internet Programming but also as a general-purpose, object-oriented programming language. SUN releases Java Development Kit 1.0
1997	Sun releases Java Development Kit 1.1 (JDK 1.1)
1998	Sun releases the Java 2 with version 1.2 of Software Development Kit (SDK 1.2).
1999	Sun releases the Java 2 Platform, Standard Edition(J2SE) and Enterprise Edition(J2EE).
2000	J2SE with SDK 1.3 was released
2002	J2SE with SDK 1.4 was released
2004	J2SE with JDK 5.0 (instead of JDK 1.5) was released. This is known as J2SE 5.0.

❖ Java Environment

Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of the system known as **Java Development Kit** (JDK) and the classes and methods are part of the **Java Standard Library** (JSL), also known as **Application Programming Interface** (API).

1. **Java Development Kit:** The Java Development Kit **(JDK)** comes with a collection of tools that are used for developing and running Java Programs. They include:

appletviewer -> (for viewing Java applets)
javac -> (Java Compiler, which translates Java Sourcecode into bytecode files that the interpreter can understand.)
java -> (Java Interpreter, which runs applet and applications by reading interpreting bytecode files.)
javadoc -> (Creates HTML-format documentation from Java source code files.)
javah -> (Produces header files for use with native methods)
javap -> (Java disassembler, which enables us to convert bytecode files into a program description.)
jdb -> (Java debugger, which helps us to find errors in our programs.)

❖ Define Term: Primitive Data Type

A data type which is directly operated on system is known as primitive data type. Such as int, float, double, long etc.

❖ Define Term: The 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). It means the output of Java compiler is not executable code but it is bytecode.

❖ Why Byte code is necessary to execute the code of Java?

1. The reason is straightforward 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 interpret the same Java bytecode.

2. The execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system.

3. The use of bytecode enables the Java run-time system to execute programs much faster than you might expect.

4. SUN supplies its **Just In Time(JIT)** compiler for bytecode, which is included in the Java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not possible to compile an entire Java program into executable code at all once, because Java performs various run-time checks that can be done only at run time. However the JIT compiles code as it is needed, during execution.

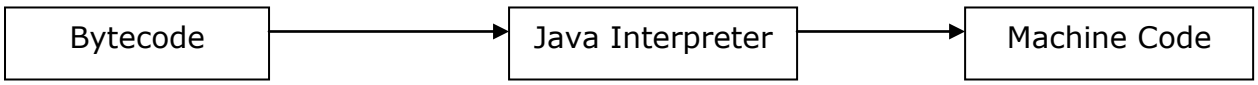
❖ What do you mean by JVM?

All language compilers translate source code into machine code for a specific computer. Java compiler also does the same thing. Java compiler done this by producing an intermediate code known as byte code for a machine that does not exist. This machine is called the **Java Virtual Machine** and it exists only inside the computer memory.

It is a virtual computer within the computer and does all major functions of a real computer. Figure shows the process of compiling a Java Program into bytecode which is also referred to as virtual machine code.

Process of Compilation

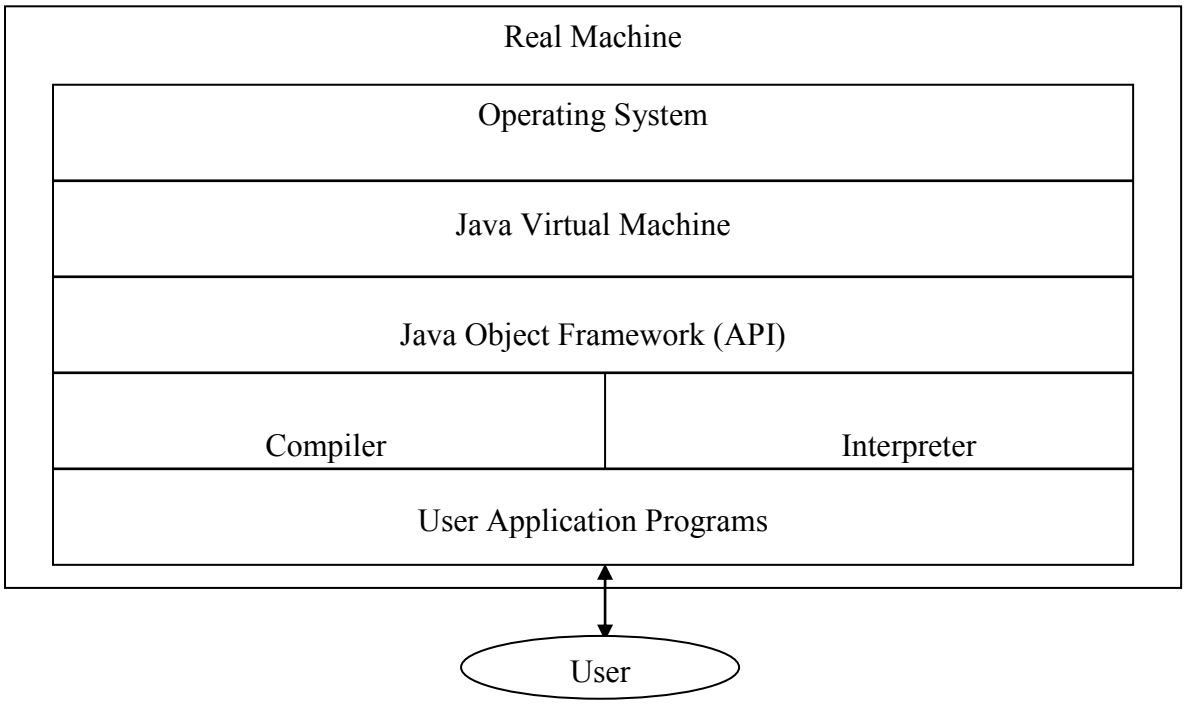
The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by the Java interpreter by acting as an intermediary between the virtual machine and the real machine as shown in the following figure. Remember that the interpreter is different for different machines.



Virtual Machine **Real Machine**

Process of converting bytecode into machine code

Figure shows how Java works on a typical computer. The Java object framework (Java API) acts as the intermediary between the user programs and the virtual machine which in turn acts as the intermediary between the operating system and the Java object framework.



Layers of interaction for Java programs

❖ Java Buzzwords or Java features

The key consideration of Java describes by SUN Microsystems are as follow:

1 Simple
2 Secure
3 Portable
4 Object-oriented
5 Robust
6 Architecture-neutral
7 Multithread
8 Interpreted
9 High performance
10 Distributed
11 Dynamic

(1) Simple

Java was designed simple way to develop program for professional programmer. Assuming that you have some programming experience, you will not find Java hard to master. If you already have knowledge of OOPS; it will be easy to learn Java. Because Java inherits the C/C++ syntax and many of the object oriented features of C++. Some of the more confusing concepts from c++ are eliminated in Java and implemented in a cleaner, more approachable manner.

(2) Secure

Security becomes an important issue for a language that is used for programming on Internet. Threat of viruses and abuse of resources are everywhere. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

(3) Portable

Java ensures portability in two ways.

1. Java compiler generates bytecode instructions that can be implemented on any machine.
2. The size of the primitive data types are machine independent.

(4) Object-Oriented

Java is a true object-oriented language. Almost everything in Java is an object. All program code and data reside within objects and classes. Java comes with an extensive set of classes, arranged in packages that we can use in our programs by inheritance. The object model in Java is simple and easy to extend, while simple types, such as integers are kept as high-performance nonobjects.

(5) Robust

Java is robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. So Java is strictly typed language.

To better understand how Java is robust, consider two of the main reasons for program failure:

1 Memory management mistakes
2 Mishandled exceptional conditions (that is, run-time errors).

1. **Memory Management:** It is very tedious task in traditional programming environments. For example, C/C++, the programmer must manually allocate and free 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 de allocation for you.

2. **Exceptional conditions:** Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found". Java helps in this area by providing object-oriented exception handling. In well-written Java program, all run-time errors can and should be managed by your program.

(6) Multithread

Multithread is a process by which more than two programs can be executed simultaneously. Java supports multithread programs. This means that we do not need to wait for the application to finish one tasks before beginning another.

For example:

(1) We can listen to an audio clip while scrolling a page and at the same time download an applet from distant computer. This feature improves the interaction performance of graphical applications.

(2) We can send tasks such as printing into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs.

(7) Architecture-Neutral

A central issue for the Java designer was that 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 fail. So, the Java designer's main goal was "**Write once; run anywhere, any time, forever.**" To great hard work this goal was achieved.

(8) Interpreted

Normally, computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two-stage system.

First Java compiler translates source code into special highly optimized instruction code known as **bytecode**. **Bytecode** are not machine instructions

Second stage Java interpreter generates machine code that can directly executed by the machine that is running on Java program. Therefore Java is both a compiled and an interpreted language.

(9) High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any machine that provide Java Virtual Machine. Java, however was designed to perform well on very lower CPUS.

It is true that Java was engineered for interpretation, 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.

(10) Distributed

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. You can also create an application on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project. Java recharged these interfaces in a package called **Remote Method Invocation (RMI)**. This feature brings an unparalleled level of abstraction to **client/server programming**.

(11) Dynamic

Java programs carry with them large amounts of run-time type information that is used to verify and determine accesses to objects at run time. This makes it possible to dynamically link code in a safe and convenient manner.

==========
==========

Q. **What is the meaning of each keyword in following statement in java?**

public static void main(String args[])

Public	The keyword public is an access specifier that declares the main() as unprotected and therefore making it accessible to all other classes. So, main() must be declared as public, because it must be called by code outside of its class when the program is started. This is similar to the C++ public modifier.
Static	static declares this method as one that belongs to the entire class and not a part of any objects of the class. The main must always be declared as static because the interpreter uses this method before any objects are created.
Void	The type modifier void states that the main method does not return any value.
String args[]	Any information that you need to pass into main() is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. String args[] declares a parameter named args, which is an array of objects of the class String. Objects of type String store character strings. In this case, args receives any command-line argument present when the program is executed.

Lexical Issues: Java programs are a collection of whitespaces, identifiers, comments, literals, operators, separators and keywords.

- **Whitespace:** Java is a free-form language. Free-form language means that you do not need to follow any special indentation rules. You can write java program in single line as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In java **whitespace is a space, tab or newline**.

For example

```
class free_form
{
      public static void main(String arg[])
      {int i,len;len=arg.length;for(i=0;i<len;i++)
{System.out.println("Argument " +(i+1) +": " +arg[i]);}}}
```

- **Identifiers:** Identifiers are used for class names, method names and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers or the underscore and dollar-sign characters. It must not begin with a number. Java is case sensitive, so **NUM** is a different than **Num**.

Some examples of valid identifiers are:

AvgTemp count a5 $test this_is_ok

Invalid variable names include:

3count high-temp Not/Ok

- **Literals:** A constant value in Java is created by using a literal representation of it. For example,

1. 100 integer literal
2. 98.78 floating literal
3. 'S' character literal
4. "This is a test" string literal

- **Comments:** There are three types of comments defined by Java.

1. Single line comment **("//")**
2. Multiline comment **("*.....*/")**
3. Documentation comment **("/**.....*/")**

Purpose of documentation comment: This type of comment is used to produce an HTML file that documents your program.

- **Separators:**

In Java there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	1. Used to contain list of parameters in method/function definition and when call it. 2. Used to defining precedence in expressions 3. Used as expression in control statements, loop statements and switch statements. 4. Used to surrounding cast types.
{ }	Braces	1. Used to initialize the value automatically in arrays.
[ ]	Brackets	1. Used to declare array types and when dereferencing array values
;	Semicolon	1. Terminates statements
,	Comma	1. Used to separates consecutive identifiers during variable declaration 2. Used to separates consecutive identifiers during variable declaration as arguments in function 3. Used to chain statements together inside a for loop.
.	Period	1. Used to separate package names from subpackagees and classes. 2. Used to separate a variable or method from reference variable.

- **The Java Class Libraries:** The Java environment relies on several built-in class libraries that contain many built in methods that provide support for such things as I/O, string handling, networking and graphics. Thus, Java as a totally is a combination of the Java language itself and its standard classes.

❖ Basic Concept of Object Oriented Programming(OOP):

It is necessary to understand some of the concepts used extensively in object oriented programming. These include:

1. Objects
2. Classes
3. Data Encapsulation and abstraction
4. Inheritance
5. Polymorphism

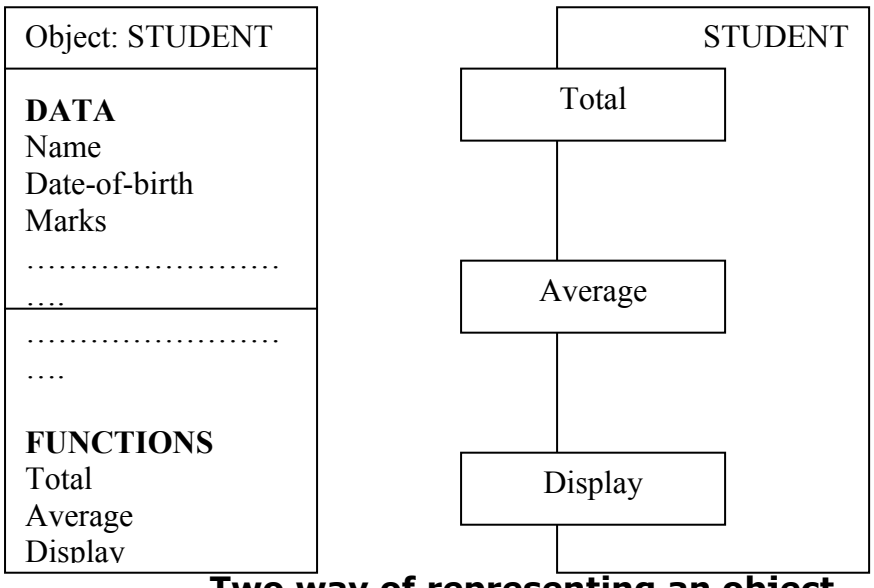
❖ Objects:

Objects are the basic run-time entities in an object-oriented system. It may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. So, class variable is known as object.

Programming problem is analyzed in terms of objects and nature of communication between them. Objects take up space in memory and have an associated address like a record in Pascal or a structure of C.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. Figure shows two notations that are popularly used in object-oriented analysis and design.



Two way of representing an object

❖ Classes:

A class is a collection of objects of similar type. For example mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer in C.

For example,

```
fruit mango;
```

will create an object mango belonging to the class fruit.

❖ Data abstraction and Encapsulation

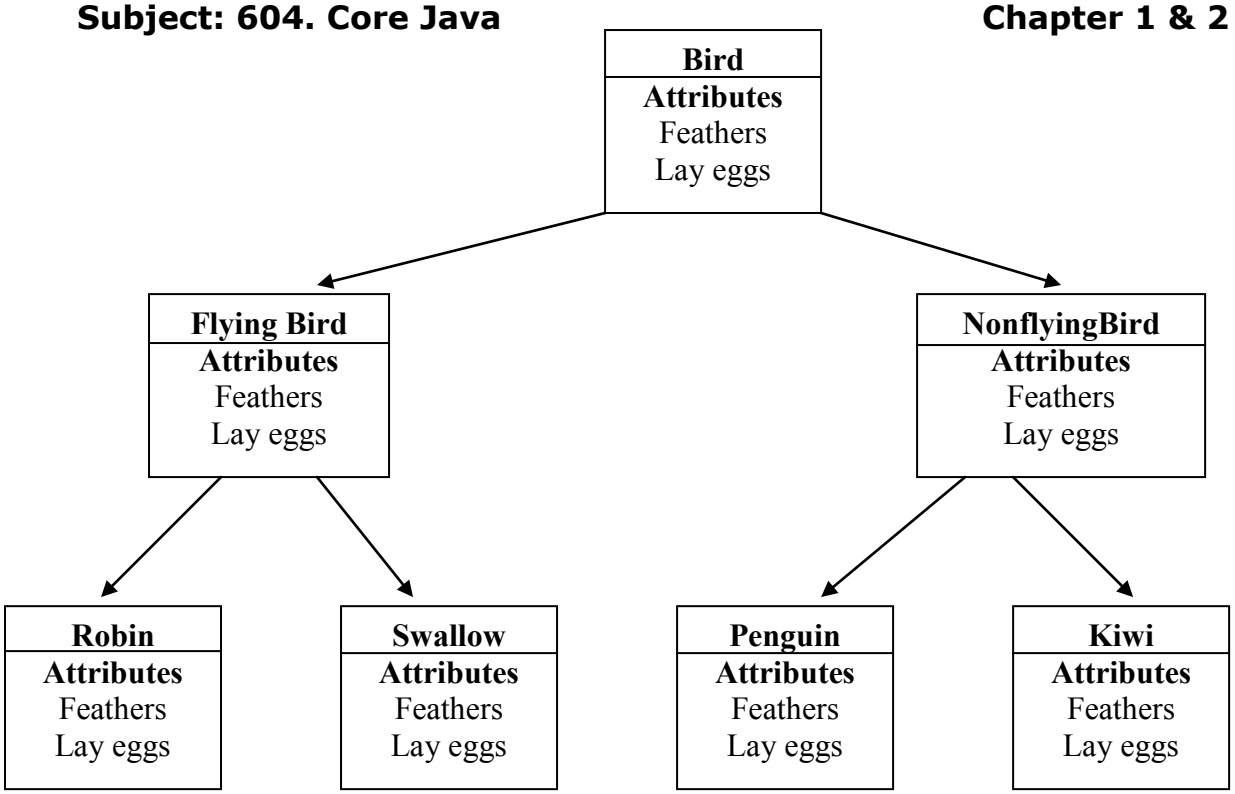
The wrapping up of data and functions into a single unit (called class) is known as **encapsulation**. Data encapsulation is the most striking features of a class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called **data hiding or information hiding**.

Abstraction refers to the act of representing essential features without including the background details or explanations. . **For example**, floating-point numbers are abstracted in programming language. You are not required to know how a floating point number is represented in binary while assigning a value to it.
Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate on these attributes.

They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called **data members** because they hold information. The functions that operate on these data are sometimes called **methods or member functions.**

❖ Inheritance:

Inheritance is the process by which objects of one class can share the properties of objects of another class. It supports the concept of hierarchical classification.



Property inheritance

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both classes.

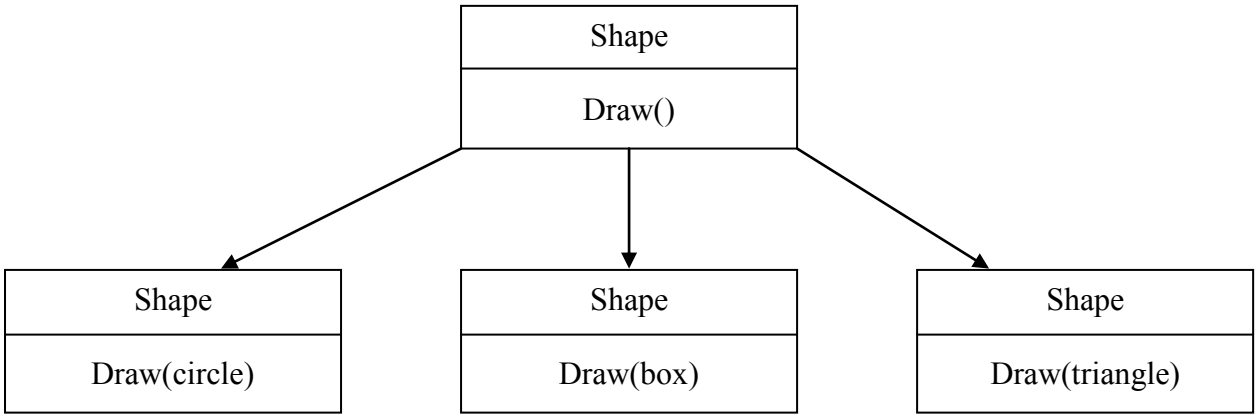
The real appeal and power of the inheritance mechanism is that it allows the programmer to reuses class that is almost but not exactly what he wants and to tailor the class in such a way that it does not introduce any undesirable size effects into the rest of class.

- **Polymorphism:**

Polymorphism means '**one name, multiple methods.**' The concept of polymorphism is implemented using the overloaded functions and operators.
Operator Overloading means an operator to exhibit different behaviors in different instances is known as operator overloading.

Function overloading means multiple functions with single name to perform different types of tasks are known as function overloading.

The following figure shows the idea of polymorphism



Polymorphism

Q. Benefits of Java.

1. Free open source and there is not any development cost.
2. Garbage collection built into the language.
3. Write once Run anywhere (Byte code+ relative JVM)
4. No pointers
5. Well implemented Data Structure APIs (Collection)
6. Well Support APIs for web (JEE)
7. Well Support APIs for mobile (JME, partially Android)
8. No need to reinvent the wheel, lots of third party APIs available for support.
9. Support, and easy integration for many functional and scripting language like Python, Scala, Grovee etc.
10. Great Document.
11. Synchronization built into the language
12. Threading built into the language
13. Late linking makes some development and deployment smoother.
14. Platform-independent binaries, and rigorous code checking, make it possible to safely deploy code within another process (e.g. user-supplied builtins, application servers)

❖ **Give reason: Java is strongly typed language.**

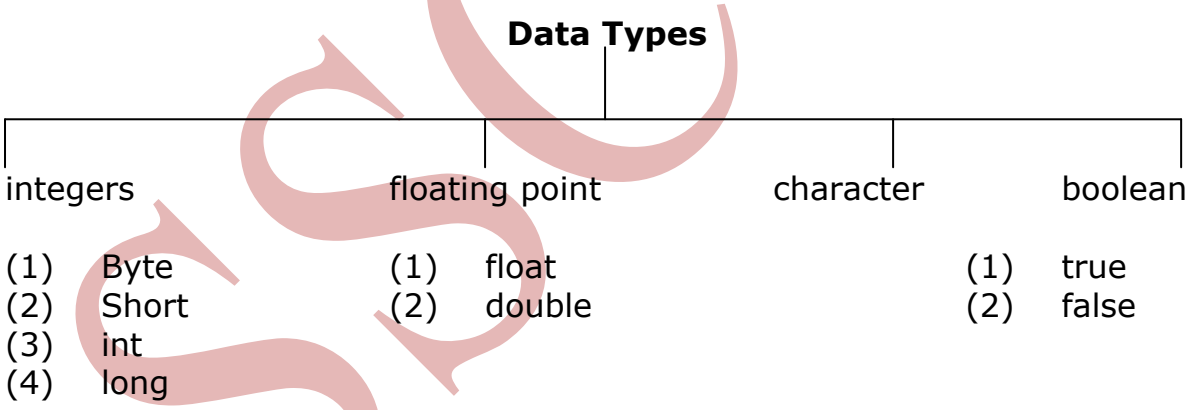
1. Every variable has a type, every expression has a type and every type is strictly defined.
2. All assignments whether explicit or via parameter passing in methods calls are checked for type compatibility.
3. There are no automatic conversions of conflicting types as in some languages.
4. The Java compiler checks all expressions and parameters to ensure that the types are compatible.
5. Any types mismatches are errors that must be corrected before the compiler will finish compiling the class

For example;

```
int x;
float y =45.67;
x=y;  //valid in C/C++ but not in Java
```

❖ **How many data types are available in Java?**

Java defines eight simple or elemental data types**. They are byte, short, int, long, char, float, double and Boolean**. All these data types can be put in four groups:



(1) Integers

Java defines four integer types: **byte, short, int, long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them.

The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
Long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Int	32	-2,147,483,648 to 2,147,483,647
Short	16	-32,768 to 32,767
Byte	08	-128 to 127

- **byte:** The smallest integer type is byte. This is a signed 8-bit types that has a range from **-128 to 127**. Byte data type is used when you are working with a stream of data from a network or file.

  The following example declares two byte variables called b & c.

  ```
  byte b,c;
  ```

- **short:** Short is a signed 16-bit type. It has range from -32,768 to 32,767. it is probably the least-used Java type, since it is defined as having its high byte first (called big-endian format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce.

  The following example declares two short variables called s & t.

  ```
  short s;
  short t;
  ```

- **int:** The most commonly used integer type is **int.** It is a signed 32-bit type that has a range from **-2,147,483,648 to 2,147,483,647**. Int type variables are commonly used to control loops and to index arrays. Any time you have an integer expression involving **bytes, shorts, ints and literal numbers,** the entire expression is promoted to **int** before the calculation is done.

  The following example declares two int variables called i & j.

  ```
  int i, j;
  ```

- **long: long** is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is very large. Long data type is used when big numbers are needed.

  The following example declares two long variables called d & s.

  ```
  long d, s;
  ```

(2) Floating-point value

Floating point numbers are also known as **real numbers**. It is useful when we want to calculate square root, sine and cosine which require a floating-point type. There are two types of floating point types float and double respectively. Their width and ranges are shown here:

Name	Width	Range
Double	64	4.93-324 to 1.8e+308
Float	32	1.4e-045 to 3.4e+038

- **float:** This type float specifies a single-precision value that uses **32 bits** of storage. Single precision is faster on some processors and takes half as much space as double precision. It will become rough when the values are either very large or very small. Variables of type float are useful when you need a fractional component but don't require a large degree of precision.

  The following example declares two float variables called f1 & f2.

  ```
  float f1,f2;
  ```

- **double:** This type double specifies a double-precision value that uses **64 bits** of storage. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All math functions, such as **sin()**, **cos()** and **sqrt()** return double values. When you need to maintain accuracy over many iterative calculations or are manipulating large value numbers **double** is the best choice.

  The following example declares two double variables called d1 & d2.

  ```
  double d1,d2;
  ```

Here is a short program that uses double variables to compute the area of a circle.

```
//compute the area of a circle.

class area
{       public static void main(String arg[])
        {       double pi,r,a;
                r=10.8;
                pi=3.14;
                a=pi*r*r;
                System.out.println("Area of circle is: " +a);
        }
}
```

(3) characters

Java used **char type** to store character value. There is difference between char data type in C/C++ and Java. **char** is an integer type and 8 bits wide in C/C++. This is not in the case of Java. Java uses Unicode to represent characters. **Unicode defines** a fully international character set that can represent all of the characters found in all human languages. It defines the set of characters of various languages such as **Latin, Greek, Arabic, Cyrillic, Hebrew, Hangul and many more**.

For the purpose of Unicode **char** data type requires 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as **ASCII** and its range from 0 to 127. The remaining characters are 8 bit wide and its range from 0 to 255.

The following example declares two char variables called c1 & c2.

```
char d1,c2;
```

Here is the program that demonstrates **char** variables:

```
class chardemo
{
	public static void main(String arg[])
	{
		char ch1,ch2;
		ch1=88;
		ch2='Y';
		System.out.println("ch1 and ch2:");
		System.out.println(ch1 +" " +ch2);
	}
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Note: You can found more information about Unicode on website:
http://www.unicode.org

(4) booleans

Java has a simple type called Boolean for logical values. It can have only one of two possible values **true or false**. All the relational operator, if statement and loop statement return logical value either true or false.

here is the program that demonstrates **boolean** variables:

```
// Demonstrate boolean values.
class booltest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

- **Literal or Constants:** Constant in Java means fixed values that do not change during the execution of program. Java support several types of constants given as bellow.

Integer Literal

An integer constant refers to a sequence of digits. There are three types of integers literal namely, decimal integer, octal integer and hexadecimal integer.

Decimal integer literals consist of a set of digits, 0 through 9, preceded by an optional minus sign. Valid examples of decimal integer constants are:

123 -321 0 456892

An octal integer literal consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 0557

Hexadecimal integer literal consist of digit from the set 0 through 15, so A through F (or a through f) are substituted for 10 through 15. You suggest a hexadecimal constant with a leading zero-x (**0X or 0x**). Some examples of hexadecimal integer are:

0X2 0X9 0X12 0X15

Integer literal create an int value, which is a 32-bit integer value. You can assign an integer literal to of Java's other integer types, such as **byte or long** without causing a type mismatch error. When a literal value is assigned to a **byte, short or long** variable, no error is generated if the literal value is within the range of the target type. However to specify a long literal, you will need to explicitly tell the compiler that the literal value is of type long. You do this by an appending an **upper or lowercase L/l** to the literal.

For example,

```
class longint
{
    public static void main(String arg[])
    {
    long l1=4578909897654;          //    error!!! you have to use
                                    //    long
      l1=4578909897654L
      l1=56489;
```

```
        int i=l;        // you can assign long value to int if it is within
                        // the range of target data type.

        System.out.println("LONG=" +l1 +"\ninteger" +i);
    }
}
```

Floating-Point Literal

Floating point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation.

Standard notation consists of a whole number component followed by a decimal point followed by a fractional component. **For example,** 2.0, 3.14159 and 0.6667 represent valid standard notation floating point numbers. Some standard notation floating variables may be distance, heights, temperatures and prices and so on.

Scientific notation uses a standard-notation, floating point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. **The general form is**

mantissa e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer with an optional plus or minus sign. The letter e separating the mantissa and the exponent can be written in either uppercase or lowercase.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5e9 or 75e8. Similarly -0.000000368 is equivalent to -3.68e-7.

For example, 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by $\mathbf{10^2}$.

Floating point literals in Java default to double precision. To specify a float literal, you must append an **F or f** to the constant. You can also explicitly specify a double literal by appending a **D or d**.

For example,

```
class floatval
{
    public static void main(String arg[])
    {
        float sum=0.0f;
        float num1=10.50F,num2=10.50F;
        sum=sum+num1+num2;
        System.out.println("sum=" +sum);
    }
}
```

Boolean literal

Boolean literals are simple. There are only two logical values that a Boolean value can have, **true and false**. The values of true or false do not convert into any numerical representation. The true literal in Java does not equal **1**, nor does the false literal equal **0**.

Character literal

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as addition and subtraction operators.

A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes such as 'a', 'z' and '@'.

For characters that are impossible to enter directly, there are several **escape sequences**, which allow you to enter the character you need, such as '\" for the single-quote character itself, and '\n' for the newline character.

There is also a mechanism for directly entering the value of a character in **octal or hexadecimal**. For octal notation use the backslash followed by the three-digit number.

For example, **'\141'** is the letter 'a'

For hexadecimal notation, you enter a **backslash-u (\u)**, then exactly four hexadecimal digits. For example, **'\u0061'** is the ISO-Latin-1 'a' because the top byte is zero.

Following table shows the character escape sequences.

Escape Sequence	Description
\ddd	Octal Character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\’	Single quote
\””	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\t	Tab
\b	Backspace
\f	Form Feed

Character Escape Sequences

String Literals

String literals in Java are specified like same as specified in other languages. Strings are actually object types. Strings are not implemented as arrays of characters like in other languages.

One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there in other languages. **Examples** of string literals are as bellow

“Hello World” “two\n lines” “\ “This is in quotes\””

“WEL COME” “I.N.D.I.A” “3+2” “S”

Consider the following program

```
class strdemo
{
	public static void main(String arg[])
	{
		String s1=”Radhe Krishna”;
		System.out.println(“S1=” +s1);
		int len=s1.length(),i=0;

		while(i<len)
		{
		System.out.println(“Character ” +(i+1) +”: “ +s1.charAt(i));
			i++;
		}
```

```
            String s2="Radhe";
            String s3="Krishna";
            String s4=s2 +" " + s3;
            System.out.println("String s4=" +s4);
      }
}
```

❖ **How to declare and dynamically initialized variables?**

Variable: The variable is the basic unit of storage in Java program. A variable is defined by the combination of an identifier, a type and an optional initialize. All variables have a scope, which defines their visibility and a lifetime.

The syntax to declare variable:

type identifier [=value][,identifier [=value]…];

type = any valid Java data type or name of class or interface.
identifier = name of variable
=value : you can initialize the variable by specifying an equal sign and a value.
Here are several examples of variable declaration of various types.

```
int a,b,c;                  //declares three int a,b and c.
int d=3,e,f=5;              //declares three more ints initializing d and f.
byte z=22;                  // initialized z by 22
double pi=3.14159;          //declares an approximation of pi.
char x='x';                 // the variable x has the value 'x'.
```

Dynamic initialization: Java allows variables to be initialized dynamically, using any expression valid at the time of variable declaration.

The key point here is that the initialization expression may use any element valid at the time of initialization including calls to methods, other variables or literals.

For example,

//demonstrate dynamic initiazliztion.

```
class dyndemo
{
      public static void main(String arg[])
      {
            double a=3.0,b=4.0;
```

```
                    //c is dynamically initialized
                double c=Math.sqrt(a*a + b* b);
                System.out.println("c=" +c);
}}
```

- **What is Scope and lifetime of variables?**

Java variables are actually classified into three kinds:

1. instance variable (class object)
2. class variables (class data member)
3. local variables

Instance and class variables are declared inside a class. Instance variables are created when the objects are created and therefore they are associated with the objects. They take different values for each object.

Class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable.

Variables declared and used inside methods are called **local variables**. They are called local variables because they are not available for use outside the method definition.

Local variables can also declared inside program blocks that are defined between an opening brace { and a closing brace }. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will be destroyed.

Each block can contain its own set of local variable declarations. We cannot declare variable to have the same name as one in an outer block. It means any block can not have the same variable name.

Scope: The area of the program where the variable is accessible (i.e usable) is called its scope.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope.

To understand the effect of nested scopes, consider the following program.

// Demonstrate block scope.

```
class scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("X and Y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("X is " + x);
  }
}
```

Output:

```
X and y: 10 20
X is 40
```

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. **For example**, consider the following program.

```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y iz: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

Output:

```
y is: -1
y is now: 100
y is: -1
```

```
y is now: 100
y is: -1
y is now: 100
```

❖ **Type Conversion and Casting:**

There are two types of conversion in Java.

(1) Automatic Type conversion
(2) Casting Incompatible Types.

➢ **Automatic Type Conversion:**

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

1. The two types are compatible
2. The destination type is larger than the source type.

When these two conditions are met, a **widening (promotion) conversion** takes place. For example the **int** type is always large enough to hold all valid short & byte values. So explicit cast statement is not required.

For example,

```
short s1=5432;
float f1;
double d1;
int i;
i=s1;
f1=i1;
d1=f1;
```

The following table list which are guaranteed to result in no loss of information.

From (Source Type)	To (Destination Type)
Byte	short, int, long, float, double, char
Short	int, long, float, double
Int	long, float, double
Long	float, double
Float	Double
Char	int, long, float, double

- **Casting Incompatible Types:**

When you want to assign larger data type value into smaller data type automatic conversion will not be possible. So, you have to explicitly convert source data type into destination data type. This kind of conversion is Sometime known as narrowing conversion.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has the following general form.

(target_type) value

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

- **Automatic Type Promotion In Expressions:**

In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, consider the following expression:

```
byte a=40,b=50,c=100;
int d=a*b/c;
```

The result of intermediate term **a*b** easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte or short operand to **int** when evaluating an expression.

This means that the sub expression **a*b** is performed using integers-not bytes. Thus, 2000 the result of the intermediate expression, 50*40 is legal even though **a and b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, the following code generates a problem:

```
byte b=50;
b=b*2;     //error! Can't assign an int to a byte!
                    Or
byte b=50,b1=2;
b=b*b1;    //error! Can't assign an int to a byte!
```

The code generates compile-time error, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of cast.

In this case you should use an explicit cast, such as

```
byte b=50;                    or        byte b=50,b1=2;
b=(byte) (b*2);                         b=(byte)(b*b1);
```

- **The type promotion rules:**

Java defines several type promotion rules that apply to expressions. They are as follows.

1. All byte and short values are promoted to **int**. If one operand is a long, the whole expression is promoted to **long**.
2. If one operand is a **float**, the entire expression is promoted to **float**.
3. If one operand is **double**, the result is **double**.

The following program demonstrates some type conversions in expression.

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
```

```
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

❖ Introduction to arrays in Java

Array: An array is 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.

1-D Array: Like any other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps:

1. Declaring the array
2. Creating memory locations
3. Initialization of arrays

- **Declaration of arrays:**
  Arrays in Java may be declared in two forms:

  Form 1: type arrayname[];
  Form 2: type [] arrayname;

For example;

```
int number[];
int month_days[];
        or
int [] counter;
float [] average;
```

The value of above all variables is set to **null** which represents an array with no value.

- **Creation of arrays:** After declaring an array, we need to create in the memory. Java allows us to create arrays using new operator only, as shown below:

```
arrayname=new type[size];
```

For example;

```
month_days=new int[12];
average=new float[10];
```

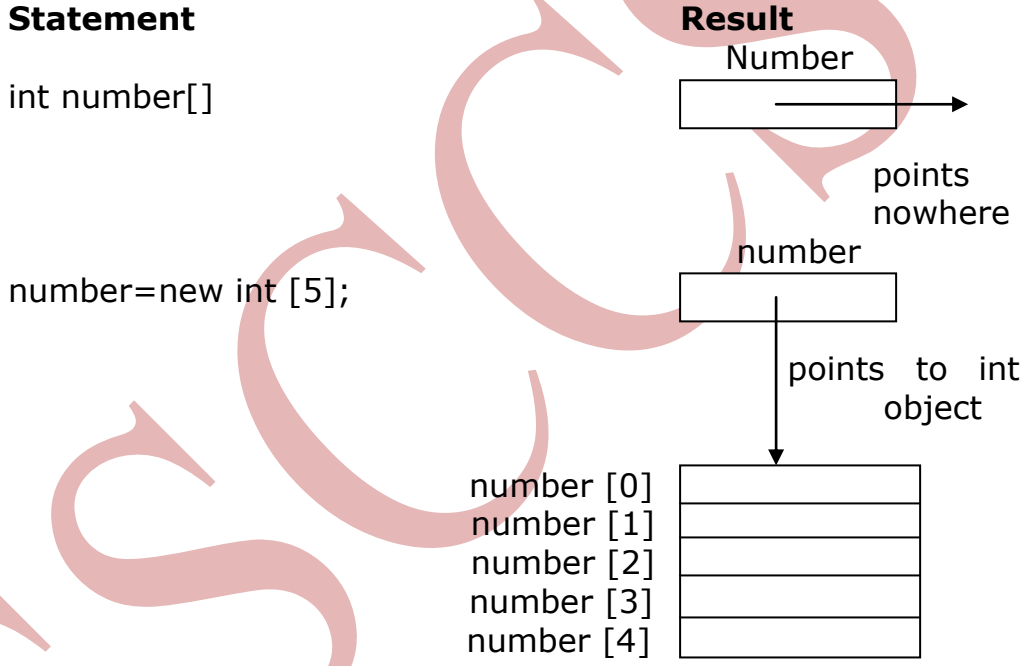
These lines create necessary memory locations for the arrays number and average and designate them as **int** and **float** respectively. Now, the

variable **month_days** refers to an array of 12 integers and average refers to an array of 10 floating point values.

It also possible to combine the two steps-declaration and creation-into single statement as shown bellow:

```
int number[]=new int[5];
float average[]=new float[3];
float  marks[]=new float[3];
char c1[]=new char[3];
String s1[]=new String[3];
```

Figure illustrates creation of an array in memory.



- **Initialization of Arrays:**

The final step is to put values into the array created. This process is known as initialization. This is done using the array subscripts as shown below:

```
arrayname[subscript]=value;
```

Example:

```
number[0]=10;
number[1]=20;
number[2]=30;
……..
……..
Number[N]=50;
```

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

type arrayname[]={list of values};

For example;

```
int number[]={10,20,30,40,50};
String name[]={"MUMBAI","CHENNAI","DELHI","CALCUTTA"};
```

It is possible to assign an array object to another array object. For example

```
int a[]={1,2,3};
int b[];
b=a;
```

are valid in Java. Both the array will have the same values.

➢ Array Length:

In Java, all arrays store the allocated size in a variable named length. We can obtain the length of the array **a** using **a.length**. For example,

```
int a[]={10,20,30};
int len;
len=a.length;
```

This information will be useful in the manipulation of arrays when their sizes are not known:

➢ Remember Points about Arrays:

1. Java creates arrays starting with the subscript 0 and ends with a value one less than the size specified.
2. Unlike C, Java protects arrays from overruns and under runs. Trying to access an array bound its boundaries will generate an error message.

❖ **2-D Array:** A collection of rows and columns is known as 2-D Array.

Declaration:

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

or

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

This allocates a 2 by 2 array and assigns it to twod. Internally this matrix is implemented as an array of arrays of int. **Conceptually**, this array will look like the one shown in the following figure.

	Col 0	Col 1	Col 2
Row->0	[0] [0]	[0] [1]	[0] [2]
Row->1	[1] [0]	[1] [1]	[1] [2]
Row->2	[2] [0]	[2] [1]	[2] [2]

Conceptual view of a 3 by 3 matix, two dimensional array.

- **Multidimensional Array or Irregular Array:** In 2-D array you need only specify the memory for the **first (leftmost) dimension**. You can allocate the remaining dimension separately.

For example, this following code allocates memory for the first dimension of twod when it is declared.

```
int twod[][]=new int[4][];

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

The following program creates a two dimensional array in which the sizes of the second dimension are an unequal.

// Manually allocate differing size second dimensions.

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

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix.

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {{ 1, 2, 3 },{ 4,5,6 },{ 7,8,9}};
    int i, j;
    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

❖ **3-D Array:** A collection of tables is known as **3-D Array**.

The following program creates a 3by 4 by 5, three dimensional array. It then loads each element with the product of its indexes.

```
// Demonstrate a three-dimensional array.
class threeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
        threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

❖ What is the meaning of String in Java?

String is neither simple type nor it is an array of characters (as are strings in C/C++). String defines an object and full description of it requires an understanding of several object-oriented features. **String** objects have many special features and attributes that make them quite powerful and easy to use.

❖ Why Java does not support Pointer?

Java does not support or allow pointers because it stores the address of another variable in memory. So it would allow Java applets to break the firewall between the Java execution environment and the host computer. Since C/C++ makes extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the boundaries of the execution environment, you will never need to use pointer or would there be any benefit in using one.

Java provides a rich operator environment. Most of its operators can be dividend into the following four categories.

1. Arithmetic Operator
2. Bitwise Operator
3. Relational Operator
4. Logical Operator

❖ Arithmetic Operators

Arithmetic Operators are used in mathematical expressions in the same way that they are used in algebra. The following tables lists the arithmetic operators.

Operator	Result
+	Addition
-	Substraction (unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition assignment
-=	Substraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

The operands of the arithmetic operators must be of a numeric type. You cannot use them on Boolean types.

➢ The Modulus Operator:

The modulus operator, % returns the reminder of a division operation. It can be applied to floating-point types as well as integer types. (This differs from C/C++, in which the % can only be applied to integer types). The following program demonstrates the %.

```
// Demonstrate the % operator.
class Modulus {
  public static void main(String args[]) {
    int x = 42;
    double y = 42.25;
    System.out.println("x mod 10 = " + x % 10);
    System.out.println("y mod 10 = " + y % 10);
  }}
```

Output:

```
x mod 10=2
y mod 10=2.25
```

- **Arithmetic Assignment Operators:**

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming.

```
a=a+3;
```

In Java, you can rewrite this statement as shown here.

```
a+=3;
```

Here is another example,

```
a=a%2;
```

Which can be expressed as

```
a%=2;
```

There are assignment operators for all of the arithmetic, binary operators. Thus any statement of the form

```
var=var op expression;
```

can be rewritten as

```
var op=expression;
```

- **Benefit of assignment operators:**

1. They save you a bit of typing, because they are 'shorthand" for their equivalent long forms.
2. They are implemented more efficiently by the Java run time system than are their equivalent long forms.

- **Increment and Decrement Operators**

The ++ and – are Java's increment and decrement operators. They have some special properties that make them quite interesting. The increment operator increases its operand by one. The decrement operator decreases its
its operand by one.

For example,

```
x=x+1;
y=y-1
```

can be rewritten like this by use of the increment operator:

```
x++; y--;
```

In this operator there are two differences between prefix form and postfix form. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
In the postfix form, the previous value is obtained for use in the expression and then the operand is modified.

```
x=42;
y=++x;
```

In this case, y is set to 43 as you would expect, because the increment occurs before x is assigned to y. thus, the line y=++x; is the equivalent of these two statements.

```
x=x+1;
y=x;
```

However, when written like this

```
x=42;
y=x++;
```

The value of x is obtained before the increment operator is executed, so the value of y is 42. Of course, in both cases x is set to 43. Here, the line y=x++; is the equivalent of these two statements.

```
y=x;
x=x+1;
```

The following program demonstrates the increment operator.

```
// Demonstrate ++ and --.
class incdec {
    public static void main(String args[]) {
      int a = 1;
      int b = 2;
      int c;
      int d;

      c = ++b;
      d = a++;
      c++;
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      System.out.println("c = " + c);
      System.out.println("d = " + d);
  }
}
```

Bitwise Operators

- Java support special operators known bitwise operators for manipulation of data at bit level either 0 or 1.
- Bitwise operator can be applied to the integer types, long, int, short, char and byte. These operators can't be applied to float or double data type.
- Bitwise operators are use for testing bits, shifting bits to left, shifting bits to right & shifting bits to right zero fill. They are summarized in the following tables.

Operator	Result
~	Bitwise Unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
>>	Bitwise Shift Right
<<	Bitwise Shift Left
>>>	Bitwise Right Zero Fill
&=	Bitwise AND Assignment
|=	Bitwise OR Assignment
^=	Bitwise Exclusive OR Assignment
>>=	Bitwise Shift Right Assignment
<<=	Bitwise Shift Left Assignment
>>>=	Bitwise Right Zero Fill Assignment

The Bitwise Logical Operators:

- The Bitwise Logical Operators are &,|, ^ and -. The following table shows the outcome of each operation. It is remembered that the bitwise operator are applied to each individual bit within each operand.

A	B	A|B	A & B	A^B
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	0

The Bitwise & (AND)

Bitwise AND operator represented by & or AND. The result of AND operation is 1 if both bits have values **1**(One) otherwise **0 (Zero).**

For Ex. If the value of x is 17 and value of y is 28

X=17	-→	0001	0001	**8 bit**
Y=28	-→	0001	1100	**8 bit**
X & Y	-→	0001	0000	**8 bit**

Assignment: Write a program to find out even and odd number by using Bitwise AND Operator.

The Bitwise | Operator (OR)

The Bitwise OR Operator is represented by |. The result of OR operation is **1 (One)** at least one bit has a value **1 (One)** otherwise **0 (Zero)**.

For Ex. If the value of x is 17 and value of y is 28

X=17	-→	0001	0001	**8 bit**
Y=28	-→	0001	1100	**8 bit**
X | Y	-→	0001	1101	**8 bit**

Assignment: Write a program to which use Bitwise OR Operator.

The Bitwise Exclusive ^ Operator (Exclusive OR)

This operator is represented by ^. The result of exclusive or is 1 (One) if only one of the bits is **1 (One)** otherwise **0 (Zero).**

For Ex. If the value of x is 17 and value of y is 28.

X=17	-→	0001	0001	**8 bit**
Y=28	-→	0001	1100	**8 bit**
X ^ Y	-→	0000	1101	**8 bit**

Assignment: Write a program to which use Bitwise OR Operator.

The Bitwise ~ Operator (NOT)

This operator is represented by ~. This operator is also known as bitwise complement or unary NOT operator. The result of NOT operator is **1 (one)** if bits is **0 (Zero)** Otherwise **0 (Zero)** if bits is **1 (One)**.

For Ex. If the value of x is 17.

X=17	-→	0001	0001	**8 bit**
~X	-→	1110	1110	**8 bit**

Assignment: Write a program to which use Bitwise OR Operator.

Using the Bitwise Logical Operators:

```
// Demonstrate the bitwise logical operators.
class BitLogic {
  public static void main(String args[]) {
    String binary[] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };

    int a = 3; // 0 + 2 + 1 or 0011 in binary
    int b = 6; // 4 + 2 + 0 or 0110 in binary
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b) | (a & ~b);
    int g = ~a & 0x0f;

    System.out.println("        a = " + binary[a]);
    System.out.println("        b = " + binary[b]);
    System.out.println("      a|b = " + binary[c]);
    System.out.println("      a&b = " + binary[d]);
    System.out.println("      a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println("       ~a = " + binary[g]);
  }
}
```

❖ The Bitwise Left Shift Operator (<<)

This operator is represented by <<. The left shift operator <<, shifts all of the bits in a value to the left a specified number times.

Each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2. But you need to watch out. If you shift a 1 bit into high-order position (bit 31 or 63), the value will become negative.

It has the following general form:

Syntax: Value<<num

Where **Value=** It specifies the value that we want to shifts as bits to the left direction.
Num= How many bits shifted in the left direction.

For Ex. We assume 8 bit value in this example. The value of x is 17 and shifts that value by 3 bit at left direction.

X=17	-→	0001	0001	**8 bit**
X<<3	**-→**	***		

X=X<<3	-→	1000	1000	**8 bit**

Note: (* means 3 bits shifts from the left. So these 3 bits will be lost at left direction and the next all bits will be replaced with that lost bits and remaining bits will be filled with 0.)

Assignment: Write a program to which use Bitwise Left Shift Operator.

❖ The Bitwise Right Shift Operator (>>)

This operator is represented by >>. The right shift operator >>, shifts all of the bits in a value to the right a specified number times.

Each right shift has the effect to divide that value by two and discards any remainder. You can take an advantage of this for high-performance integer division by 2. Of course, you must be sure that you are not shifting any bits off the right end.

It has the following general form:

Syntax: Value>>num

Where **Value=** It specifies the value that we want to shifts as bits to the right direction.
Num= How many bits shifted in the right direction.

For Ex. Here we assume **8 bit** in this example. The value of x is 35 and shifts that value by 2 bit at right direction.

X=35	-→	0010	0011	**8 bit**
X>>2	**-→**		**	
X=X>>2	-→	0000	1000	**8 bit**

Note: (** means 2 bits shifts from the right. So these bits will be lost at right direction and the next all bits will be replaced with that lost bits and remaining bits will be filled with 0.)

Assignment: Write a program to which use Bitwise right Shift perator.

- **How java represents negative numbers?**

All of the integer types are represented by binary numbers of varying bit widths. For example, the byte value for 42 in binary is **00101010,** where each position represents a power of two, starting with 2^0 at the rightmost bit. The next bit position to the left would be 2^1 or 2, continuing toward the left with 2^2 or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3 and 5 (counting from 0 at the right); thus 42 is the sum of $2^1 + 2^3 + 2^5$, which is 2+8+32.

42 (Binary) = 00101010
42 (Decimal) =
76543210
$2^5\ 2^3\ 2^1 = 42$

All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as two's complement.

2's complement means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example , -42 is represented by inverting all of the bits in 42, or 00101010, which holds 11010101, then adding 1, which results in 11010110 or -42. To decode a negative number, first invert all of the bits, and then add 1. -42 or 11010110 inverted holds 00101001 or 41, so when you add 1 you get 42.

```
Binary of 42    =    00101010
Binary of -42   =    11010101 (By inverting all of the bits in a value)
                          +1 (add 1)
                     -----------
                     11010110
```

Sign Extension: When you are shifting right, the top (leftmost) bits exposed by the right shifts are filled in with the previous contents of the top bit. This is called sign extension and serves to preserve the sign of negative numbers when you shift them to right. For example, -42>>3 is -6

For Ex. Here we assume **8 bit** in this example. The value of x is -42 and shifts that value by 3 bit at right direction.

```
X=-42    -→          1101      0110      8 bit
X>>3     -→                     ***
-------------------------------------------------------
X=X>>3   -→          1111      1010      8 bit
```

Note: (* means 3 bits shifts from the right. So these bits will be lost at right direction and the next all bits will be replaced with that lost bits and remaining bits will be filled with 1 to preserve the negative sign.)

Assignment: Write a program to which use Bitwise right Shift Operator for negative value.

The Unsigned Right Shift Operator (>>>)
Or
Shifts Right Zero Fill Assignment Operator (>>>)

This operator is represented by >>>. The unsigned right shift operator >>>, shifts all of the bits in a value with 0 (zero) to the right a specified number times.

As you have just seen, the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However sometimes this is undesirable.

For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an **unsigned shift.**

To accomplish this, you will use java's unsigned, shift-right operator, **>>>** which always shifts zeros into the high-order bit.

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

```
a=-1          11111111  11111111  11111111  11111111  =-1
a=a>>>24      00000000  00000000  00000000  11111111  =255
```

The **>>>** operator is often not as useful as you might like, since it is only meaningful for 32 or 64-bit values. Remember, smaller values are automatically promoted to **int** in expressions. This means that sign-extension occurs and that the shift will take place on a 32-bit rather than on an 8 or 16 bit value.

Assignment: Write a program to which use Bitwise unsigned right Shift Operator for negative value of integer data type and shift the bit from 20 to 24 bit and 25 to 32 bits.

Q 1: What do you mean by unsigned shift.
Q 2: What is the actual purpose of unsigned right shift operator?

❖ Bitwise Operator Assignments:

All of the binary bitwise operators have a shorthand form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.

For example the following two statements, which shifts the value in a right by four bits are equivalent.

```
a=a>>4;
a>>=4;
```

Likewise, the following two statements, which result in **a** being assigned the bitwise expression **a** OR **b**, are equivalent:

```
a = a | b;
a |= b;
```

The following program creates a few integer variables and then uses the shorthand form of bitwise operator assignments to manipulate the variables:

```
class opbitEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a |= 4;          //a=a|4 = 1 | 4 = 0001 | 0100 = 0101 = 5
    b >>= 1;         // 1
    c <<= 1;         // 6
    a ^= c;          // a ^ c = 5 ^ 6 = 0101 ^ 0110= 0011= 3
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

//0011= $2^0 + 2^1 = 1 + 2 = 3$

❖ Relational Operators:

The relational operators determine the relationship that one operand has to other. They determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expression that control the **if** statement and the various **loop statements**.
As stated, the result produced by a relational operator is a **boolean** value.

For example, the following code fragment is perfectly valid.

```
int a=4;
int b=1;
boolean c=a<b;
```

In this case the result of **a<b** is **false** stored in **c**.

If you are coming from a C/C++ background, please not the following.

```
int done;
//……
if(!done) …..// valid in C/C++
if(done)  …..// but not in Java
```

In Java, these statements must be written like this:

```
if(done==0) …    // This is Java-style.
if(done!=0) …    //
```

The reason is that Java does not define **true and false** in the same way as C/C++. In C/C++, **true** is any **nonzero value** and **false is zero**. **In Java**, **true and false** are nonnumeric values which do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

For example,

```
boolean b;
b=true;

if(b){          //valid in Java, condition will become true
…..
…..
}
```

❖ Boolean Logical Operators

The Boolean logical operators shown here operate only on Boolean operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR(exclusive OR)
||	Short-Circuit OR
&&	Short-Circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The following table shows the effect of each logical operation:

A	B	A|B	A&B	A^B	!A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

The following program demonstrates the Boolean logical operators.

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;

    System.out.println("       a = " + a);
    System.out.println("       b = " + b);
    System.out.println("     a|b = " + c);
    System.out.println("     a&b = " + d);
    System.out.println("     a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("      !a = " + g);
  }
}
```

Output:

```
              a= true
              b= false
            a|b= true
            a&b= false
            a^b= true
      a&b|a&!b= true
              !a=false
```

❖ Short-Circuit Logical Operators:

Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the **Boolean AND and OR operators**, and are known as short-circuit operators.

You know very well that the **OR operator** results in **true** when **A is true**, no matter what **B is**. Similarly, the **AND operator** results in **false** when **A is false**, no matter **what B** is.

If you use the **|| and &&** forms, rather than the **| and &** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the **left operand** alone.

This is very useful when the **right-hand operand** depends on the left one being true or false in order to function properly.

For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
int denom=0,num=10
If(denom!=0 && num/denom=10)
```

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

For example,

```
int c=1,e=99;
if (c==2 || e++ <100)
d=100;
```

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

❖ The Assignment Operator:

The Assignment Operator is the single equal sign=. The assignment operator works in Java much as it does in any other computer language. It has this general form:

Var=expression;

Here, the type of **var** must be compatible with the type of expression.
The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x,y,z;
x=y=z=100;       //set x, y, and z to 100
```

This fragment sets the variables **x, y and z** to 100 using a single statement.
This works because the = is an operator that holds the value of the right-hand expression.

Therefore the value of z=100 is 100, which is then assigned to y, which in turn is assigned to x. using a "chain of assignment" is an easy way to set a group of variables to a common value.

❖ The ? Operator:

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operators is the **?** and it works in Java much like it does in **C, C++ and C#**. The **?** has this general form:

expression1 ? expression2:expression3

Here, expression1 can be any expression that evaluates to a **Boolean value**. If expression1 is true, then expression2 is evaluated;otherwise, expression3 is evaluated.

The result of the **?** Operation is that of the expression evaluated. Both **expression2 and expression3** are required to return the same type, which can't be **void.**

For example,

```
int  i, j,ans;
i=10;
j=20;
ans=(i>j) ? i : j;
```

When Java evaluates this assignment expression, it first looks at the expression to the **left** of the question mark. If i is greater than j then the expression between **question mark** and **colon** is evaluated and used as the value of the entire **?** expression. If **I** is not greater than **j** then the expression after the **colon** is evaluated and used for the value of the

entire **? expression.** The result produced by the **?** operator is then assigned to **ans.**

Here , is a program that demonstrate the ? operator. It uses it to obtain the largest value between two variables.

For example,

```
// Demonstrate ?.
class largest {
  public static void main(String args[]) {
    int i, j,k;

    i = 10,j=20;
    k = i > j ? i : j;        // get largest value
   System.out.println(i + " is " + k);
  }}
```

❖ Using Parentheses:

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

```
a>>b+3;
```

This expression first adds 3 to b and then shifts a right by that result. That is this expression can be rewritten using redundant parentheses like this:

```
a>> (b+3)
```

However, if you want to first shift **a right by b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

```
(a>>b) +3
```

Java's program control statements can be put into the following categories:

1. Selection
2. Iteration
3. Jump

- **Selection statement** allows your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- **Iteration statements** enable program execution to repeat one or more statements. (Iteration statements mean all loops).
- **Jump statements** allow your program to execute in a nonlinear fashion.

(1) Java's selection statements:

Java supports two selection statements: **if** and **switch. These** statements allow you to control the flow of your program's execution based upon conditions known only during at run time.

❖ If statement:

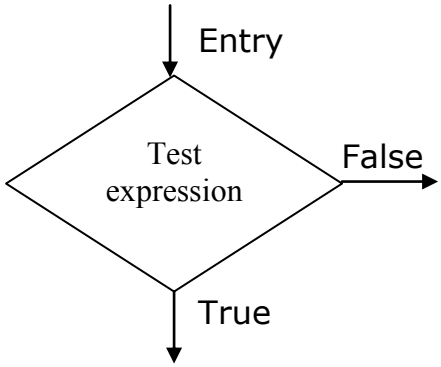
If statement is Java's conditional branch statement. It is powerful decision making statement and is used to control the flow of execution of statements. It is basically two-way statement and is used in conjunction with an expression.

Here the general form of the **if** statement:

```
If(condition)
        statement1;
else
        statement2;
```

Here each statement may be a single statement or a compound statement enclosed in curly braces (that is a block). The **condition** is any expression that returns **boolean** value. The **else** clause is optional.

If works like this: If the condition is **true**, then **statement1** is executed. Otherwise **statement2** (if it exists) is executed. This point of program has two paths to follow, one for the **true** condition and the other for the **false** condition as shown in **following figure**.



Two-Way Branching

Consider the **following program:**

```
int a,b;
// …
if(a<b)
        a=0;
else
        b=0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Most often, the expression used to control the **if** will involve the relational operators. However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable, as shown in this code fragment.

```
boolean dataavailable;
// …
        if (dataavailable)
                processdata();
        else
                waitformoredata();
```

Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you will need to create a block, as in this fragment.

```
int bytesavailable;
        // …
        if (byteavailable>0)
        {
                processdata();
                bytesavailable-=n;
        }
        else
                waitformoredata();
```

Here, both statements within the if block will execute if **bytesavailable** is greater than zero.

- **Nested ifs:** A nested **if** is an **if statement** that is the target of another **if or else**. **Nested ifs** are very common in programming. When you nest ifs, the main thing to remember is that an **else statement** always refers to the nearest **if statement** that is within the same block as the else and that is not already associated with an else. Here is an example.

```
        If(i==10)
        {
              if(j<20)
                    a=b;
              if(k>100) c=d;
              else
                    a=c;
        }
else    a=d;
```

As the comments indicate, the final **else** is not associated with **if(j<20)**, because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)**, because it is the closest **if** within the same block.

- **The if-else-if ladder:**

A common programming construct that is based upon a sequence of nested **ifs** is the **if-else-if** ladder. It looks like this:

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

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true,** the statement associated with that **if** is executed, and the rest of the ladder is bypassed.

If none of the conditions is **true**, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.

If there is no final **else** and all other conditions are **false**, then no action will take place.

Consider the following program that uses an **if-else-if** ladder to determine which season a particular month is in.

```
// Demonstrate if-else-if statements.
class Ifelse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

(2) Switch Statement:

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.

It often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement.

```
switch(expression)
{
        case value 1:
                //statement sequence
                break;
        case value 2:
                //statement sequence
                break;
        .
        .
        .
        case valueN:
                //statement sequence
                break;

        default:
                //default statement sequence
}
```

The expression must be of type **byte, short, int or char**; each of the values specified in the **case** statements must be of a type compatible with the expression.

Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The switch statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed.

If none of the constants matches the value of the expression, then the default statement is executed. However, the default statement is an optional. If no case matches and no default is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire switch statement.

Here is a simple example that uses a switch statement.

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

Output:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

- The **break** statement is an optional. If you omit the break, execution will continue on into the next case. It is sometimes desirable to have multiple **cases** without break statements between them. For example, consider the following program.

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more.");
      }
  }
}
```

- The previous example is, of course unnatural for the sake of illustration, omitting the **break** statement has many practical applications in real programs. Consider the following rewrite of the season example shown earlier.

// An improved version of the season program.

```
class Switch {
   public static void main(String args[]) {
      int month = 4;
      String season;
      switch (month) {
        case 12:
        case 1:
        case 2:
          season = "Winter";
          break;
        case 3:
        case 4:
        case 5:
          season = "Spring";
          break;
```

```
        case 6:
        case 7:
        case 8:
          season = "Summer";
          break;
        case 9:
        case 10:
        case 11:
          season = "Autumn";
          break;
        default:
          season = "Bogus Month";
      }
      System.out.println("April is in the " + season + ".");
   }
}
```

❖ **Nested Switch Statements:**

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a nested **switch**. For example consider the following fragment is perfectly valid:

```
switch(count)
{
      case 1:
            switch(target)
            {
                  case 0:
                        System.out.println("target is zero");
                        break;
                  case 1:
                        System.out.println("target is one");
                        Break;
            }
      break;
      case 2:     //      …..
}
```

Here, the **case 1:** statement in the **inner switch** does not conflict with the **case 1:** statement in the outer switch. The **count** variable is only compared with the list of cases at the outer level. If **count is 1**, then target is compared with the inner list cases.

There are three important features of the **switch** statement to note:

1. The **switch** differs from the **if** in that **switch** can only test equality, whereas **if** can evaluate any type of **Boolean expression**. That is, the

switch looks only for a match between the value of the expression and one of its **case** constants.

2. No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement enclosed by an outer **switch** can have **case** constants in common.

3. A **switch** statement is usually more efficient than a set of nested **ifs**.

❖ **How Java compiler compiles switch statement?**

Or

❖ **When switch statement is better than if-else ?**

When Java compiler compiles a **switch** statement, the Java compiler will inspect each of the **case constants** and create a **"jump table"** that it will use for selecting the path of execution depending on the value of expression.

Therefore, if you need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of **if-elses**. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

(3) Iteration Statements:

Java's iteration statements are **for**, **while** and **do-while.** These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

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

```
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.

Here is a while loop that counts down from 10, printing exactly ten lines of value.

```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

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.

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. For example, consider the following program:

```
class nobody
{
        public static void main(String arg[])
        {
                int i,j;
                i=100;
                j=200;

                //find midpoint between i and j
                while(++I < --j) ;        //no body in this loop

                System.out.println("Midpoint is" +i);
        }
}
```

Output: Midpoint is 150

❖ do-while loop:

We know that, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while 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. Fortunately, Java supplies a loop that does just that:

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 the 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.

Here is the reworked version of the previous program that demonstrates the **do-while** loop.

```
class dowhile
{
        public static void main(String arg[])
        {
                int n=10;
                        do
                        {
                                System.out.println("n=" +n);
                                n--;
                        }while(n>0);
        }
}
```

The above loop can be written more efficiently as follows:

```
do
{
        System.out.println("n=" +n);
}while(--n>0);
```

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

Consider the following program which implements a very simple help system for Java's selection and iteration statements:

// Using a do-while to process a menu selection -- a simple help // system.

```
class Menu {
  public static void main(String args[])
    throws java.io.IOException {
    char choice;

    do {
      System.out.println("Help on:");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. while");
      System.out.println("  4. do-while");
      System.out.println("  5. for\n");
      System.out.println("Choose one:");
      choice = (char) System.in.read();
    } while( choice < '1' || choice > '5');

    System.out.println("\n");
     switch(choice) {
      case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
      case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println("  case constant:");
        System.out.println("    statement sequence");
        System.out.println("  break;");
        System.out.println("  // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
      case '4':
```

```
        System.out.println("The do-while:\n");
        System.out.println("do {");
        System.out.println("  statement;");
        System.out.println("} while (condition);");
        break;
      case '5':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
    }
  }
}
```

Notice that characters are read from the keyboard by calling **System.in.read().** This is one of Java's console input functions. **System.in.read()** is used here to obtain the user's choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**).

By default, standard input is line buffered, so you must press **ENTER** before any characters that you type will be sent your program.

Because **System.in.read()** is being used, the program must specify the throws java.io.IOException clause. This line is necessary to handle input errors. It is a part of Java's exception handling features.

❖ **For loop:**

The general form of the **for** statement:

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

If only one statement is being repeated, there is no need for the curly braces.

The for loop operates as follows:

1. 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. So, initialization expression is only executed once.

2. 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.

3. 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);
        }
}
```

When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. **For example**, here is a simple program that tests for prime numbers.

// Test for primes.

```
class findprime {
  public static void main(String args[]) {
    int num;
    boolean isPrime = true;

    num = 14;
    for(int i=2; i <= num/2; i++) {
      if((num % i) == 0) {
        isPrime = false;
        break;
      }
    }
    if(isPrime) System.out.println("Prime");
    else System.out.println("Not Prime");
  }
}
```

❖ Using the Comma:

There will be times when you will want to include more than one statement in the initialization and iteration portion of the for loop. Java allow two or more variables to control a **for** loop, Java permits you to include multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by the comma.

// Using the comma.

```
class Comma {
  public static void main(String args[]) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

❖ Some for loop variations:

The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts, the initialization, the conditional test, and the iteration, do not need to be used for only those purposes.

One of the most common variations involves the conditional expression. Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any valid **Boolean** expression.

For example,

```
boolean done=false;

        for(int i=1;!done;i++)
        {
                //…..
                if(interrupted())
                done=true;
        }
```

Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent as in the next program.

```
// Parts of the for loop can be empty.
class ForVar {
  public static void main(String args[]) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i is " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty. For example,

```
for(;;)
{
        //…..
}
```

This loop will run forever, because there is no condition under which it will terminate. Although there are some programs, such as operating system command processor that require a **nonfinite loop**, most "infinite loops" are really just loops with special termination requirements.

❖ **Nested loops:**

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

(4) Jump statements:

Java supports three **jump** statements: **break, continue, and return**. These statements transfer control to another part of your program. Each is examined here.

❖ Using break:

In Java, the **break** statement has three uses.

1. It terminates a statement sequence in a **switch** statement.
2. It can be used to **exit loop**.
3. It can be used as a "civilized" form of **goto**.

❖ Using break to Exit a Loop:

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
// Using break to exit a loop.
class breakloop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

The **break statement** can be used with any of Java's loops, including intentionally infinite loops. **For example**, here is the preceding program coded by use of a **while** loop.

```
//  Using break to exit a while loop.

class breakloop2 {
  public static void main(String args[]) {
    int i = 0;

    while(i < 100) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
      i++;
    }
```

```
    System.out.println("Loop complete.");
  }
}
```

When used inside a set of nested loops, the break statement will only break out of the inner

```
// Using break with nested loops.
class breakloop3 {
  public static void main(String args[]) {
    for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  } }
```

❖ Using break as a Form of Goto:

The **break statement** can also be used by itself to provide a "elegant" form of the **goto** statement.

Java does not have a **goto** statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations.

However sometimes **goto** can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the break statement. By using this form of **break,** you can break out of one or more block of code. These blocks need not be part of a loop or a **switch.** They can be any block.

The general form of the **labeled break** statement is shown here.

```
break label;
```

Here **label=**it is the name of label that identifies block of code.

When this form of break executes, control is transferred out of the named block of code. The labeled block of code must enclose the break statement, but it does not need to be the immediately enclosing block.

This means that you can use a labeled **break** statement to exit from set of nested blocks. But you cannot use break to transfer control to a block of code that does not enclose **break** statement.

For example, the following program shows three nested blocks, each with its own label. The break statement causes execution to jump forward, past the end of the block labeled secod, skipping the two println() statements.

// Using break as a civilized form of goto.

```
class breakblock {
  public static void main(String args[]) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Before the break.");
          if(t) break second; // break out of second block
          System.out.println("This won't execute");
        }
        System.out.println("This won't execute");
      }
      System.out.println("This is after second block.");
    }
  }
}
```

One of the most common uses for a **labeled break** statement is to exit from nested loops. For example, consider the following program, the outer loop executes only once.

// Using break to exit from nested loops

```
class breakloop4 {
  public static void main(String args[]) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

keep in mind that you can't break to any label which is not defined for an enclosing block. **For example** consider the following program that will not compile.

```
// This program contains an error.

class breakerr {
  public static void main(String args[]) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one; // WRONG
      System.out.print(j + " ");
    }}}
```

❖ **Using continue statement as jump:**

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end.
Here is an example program that uses **continue** to reason two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;System.out.println("");}}}
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue.

Here is an **example program** that uses continue to print a triangular multiplication table for 0 through 9.

```
// Using continue with a label.

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

```
          continue outer;
        }
        System.out.print(" " + (i * j));
      }}
    System.out.println();
}}
```

❖ **Return statement:**

The last control statement is **return.** The return statement is used to explicitly return from a method. That is it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

Here **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main()**

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
    boolean t = true;

    System.out.println("Before the return.");
    if(t) return; // return to caller
    System.out.println("This won't execute.");
  }}
```

In the **preceding program**, the if(t) statement is necessary. Without it the Java compiler would flag an "unreachable code" error, because the compiler would know that the last **println()** statement would never be executed.

- **Class fundamentals:** The class is at the core of Java. it is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java.

- **Define term class:** Class is a user defined data type.

- **The general form of a class:**

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or data, most real-world classes contain both.

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. The 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** or **data members.**

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

All methods have the same general form as **main(),** which we have been using this far. However, most methods will not be specified as **static** or **public.**

Notice that the general form of class does not specify a **main()** method. Java classes do not need to have a **main().** You only specify one if that class is the starting point for your program. Further, the applets don't require a **main()** method at all.

❖ **Simple class:**

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width, height and depth.**

For example,

```
class box
{
        double width;
        double height;
        double depth;
}
```

here class defines a new type of data is called **box.** You will use this name to declare objects of type **box.** It is important that a **class** declaration only creates a template; it does not create an actual object.

To actually create a **box** object, you will use a statement like the following:

```
box mybox=new box();        //create a box object called mybox
```

After this statement executes, **mybox** will be an **object/instance** of box.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Therefore, every **box** object will contain its own copies of the instance variables **width, height and depth**.

To access these variables, you will use the **dot(.)** operator. The dot operator links the name of the object with the name of an instance variable.

For example mybox.width=100;

```
/* A program that uses the Box class.

Call this file BoxDemo.java
*/
class box {
  double width;
  double height;
  double depth;
}
// This class declares an object of type Box.
class boxdemo {
  public static void main(String args[]) {
    box mybox = new box();
    double vol;

    // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

    System.out.println("Volume is " + vol);
  }}
```

It is important to understand that changes to the **instance variables** of one object have no effect on the instance variables of another.
For example the following program declares two **box** objects:

```
// This program declares two Box objects.

class box {
  double width;
  double height;
  double depth;
}

class boxdemo2 {
  public static void main(String args[]) {
    box mybox1 = new box();
    box mybox2 = new box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;
    /* assign different values to mybox2's
```

```
    instance variables */
  mybox2.width = 3;
  mybox2.height = 6;
  mybox2.depth = 9;

  // compute volume of first box
  vol = mybox1.width * mybox1.height * mybox1.depth;
  System.out.println("Volume is " + vol);

  // compute volume of second box
  vol = mybox2.width * mybox2.height * mybox2.depth;
  System.out.println("Volume is " + vol);
 }}
```

- **Declaring objects:**

When you create a class, you are creating a new data type. You can use this type to declare objects of that type. There are **two** step processes to obtain objects of a class.

1. You must declare a variable of the class type. This variable does not define an object. It is simply a variable that can refer to an object.

2. 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 memory for an object and returns a reference to it. This reference is more or less, the address in memory of the object by allocated by **new**.

This reference is then stored in the variable. Thus, in all class objects must be dynamically allocated.

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

```
box mybox=new box();
```

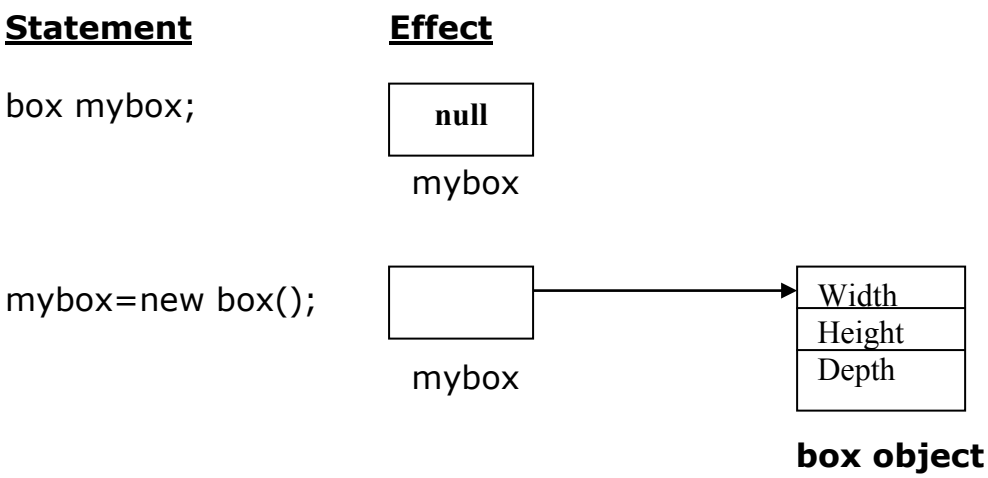
This statement can be rewritten like this to show each step more clearly;

```
box mybox;
mybox=new box();
```

- The first line declares mybox as reference to an object of type box. After this line executes, mybox contains the value null, which

indicates that it does not still point to 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 in the following figure.



- **A closer Look at new:**

Or

- **Write a short note about new**

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

Syntax:

class-var=new classname();

Where
class-var = it is a variable of class type or an object of class type.
classname = it is the name of the class under which the object of that class
is being created.

The class name followed by parentheses specifies the default constructor of
the class.

For example

box mybox=new box(); [box() specifies the default constructor].

New allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However computer has fixed memory, so **new** may not be allocate enough memory to your program. If this happens, a run-time exception will occur. You can handle this situation by creating exception handling mechanism in your program.

- **Why do not need to use new for simple data types (int, char etc)?**

Java's simple types are not implemented as objects. Simple types are implemented as "normal" variables. This is required for the efficiency of variables. Because objects have many features and attribute that require java to take care of them differently than it treats the simple data types.

- **What is the difference between class and object?**

Class	Object
(1) Class is a logical framework	(1) object is a physical framework
(2) Class has not space of memory	(2) object has space of memory

- **How to assign object as reference variable?**

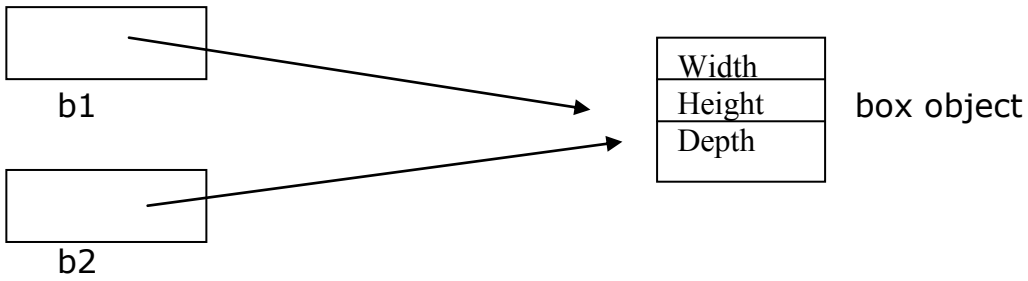
You can assign an object as reference variable to same class variable.

For example

```
box b1=new box();
box b2=b1;
```

In above statement both b1 and b2 will refer same object after execution. The assignment of b1 and b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Therefore any changes occur by object b2 will affect the object b1, because they are the same object.

This situation is depicted here:



b1 and b2 both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2.

For example,

```
box b1=new box();
box b2=b1;
//…..
B1=null;
```

❖ Introduction of methods or declaration of methods.

Classes usually consist of two things: Instance variables and methods.
The objects created by such a class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class but immediately after the declaration of data member or instance variable.

This is the general form of a method:

```
type methodname(parameter list)
{
```

```
        //body of method
}
```

Method declarations have four basic parts:

1. The name of the method (methodname)
2. The type of the value the method returns (type)
3. A list of parameters (parameter-list)
4. The body of the method

Here, type specifies the type of value the returned by method. This can be any valid simple data type such as int, float as well as any class type. If the method does not return a value, its return type must be void.
The method name is a valid identifier. The parameter-list is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input. The variables in the list are separated by commas. In the case where no input data are required, the declaration must retain the empty parentheses.

For example,

```
        Setval(int x, float y, double z)      //three parameters
        getdata()                             //empty list
```

let us consider the box class again and add a method setdim() to it.

```
class Box {
  double width;
  double height;
  double depth;

  // sets dimensions of box
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}
```

- **How to add method to the box class?**

Most of the case you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes.

The following program shows you how to add method to the box class.

```
// This program includes a method inside the box class.
```

```
class Box {
  double width;
  double height;
  double depth;

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
  }
}

class BoxDemo3 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // display volume of first box
    mybox1.volume();

    // display volume of second box
    mybox2.volume();
  }
}
```

This program generates the following output, which is the same as the previous version.

Volume is 3000.0
Volume is 162.0

❖ How to return value by method to the box class?

The following example is an improved version of the preceding program.
// Now, volume() returns the volume of a box.

```
class Box {
  double width;
  double height;
```

```
  double depth;

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

class BoxDemo4 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

There are two important things to understand about returning values.

The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **Boolean**, you could not return an integer.

The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

Note: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume()** could have been used in the **println()** statement directly, as shown here:

system.out.println("volume is " +mybox1.volume());

- **How to add method that takes parameters?**

A parameterized method can operate on a variety of data and be used in a number of slightly different situations. Here is a method that returns the square of the number 10:

```
int square()
{
      return 10*10;
}
```

Here , use of this method is very limited. However, you can modify the method so that it takes a parameter as shown next, then you can make **square()** much more useful.

```
int square(int i)
{
      return  i*i;
}
```

square() is now a general-purpose method that can compute the square of any integer value rather than just 10.

Here is an example:

```
int  x, y;    x=square(5);x=square(9);y=2;x=square(y);
```

A **parameter** is a variable defined by a method that receives a value when the method is called. For example, in **square()**, i is a parameter. An **argument** is a value that is passed to a method when it is called.

For example **square(100)** passes 100 as an argument. Inside **square()**, the parameter i receives that value.

//This program uses a parameterized method.

```
class Box {
  double width;
  double height;
  double depth;

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

```
  // sets dimensions of box
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}

class BoxDemo5 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // initialize each box
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

❖ CONSTRUCTORS:

A **constructor** is a special method which has same name as class name that is used to initialize the instance variable of class when we create an object of class.

Constructor is automatically called when we create an object of class. Constructor looks like a strange because they have no return type, not even **void**.

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/

class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
```

```
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

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

class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

❖ Parameterized constructors:

While the **box()** constructor in the preceding example does initialize a **Box** object, it is not very useful because of all boxes have the same dimensions.

However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are known as **parameterized constructors**.

For example, the following version of **Box** defines a parameterized constructor which sets the dimensions of a box as specified by those parameters.

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
```

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

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

class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
```

❖ This keyword:

Sometimes a method will need to refer to the object that called it. To allow this, java defines the **this** keyword. **this** can be used inside any method to refer to the current object. It means **this** is always a reference to the object on which the method was called. You can use **this** anywhere a reference to an object of the current class type is permitted.

Consider the following example to understand the following version of Box().

```
//a redundant use of this
box(double w,double h,double d)
{
```

```
        this.width=w;
        this.height=h;
        this.depth=d;
}
```

❖ **Instance variable hiding:**

Sometimes the name of instance variable (class data member) and the variable name of parameterized method may be same. When a local variable of parameterized method has the same name as an instance variable, the local variable hides the **instance variable.**

We can use **this** keyword to resolve the problem between instance variable and local variable by using **this** keyword. Because **this** directly refer the current object of class.

For example, here is another version of box() which uses width, height and depth for parameter names and then uses this to access the instance variables by the same name:

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
  }
  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
  class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);
    double vol;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
```

❖ Garbage collection:

Objects are dynamically allocated by using the **new** operator and destroyed the objects to release the memory for later reallocation in C++ language.
In C++ dynamically allocated objects must be manually released by use of **delete** operator.
Java takes a different approach it handles deallocation for you automatically. The technique that accomplishes this is called **garbage collection**.

It works like this: When no references to an object exist, that object is assumed to be no longer needed and the memory occupied by the object can be reclaimed. There is no forcefully need to destroy objects as in C++.

When one or more objects exist and it is not used for longer time, garbage collection occurs infrequently during the execution of your program.

❖ The finalize() method:

Sometimes an object will need to perform some action when it is destroyed. **For example**, if an object is type of some non-Java resource such as close connection object of database file, record set file, file handle (open, read and write) or window character font, then you need to free these resources before an object is destroyed. To handle such situations, Java provides a mechanism called finalization.

Finalization method will be called when object need to be destroyed by the garbage collection.

To add a finalize method to a class, you simply define the **finalize()** method. 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.

The **finalize()** method has this general form:

```
protected void finalize()
{
      //finalization code here
}
```

// This Java program will fail after 5 calls to f().

```
class X {

    static int count = 0;

  // constructor
  X() {
   if(count<MAX) {
     count++;
    }
    else  {
      System.out.println("Error -- can't construct");
      System.exit(1);
    }
  }

  // finalization
  protected void finalize() {
    count--;
  }

  static void fun1()
  {
    X ob = new X(); // allocate an object
    // destruct on way out
  }

  public static void main(String args[]) {
    int i;

    for(i=0; i < 5; i++) {
      fun1();
      System.out.println("Current count is: " + count);
    }
  }
}
```

- **Method Overloading:** When two or more methods have same name within the same class but the parameter declarations are different is known as **method overloading**. Method overloading is one of the ways that java implements polymorphism.

- Java uses the type and number of arguments to determine which version of the overloaded method will be called when an overloaded method is called. Therefore, overloaded methods must differ in the type and 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 call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is simple example that shows the 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);
  }}
```

- When an overloaded method is called, java looks for a match between the arguments used to call the method and method's parameters. However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution. For example, consider the following program.

/ Automatic type conversions apply to overloading.

```
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // overload test for a double parameter and return type
  void test(double a) {
    System.out.println("Inside test(double) a: " + a);
  }
}

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

    ob.test();
    ob.test(10, 20);

    ob.test(i);              // this will invoke test(double)
    ob.test(123.2);          // this will invoke test(double)
  }
}
```

- **How methods overloading support polymorphism in java?**

Polymorphism means "one interface with multiple methods". The language that does not support method overloading must be given a unique name. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of t his function each with a slightly different name.

For example in C, the function **abs()** returns the absolute value of an integer.

labs() returns absolute value of a long integer and
fabs() returns the absolute value of floating point value.

C language does not support overloading, so each function has its own name even though all three functions do essentially the same thing.

This situation does not possible in Java, because each absolute value method can use the same name. Java's standard class library includes an absolute value method called **abs()**. This method is overloaded by Java's **Math** class to handle all numeric types. Java determines which version of **abs()** to call based upon the type of argument.

❖ Overloading Constructors:

As like to overload methods we can also overload constructor. To understand how it is done consider the following example.

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.

*/
class Box {
  double width;
  double height;
  double depth;

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
class OverloadCons {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    double vol;
```

```
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
  }
}
```

❖ **Using Objects as Parameters:**

We can also pass object as parameters in methods similar to pass simple type as parameters.

```
// Objects may be passed to methods.
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }
  // return true if o is equal to the invoking object

  boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}
class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equals(ob2));

    System.out.println("ob1 == ob3: " + ob1.equals(ob3));
  }
}
```

❖ Using Objects as Parameters in constructors

One of the most common uses of object parameters involves constructors. You can construct new object by passing argument as object in the constructor of existing object.

// Here, Box allows one object to initialize another.

```
class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

 // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
 // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }
 // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }
 // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
class OverloadCons2 {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    Box myclone = new Box(mybox1);
```

```
    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

    // get volume of clone
    vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
  }
}
```

❖ **What do you mean by call by value & call by reference?**

- **Call by value:** When simple type is passed to a method, this method is done by use of call by value. This method copies the value of an argument into the formal parameter of the method. Therefore, changes made to the parameter of the method have no effect on the argument.

- **Call by reference:** When objects are passed to methods, this method is done by use of call by reference. In this method, a reference to an argument is passed to the parameter.

Inside the method, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the method.

Consider the following program,

```
// Simple Types are passed by value.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();
    int a = 15, b = 20;

    System.out.println("a and b before call: " +
                       a + " " + b);
```

```
    ob.meth(a, b);

    System.out.println("a and b after call: " +
                       a + " " + b);
  }
}
```

The output from this program is shown here:

```
a and b before call:    15      20
a and b after call:     15      20
```

When you pass an object to a method, the situation is totally changed because objects are passed by reference. This effectively means that objects are passed to methods by use of **call-by-reference**.

For example consider the following program.

// Objects are passed by reference.

```
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // pass an object
  void meth(Test o) {
    o.a *=  2;
    o.b /= 2;
  }
}
class CallByRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);

    System.out.println("ob.a and ob.b before call: " +
                       ob.a + " " + ob.b);

    ob.meth(ob);

    System.out.println("ob.a and ob.b after call: " +
                       ob.a + " " + ob.b);
  }
}
```

The output from this program is shown here:

```
Ob.a  and ob.b before call:    15      20
Ob.a  and ob.b after   call:   30      10
```

❖ **Returning Objects:**

A method can return any type of data including class types that you create.

For example consider the following program.

```
// Returning an object.
class Test {
  int a;

  Test(int i) {
    a = i;
  }

  Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}

class RetOb {
  public static void main(String args[]) {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: "
                       + ob2.a);
  }
```

❖ **Recursion:** Recursion means a function which call itself. The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1*2*3 or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
  // this is a recusive function
  int fact(int n) {
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}
```

```
class Recursion {
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factorial of 3 is " + f.fact(3));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

❖ **How factorial function does works?**

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

New local variables and parameters of a method are storage on the stack and the method code is executed with the new variables from the beginning. A recursive call does not make a new copy of the method. Only the arguments are new.

It means as each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the call inside the method.

❖ **What are the main advantages of recursive method?**

1. It can be used to create clearer and simpler versions of several algorithms than to use iterative relatives. For example, quicksort algorithm is quite difficult to implement in an iterative way.
2. Some problems especially related to AI-related ones can be solve themselves to recursive problem.

❖ **What are the main advantages of recursive method?**

1. Recursive versions of many methods may execute a bit more slowly than the iterative statement because of the added overhead of the additional function calls.

2. Many recursive calls to a method could reason a stack overrun. Because storage for arguments and local variables is on the stack and each new call creates a new copy of these variables, therefore it is possible that the stack could be tired. If this occurs, the java run-time system will generate an exception.

```
// Another example that uses recursion.
        class RecTest {
          int values[];

          RecTest(int i) {
            values = new int[i];
          }

          // display arrary -- recursively
          void printArray(int i) {
            if(i==0) return;
            else printArray(i-1);
            System.out.println("[" + (i-1) + "] " + values[i-1]);
          }
        }

        class Recursion2 {
          public static void main(String args[]) {
            RecTest ob = new RecTest(10);
            int i;

            for(i=0; i<10; i++) ob.values[i] = i;

            ob.printArray(10);
          }
}
```

❖ **Introducing Access Control:**

Encapsulation links data with the code that manipulates it. Access control provides way to make control over the accessibility of a class data member. So you can prevent misuse of your data member.

For example consider the stack class in which the method **push()** and **pop()** do provide restricted boundary to the stack, this interface is not compulsory.

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

Let's begin by defining **public and private**.

- **Public:** When a member of a class is modified by the public specifier, then that member can only be accessed by any other code.
- **Private:** When member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- **Default access:** when no access specifier is used then by default the member of a class is public within its own package, but can't be accessed outside its package.

Now you can understand why **main()** has always been preceded by the **public** specifier. It is called by code that is outside the program – that is by the java run-time system.

```
/* This program demonstrates the difference between
   public and private.
*/

class Test {
  int a;                    // default access
  public int b;             // public access
  private int c;            // private access

  // methods to access c
  void setc(int i) {        // set c's value
    c = i;
  }
  int getc() {              // get c's value
    return c;
  }
}

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

// These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

// This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK

    System.out.println("a, b, and c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

To see how access control can be applied to more practical example, consider the following improved version of the **stack** class

```
// This class defines an integer stack that can hold 10 values.
class Stack {

  /* Now, both stck and tos are private.  This means
     that they cannot be accidentally or maliciously
     altered in a way that would be harmful to the stack.
  */
  private int stck[] = new int[10];
  private int tos;
```

```
// Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
class TestStack {
public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

// push some numbers onto the stack

    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

// pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
      System.out.println(mystack2.pop());

// these statements are not legal
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
  }
}
```

❖ **What do you mean by static in java?**

- Sometimes there will be situation generates when we want to access class data member without using the reference of class object. It is possible to create a class member that can be used by itself without reference of class object. To create such a member, precede its declaration with the keyword **static.**

- When a member is declared as **static**, it can be accessed before any objects of its class are created and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is **main()** method. **main()** is declared as **static** because it must be called before any object exists.

- Instance variables declared as **static** are, essentially global variables. When objects of its class are declared, no copy of a **static** variable is made. All instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions.

1. They can only call other **static** methods.
2. They must only access **static** data.
3. They cannot refer to **this** or **super** in any way.

If you want to initialize your **static** variable, you have to declare a static block and initialize static variable inside that block.

The following example shows a class that has a **static** method, some **static** variables and **static** initialization block.

// Demonstrate static variables, methods, and blocks.

```
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(48);
  }
}
```

Here is the output of the program:

```
Static block initialized
x=42
a=3
b=12
```

You can access the **static variables** and **methods** from the outside class by specifying the name of their class followed by the dot operator. Consider the following example.

Static method: **classname.method()**
Static variable: **classname.static_variable.**

Here is an example. Inside **main()** the static method **callme()** and the **static** variable **b** are accessed outside of their class.

```
class StaticDemo {
  static int a = 48;                          //static variable a
  static int b = 99;                          //static variable b
  static void callme() {                      //static method
    System.out.println("a = " + a);
  }
}
class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();                              //static method
    System.out.println("b = " + StaticDemo.b);        //static variable
  }
}
```

❖ **Final keyword for variable:**

A variable can be declared as **final**. When you declare variable as final it means you cannot change contents of variable so you must initialize a final variable when it is declared. (final is similar to cost in c/c++/c#).

Consider the following example,

```
final int a=107;
final int b=201;
final float f1=3.5f;
```

❖ **New version of Array:**

We can get the length of an array by using the length instance variable. All arrays have this variable and it will always hold the size of the array. Here is a program that demonstrates this property:

// This program demonstrates the length array member.

```
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }}
```

You can put the **length** member to good use in many situations.

❖ **Introducing Nested and Inner classes:**

Nested class: To define a class within another class; such classes are known as nested classes.

The scope of a nested class is limited upto the scope of its enclosing class. Therefore if class **B** is defined within class **A**, then **B** is known to **A** but not outside of **A**.

It means class **B** can access all the members including private members of class in which it is nested. (Here for class **A**). However, the enclosing class (**class A**) does not have access to the members of the nested class (**class B**).

There are two types of nested classes: **static and non-static.**

Static nested class is one which has the **static** modifier applied. Because it is **static** it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction **static nested classes** are seldom used.

Non-static or Inner Class: An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

// Demonstrate an inner class.

```
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }
```

```
 // this is an innner class
 class Inner {
   void display() {
     System.out.println("display: outer_x = " + outer_x);
   }
 }
}
class InnerClassDemo {
 public static void main(String args[]) {
   Outer outer = new Outer();
   outer.test();
 }
}
```

Output from this application shown here:

```
        Display:        outer_x=100
```

In the above program, an inner class named Inner is defined within the scope of class **outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**.

display() method is defined inside the **Inner** class. This method displays value of **outer_x** on the screen. The main() method of InnerClassDemo create an object of class **outer** and call its **test()** method. This method creates an object of class **Inner** and **display()** method is called.

Inner class is known only within the scope of class **outer**. The java compiler generates an error message if any code outside of **class outer** tries to create an object of class Inner. It means inner class is only known within its enclosing scope.

For example

```
// This program will not compile.
class Outer
{
 int outer_x = 100;

        void test()
        {
                Inner inner = new Inner();
                inner.display();
        }
        // this is an innner class
        class Inner
        {
                int y = 10;                             // y is local to Inner
                void display()
                 {
                        System.out.println("display: outer_x = " + outer_x);
                 }
        }
```

```
        void showy()
        {
                System.out.println(y);          // error, y not known here!
        }
}
class InnerClassDemo
{
 public static void main(String args[])
 {
   Outer outer = new Outer();
   outer.test();
 }
}
```

❖ **Define Inner Class within a for loop.**

You can define a nested class within the block defined by a method or even within body of **a for loop**, as this next program shows.

// Define an inner class within a for loop.

```
class Outer
{
        int outer_x = 100;

        void test()
        {
                for(int i=0; i<10; i++)
                {
                        class Inner
                        {
                                void display()
                                {
                                System.out.println("display: outer_x = " + outer_x);
                                }
                        }
                                Inner inner = new Inner();
                                inner.display();
                }
        }
}
class InnerClassDemo
{
        public static void main(String args[])
        {
                Outer outer = new Outer();
                outer.test();
        }
}
```

❖ **String class:**

String is probably the most commonly used class in Java's class library.

There are two important thing about string in java.

1. **String** you create is an **object** of type **String**. Even string constants are actually **String** objects. For example, in the following statement

   System.out.println("This is a string, too");

the string **"This is a String, too"** is a **string constant**. Fortunately, java handles String constants in the same way that other computer languages handle "normal" strings, so you don't have to worry about this.

2. The objects of type **String** is created; its contents cannot be changed. There are two reasons behind this to make restrictions.

   - If you need to change a string, you can always create a new one that contains the modifications.
   - Java defines a class of string called **StringBuffer** which allows strings to be changed.

Strings can be constructed a variety of ways. The easiest is to use a statement like this:

String mystring = "this is a test";

You can use it anywhere to display value of **mystring** variable on the screen.

System.out.println(mystring);

Java defines one operator for **String** objects: **+.** It is used to concatenate two strings. For example, this statement

String mystring = "I" + " like " +"Java." ;

results in **mystring** containing "I like Java."

The given program demonstrates the **String** concept.

// Demonstrating Strings.

```
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
```

```
    System.out.println(strOb3);
  }
}
```

The **String** class contains several methods that you can use. Here are a few. The general syntax of these three methods are shown here:

```
boolean equals(String object)
int length()
char charAt(int index)
```

// Demonstrating some String methods.

```
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

This program generates the following output:

```
Length of strob1:12
Char at index 3 in strob1: s

strob1 != strob2
strob1==strob3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example;

// Demonstrate String arrays.

```
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

❖ **How to use command line arguments?**

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution. This is achieved in java programs by using what are known as **command line arguments**.

Command line arguments are parameters that are supplied to the application program at the time of calling it for execution.

We can write Java programs that can receive and use the arguments provided in the command line argument. Use the signature of the **main()** method used in our java programs.

Public static void main(String args[])

As pointed out earlier, **args** is declared as an array of strings **(known as string objects)**. Any arguments provided in the command line (at the time of execution) are passed to the array **args** as its elements.

We can simply access the array elements and use them in the program as we wish.

For example consider the following command line argument

java test BASIC FORTRAN C++ JAVA

This command line contains four arguments. These are assigned to the array **args** as follows:

BASIC -→ args[0]
FORTRAN -→ args[1]
C++ -→ args[2]
JAVA -→ args[3]

```
/*
        *       This program uses command line
        *       Arguments as input
/*
class comlinetest
{
        public static void main(String args[])
        {
                int count, i=0;
                String string;
                Count=args.length;
                System.out.println("Number of arguments=" +count);
                while( i<count)
                {
                        string=args[i];
                        i=i+1;
                        System.out.println(I +":" +"Java is" +string +"!");
                }
        }
}
```

Run the above program with the command line argument as follows:

```
java comlinetest BASIC FORTRAN C++ JAVA
```