UNIT -1

Introduction to JAVA

Java is a general-purpose, object-oriented programming language. It is a blend of C & C++. It was developed by Sun Microsystems of USA, by a team headed by **James Gosling** in the year 1991. It was first named as 'Oak'.

It was primarily developed to be a portable, platform-independent language, means that the programs built in java can be executed in any platform and under different environments. So that it can be embedded in various electronic devices, such as microwave ovens, mobile phones, remote control etc.

C and C++ languages have problems in terms of reliability and portability. So the language java was formed, by removing the sources of problems of C and C++ and making the new language java simple, reliable , portable and powerful.

Java is also used in web applications, because of its portability and security. Applets and servlet programs were developed, which acts as web browser and web server respectively. World Wide Web emerged at about the same time that Java was being implemented. Internet wanted portable programs, so Java was used in its programs. It was a perfect response to the demands of the newly emerging, highly distributed computing universe.

JDK (Java Development Kit)

It is the Java tool which contains all the classes , methods and variables used to develop and build a Java program. It consists of –

- appletviewer (for viewing Java applets)
- javac (Java compiler)
- java (Java interpreter)
- javap (Java disassemble)
- javah(for C header files)
- javadoc(for creating HTML documents)
- jdb (Java debugger)

The program done in a computer in Java is compiled using java compiler and is then interpreted in another remote system (if required). The Java debugger is used to locate errors in the program. Java is an evolving programming language that began with the release of JDK. In the next phase, the Java development team included more interfaces and libraries as programmers demanded.

These API extensions were added to the JDK and called as Java 2 Standard Edition (J2SE), Java 2 Enterprise Eidition (J2EE) has the API to build applications for a multi-tier architecture. Java 2 Mobile Edition(J2ME) contains the API used to create wireless Java applications.

JRE (Java Runtime Environment) consists of tools required to execute the java code, eg. JVM. A program cannot take any action outside this boundary during runtime.

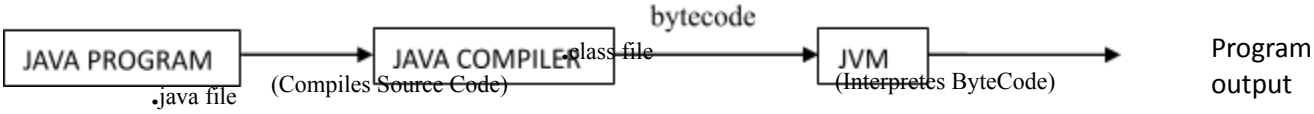
Java is interpreted

A Java source file is first compiled using Java compiler, this produces the bytecode. Byte code is highly optimized set of instruction. It is an intermediate stage during the execution. The byte code is than interpreted by the Java Virtual Machine (JVM), to produce the output in the expected system.

Due to the interpretation behavior of Java, this language is platform independent and portable .

JVM(Java Virtual Machine)

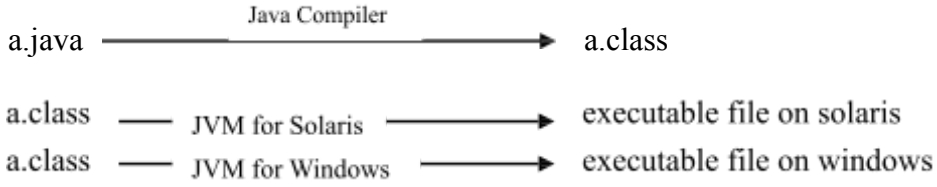
A Java Virtual Machine (JVM) is a virtual machine capable of executing Java bytecode. JVM enables Java to be both secure and portable.



The Java compiler generates the bytecode, which is a highly optimized set of instructions. JVM is an interpreter for bytecode and creates the executable file. Thus Java is called an interpreted language.

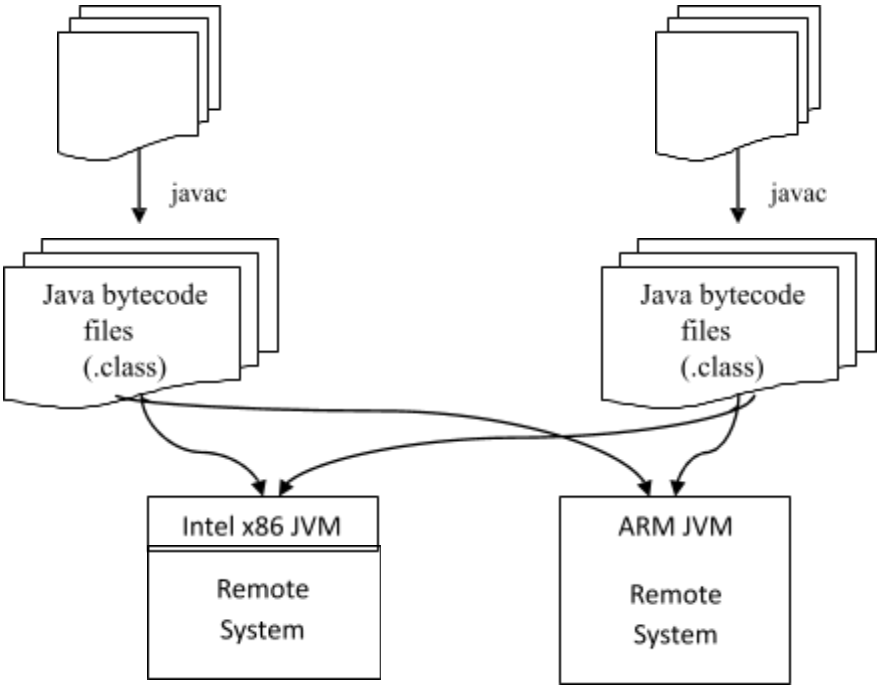
There are different JVMs for different operating systems. So only JVM need to be implemented for any other platform. Thus JVM helps in the portability of Java program. JVM also helps to make Java secure. Java code is executed by JVM and thus JVM has the control to prevent program from generating side effects to the system.

The program **compiled once**, can be **run anywhere**, **any number of times**.



System1 Java source file
(.java)

System2 Java source file
(.java)



How Java execution is faster than other languages?

In C or C++ , the preprocessor statements or header files are replaced with those large header files and then the whole program is compiles. So it takes more time for compilation.

But Java doesnot contain header files or preprocessor statements. The compilation process consists of just creating a bytecode. During interpretation by JVM, if any packages are included the required functions are executed and only the result is sent to the calling java program.

How is Java secure?

- Java compiler catches more compile-time errors.
- It does not use pointers, so addresses are not accidentally accessed.
- The programs are run in Java Runtime Environment and programs cannot take any action outside this boundary. Programs are prohibited from –
    - Making network connection to any host, except the host from which applet was downloaded.
    - Creating a new process, etc.

Difference between C , C++ and Java

Feature	**C**	**C++**	**Java**
Paradigms	Procedural	Procedural, OOP	OOP
Platform Dependence	Yes	Yes	No
Result from compiler	Executable Code	Executable Code	Java bytecode
Interpreter	Not used	Not used	Used
Memory management	Manual	Manual	Managed using a garbage collector
Pointers	Yes	Yes	No
Preprocessor	Yes	Yes	No
String Type	Character arrays	Character arrays, objects	Objects
Complex Data Types	Structures, unions	Structures, unions, classes	Classes
Inheritance	No	Multiple class inheritance	Single class inheritance, and interface
Operator Overloading	No	Yes	No
Empty argument list	Func(void)	Func() or Func(void)	Func()
Global Variable	Yes	Yes	No
Goto Statement	Yes	Yes	No
Header Files	Yes	Yes	No
Destructor	No	Yes	Uses finalize() method

Java Features (buzzwords)

The developers of Java wanted to design a language that solves all the problems of modern high-level language. They not only wanted secure and portable language but also a simple, compact and interactive language.

The key attributes of Java are –

- Simple
- Secure
- Portable
- Object – Oriented
- Robust
- Multithreaded
- Architecture – neutral

- Interpreted
- High performance
- Distributed and
- Dynamic

Simple – Java is a simple language to learn and program. Many features of C and C++ are used in Java. It is very easy to familiarize Java as the same concepts of C, C++ and object – oriented programming are used here. Java is a simplified version of C++.

Secure - Systems connected to Internet must be secure. There are more chances of virus attack while downloading. Viruses may affect your system or take important information like account no, password etc. from your system. Java achieved security by confining an applet to Java execution environment and not allowing it access to other parts of the computer.

Portability – portability means the code written and executed in a system must be able to execute in other systems of different environment also. This is achieved in Java by JVM. Changes and upgrades in operating system, processors and system resources will not force any changes in Java programs. Changes in the platform like windows, linux etc. will not stop the java program from execution.

Object-oriented – Java is true object-oriented language. Everything in Java is an object. All the program code resides in a class or object. The classes are arranged in packages and can be inherited.

Robust – Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. The major two reasons for any program failure are memory management mistakes and mishandled exceptions. These are handled very well in Java. It is designed as a garbage-collected language relieving the programmers from all memory management problems. It incorporates exception handling – handles serious errors and eliminates any risk of crashing the system.

Multithreaded – Java supports multithreading. Java program has the capability of handling multiple tasks simultaneously. This means that we need not wait for the application to finish one task before beginning another. Example - we can listen to an audio clip while scrolling a page and at the same time download an applet from a distant computer.

Architectural-Neutral - The main issue of Java developers was t hat the code **longevity** and **portability**. The code executed in a system, may not execute in the same system due to the

upgrades of OS, processors and change in core system resources. The Java designers made the language and built JVM such that a code written once can run anytime, anywhere and forever.

Interpreted and High performance - A Java program is compiled to form an intermediate Java bytecode. This can be executed in any system that has implemented JVM. JVM interprets the bytecode to form an executable file. The bytecode is highly optimized and so there is high performance.

Distributed – Java is designed as a distributed language, as it can share data and programs from Internet. Java handles TCP/IP protocol. Java also supports RMI, which enables a program to invoke remote methods.

Dynamic – Java is a dynamic language. It is capable of dynamically linking in new class libraries, methods and objects in a safe manner.

Object – Oriented Programming

Java is an object-oriented programming language. Object-oriented programming organizes a program. The main characteristics of object-oriented program is that it controls the access of data. Java is full of classes. Even a small program in Java contains a class. An object is an instance of a class.

Object

- An Object is an instance of user-defined data type called Class. It can also be said that object is a variable whose type is a user-defined Class.
- Object is a real – time entity.
- An object in Java is a representation of some physical, logical, or conceptual entity. For example, an object can be a vehicle, bank-account, animal, machine etc.

Class

- Class is a template for instantiating an object.
- It encapsulates the attributes and behavior of an entity.
- Similar objects belong to a class. For example, cat and dog belongs to animal class.

Object is a run-type entity in object-oriented programming where as Class is just a template for generating an object.

Abstraction

- Abstraction is representation of only the essential features of an entity. For example, to drive a car, you only need to know about accelerator, brake, clutch, and some other controls without ever knowing how the things inside the car work.

- Abstraction helps in managing the complexity of the software.

Encapsulation

- Encapsulation is bundling together the attributes and behaviors of an entity.

- Encapsulation together with data/information hiding feature allows limited visibility to the class users. Access specifiers (private, public, and protected) in Java are used to implement data hiding.

- Object-oriented approach is a data-centric approach. Data together with operations that can manipulate it are encapsulation into a Class.

Inheritance

- Accessing the properties and operations of a class by another class is called inheritance

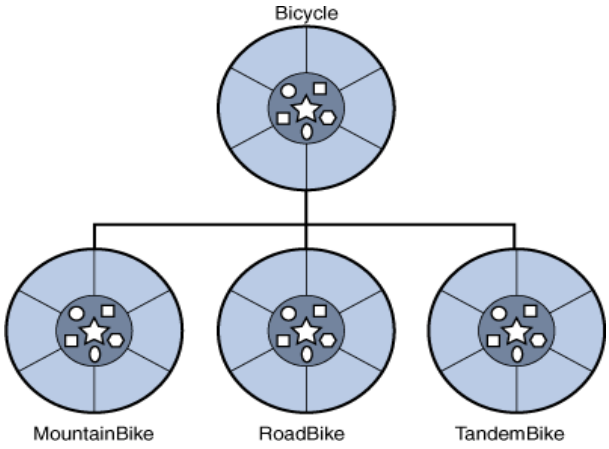
- Inheritance is similar to inheriting characteristics from our parents.

- Software development is costly and time-consuming. Object-orientation not only helps in managing complexity but also promotes software reusability through its 'inheritance' feature.

- Using inheritance, a class (called derived-class) inherits already well-developed methods from another class (called base-class); thereby saving lot of time and money.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this example, Bicycle now becomes the superclass of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses:



This gives Mountain Bike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read.

Polymorphism

- Literally, polymorphism means many forms.
- A function can exhibit many forms depending on the class of which it is a part. For example , we have a function called make_sound(). If is part of human object, make_sound() returns 'speak' and if it is a part of dog object then it returns 'bark'

A First Simple Program –

Java is full of classes, every method and variables in the program are from a class. The program name in Java should have the name of the class having the main function in it. The class name is extended with '.java' extension to form the program name.

Example - A program to print a statement.

```
/*This is a simple program to
print a statement at the console */
class Sample
{
        public static void main(String args[ ])
        {
                System.out.println("This is java");
        }
}
```

Output is :

```
c:\> javac Sample.java
c:\> java Sample
This is java
```

The line1 and 2 in the program are comment statements, there are mainly 2 types of comments in Java. '//' is a single line comment, using the // symbol comments the entire line starting from that symbol. The other type of comment is the multiline comment, the whole statements starting from '/*' and ending with'*/' are commented.

The line - **class Sample** uses the keyword class to declare that new class is being defined. Sample is an identifier that is the name of the class. The class members are included in the '{' & '}'.

The line **public static void main(String args[ ])** begins the main() method. **public** keyword is an access specifier, which allows the programmer to control the visibility of the class members. The main method must be declared as public, since it must be called by code outside of its class when the program is stated, JVM calls the main() function. **static** means that this variable or method is for the class not for any objects in particular. The main() method must be static as it is called without having to instantiate any class. **void** tells that main() does not return any value.

Any information needed to pass to the method is received by the array specified in parenthesis of the main function. This is an array of strings.

The line **System.out.println("This is java");** is used to display a line specified within double quotes.

Execution of Java program

The above program should have the filename as 'Sample.java'. The filename should have the same case as the class name, as Java is case-sensitive.

Java source file is compiled using Java compiler at the prompt as,

C;\> javac Sample.java

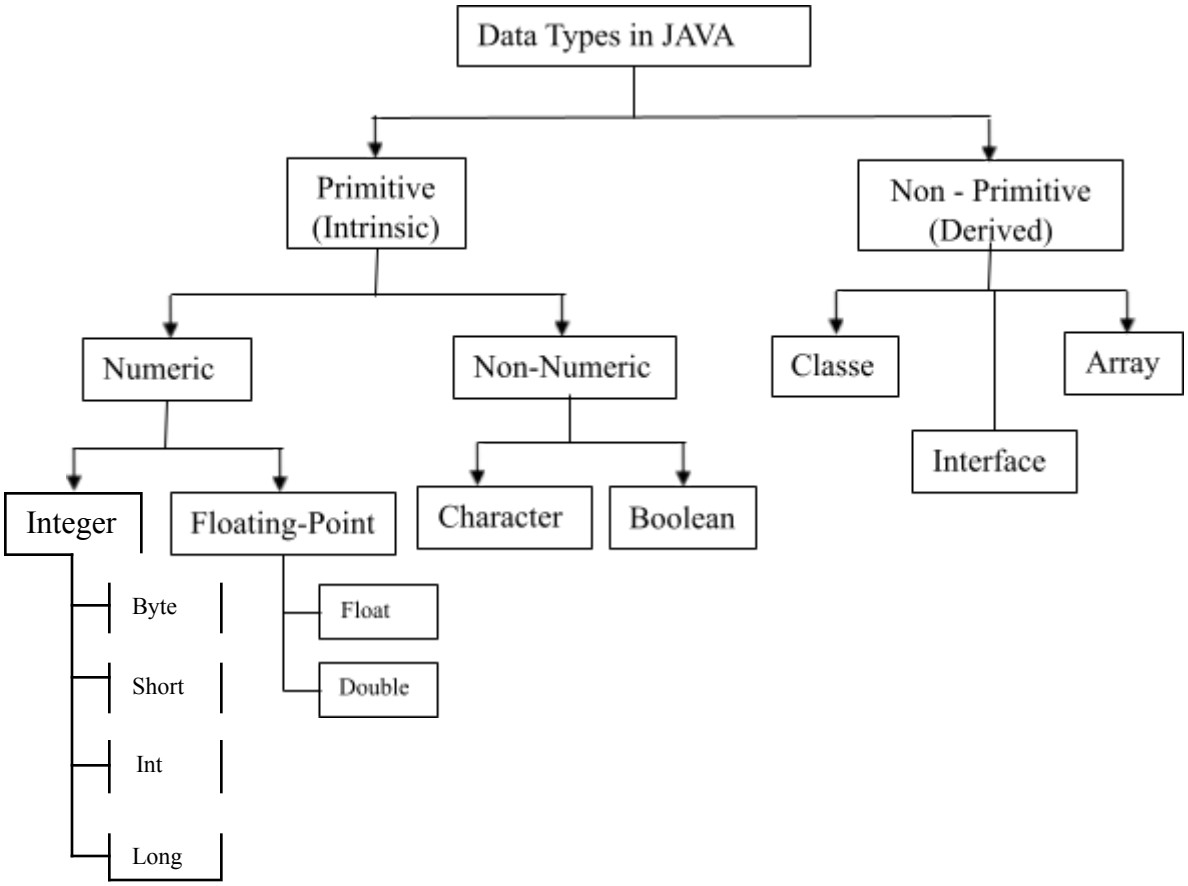
This creates a file called **Sample.class** that contains the bytecode version of the program. This cannot be directly executed it has to be interpreted by JVM as follows to get the output.

C:\>java Sample

The Java compiler will create separate class file for each class in the program. Then the **java** command is given with the name of the class which you want to execute.

Data Types

Every variable in Java has a data type. Data types specify the size and type of values that can be stored. The variety of data types available, allow the programmer to select the type appropriate to the needs of the application. Java data types are as follows –



Note: All the data types have a strictly defined range, regardless of the execution environment, ie. int is always 32 bits. This guarantees to achieve portability.

Integer Types

Java defines four integer types: **byte, short, int**, and **long**. They can hold only whole numbers such as 123,-87. These integer data types differs in the size of the values that can be stored. The memory size and range of all the four integer data types are shown in table below –

Type	Size (in bytes)	Minimum Value	Maximum Value
byte	1	-128	127
short	2	-32,768	32,767
int	4	-2,147,483,648	2,147,483,647
long	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Floating Point Types

Integer types can hold only whole numbers and therefore we use another type known as floating point type to hold numbers containing fractional parts such as 27.45 and -1.345. There are two kinds of floating point storage in Java – float type and double type. The float type values are single-precision numbers while the double types represent double-precision numbers.

Type	Size (in bytes)	Minimum Value	Maximum Value
float	4	3.4e-038	3.4e+038
double	8	1.7e-308	1.7e+308

Floating point numbers are always treated as double-precision numbers. To force them into single-precision mode, f or F must be appended to numbers. Example - 1.234f, 3.53 F

Character Type

The data type used to store characters is char. It has a size of 2 bytes. It uses Unicode to represent characters. It holds a single character.

Boolean Type

Java has a primitive type, called Boolean, for logical values. It can have only one of two possible values, true or false.

```
//A simple program to show the use of data type
class DTypes
{
        public static void main(String args[ ])
        {
                byte b =34;       //Declaration of byte variable
                int i=1000;       //Declaration of integer variable
                char c1,c2;       //Declaration of character variables
                c1 = 89;          // Initialization of character
                variable c2 = 'Y';
                double dval=160000000000;            //Declaration of double variable
                boolean b = true;        //Declaration of Boolean variable

                System.out.println("Value of byte variable b is " +b);
                System.out.println("Value of integer variable i is " +i);
                System.out.println("Value of character variable c1 is " +c1);
                System.out.println("Value of character variable c2 is " +c2);
                System.out.println("Value of double variable dval is " +dval);
                if (b)
                    System.out.println("This is printed as b is true");
```

```
            else
                System.out.println("This is not printed as b is true");
        }
}
```

Output of this program is –

```
Value of byte variable b is 34
Value of integer variable i is 100
Value of character variable c1 is Y
Value of character variable c2 is Y
Value of double variable dval is 160000000000
This is printed as b is true
```

Default Values

Fields that are declared but not initialized will be set to a default by the compiler. This default will be zero or null, depending on the data type. The default values for the above data types are -

Data Type	**Default Value**
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Strings

Strings are created using two classes in Java, **String** and **StringBuffer**. An object of String class is instantiated to store the value of strings. A Java string is not a character array and is not NULL terminated.

Syntax for declaring a string

```
String strname;
strname = new String("New string");
```

Eg –

String firstName;
firstName = new String(“Sajeev”);

These statements can be combined as ,
String firstName = new String(“Sajeev”);

An array of strings can be created. The
statement String arg[] = new String[4];
This statement creates an array ‘arg’ to hold four string constants.

String Methods

The String class contains many methods to translate the string variables. Eg – toLowerCase(), trim(), replace(), concat() etc. The table below shows the lists of string methods. Here **i** and **j** are int indexes into a string, **s** and **t** are Strings, and **c** is a char**. cs** is a CharacterSequence.

Method	**Task Performed**
i = *s*.length()	length of the string *s*
i = *s*.compareTo(*t*)	compares t to s. returns <0 if s<t, 0 if ==, >0 if s>t
b = *s*.equals(*t*)	true if the two strings have equal values
b = *s*.equalsIgnoreCase(*t*)	true if the two strings have equal values, case is not considered.
i = *s*.indexOf(*t*)	index of the first occurrence of String *t* in *s*
i = *s*.indexOf(*t*, i)	index of String *t* at or after position *i* in *s*
c = *s*.charAt(*i*)	Returns char at position *i* in *s*
s1 = *s*.substring(*i*)	substring from index *i* to the end of *s*
s1 = *s*.substring(*i*, *j*)	substring from index *i* to before index *j* of *s*
s1 = *s*.toLowerCase()	new String with all chars lowercase
s1 = *s*.toUpperCase()	new String with all chars uppercase
s1 = *s*.trim()	new String with whitespace deleted from front and back
s1 = *s*.replace(*c1*, *c2*)	new String with all *c1* characters replaced by character *c2*
b = *s*.matches(*regexStr*)	true if regular expression ‘regexStr’ matches the entire string in s
s = String.valueOf(*x*)	Converts x to String, where x is *any* type value

```
//A program to sort an array of strings in alphabetical order.
class StringOrdering
{
  static String name[ ] ={“Madras”,”Delhi”,”Ahmedbad”,”Calcutta”,”Bombay”};
  public static void main(String mar[])
  {
        int size = name.length;
        String temp = null;
        for (int i = 0; i < size ; i++)
        {
```

```
                for( int j =i+1; j < size ; j++)
                {
                    If(name[i].compareTo(name[j]) >0)
                    {
                        //swap the strings
                        temp = name[i];
                        name[i] = name[j];
                        name[j] = temp;
                     }
                }
        }
        for (int i = 0; i < size ; i++)
        {
                 System.out.println(name[i]);
        }
   }
}
```

Variables

The variable is the basic unit of storage in a Java program. A variable may take different values at different times, unlike constants that remain unchanged during the execution of program. All variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable

In Java, all variables must be declared before they can be used.

The **syntax** of declaring a variable -

type varname [= value][, varname [= value] ...] ;

here ‘type’ - one of Java's atomic datatypes, or the name of a class or interface,
‘varname’ - name of the variable.

The variable can be initialized by specifying an equal sign and a value. It must be of the same type (or compatible) as that specified for the variable. Use a comma to declare more than one variable of the specified type.

Eg -

```
int a, b, c; // declares three variables of int type a, b, and c.
int d = 3, e, f = 5; // declares three int type variable , d and f are initialized
byte z = 22; // declares one byte type variable z.
double pi = 3.14159; // declares and initializes a double variable.
char x = 'x'; // declares a character variable x and initializes to value 'x'.
```

The Scope and Lifetime of Variables

Java allows variables to be declared within any block, which begins with an opening brace '{' and ended by a closing brace '}'. A block defines a *scope.* The variables declared in a block are visible (accessible) only in that block.

```
// 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;
                        // x and y both known here.
                        System.out.println("x and y: " + x + " " + y);
                        x = y * 2;
                }
                System.out.println("Variable y is not known here ");
                // x is still known here .
                System.out.println("x is " + x);
        }
}
```

Output –

x and y: 10 20

Variable y is not known here

x is 40

Type Casting

The conversion of a variable from one data type to another data type is called type casting.

- Four integer types (byte,short,int and long)can be cast to any other data type except Boolean.
- Floating point datatypes(float and double) can also be casted to any other type except Boolean.
- Casting into a smaller type results in a loss of data.
- Floating point values if type casted to integer, it losses the fractional part of the floating value.

Casts that results in No Loss of information

From	To
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	Double

Java's Automatic Conversions – widening conversion

Type casting is automatically done from lower types to higher types. The casting takes place automatically when the following two conditions are met:

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

When these two conditions are met, a widening conversion takes place. For example, the

int type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

Eg: byte b=108; // b is of type byte
int a =b; // value of byte 'b' is automatically taken into integer 'a'.

Note: 1) When an expression is evaluated, the datatype of expression is automatically converted into int type.(if the expression contains smaller datatypes than an int).

2) If the expression has datatype which is higher than int, the whole expression is evaluated to the highest datatype in the expression.

Casting incompatible types

Sometimes it is required to convert higher datatype to lower datatype, eg. from **int** to **byte**. This conversion makes the value narrower so that is will fit into the target type. This conversion is called narrowing conversion. The syntax of explicit conversion is

Type var1 = (newtype) var2;

Eg: int a;
a=105;
byte b = (byte) a;

Access Specifiers

The access to the code and variables in the class is controlled by encapsulation. The access specifier in the declaration determines the accessibility of its members. The access specifiers in Java are – **public, private**, and **protected**.

When the member has a public specifier, then that member can be accessed by any other code. When the member is specified as private, then that member can only be accessed by other members of its class. protected applies only when inheritance is involved. So the main() function is specified as public, as it has to be invoked by Java run-time system.

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

The Basic Arithmetic Operators

The basic arithmetic operations-addition, subtraction, multiplication, and division- all do the same operations as in mathematic. The minus operator is also used as a unary operator, to negate the single operand. Unary operator means it acts on single operator. The division operator when applied to an integer type, there will be no fractional component attached to the result.

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

The following simple example program demonstrates the arithmetic operators and modulus operator. It also illustrates the difference between floating-point division and integer division.

```
class BasicMath
{
        public static void main(String args[])
        {
                // arithmetic using integers
                System.out.println("Integer
                Arithmetic"); int a = 1 + 1;
                int b = a *
                3; int c = b /
                4; int d = c -
                a; int e = -d;
                int x = 42
                System.out.println("a = " + a);
                System.out.println("b  =  "  +
                b); System.out.println("c = " +
                c); System.out.println("d = " +
                d);

               System.out.println("e = " + e); System.out.println("42 mod 10= " + 42 % 10);
                // arithmetic using doubles
                System.out.println("Floating Point Arithmetic");
                double da = 1 + 1;
                double db = da * 3;
                double dc = db / 4;
                double dd = dc - a;
                double de = -dd;
                System.out.println("da = " + da);
                System.out.println("db = " + db);
                System.out.println("dc = " + dc);
                System.out.println("dd = " + dd);
                System.out.println("de = " + de);
                System.out.println("42.3 mod 10 = " +42.3 % 10);
```

```
        }
}
```
Output is –
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
42 mod 10 = 2
Floating Point Arithmetic
da = 2
db = 6
dc = 1.5
dd = -0.5
de = 0.5
42.3 mod 10 = 2.3

Control Statements

Control statements causes the flow of execution to advance or branch based on changes to the state of a program. Program control statements can be put into the following categories: **selection, iteration, and jump.**

Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

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

if

The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

Syntax –

```
if (condition is true )
        statement1;
else
        statement2;
```

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

If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed.

Eg1 –

```
……
int a, b,res;
a=6;
b=0;
if( b != 0)
   res = a/b;
……
….
```

Here, if **b** is not zero, then a is divided by b.

Eg 2–

```
String passwd = “sai”;
……
if( passwd == “vidya”)
   System.out.println(“Valid Password”);
else
   System.out.println(“In-valid Password”);
…….
……..
```

Nested if

A *nested* **if** is an **if** statement that comes in the statement of another **if** or **else**. The **else** statement always refers to the nearest **if** statement that is within the same block as the **else.**

Eg –

```
if(i == 10)
{
        if(j < 20) a = b;
        if(k > 100)
            c = d; // this if is
        else
            a = c; // associated with this else

         }
else a = d; // this else refers to if(i == 10)
```

The final **else** is not associated with **if(j<20)**, because it is not in the same block. But, it 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

When a series of decisions are involved, a sequence of nested **if**s and **else**s are required. The ***if else-if*** *ladder* looks like this:

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

When one of the **if** statement conditions is **true**, the statement associated with that **if** is executed, and the rest of **else if** conditions are 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.

```
//A program to demonstrate if-else-if statements, to find the number of days in a month .
class IfElse
{
        public static void main(String args[])
        {
                int month = 4; // April
                int days;
                String year = "leapyear";
                if(month =2)
                {
                        if(year = "leapyear")
                            days = 29;
                        else
                            days = 28;
                }
                else if(month == 1 || month == 3 || month == 5|| month == 7|| month == 8|| month
                        == 10|| month == 12)
```

```
                days = 31;
        else
                days = 30;
        System.out.println("April has " + days + "days.");
    }
}
```

Ouput is –

April has 30 days.

Iteration Statements

The iteration statements or loops of Java are **for**, **while**, and **do-while**. A loop repeatedly executes the same set of instructions until a termination condition is met.

while

The **while** loop is the most fundamental looping statement of Java. It repeats a statement or block while its controlling expression is true.

Syntax –

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

The condition can be any Boolean expression(An expression which returns true or false). 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.

```
//A program to demonstrate the while loop.
class While

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

Output is –

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

The body of the loop will not execute even once if the condition is false to begin with.

do-while

Sometimes it is required to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In **do-while** loop the test of expression is done at the end.

Syntax

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

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. The **do-while** loop *condition* must be a Boolean expression.

```
// A program to demonstrate the do-while loop.
class DoWhile
{
        public static void main(String args[])
        {

           int n = 10; do {
                System.out.println("tick " +
                n); n--;
                } while(n > 0);
        }
}
```

The loop in the preceding program, can also be written as follows:

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

For loop

There are two for loops in Java. The first is the traditional form that is used in other languages also. The second is the new “for-each” form.

The **syntax** of the **for** statement –

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

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. The initialization expression is only executed once.
Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

```
//A program to 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);
        }
}
```

As the for loop contains only 1 statement, there is no need for the curly braces. Often the control variable of the **for** loop is only used within that loop, So it is possible to declare the variable inside the initialization portion of the **for**.

For example, the preceding program can be changed as follows as the control variable **n** is used inside the **for** only.

```
……………
……
for(int n=10; n>0; n--)
System.out.println("tick " + n);
}
}
```

Using the Comma in for loop
Initialization and iteration portions of the **for** loop may need more than one statement, this can be implemented by using a comma in the for loop. When the loop is controlled by the interaction of two or more variables, it would be useful if both variables could be included in the same **for** statement.
Eg –

```
// 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);
                }
        }
}
```

Output –

a = 1

b = 4

a = 2

b = 3

In this example, the initialization portion sets the values of both **a** and **b**. A comma is used to separate the initialization and iteration statements. The statements in the iteration portion are executed each time the loop repeats.

For-Each Statement

From JDK 5, a second form of **for** was defined that implements 'for-each' style loop. The 'for-each' loop moves through a collection of objects, such as an array, in strictly sequential fashion from start to finish. In Java this is built as an enhancement to for loop, so this style is also calles as enhanced for loop.

Syntax –

```
for(type itr-var : collection)
{
        statements;
}
```

Here, type specifies the data type and itr-var is the name of an iteration variable that will receive the elements from a collection, one by one from beginning to end of the collection.

This style of for loop eliminates the need to establish a loop counter and manually index the array. It automatically cycles through the entire array, obtaining one element at a time.

Eg –

```
//A program to demonstrate the use of for-each style of for loop
class ForEach
{
        public static void main(String args[])
        {
                int nums[ ] ={1,2,3,4,5,6,7,8,9,10};
                int sum = 0;
```

```
        for(int x : nums)
        {
            System.out.println("Value is : " + x);
            sum = sum + x;
        }
        System.out.println("Summation: " + sum);
    }
}
```

Output –
Value is : 1
Value is : 2
Value is : 3
Value is : 4
Value is : 5
Value is : 6
Value is : 7
Value is : 8
Value is : 9
Value is :10
Summation : 55

Note: 1) **break;** statement can be used to terminate the loop early.
2) Even if the value of the iteration variable is changed, it will not change the value in collection.

Nested Loops
Java allows loops to be nested. That is, one loop may be inside another loop.
Eg –

```
// 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();
        }
    }
}
```

Output –

.....

.....

.....

.....

.....

.....

....

...

..

.