Module - 2

Introduction to Java

A **programming language** is an artificial language that can be used to control the behavior of a machine, particularly a computer.

Procedural program—the functions are interdependent and therefore difficult to separate one from another. These interdependent functions cannot be reused in other programs. This made program development a complex task.

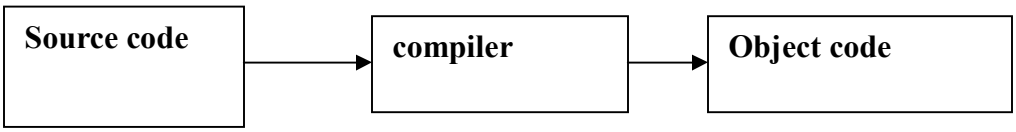
Data in procedural programming is visible and accessible throughout the program,

Ex: COBOL, Pascal, BASIC

Java is a programming language originally developed by Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform.

The language derives much of its syntax from C and C++

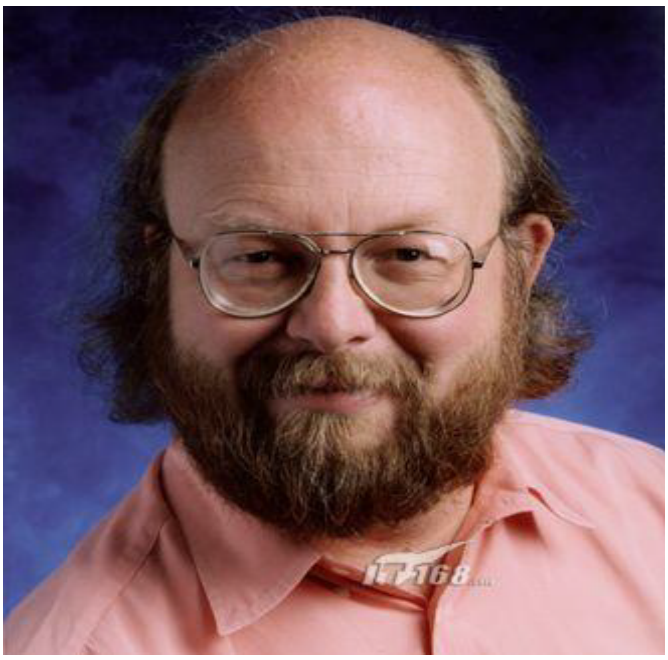
A **compiler** is a computer program (or set of programs) that translates text written in a computer language (the *source language*) into another computer language (the *target language*). The original sequence is usually called the *source code* and the output called *object code*.



In computer science, a **linker** or **link editor** is a program that takes one or more objects generated by compilers and assembles them into a single executable program.

In computing, an **executable** (file) causes a computer "to perform indicated tasks according to encoded instructions,"[1] as opposed to a file that only contains data. Files that contain instructions for an interpreter or virtual machine may be considered executable, but are more specifically called scripts or bytecode.

Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM).



The Java language was created by James Gosling in June 1991 for use in one of his many set-top box projects.

A **set-top box** (STB) or **set-top unit** (STU) is a device that connects to a television and an external source of signal, turning the signal into content which is then displayed on the television screen.

A **set-top box** (STB) or **set-top unit** (STU) is a device that connects to a television and an external source of signal, turning the signal into content which is then displayed on the television screen.

The language was initially called *Oak*, after an oak tree that stood outside Gosling's office—and also went by the name *Green*—and ended up later being renamed to *Java*,

The first public implementation was Java 1.0 in 1995.

The slogan of sun microsystem is (WORA) Write Once Run Anywhere

History of Java:

- In1990, Sun Micro Systems Inc. (US) was conceived a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called Stealth Project but later its name was changed to Green Project.
- In January 1991, Project Manager James Gosling and his team members Patrick Naughton, Mike Sheridan, Chris Wrath, and Ed Frank met to discuss about this project.
- Gosling thought C and C++ would be used to develop the project. But the problem he faced with them is that they were system dependent languages. The trouble with C and

C++ (and most other languages) is that they are designed to be compiled for a specific target and could not be used on various processors, which the electronic devices might use.

- James Gosling with his team started developing a new language, which was completely system independent. This language was initially called OAK. Since this name was registered by some other company, later it was changed to Java.
- James Gosling and his team members were consuming a lot of coffee while developing this language. Good quality of coffee was supplied from a place called "Java Island'. Hence they fixed the name of the language as Java. The symbol for Java language is cup and saucer.
- Sun formally announced Java at Sun World conference in 1995. On January 23rd 1996, JDK1.0 version was released. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.
- The similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences.
- While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java.
- One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

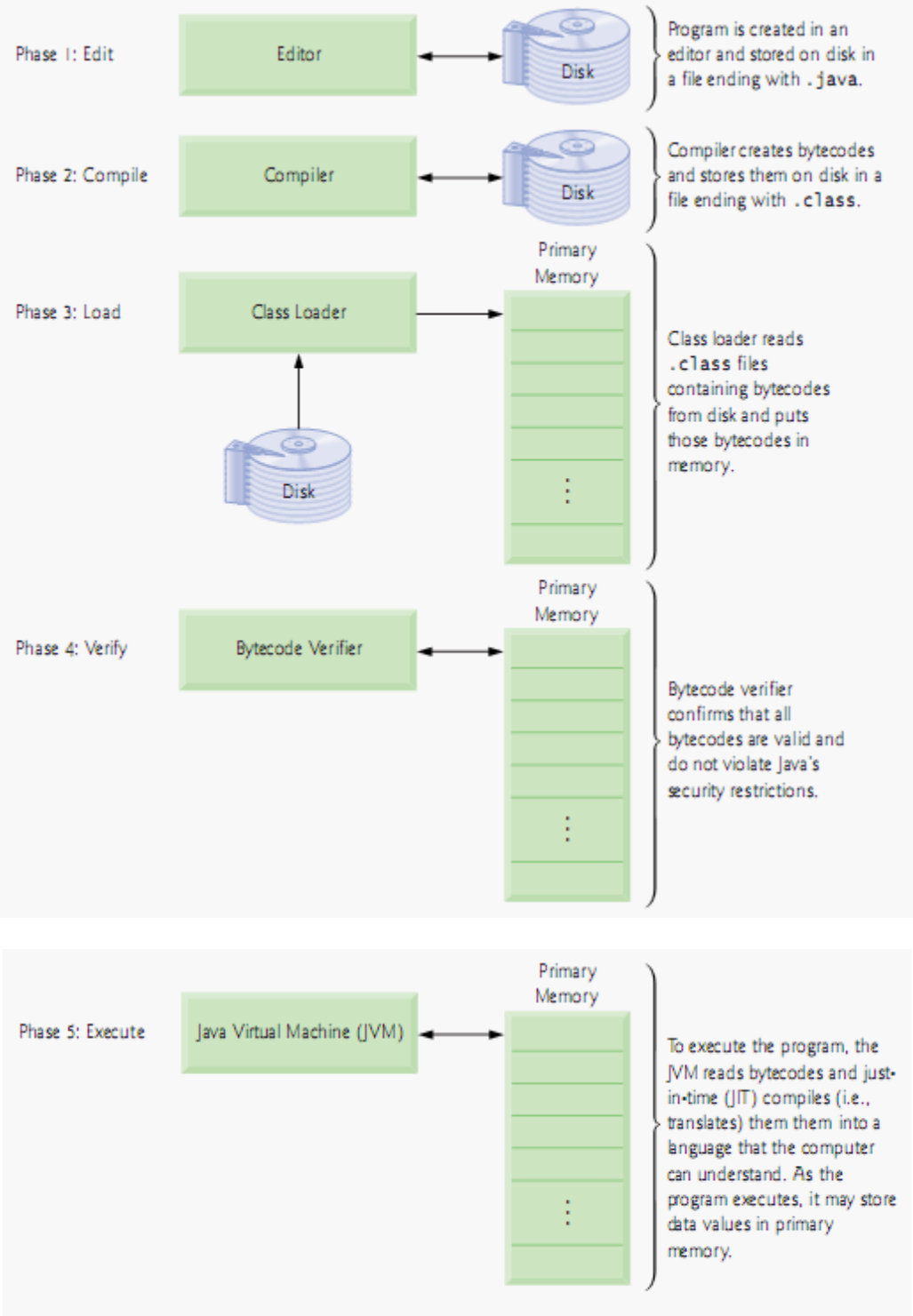


Fig. 1.1 | Typical Java development environment.

Table 2–1 Java Jargon

Name	Acronym	Explanation
Java Development Kit	JDK	The software for programmers who want to write Java programs
Java Runtime Environment	JRE	The software for consumers who want to run Java programs
Standard Edition	SE	The Java platform for use on desktops and simple server applications
Enterprise Edition	EE	The Java platform for complex server applications
Micro Edition	ME	The Java platform for use on cell phones and other small devices
Java 2	J2	An outdated term that described Java versions from 1998 until 2006
Software Development Kit	SDK	An outdated term that described the JDK from 1998 until 2006
Update	u	Sun's term for a bug fix release
NetBeans	—	Sun's integrated development environment

Table 2–2 Java Directory Tree

Directory Structure	Description
jdk	(The name may be different, for example, `jdk1.8.0`)
`bin`	The compiler and tools
`demo`	Look here for demos
`docs`	Library documentation in HTML format (after expansion of `j2sdk`*version*`-doc.zip`)
`include`	Files for compiling native methods (see Volume II)
`jre`	Java runtime environment files
`lib`	Library files
`src`	The library source (after expanding `src.zip`)

Obtaining the Java Environment:

- Install JDK after downloading, by default JDK will be installed in

C:\Program Files\Java\jdk1.5.0_05 (Here jdk1.5.0_05 is JDK's version)

Setting up Java Environment: After installing the JDK, we need to set at least one environment variable in order to able to compile and run Java programs. A PATH environment variable enables the operating system to find the JDK executable when our working directory is not the JDK's binary directory.

- Setting environment variables from a command prompt: If we set the variables from a command prompt, they will only hold for that session. To set the PATH from a command prompt: set PATH=C:\Program Files\Java\jdk1.5.0_05\bin;.%PATH%
- Setting environment variables as system variables: If we set the variables as system variables they will hold continuously.

o Right-click on My Computer

o Choose Properties

o Select the Advanced tab

o Click the Environment Variables button at the bottom

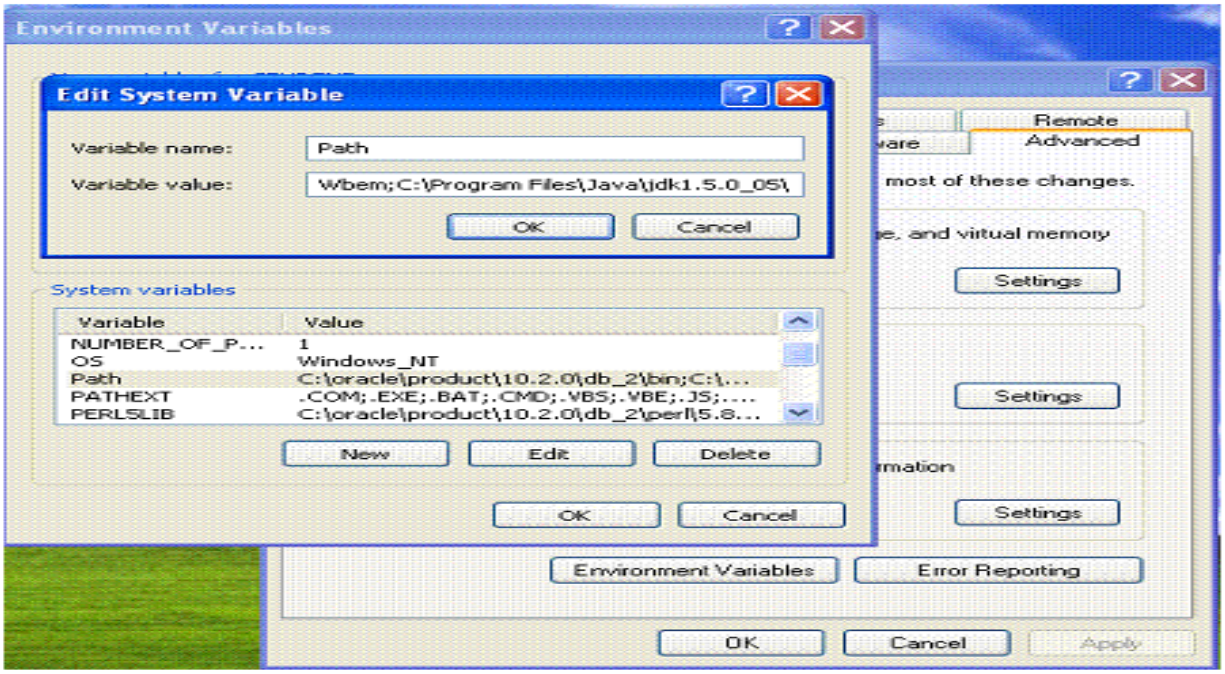
o In system variables tab, select path (system variable) and click on edit button

o A window with variable name- path and its value will be displayed.

o Don't disturb the default path value that is appearing and just append (add) to that path at the end:;C:\ProgramFiles\Java\ jdk1.5.0_05\bin;.;

o Finally press OK button.

- Repeat the process and type at:
  Variable name class path
  Variable value: C:\ProgramFiles\Java\ jdk1.5.0_05\lib\tool.jar;
- Finally press OK button.



JAVA Applications:

Features of Java (Java buzz words):

- Simple: Learning and practicing java is easy because of resemblance with c and C++.
- Object Oriented Programming Language: Unlike C++, Java is purely OOP.
- Distributed: Java is designed for use on network; it has an extensive library which works in agreement with TCP/IP.
- Secure: Java is designed for use on Internet. Java enables the construction of virus-free, tamper free systems.
- Robust (Strong/ Powerful): Java programs will not crash because of its exception handling and its memory management features.
- Interpreted: Java programs are compiled to generate the byte code. This byte code can be downloaded and interpreted by the interpreter. .class file will have byte code instructions and JVM which contains an interpreter will execute the byte code.
- Portable: Java does not have implementation dependent aspects and it yields or gives same result on any machine.
- Architectural Neutral Language: Java byte code is not machine dependent, it can run on any machine with any processor and with any OS.
- High Performance: Along with interpreter there will be JIT (Just In Time) compiler which enhances the speed of execution.
- Multithreaded: Executing different parts of program simultaneously is called multithreading. This is an essential feature to design server side programs.
- Dynamic: We can develop programs in Java which dynamically change on Internet (e.g.: Applets).

JAVA DEVELOPMENT KIT – (JDK)

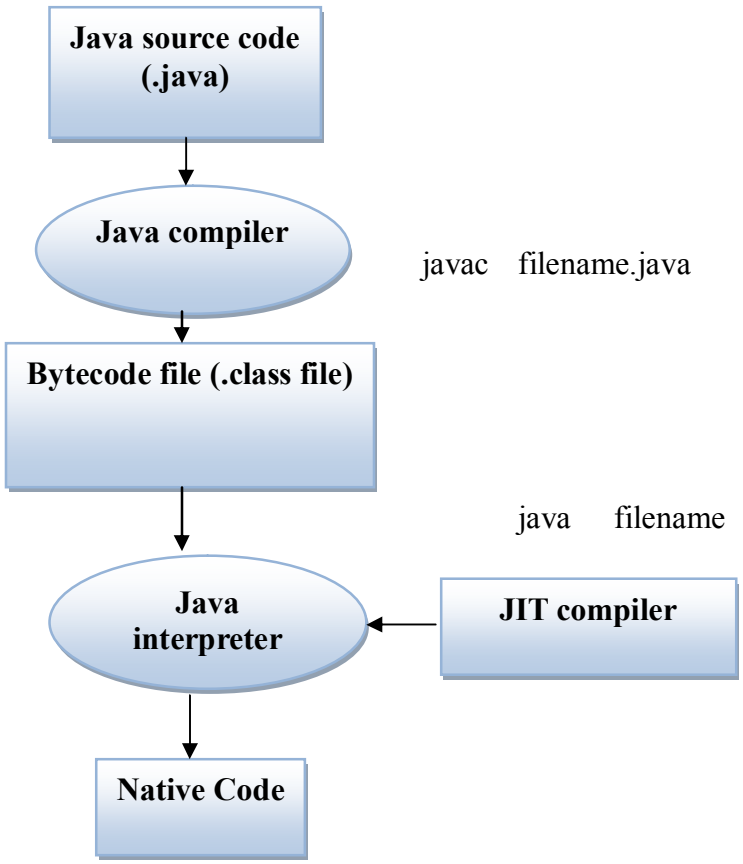
As the name suggests refers to a collection of tools that are used for developing and running java programs. The jdk consists of a collection of tools as

- appletviewer (to launch applets)
- javac (java compiler)
- java (java interpreter)
- javap (java disassembler)
- javah (for C header files)
- javadoc (for HTML documents)
- jdb (java debugger)

A source program written in java is compiled using "javac" (java compiler) and executed using "java" (java interpreter). The "jdb" (java debugger) is used to locate errors if any in the source file.

JAVA IS INTERPRETED:

Java as a language initially gained popularity mainly due to its platform independent architecture or portability feature. The reason for java to be portable is that it is interpreted.



BYTE CODE:

1. The concept of "Bytecodes" is the main reason for java to achieve portability.
2. When the source program is given as an input to the java compiler, the java compiler does not generate the machine level language executable code. Rather it generates bytecodes.
3. Bytecodes are highly optimized set of instructions designed to be executed by the java runtime system called as the JVM.

The Java Virtual Machine (JVM):

Java Virtual Machine (JVM) is the heart of entire Java program execution process. First of all, the .java program is converted into a .class file consisting of byte code instructions by the java compiler at the time of compilation. Remember, this java compiler is outside the JVM. This .class file is given to the JVM. Following figure shows the architecture of Java Virtual Machine.

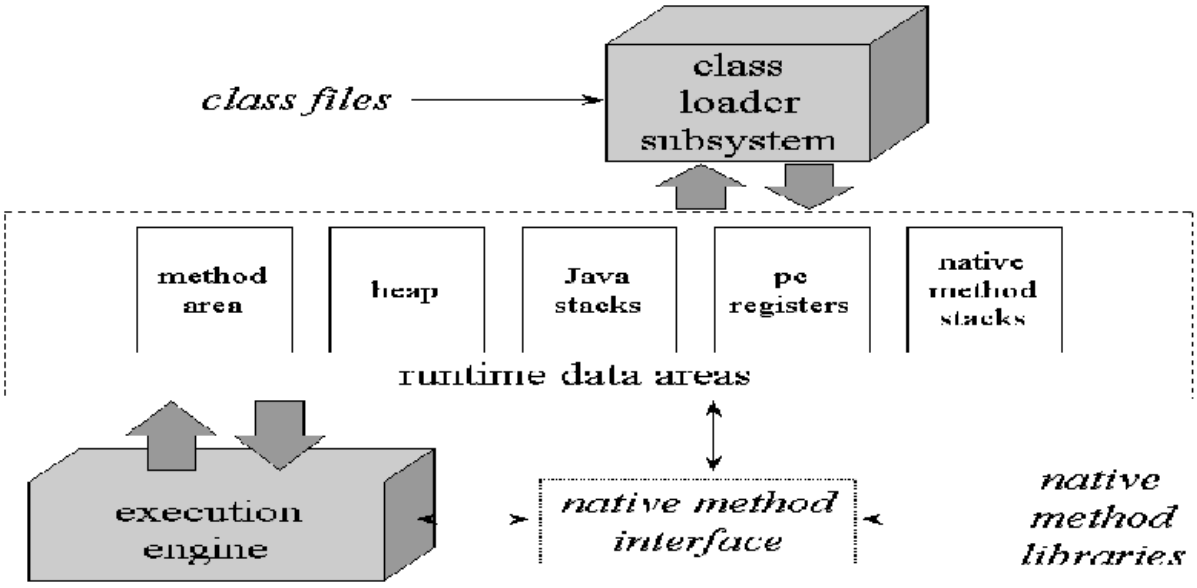


Figure: The internal architecture of the Java virtual machine.

In JVM, there is a module (or program) called class loader sub system, which performs the following instructions:

- First of all, it loads the .class file into memory.
- Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
- If the byte instructions are proper, then it allocates necessary memory to execute the program. This memory is divided into 5 parts, called run time data areas, which contain the data and results while running the program. These areas are as follows:
- **Method area:** Method area is the memory block, which stores the class code, code of the variables and code of the methods in the Java program. (Method means functions written in a class).
- **Heap:** This is the area where objects are created. Whenever JVM loads a class, method and heap areas are immediately created in it.
- **Java Stacks:** Method code is stored on Method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on Java Stacks. So, Java Stacks are memory area where Java methods are executed. While executing methods, a separate frame will be created in the Java Stack, where the method is executed. JVM uses a separate thread (or process) to execute each method.

- **PC (Program Counter) registers:** These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instruction of the methods.
- **Native Method Stacks:** Java methods are executed on Java Stacks. Similarly, native methods (for example C/C++ functions) are executed on Native method stacks. To execute the native methods, generally native method libraries (for example C/C++ header files) are required. These header files are located and connected to JVM by a program, called **Native method interface**.

- **Execution Engine** contains interpreter and JIT compiler which translates the byte code instructions into machine language which are executed by the microprocessor. Hot spot (loops/iterations) is the area in .class file i.e. executed by JIT compiler. JVM will identify the Hot spots in the .class files and it will give it to JIT compiler where the normal instructions and statements of Java program are executed by the Java interpreter.

OBJECT ORIENTED PROGRAMMING:

Java is basically an object oriented programming (OOP) language. OOP organizes a program around its data and a set of well-defined interfaces to that data.

The 3 main principles of OOP are-

(i) Encapsulation: is a mechanism that binds together code and the data it manipulates. It ensures that the data is safe and is not freely available for outside interferences and missuses.

(ii) Inheritance: is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. By the use of inheritance, a class has to define only those qualities that make it unique. The general qualities can be inherited from the parent class.

(iii) Polymorphism: in general refers to “one interfaces multiple methods” philosophy. It means that it is possible to generate a generic interface for a group of related activities.

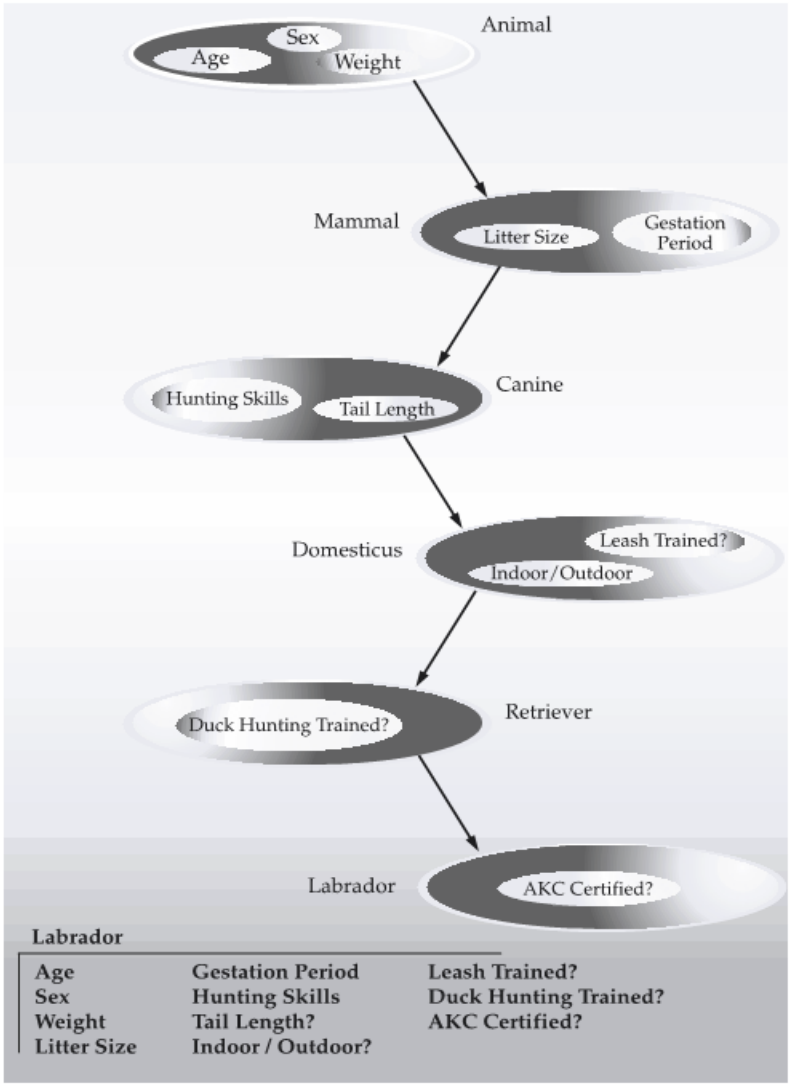
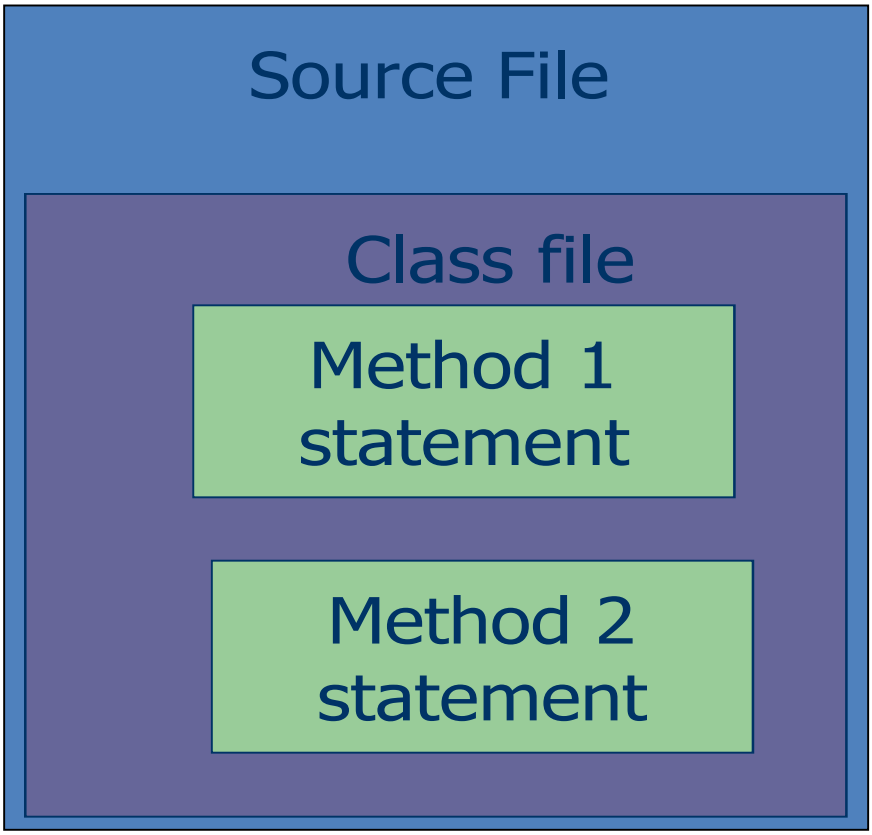


Figure 2-2. *Labrador inherits the encapsulation of all of its superclasses*

SIMPLE JAVA PROGRAMS:

Code Structure in Java



Code Structure of class

A source code file (with the .java extension) holds one class definition. A class has one or more methods. Method code is basically a set of statements. Every java class has to have at least one class and main method (one main per application any number of class).

```
Public class One
{


}
```

Comments: Comments are description about the aim and features of the program. Comments increase readability of a program. Three types of comments are there in Java:

- Single line comments: These comments start with //

e.g.: // this is comment line

- Multi line comments: These comments start with /* and end with */

e.g.: /* this is comment line*/

- Java documentation comments: These comments start with /** and end with */

These comments are useful to create a HTML file called API (application programming Interface) document. This file contains description of all the features of software.

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- Assignment expressions
- Any use of ++ or --
- Method invocations
- Object creation expressions

Such statements are called *expression statements*. Here are some examples of expression statements.

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements*. A *declaration statement* declares a variable. You've seen many examples of declaration statements already:

```
// declaration statement
double aValue = 8933.234;
```

Finally, *control flow statements* regulate the order in which statements get executed.

Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, BlockDemo, illustrates the use of blocks:

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

Structure of the Java Program:

As all other programming languages, Java also has a structure.

- The first line of the C/C++ program contains include statement. For example, <stdio.h> is the header file that contains functions, like printf (), scanf () etc. So if we want to use any of these functions, we should include this header file in C/ C++ program.
- Similarly in Java first we need to import the required packages. By default java.lang.* is imported. Java has several such packages in its library.
- A package is a kind of directory that contains a group of related classes and interfaces. A class or interface contains methods.
- Since Java is purely an Object Oriented Programming language, we cannot write a Java program without having at least one class or object.
- So, it is mandatory to write a class in Java program. We should use class keyword for this purpose and then write class name.
- In C/C++, program starts executing from main method similarly in Java, program starts executing from main method. The return type of main method is void because program starts executing from main method and it returns nothing.

Sample Program:

```
//A Simple Java Program
import java.lang.System;
import java.lang.String;
class Sample
{
        public static void main(String args[])
                {
                        System.out.print ("Hello world");
                }
}
```

- Since Java is purely an Object Oriented Programming language, without creating an object to a class it is not possible to access methods and members of a class.
- But main method is also a method inside a class, since program execution starts from main method we need to call main method without creating an object.
- Static methods are the methods, which can be called and executed without creating objects.
- Since we want to call main () method without using an object, we should declare main () method as static. JVM calls main () method using its Classname.main () at the time of running the program.
- JVM is a program written by Java Soft people (Java development team) and main () is the method written by us. Since, main () method should be available to the JVM, it should be declared as public. If we don't declare main () method as public, then it doesn't make itself available to JVM and JVM cannot execute it.
- JVM always looks for main () method with String type array as parameter otherwise JVM cannot recognize the main () method, so we must provide String type array as parameter to main () method.
- A class code starts with a {and ends with a}.
- A class or an object contains variables and methods (functions). We can create any number of variables and methods inside the class.
- This is our first program, so we had written only one method called main ().
- Our aim of writing this program is just to display a string "Hello world".
- In Java, print () method is used to display something on the monitor.
- A method should be called by using objectname.methodname (). So, to call print () method, create an object to PrintStream class then call objectname.print () method.
- An alternative is given to create an object to PrintStream Class i.e. System.out.
- Here, System is the class name and out is a static variable in System class, out is called a field in System class.
- When we call this field a PrintStream class object will be created internally. So, we can call print() method as: System.out.print ("Hello world");

- println () is also a method belonging to PrintStream class. It throws the cursor to the next line after displaying the result.
- In the above Sample program System and String are the classes present in java.lang package.

Escape Sequence: Java supports all escape sequence which is supported by C/ C++. A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.

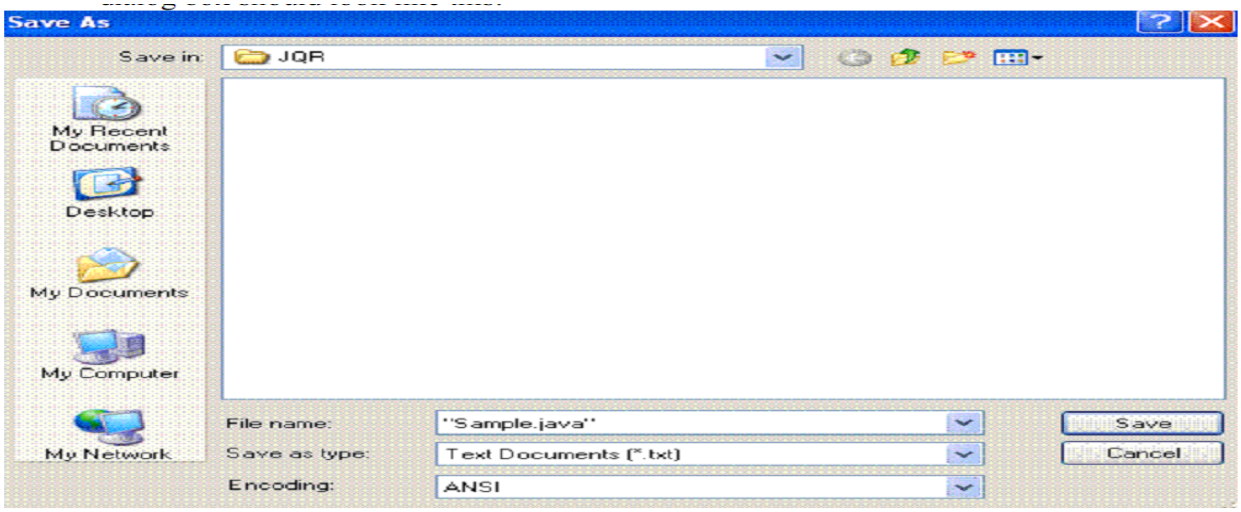
When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a form feed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

Creating a Source File:

- Type the program in a text editor (i.e. Notepad, WordPad, Microsoft Word or Edit Plus). We can launch the Notepad editor from the Start menu by selecting Programs > Accessories > Notepad. In a new document, type the above code (i.e. Sample Program).
- Save the program with filename same as Class name (i.e. Sample.java) in which main method is written. To do this in Notepad, first choose the File > Save menu item. Then, in the Save dialog box:

- Using the Save in combo box, specify the folder (directory) where you'll save your file. In this example, the directory is JQR on the D drive.
- In the File name text field, type "Sample.java", including the quotation marks. Then the dialog box should look like this:



- Now click Save, and exit Notepad.

Compiling the Source File into a .class File:

- To Compile the Sample.java program go to DOS prompt. We can do this from the Start menu by choosing Run... and then entering cmd. The window should look similar to the following figure.

- The prompt shows current directory. To compile Sample.java source file, change current directory to the directory where Sample.java file is located. For example, if source directory is JQR on the D drive, type the following commands at the prompt and press Enter:

Now the prompt should change to D:\JQR>

- At the prompt, type the following command and press Enter.

javac Sample.java

- The compiler generates byte code and Sample.class will be created.

Executing the Program (Sample.class):

- To run the program, enter java followed by the class name created at the time of compilation at the command prompt in the same directory as:

java Sample.

- The program interpreted and the output is displayed.

DATA TYPES AND OTHER TOKENS:

There are 8 primitive data types

- Integer types
  - byte, short, int, long
- floating point type
  - float ,double
- Boolean type
- character type

• **Integers**: This group includes byte, short, int, and long, which are for whole-valued signed numbers.

• **Floating-point numbers**: This group includes float and double, which represent numbers with fractional precision.

• **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.

• **Boolean:** This group includes boolean, which is a special type for representing true/false values.

The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range.

// Compute distance light travels using long variables.

```
class Light
{
        public static void main(String args[])
        {
                int lightspeed;
                long days;
                long seconds;
                long distance;
                // approximate speed of light in miles per second
                lightspeed = 186000;
                days = 1000; // specify number of days here
                seconds = days * 24 * 60 * 60; // convert to seconds
                distance = lightspeed * seconds; // compute distance
                System.out.print("In " + days);
```

```
            System.out.print(" days light will travel about ");
            System.out.println(distance + " miles.");
        }
}
```

This program generates the following **output:**

In 1000 days light will travel about 16070400000000 miles.

Clearly, the result could not have been held in an int variable.

// Compute the area of a circle.

```
class Area
{
        public static void main(String args[])
        {
            double pi, r, a;
            r = 10.8; // radius of circle
            pi = 3.1416; // pi, approximately
            a = pi * r * r; // compute area
            System.out.println("Area of circle is " + a);
        }
}
```

// Demonstrate char data type.

```
class CharDemo
{
        public static void main(String args[])
        {
            char ch1, ch2;
```

```
        ch1 = 88;  // code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

This program displays the following

output:

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

// 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 statementChapter 3: Data Types, Variables, and Arrays 39
        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));
```

```
        }
}
```

The **output** generated by this program is shown here:

b is false

b is true

This is executed.

10 > 9 is true

Size and range of data types:

Group	Data types	Size	Range	Default value
Integer	Byte	1 byte	-2^{7} to 2^{7-1} (signed) Unsigned does not support	0
	Short	2 byte	-2^{15} to 2^{15-1}	0
	Int	4 byte	-2^{31} to 2^{31-1}	0
	Long	8 byte	-2^{63} to 2^{63-1}	0
Float	Float	4 byte	$3.4e^{-038}$ to $3.4e^{+038}$	0.0
	Double	8 byte	$1.7e^{-308}$ to $1.7e^{+308}$	0.0
Boolean	Boolean	1 bit	True or false	False
Character	char	2 byte	A single character	null

OPERATORS:

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

Operators: An operator is a symbol that performs an operation. An operator acts on variables called operands.

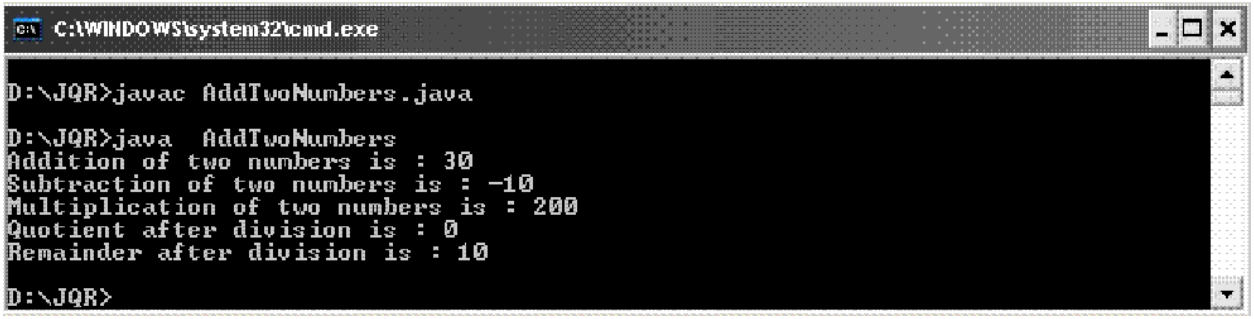
- **Arithmetic operators:** These operators are used to perform fundamental operations like addition, subtraction, multiplication etc.

Operator	Meaning	Example	Result
+	Addition	3 + 4	7
-	Subtraction	5 - 7	-2
*	Multiplication	5 * 5	25
/	Division (gives quotient)	14 / 7	2
%	Modulus (gives remainder)	20 % 7	6

Program : Write a program to perform arithmetic operations

```
//Addition of two numbers

class AddTwoNumbers

{    public static void mian(String args[])

   {  int i=10, j=20;

     System.out.println("Addition of two numbers is : " + (i+j));

     System.out.println("Subtraction of two numbers is : " + (i-j));

     System.out.println("Multiplication of two numbers is : " + (i*j));

     System.out.println("Quotient after division is : " + (i/j) );

     System.out.println("Remainder after division is : " +(i%j) );

   }

}
```

Output:



- **Assignment operator:** This operator (=) is used to store some value into a variable.

Simple Assignment	Compound Assignment
x = x + y	x += y
x = x – y	x -= y
x = x * y	x *= y
x = x / y	x /= y

Here is a sample program that shows several op= assignments in action:

```
// Demonstrate several assignment operators.
class OpEquals
{
	public static void main(String args[])
	{
		int a = 1;
		int b = 2;
		int c = 3;
		a += 5;
		b *= 4;
		c += a * b;
		c %= 6;
		System.out.println("a = " + a);
		System.out.println("b = " + b);
		System.out.println("c = " + c);
	}
```

}

The output of this program is shown here:

a = 6

b = 8

c = 3

- **Unary operators:** As the name indicates unary operator's act only on one operand.

Operator	Meaning	Example	Explanation
-	Unary minus	j = -k;	k value is negated and stored into j
++	Increment Operator	b++; ++b;	b value will be incremented by 1 (called as post incrementation) b value will be incremented by 1 (called as pre incrementation)
--	Decrement Operator	b--; --b;	b value will be decremented by 1 (called as post decrementation) b value will be decremented by 1 (called as pre decrementation)

The following program demonstrates the increment operator.

```
// Demonstrate ++.
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);
        }
}
```

The output of this program follows:

a = 2

b = 3

c = 4

d = 1

- **Relational operators:** These operators are used for comparison purpose.

Operator	Meaning	Example
==	Equal	x == 3
!=	Not equal	x != 3
<	Less than	x < 3
>	Greater than	x > 3
<=	Less than or equal to	x <= 3

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

The following program, Comparison Demo, tests the comparison operators:

```
class ComparisonDemo {

   public static void main(String[] args){
      int value1 = 1;
      int value2 = 2;
      if(value1 == value2)
         System.out.println("value1 == value2");
      if(value1 != value2)
         System.out.println("value1 != value2");
      if(value1 > value2)
         System.out.println("value1 > value2");
      if(value1 < value2)
```

```
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

Output:

```
value1 != value2
value1 <  value2
value1 <= value2
```

The Type Comparison Operator instanceof

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

The following program, InstanceofDemo, defines a parent class (named Parent), a simple interface (named MyInterface), and a child class (named Child) that inherits from the parent and implements the interface.

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

        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}

class Parent {}
class Child extends Parent implements MyInterface {}
```

interface MyInterface {}

Output:

obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true

When using the instanceof operator, keep in mind that null is not an instance of anything.

- **Logical operators:** Logical operators are used to construct compound conditions. A compound condition is a combination of several simple conditions.

Operator	Meaning	Example	Explanation
&&	and operator	if(a>b && a>c) System.out.print("yes");	If a value is greater than b and c then only yes is displayed
||	or operator	if(a==1 || b==1) System.out.print("yes");	If either a value is 1 or b value is 1 then yes is displayed
!	not operator	if(!(a==0)) System.out.print("yes");	If a value is not equal to zero then only yes is displayed

The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state:

!true == false and !false == true.

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

Here is a program that is almost the same as the BitLogic example shown earlier, but it operates on boolean logical values instead of binary bits:

// 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);
    }
}
```

After running this program, you will see that the same logical rules apply to Boolean values as they did to bits. As you can see from the following **output,** the string representation of a Java boolean value is one of the literal values true or false:

a = true

b = false

a|b = true

a&b = false

a^b = true

a&b|a&!b = true

!a = false

- **Bitwise operators:** These operators act on individual bits (0 and 1) of the operands. They act only on integer data types, i.e. byte, short, long and int.

Operator	Meaning	Explanation
&	Bitwise AND	Multiplies the individual bits of operands
|	Bitwise OR	Adds the individual bits of operands
^	Bitwise XOR	Performs Exclusive OR operation
<<	Left shift	Shifts the bits of the number towards left a specified number of positions
>>	Right shift	Shifts the bits of the number towards right a specified number of positions and also preserves the sign bit.
>>>	Zero fill right shift	Shifts the bits of the number towards right a specified number of positions and it stores 0 (Zero) in the sign bit.
~	Bitwise complement	Gives the complement form of a given number by changing 0's as 1's and vice versa.

Program : Write a program to perform Bitwise operations

```
//Bitwise Operations

class Bits

{

        public static void main(String args[])

        {

                byte x,y;

                 x=10;

                   y=11;

                   System.out.println ("~x="+(~x));

                   System.out.println ("x & y="+(x&y));

                   System.out.println ("x | y="+(x|y));

                   System.out.println ("x ^ y="+(x^y));

                   System.out.println ("x<<2="+(x<<2));

                   System.out.println ("x>>2="+(x>>2));
```

```
        System.out.println ("x>>>2="+(x>>>2));
    }
}
```

Output:

- **Ternary Operator or Conditional Operator (? :):** This operator is called ternary because it acts on 3 variables.

The syntax for this operator is:

Variable = Expression1? Expression2: Expression3;

First Expression1 is evaluated. If it is true, then Expression2 value is stored into variable otherwise Expression3 value is stored into the variable.

e.g.: max = (a>b) ? a: b;

Here is a program that demonstrates the ? operator. It uses it to obtain the absolute value of a variable.

```
// Demonstrate ?.
class Ternary {
public static void main(String args[]) {
int i, k;
i = 10;
k = i < 0 ? -i : i; // get absolute value of i
```

```
System.out.print("Absolute value of ");

System.out.println(i + " is " + k);

i = -10;

k = i < 0 ? -i : i; // get absolute value of i

System.out.print("Absolute value of ");

System.out.println(i + " is " + k);

}

}
```

The **output** generated by the program is shown here:

```
Absolute value of 10 is 10

Absolute value of -10 is 10
```

Now that you've learned how to declare and initialize variables, you probably want to know how to *do something* with them. Learning the operators of the Java programming language is a good place to start.

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

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order.

The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence.

Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence

Operators	Precedence
Postfix	*expr*++ *expr*--
Unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
Additive	+ -
Shift	<< >> >>>
Relational	< > <= >= instanceof
Equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
Ternary	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that

you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

TYPE CONVERSION AND CASTING:

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.

For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between in compatible types. To do so, you must use a cast, which performs **an explicit conversion** between incompatible types.

Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

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:

• **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.

For **widening conversions**, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

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

(target-type) value;

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;

byte b;

// ...

b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some **type conversions that require casts:**

```
// Demonstrate casts.

class Conversion

{

        public static void main(String args[])

        {

                byte b;

                int i = 257;

                double d = 323.142;

                System.out.println("\nConversion of int to byte.");

                b = (byte) i;

                System.out.println("i and b " + i + " " + b);

                System.out.println("\nConversion of double to int.");

                i = (int) d;

                System.out.println("d and i " + d + " " + i);

                System.out.println("\nConversion of double to byte.");
```

```
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following **output:**

Conversion of int to byte.

i and b 257 1

Conversion of double to int.

d and i 323.142 323

Conversion of double to byte.

d and b 323.142 67Let's look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case.

When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example,

examine the following expression:

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

The result of the 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, short, or char operand to int when evaluating an expression.

This means that the subexpression a*b is performed using integers—not bytes. Thus, 2,000, 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, this seemingly correct code causes a problem:

```
byte b = 50;

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

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

In cases where you understand the consequences of overflow, you should use an explicit

cast, such as

```
byte b = 50;

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

which yields the correct value of 100.

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions.

They are as follows:

First, all byte, short, and char values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands is double, the result is double.

The following program demonstrates how each value in the expression gets promoted

to match the second argument to each binary operator:

```
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);

}

}
```

Let's look closely at the type promotions that occur in this line from the program:

```
double result = (f * b) + (i / c) - (d * s);
```

In the first subexpression, f * b, b is promoted to a float and the result of the subexpression is float. Next, in the subexpression i/c, c is promoted to int, and the result is of type int. Then, in d*s, the value of s is promoted to double, and the type of the subexpression is double.

Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

Conclusion of Type Conversion:

- Size Direction of Data Type
  - Widening Type Conversion (Casting down)
    - Smaller Data Type → Larger Data Type
  - Narrowing Type Conversion (Casting up)
    - Larger Data Type → Smaller Data Type
- Conversion done in two ways
  - Implicit type conversion
    - Carried out by compiler automatically
  - Explicit type conversion
    - Carried out by programmer using casting
- Widening Type Converstion
  - Implicit conversion by compiler automatically

byte -> short, int, long, float, double
short -> int, long, float, double
char -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double

- Narrowing Type Conversion
  - Programmer should describe the conversion explicitly

byte -> char
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double
- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
int i;
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

STRINGS:

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

String greeting = "Hello world!"

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a String object with its value—in this case, Hello world!

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays hello.

String Length

Methods used to obtain information about an object are known as *accessor methods*. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A *palindrome* is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string. It invokes the String method charAt(i), which returns the i^{th} character in the string, counting from 0.

public class StringDemo

```
{
   public static void main(String[] args)
   {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      char[] tempCharArray = new char[len];
      char[] charArray = new char[len];

      // put original string in an
      // array of chars
      for (int i = 0; i < len; i++)
      {
         tempCharArray[i] =
            palindrome.charAt(i);
      }

      // reverse array of chars
      for (int j = 0; j < len; j++)
      {
         charArray[j] =
            tempCharArray[len - 1 - j];
      }

      String reversePalindrome =
         new String(charArray);
      System.out.println(reversePalindrome);
   }
}
```

Running the program produces this output:

doT saw I was toD

Concatenating Strings

The String class includes a method for concatenating two strings:

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end.

You can also use the concat() method with string literals, as in:

"My name is ".concat("Rumplestiltskin");

Strings are more commonly concatenated with the + operator, as in

"Hello," + " world" + "!"

which results in

"Hello, world!"

The + operator is widely used in print statements. For example:

```
String string1 = "saw I was ";
System.out.println("Dot " + string1 + "Tod");
```

which prints

Dot saw I was Tod

ARRAYS:

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

On which memory, arrays are created in java?

Arrays are created on dynamic memory by JVM. There is no question of static memory in Java; everything (variable, array, object etc.) is created on dynamic memory only.

Arrays: An array represents a group of elements of same data type. Arrays are generally categorized into two types:

- Single Dimensional arrays (or 1 Dimensional arrays)
- Multi-Dimensional arrays (or 2 Dimensional arrays, 3 Dimensional arrays, …)

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

type var-name[ ];

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold.

A one dimensional array or single dimensional array represents a row or a column of elements. For example, the marks obtained by a student in 5 different subjects can be represented by a 1D array.

- We can declare a one dimensional array and directly store elements at the time of its declaration, as: *int marks[] = {50, 60, 55, 67, 70};*
- We can create a 1D array by declaring the array first and then allocate memory for it by using new operator, as:
  array-var = new type[size];

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by new will automatically be initialized to zero.

int marks[]; //declare marks array

marks = new int[5]; //allot memory for storing 5 elements

These two statements also can be written as:

int marks [] = new int [5];

- We can pass the values from keyboard to the array by using a loop, as given here

```
for(int i=0;i<5;i++)
{
//read integer values from keyboard and store into marks[i]
Marks[i]=Integer.parseInt(br.readLine());

}
```

Let us examine some more examples for 1D array:

float salary[]={5670.55f,12000f};

float[] salary={5670.55f,12000f};

string names[]=new string[10];

string[] names={'v','r'};

Let's review: Obtaining an array is a two-step process.

First, you must declare a variable of the desired array type.

Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

The advantage of using arrays is that they simplify programming by replacing a lot of statements by just one or two statements. In C/C++, by default, arrays are created on static memory unless pointers are used to create them. In java, arrays are created on dynamic memory i.e., allotted at runtime by JVM.

Program : Write a program to accept elements into an array and display the same.

```
// program to accept elements into an array and display the same.
import java.io.*;
class ArrayDemo1
{  public static void main (String args[]) throws IOException
  {  //Create a BufferedReader class object (br)
    BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
    System.out.println (“How many elements: “ );
int n = Integer.parseInt (br.readLine ());
//create a 1D array with size n
    int a[] = new int[n];
    System.out.print ("Enter elements into array : ");
for (int i = 0; i<n;i++)
      a [i] = Integer.parseInt ( br.readLine ());
System.out.print (“The entered elements in the array are: “);
    for (int i =0; i < n; i++)
      System.out.print (a[i] + “\t”);
  }
}
```

Output:

Here is a program that creates an array of the number of days in each month.

```
// Demonstrate a one-dimensional array.


class Array {

public static void main(String args[]) {

int month_days[];

month_days = new int[12];

month_days[0] = 31;

month_days[1] = 28;

month_days[2] = 31;

month_days[3] = 30;

month_days[4] = 31;

month_days[5] = 30;

month_days[6] = 31;

month_days[7] = 31;

month_days[8] = 30;

month_days[9] = 31;

month_days[10] = 30;

month_days[11] = 31;

System.out.println("April has " + month_days[3] + " days.");

}

}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is month_days[3] or 30.

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

int month_days[] = new int[12];

This is the way that you will normally see it done in professionally written Java programs. Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements.

The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use new. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.

class AutoArray {

public static void main(String args[]) {

int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,

30, 31 };

System.out.println("April has " + month_days[3] + " days.");

}

}
```

When you run this program, you see the same output as that generated by the previous version.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range.

For example, the run-time system will check the value of each index into month_days to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```
// Average an array of values.

class Average {

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

```
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
double result = 0;
int i;
 for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
}
//sorting of names in ascending order
import java.io.*;
import java.lang.*;
import java.util.*;
class Sorting
{
        public static void main(String[] args)
        {

                int k=args.length;
                String temp=new String();
                String names[]=new String[k+1];
                for(int i=0;i<k;i++)
                {
                        names[i]=args[i];
                }
                for(int i=0;i<k;i++)
                        for(int j=i+1;j<k;j++)
```

```
                {
                        if(names[i].compareTo(names[j])<0)
                        {
                                temp=names[i];
                                names[i]=names[j];
                                names[j]=temp;
                        }
                }
        System.out.println("Sorted order is");
        for(int i=k-1;i>=0;i--)
        {
                System.out.println(names[i]);
        }
    }
}
```

Output: Java Sorting veer ram ajay

Ajay

Ram

Veer

Multi-Dimensional Arrays (2D, 3D … arrays):

A two dimensional array is a combination of two or more (1D) one dimensional arrays. A three dimensional array is a combination of two or more (2D) two dimensional arrays.

- **Two Dimensional Arrays (2d array):** A two dimensional array represents several rows and columns of data. To represent a two dimensional array, we should use two pairs of square braces [ ] [ ] after the array name. For example, the marks obtained by a group of students in five different subjects can be represented by a 2D array.

o We can declare a two dimensional array and directly store elements at the time of its declaration, as:

int marks[] [] = {{50, 60, 55, 67, 70},{62, 65, 70, 70, 81}, {72, 66, 77, 80, 69} };

o We can create a two dimensional array by declaring the array first and then we can allot memory for it by using new operator as:

int marks[ ] [ ]; //declare marks array

marks = new int[3][5]; //allot memory for storing 15 elements.

These two statements also can be written as*: int marks [ ][ ] = new int[3][5];*

Program : Write a program to take a 2D array and display its elements in the form of a matrix.

```
//Displaying a 2D array as a matrix
class Matrix
{  public static void main(String args[])
  {  //take a 2D array
    int x[ ][ ] = {{1, 2, 3}, {4, 5, 6} };
    // display the array elements
    for (int i = 0 ; i < 2 ; i++)
    {  System.out.println ();
      for (int j = 0 ; j < 3 ; j++)
        System.out.print(x[i][j] + "\t");
    }
  }
}
```

Output:

// Demonstrate a two-dimensional array.

```
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

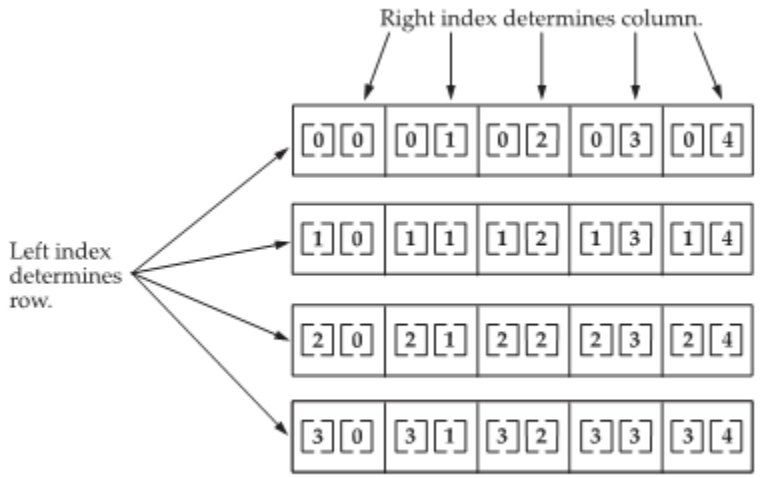
This program generates the following output:

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19



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

- **Three Dimensional arrays (3D arrays):** We can consider a three dimensional array as a combination of several two dimensional arrays. To represent a three dimensional array, we should use three pairs of square braces [ ] [ ] after the array name.

o We can declare a three dimensional array and directly store elements at the time of its declaration, as:

int arr[ ] [ ] [ ] = {{{50, 51, 52},{60, 61, 62}}, {{70, 71, 72}, {80, 81, 82}}};

o We can create a three dimensional array by declaring the array first and then we can allot memory for it by using new operator as:

int arr[ ] [ ] = new int[2][2][3]; //allot memory for storing 15 elements.

```
//example for 3-D array
class ThreeD
{
    public static void main(String args[])
    {
        int dept, student, marks, tot=0;
        int
arr[][][]={{{50,51,52},{60,61,62}},{{70,71,72},{80,81,82}},{{65,66,67},{75,76,77}}};
        for(dept=0;dept<3;dept++)
        {
```

```
                System.out.println("dept"+(dept+1)+":");
                for(student=0;student<2;student++)
                {
                        System.out.print("student"+(student+1)+"marks:");
                        for(marks=0;marks<3;marks++)
                        {
                                System.out.print(arr[dept][student][marks]+" ");
                                tot+=arr[dept][student][marks];
                        }
                        System.out.println("total:"+tot);
                        tot=0;
                }
                System.out.println();
        }
    }
}
```

arrayname.length: If we want to know the size of any array, we can use the property 'length' of an array. In case of 2D, 3D length property gives the number of rows of the array.

Alternative Array Declaration Syntax

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

type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable.

For example, the following two declarations are equivalent:

int al[] = new int[3];

int[] a2 = new int[3];

The following declarations are also equivalent:

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

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

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

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

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

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

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

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy they are stored as strings in a String array passed to the args parameter of main( ).

The first command-line argument is stored at args[0], the second at args[1], and so on.

For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.

class CommandLine {

public static void main(String args[]) {

for(int i=0; i<args.length; i++)

System.out.println("args[" + i + "]: " +args[i]);

}

}
```

Try executing this program, as shown here:

java CommandLine this is a test 100 -1

When you do, you will see the following **output:**

args[0]: this

args[1]: is

args[2]: a

args[3]: test

args[4]: 100

args[5]: -1

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

ACCESS CONTROL:

Type of modifier	Keyword	Description
Access modifier (use only one)	`public`	Data field is available everywhere (when the class is also declared `public`).
	`private`	Data field is available only within the class.
	`protected`	Data field is available within the class, available in subclasses, and available to classes within the same package.
	No access modifier	Data field is available within the class and within the package.
Use modifiers (all can be used at once)	`static`	Indicates that only one such data field is available for all instances of this class. Without this modifier, each instance has its own copy of a data field.
	`final`	The value provided for the data field cannot be modified (a constant).
	`transient`	The data field is not part of the persistent state of the object.
	`volatile`	The value provided for the data field can be accessed by multiple threads of control. Java ensures that the freshest copy of the data field is always used.

Access Specifiers: An access specifier is a key word that represents how to access a member of a class. There are mainly four access specifies in java.

- private: private members of a class are not available outside the class.
- public: public members of a class are available anywhere outside the class.
- protected: protected members are available outside the class.
- default: if no access specifier is used then default specifier is used by java compiler.
- Default members are available outside the class.
- The access modifiers supported by Java are static, final, abstract, synchronized, native, volatile, transient and strictfp.

Can u declare a class as 'private'?

No, if we declare a class as private, then it is not available to java compiler and hence a compile time error occurs. But, inner classes can be declared as private.

An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;

private double j;

private int myMethod(int a, char b) { // ...
```

To understand the effects of public and private access, consider the following program:

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

As you can see, inside the Test class, a uses default access, which for this example is the same as specifying public. b is explicitly specified as public. Member c is given private access. This means that it cannot be accessed by code outside of its class. So, inside the AccessTest class, c cannot be used directly. It must be accessed through its public methods setc( ) and getc( ). If you were to remove the comment symbol from the beginning of the following line,

```
//  ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the Stack class shown at the end.

```
// 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--];

}

}
```

As you can see, now both stck, which holds the stack, and tos, which is the index of the top of the stack, are specified as private. This means that they cannot be accessed or altered except through push( ) and pop( ). Making tos private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the stck array.

The following program demonstrates the improved Stack class. Try removing the commented-out lines to prove to yourself that the stck and tos members are, indeed, inaccessible.

```
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;

}

}
```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. However, in most real-world classes, you will need to allow operations on

data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared 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( ). **main( ) is declared as static because it must be called before any objects exist.**

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

Methods declared as static have several restrictions:

• They can only call other static methods.

• They must only access static data.

• They cannot refer to this or super in any way. (The keyword super relates to inheritance.).

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

The following example shows a class that has a static method, some static variables, and a 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(42);
}
}
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12. Then main( ) is called, which callsmeth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Here is the **output** of the program:

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

Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by **the dot operator.**

For example, if you wish to call a static method from outside its class, you can do so using the following general form:

```
classname.method( )
```

Here, classname is the name of the class in which the static method is declared. As you can see, this format is similar to that used to call non-static methods through object-reference variables.

A static variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside main( ), the static method callme( ) and the static variable b are accessed through their class name StaticDemo.

```
class StaticDemo {
static int a = 42;
```

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

Here is the output of this program:

a = 42

b = 99

Introducing final

A variable can be declared as final. Doing so prevents its contents from being modified.

This means that you must initialize a final variable when it is declared. For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for final variables.

Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant.

The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

native Modifier:

"**native**" keyword applies to methods. The code of a native method is written in some other language like C or C++. At execution time, this native code is executed separately and put with the Java output and is given.

To do all this, Java includes Java Native Interface (JNI) which is inbuilt into JDK. "native" permits the Java developer to write the machine-dependent code in other language (like C/C++) and make use in Java. Alternatively, if a C or C++ function exists with a very complex code, it is not required to convert it into Java code and instead can be used directly in a Java program.

Observe some methods of Java API.

```
public static native void sleep(long) throws InterruptedException;    // belonging to Thread class

public native int read() throws IOException;    // belonging to FileInputStream class
```

Observe **native keyword** in the above two statements. The functions' code is written in other language (means, not in Java). To execute these methods, Java takes the help of underlying OS.

volatile Modifier

The keyword **volatile** applies to variables and objects. A volatile variable value is more likely to get changed in the code. A volatile variable is treated differently by the JVM.

Volatile variables are not optimized (to minimize execution time) for **performance** by the compiler as their values are expected to get changed at any time without information. Volatile is better used with multiple threads that can change a variable value often.

Volatile can be used as follows.

```
public volatile int rate = 10;
```

Volatile does not have any meaning when applied to final variables or immutable objects or synchronized block.

The following two statements give compilation error.

```
final volatile int x = 10;
volatile final int x = 10;
```

transient Modifier:

It is used with RMI technology. "**transient**" is a keyword which means the data is not serialized. In RMI (Remote Method Invocation) technology, the objects along with the data are serialized. If the data is not required to be serialized and thereby not sent to the remote server, then declare the data as transient.

Following is the way how to declare a variable as transient.

```
public transient double goldRate = 210.5;
```

strictfp Modifier

A **floating-point value**, in a language, is platform-dependent. That is, the same floating value, in a Java program, executed on different operating systems may give different outputs (precision). To get the same precision (regardless of hardware, software and OS) on every operating system, declare the class or method as strictfp. "strictfp" is a keyword added in JDK 1.2 version.

"strictfp" is abbreviation for "strict floating-point".

```
public strictfp class Demo
    public strictfp interface Demo
    public strictfp void display()
```

If a class is strictfp, all the code in the class is evaluated with the strict floating point precision as laid in IEEE 754 standards. strictfp follows the rules formatted by IEEE 754. JVM does not apply its own precision.

Following table gives the list of access specifiers and modifiers that can be applied to variables, methods and classes.

specifier/modifier	local variable	instance variable	method	class
public	NA	A	A	A
protected	NA	A	A	NA
default	A	A	A	A
private	NA	A	A	NA
final	A	A	A	A
static	NA	A	A	NA
synchronized	NA	NA	A	NA
native	NA	NA	A	NA
volatile	NA	A	NA	NA
transient	NA	A	NA	NA
strictfp	NA	NA	A	A

A: Allowed NA: Not Allowed

LITERALS

A literal is a value that can be written directly into a Java program. The compiler can calculate the value it represents because most primitive data types have their own literal pattern.

Literal patterns:

- **whole numbers** are int values (e.g., 123).
- **whole numbers followed by an "L" (or "l")** are long values (e.g., 6720000000L). Note, long variables can be assigned a value using an int literal - a long literal is only needed when it is greater than the value an int data type can hold.
- **decimal numbers** are double values (e.g., 1.2) Note, you can also follow a decimal number with a "D" or "d" to explicitly define it as a double value.
- **decimal numbers followed by an "F" or "f"** are float values (e.g., 0.2F).
- **truth values** are boolean values (i.e., true, false).
- **single characters in single quotes** are character values (e.g., 'a').
- **characters in double quotes** are string values (e.g., "abc").
- **whole numbers preceded by "0x"** are hexadecimal numbers (e.g., 0xFF).
- **whole numbers preceded by "0"** are octal numbers (e.g., 0647).

Examples:

Using literals to assign a value to an int variable. An int literal:

```
int numOfDays = 7;
```

A hexadecimal literal:

```
int hexNumber = 0x5F;
```

CONTROL STATEMENTS:

Control statements are the statements which alter the flow of execution and provide better control to the programmer on the flow of execution. In Java control statements are categorized into selection control statements, iteration control statements and jump control statements.

- **Java's Selection Statements:** Java supports two selection statements: if and switch. These statements allow us to control the flow of program execution based on condition.

  - if Statement: if statement performs a task depending on whether a condition is true or false.

Syntax: 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 a boolean value. The else clause is optional.

Program : Write a program to find biggest of three numbers.

```
//Biggest of three numbers
class BiggestNo
{  public static void main(String args[])
  {  int a=5,b=7,c=6;
    if ( a > b && a>c)
      System.out.println ("a is big");
    else if ( b > c)
      System.out.println ("b is big");
    else
      System.out.println ("c is big");
  }
}
```

Output:

- Switch Statement: When there are several options and we have to choose only one option from the available ones, we can use switch statement.

Syntax: switch (expression)

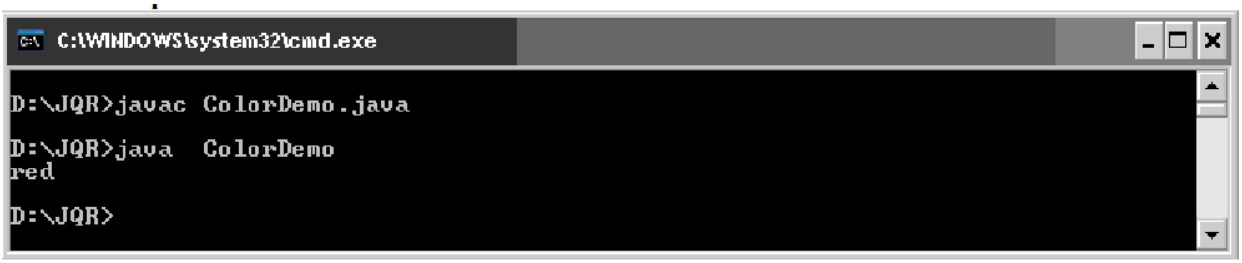
```
{  case value1:  //statement sequence
      break;
   case value2:  //statement sequence
      break;
    ……..
   case valueN:  //statement sequence
      break;
   default:  //default statement sequence
}
```

Here, depending on the value of the expression, a particular corresponding case will be executed.

Program : Write a program for using the switch statement to execute a particular task depending on color value.

```
//To display a color name depending on color value
class ColorDemo
{  public static void main(String args[])
  {  char color = 'r';
    switch (color)
    {  case 'r': System.out.println ("red");    break;
      case 'g': System.out.println ("green");  break;
      case 'b': System.out.println ("blue");    break;
      case 'y': System.out.println ("yellow");  break;
      case 'w': System.out.println ("white");  break;
      default: System.out.println ("No Color Selected");
    }
  }
}
```

Output:



- **Java's Iteration Statements**: Java's iteration statements are for, while and do-while. These statements are used to repeat same set of instructions specified number of times called loops.

A loop repeatedly executes the same set of instructions until a termination condition is met.

o while Loop: while loop repeats a group of statements as long as condition is true. Once the condition is false, the loop is terminated. In while loop, the condition is tested first; if it is true, then only the statements are executed. while loop is called as entry control loop.

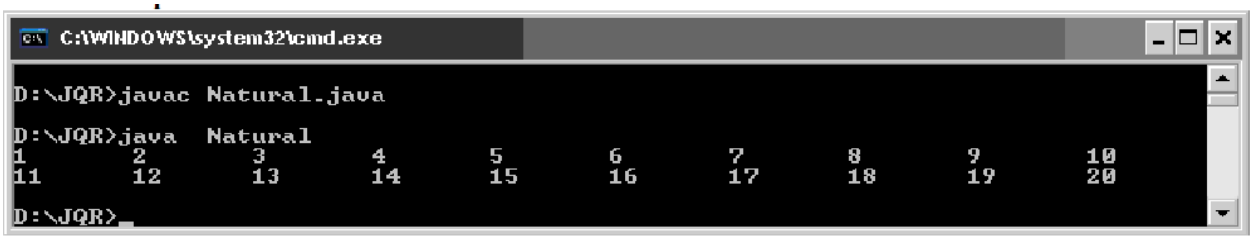
Syntax: while (condition)

```
{
 statements;
}
```

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
  {  int i=1;
   while (i <= 20)
   {  System.out.print (i + "\t");
    i++;
   }
  }
}
```

Output:



o **do…while Loop**: do…while loop repeats a group of statements as long as condition is true. In do...while loop, the statements are executed first and then the condition is tested. do…while loop is also called as exit control loop.

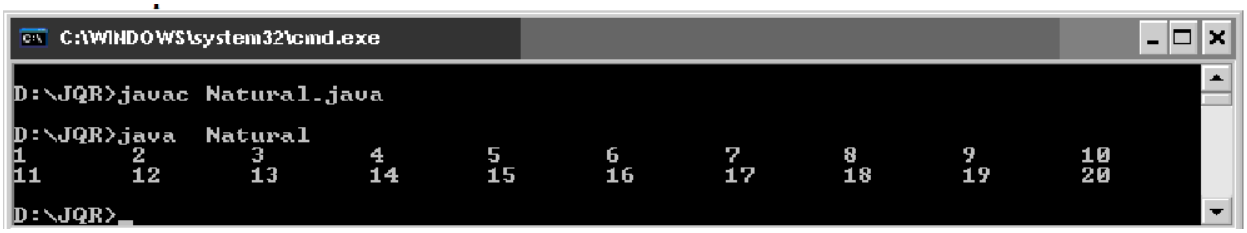
Syntax: do

```
{
 statements;
} while (condition);
```

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
  {  int i=1;
   do
   {  System.out.print (i + "\t");
    i++;
   } while (i <= 20);
  }
}
```

Output:



- **for Loop**: The for loop is also same as do…while or while loop, but it is more compact syntactically. The for loop executes a group of statements as long as a condition is true.

Syntax: for (expression1; expression2; expression3)

{ statements;

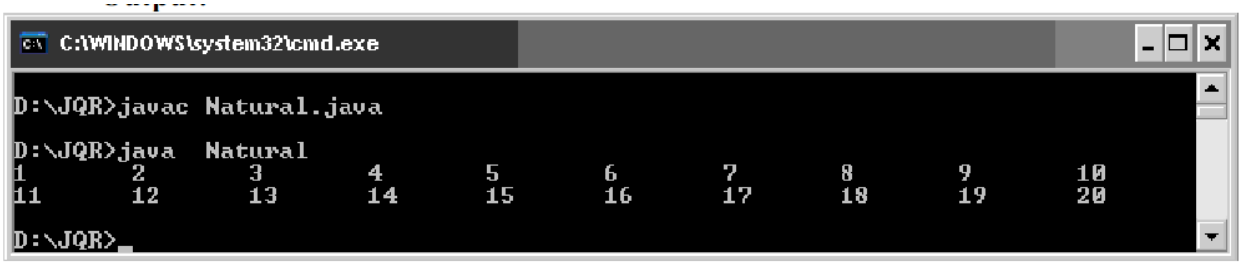
}

Here, expression1 is used to initialize the variables, expression2 is used for condition checking and expression3 is used for increment or decrement variable value.

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
  {  int i;
    for (i=1; i<=20; i++)
      System.out.print (i + "\t");
  }
}
```

Output:



- **Java's Jump Statements**: Java supports three jump statements: break, continue and return.

These statements transfer control to another part of the program.

- **break:**

✓ break can be used inside a loop to come out of it.

✓ break can be used inside the switch block to come out of the switch block.

✓ break can be used in nested blocks to go to the end of a block. Nested blocks represent a block written within another block.

Syntax: break; (or) break label;//here label represents the name of the block.

Program : Write a program to use break as a civilized form of goto.

```
//using break as a civilized form of goto
class BreakDemo
{
    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");
                }
        }
}
```

Output:

- **continue:** This statement is useful to continue the next repetition of a loop/ iteration.

When continue is executed, subsequent statements inside the loop are not executed.

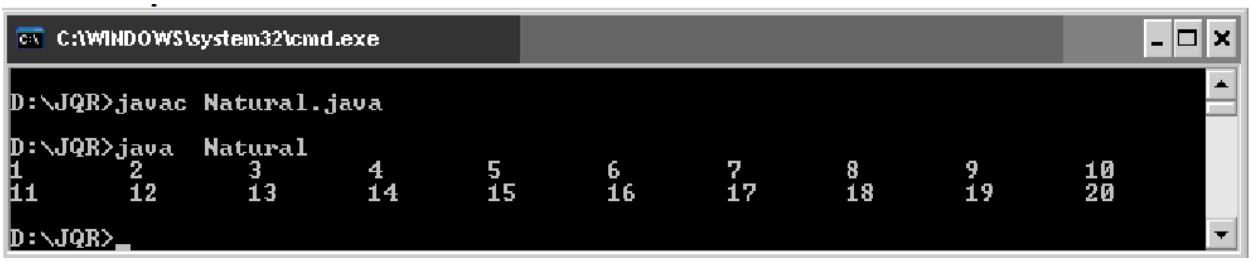
Syntax: continue;

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main (String args[])
  {  int i=1;
    while (true)
    {  System.out.print (i + "\t");
      i++;
      if (i <= 20 )
        continue;
      else
        break;
    }
```

```
  }
}
```

Output:



- **return statement:**
  - return statement is useful to terminate a method and come back to the calling method.
  - return statement in main method terminates the application.
  - return statement can be used to return some value from a method to a calling method.

Syntax: return;

(or)

return value; // value may be of any type

Program : Write a program to demonstrate return statement.

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

Output:

Note: goto statement is not available in java, because it leads to confusion and forms infinite loops.