Module 1

The Creation of Java

Java was developed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and MikeSheridan at Sun Microsystemsin 1991. It took 18 months to develop the first workingversion. This language was initially called "Oak," but was renamed "Java" in 1995. James Gosling is known as the father of java programming language.

Surprisingly, Java was not developed for the Internet. Instead, it was used to create software to be embedded in various consumer electronic devices such as microwave ovens and remote controls. C and C++ languages were also used in embedded software. If a program is written in C or C++, it may require a full C or C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier and more cost efficient solution was needed. In an attempt to find such asolution, Gosling and others began work on a portable, platform-independent language thatcould be used to produce code that would run on a variety of CPUs under differingenvironments. This effort ultimately led to the creation of Java.

How Java changed the Internet?

Java Applets

An *applet* is a tiny Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. An applet is downloaded on demand, without further interaction with the user. If the userclicks a link that contains an applet, the applet will be automatically downloaded and run inthe browser. Applets are intended to be small programs. They are typically used to displaydata provided by the server, handle user input, or provide simple functions, such as a loancalculator, that execute locally, rather than on the server. In essence, the applet allows somefunctionality to be moved from the server to the client.

Security

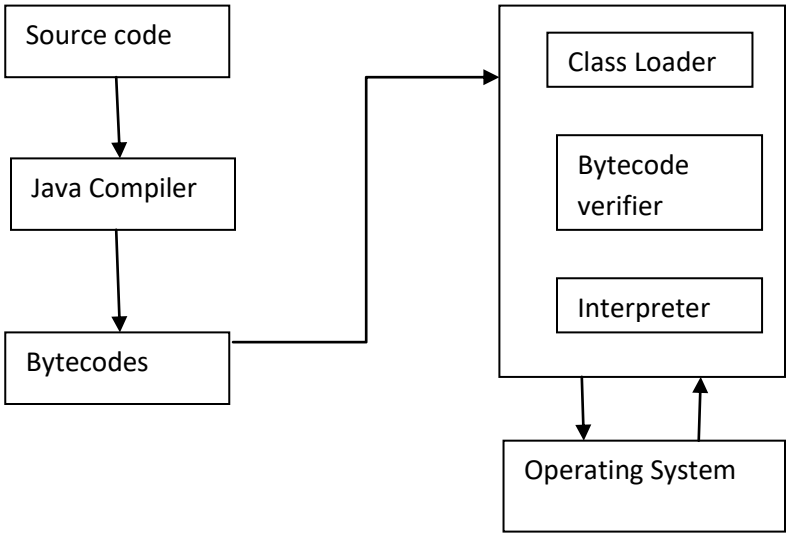
The code that is downloaded from the internet may contain virus. For example, a virus programmight gather private information, such as credit card numbers, bank account balances, andpasswords, by searching the contents of your computer's local file system. In order for Javato enable applets to be downloaded and executed on the client computer safely, it wasnecessary to prevent an applet from launching such an attack.**Java achieved this protection by allowing an applet to the Java execution environment and not allowing it access to other parts of the computer.**

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. **Java is portable means programs written in java will work on different types of computers as well as operating systems.** java has been implemented in the way of implementing a portable language

Java's Magic :The Bytecode

Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM).* In reality(essence) the original JVM was designed as an i*nterpreter for bytecode.* Only the JVM needs to be implemented for each platform. Once the jvm is installed in a given system, any Java program can run on it. Although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.



Once the source code (Eg., test.java) is compiled, it generates the class file (test.class) which is comprised of bytecodes. At run time, the JVM loads the class files, determines the semantics of each individual bytecode and performs appropriate computation. Just-in-time (JIT) is a component of the java run time environment that improves the performance of java applications at run time by compiling bytecodes into native machine code at run time.

Java's Buzzwords / Salient features of JAVA

- Simple
- Secure
- Portable
- Object Oriented
- Robust
- Multithreaded
- Architectural neutral
- Interpreted
- High performance Distributed
- Dynamic

Simple: Java was designed to be easy for the professional programmer to learn and use effectively. If one has some programming experience, he will not find Java hard to master.If you already understand the basic concepts of object-oriented programming, learning Javawill be even easier.

Secure : When we download a program from the Internet, there is a risk of viral infection. When we use java compatible web browser, we can safely download java applets without the fear of viral infection. bcz java achieved this protection by allowing an applet to the Java execution environment and not allowing it access to other parts of the computer.

Portable : Java is portable means programs written in java will work on different types of computers as well as operating systems.

Object Oriented : Java is purely object oriented programming language because without class and object it is impossible to write any Java program. Java is not pure object oriented programming language. because java supports primitive datatypes like int ,float ,boolean,double,long etc.

Robust : Java is a robust language for effective memory management and exception handling.For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers might forget to free. Java on its own eliminates these problems by managing memory allocation and deallocation automatically

Exceptional conditions arise in situations such as division by zero or "file not found,"and so on. Java handles these types of exceptions effectively at run time.

Multithreaded : Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows us to write programs that do many things simultaneously.

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

Interpreted and High performance : Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed onany system that implements the Java Virtual Machine. Most previous attempts atcross-platform solutions have done so at the expense of performance. Java bytecode was carefully designed so that it would be easy to translate directly intonative machine code for very high performance by using a just-in-time compiler.

Distributed : Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Java also supports *Remote Method Invocation (RMI).* This feature enables a program toinvoke methods across a network.

Dynamic : Java programs carry with them substantial amounts of run-time type information thatis used to verify and resolve accesses to objects at run time. This makes it possible todynamically link code in a safe The **Java Development Kit** (**JDK**) is a software **development** environment used for**developing** Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.JDK is a package of tools for developing Java-based software, whereas the JRE is a package of tools for running Java code. The JRE can be used as a standalone component to simply run

Java programs, but it's also part of the JDK. The JDK requires a JRE because running Java programs is part of developing them.

Following tools are the foundation of the JDK. They are the tools we use to create and build applications.

Tool Name	Description
Javac	The compiler for the Java programming language
Java	The launcher for Java applications
Javadoc	API documentation generator
Appletviewer	Run and debug applets without a web browser
Jar	Create and manage Java Archive (JAR) files.
Jdb	The Java Debugger
Javah	C header and stub generator. Used to write native methods
Javap	Class file disassemble

The Evolution of Java

Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. Thefeatures added by Java 1.1 were more significant and substantial. Java 1.1 added many new library elements,redefined and reconfigured many features of the 1.0 library. Italso deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus,Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the ―2" indicates ―second generation."
The creation of Java 2 was a watershed event, marking the beginning of Java's ―modernage." The first release of Java 2 carried the version number 1.2.With Java 2, Sun repackaged the Java product as J2SE (Java 2 PlatformStandard Edition). Java 2 added support for a number of new features, such as Swing and the CollectionsFramework, and it enhanced the Java Virtual Machine and various programming tools. Java2 also contained a few deprecations. The most important affected the **Thread** class in whichthe methods **suspend( )**, **resume( )**, and **stop( )** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. Although version 1.3 contained a smaller set of changes. The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new exceptions, and a channel-based I/O subsystem.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previousJava upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language. Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the developer's kit was called JDK 5. However, in order to maintain consistency, Sun decidedto use 1.5 as its *internal version* number, which is also referred to as the *developer version*number. The ―5" in J2SE 5 is called the *product version* number.

The newest release of java Java SE 6 builds on the base of J2SE 5, adding incremental improvements. Java SE 6adds no major features to the Java language proper, but it does enhance the API libraries,add several new packages, and offer improvements to the run time.

Object Oriented Programming

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

Abstraction

An essential element of object-oriented programming is *abstraction.* Humans managecomplexity through abstraction. For example, people do not think of a car as a set of tensof thousands of individual parts. They think of it as a well-defined object with its ownunique behavior. This abstraction allows people to use a car to drive to the grocery storewithout being overwhelmed by the complexity of the parts that form the car. They canignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

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

The Three OOP Principles

The three Object oriented programming (OOP) principles are**encapsulation, inheritance, and polymorphism.**

Encapsulation:*Encapsulation* is the mechanism that binds together code and the data it manipulates, andkeeps both safe from outside interference and misuse. The whole idea behind encapsulation is to hide the implementation details from users.In encapsulation, the data in one class is hidden from other classes, so it is also known as **data-hiding.**Both Abstraction & Encapsulation works hand in hand because Abstraction says what details to be made visible & Encapsulation provides the level of access right to that visible details.

Example: Power steering of a car is a complex system, which internally have lots of components tightly coupled together, they work synchronously to turn the car in the desired direction. But to the external world there is only one interface is available and rest of the complexity is hidden.

In java, the basis of encapsulation is the class. A *class* definesthe structure and behavior (data and code) that will be shared by a set of objects. For this reason, objects are sometimes referred to as*instances of a class.* Thus, a class is a logical construct; an object has physical reality. The data defined by the class are called member variables or instance variables and the code that operates on the data is referred to as member methods or just methods.The *private* methods and datacan only be accessed by code that is a member of the class. Therefore, any other code thatis not a member of the class cannot access a private method or variable.

Advantages of Encapsulation:

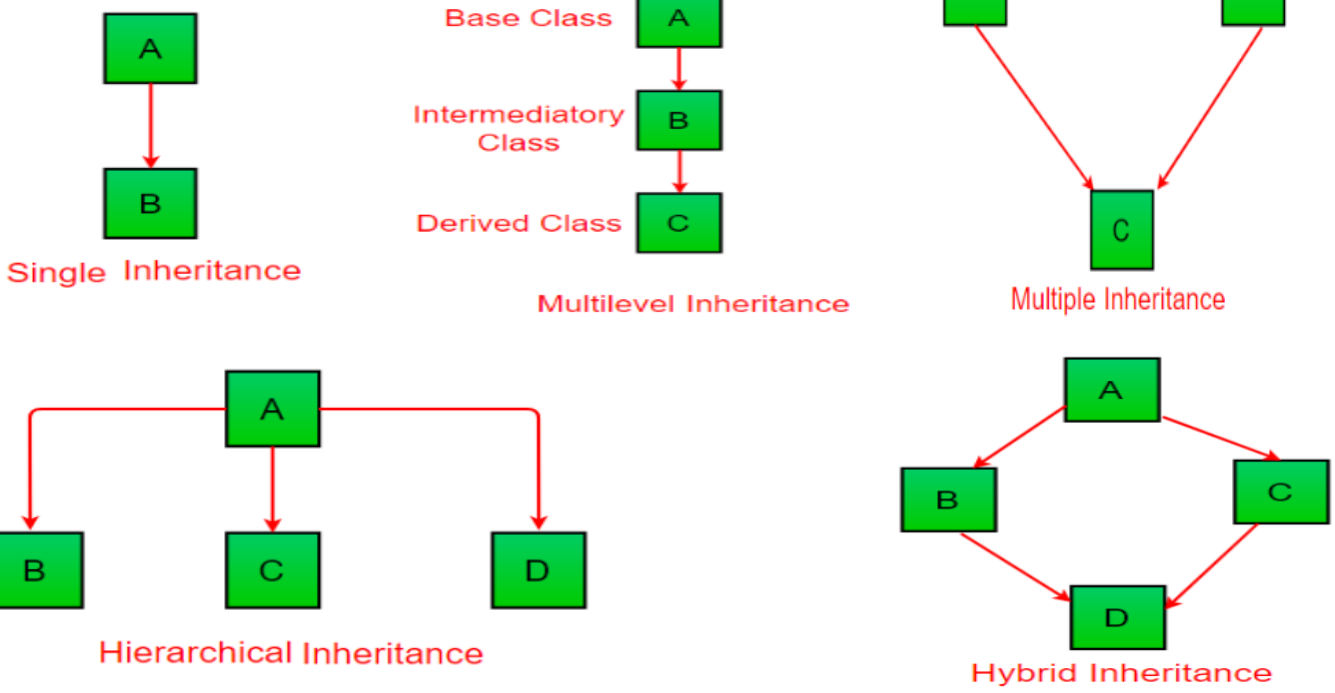
- **Data Hiding:** The user will have no idea about the inner implementation of the class
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement.
- **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Inheritance: It is the mechanism in java by which one object acquires the properties of the other object. Thisis important because it supports the concept of hierarchical classification.

Important terminology:

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of ~~"re~~usability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class.

Types of Inheritance



Example :Car is a four wheeler vehicle so assume that we have a class FourWheeler and a sub class of it named Car. Here Car acquires the properties of a class FourWheeler. Other classifications could be a jeep, tempo, van etc. FourWheeler defines a class of vehicles that have four wheels, and specific range of engine power, load carrying capacity etc. Car (termed as a sub-class) acquires these properties from

FourWheeler, and has some specific properties, which are different from other classifications of FourWheeler, such as luxury, comfort, shape, size, usage etc.

Polymorphism :Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you define one interface and have multiple implementations.

> **Example:**A car has a gear transmission system. It has four front gears and one backward gear. When the engine is accelerated then depending upon which gear is engaged different amount power and movement is delivered to the car. The action is same applying gear but based on the type of gear the action behaves differently.

Polymorphism could be static or dynamic. Method Overloading is static polymorphism while, Method overriding is dynamic polymorphism.

- **Overloading** in simple words means more than one method having the same method name that behaves differently based on the arguments passed while calling the method. This is called static because, which method to be invoked is decided at the time of compilation
- **Overriding** means a derived class is implementing a method of its super class. The call to overriden method is resolved at runtime, thus called runtime polymorphism.

First Sample program

```
/*
This is a simple Java program.Call this file "Example.java".
*/
class Example
{
// Your program begins with a call to main().
public static void main(String args[])
{
        System.out.println("This is a simple Java program.");
}
}
// compilation : c:/javac Example.java
// To run : c:/java Example
```

Two Control Statements

The Java **if** statement works much like the IF statement in any other language. Further, it is syntactically identical to the **if**statements in C, C++.

The syntax of if-then statement is

```
if (expression)
{
   //statements
}
```

The syntax of if-then-else statement is

```
if (expression)
```

```
        {
           //codes
        }
else{
        // some other codes
        }
```

Example:

```
        classIfelse{
                        publicstaticvoid main(String[]args){
                        int number =10;
                        if(number >0){
                                System.out.println("Number is positive.");
                                }
                        else {
                                System.out.println("Number is not positive.");
                                }
                                System.out.println("This statement is always executed.");
                        }
                }
```

for loop

The simplest form of the **for** loop is shown here

for(*initialization; testexpression; update*)

```
        {
        //   statements
        }
```

How for loop works?

1. The initialization expression is executed only once.
2. Then, the test expression is evaluated. Here, test expression is a boolean expression.
3. If the test expression is evaluated to true,
    - Codes inside the body of for loop is executed.
    - Then the update expression is executed.
    - Again, the test expression is evaluated.
    - If the test expression is true, codes inside the body of for loop is executed and update expression is executed.
    - This process goes on until the test expression is evaluated to false.
4. If the test expression is evaluated to false, for loop terminates.

Example:

```
// Program to print a line 10 times
        Class Loop {
                        publicstaticvoid main(String[]args){
                        for(inti=1;i<=10;++i){
                        System.out.println("Line "+i);
                        }
                        }
                }
```

Lexical Issues

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.

Whitespace: Java is a free-form language. This means that you do not need to follow any special indentationrules. For instance, the Exampleprogram could have been written all on one line or in any other strange way you felt like typing it. In Java,whitespace is a space, tab, or newline.

Identifiers: Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscoreand dollar-sign characters. They must not begin with a number, lest they be confused with anumeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.Some examples of valid identifiers areAvgTemp, a4, $test, this_is_ok.

Literals: Aconstant value in Java is created by using a *literal* representation of it. For example, the literal 100 specifies an integer, 98.5 specifies a floating point value, _a' specifies a character constant, ―I am a string" specifies a string.

Comments in java: Java supports three types of comments

1. Single line comment Eg., // This is a single line comment
2. Multiline comment
   Eg., /* this is an example
   For multiline comment */
3. Documentation Eg., /**…….*/
   This type of comment is used to produce an HTML file that documents the program automatically.

Separators:

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the table.

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

The Java Keywords: There are 50 keywords currently defined in the Java language. These keywords,combined with the syntax of the operators and separators, form the foundation of the Javalanguage. These keywords cannot be used as names for a variable, class, or method.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Java is a strongly typed language

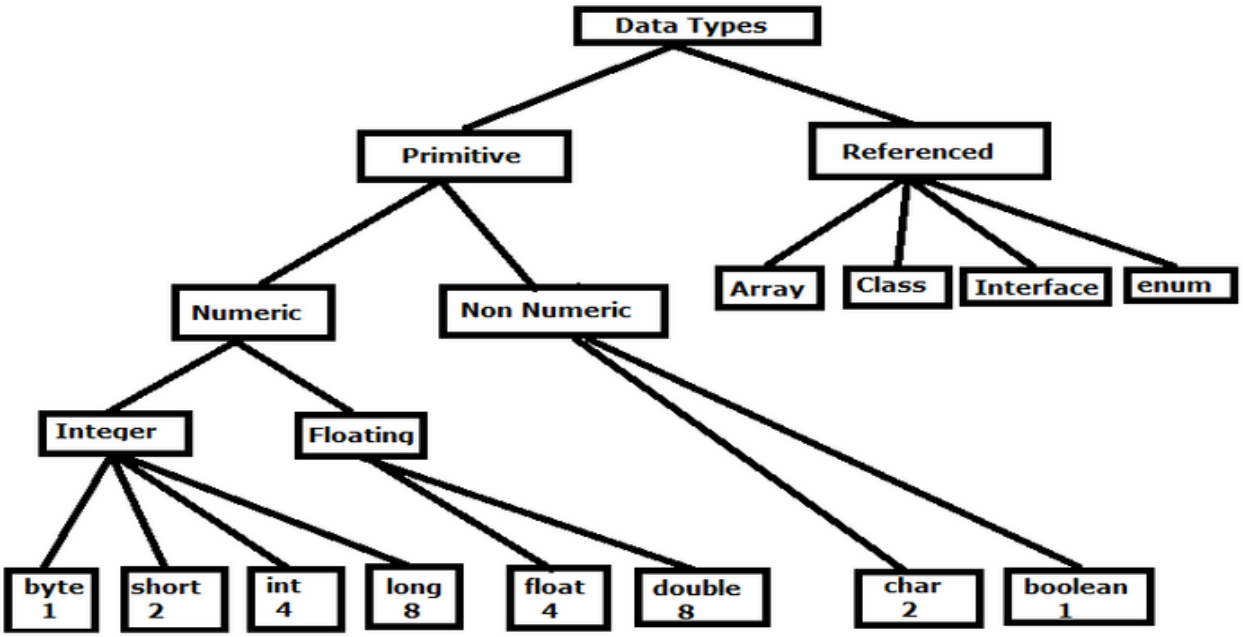
A strongly-typed programming language is one in which each type of data (such as integer, character, etc.) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.

In Java, when a variable is declared, it must be informed to the compiler what data type the variable stores like integer, float, double or string etc.For Example, consider the simple code snippet :

```
int age = 20;
String name = –Tridib";
booleanans = true;
```

In this snippet, each variable is declared with the data type.

Data Types



byte: The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Byte variables are declared by use of the byte keyword.

short: short sis a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used

Java type.This type is most likely applicable in 16-bit computers.

int: The most commonly used integer type is int. It is a signed 32-bit type. whenbyte and short values are used in an expression they are *promoted* to intwhen the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, intis often the best choice when an integer is needed.

long: long is a signed 64-bit type and is useful for those occasions where an inttype is not large enough to hold the desired value. The range of a long is quite large.

float: The type float specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision. Variables of type float are useful when you need a fractional component, but don‘t require a large degree of precision.

double: the type double uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. This is used when a large degree of precision is required.

```
// program to calculate the area of a circle
        classareacircle{
                public static void main(String arg[])
                {
                floatpi,r;
                double area;
                pi=3.14;
                  r=10.9;
                area=pi*r*r;
                System.out.println(–Area of circle is” +area);
                }
        }
```

character: Java uses Unicode to represent characters. *Unicode* defines a fullyinternational character set that can represent all of the characters found in all humanlanguages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic,Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, inJava **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s.The standard set of characters known as ASCII still ranges from 0 to .

boolean: Java has a primitive type, called **boolean**, for logical values. It can have only one of twopossible values, **true** or **false**.

Default values of data types in java

Data type	Default Value
byte	0
short	0
Int	0
long	0L
float	0.0f
double	0.0d
character	_\u000‘

boolean	False

A closer look at Literals

Integer literals: Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values,meaning they are describing a base 10 number. There are two other bases which can be usedin integer literals, *octal* (base eight) and *hexadecimal* (base 16).

Octal values are denoted in Javaby a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seeminglyvalid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range.Amore common base for numbers used by programmers is hexadecimal, which matchescleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimalconstant with a leading zero-x, (**0x**or **0X**). The range of a hexadecimal digit is 0 to 15, so *A*through *F* (or *a* through *f*)are substituted for 10 through 15.Integer literals create an **int**value, which in Java is a 32-bit integer value.

Floating point Literals: Floating-point numbers represent decimal values with a fractional component. They can beexpressed in either standard or scientific notation. *Standard notation* consists of a whole numbercomponent followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers. *Scientific notation*uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 bywhich the number is to be multiplied. The exponent is indicated by an *E* or *e* followed by adecimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05,and 2e+100.

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

Character Literals: Characters in Java are indices into the Unicode character set. They are 16-bit values that canbe converted into integers and manipulated with the integer operators, such as the additionand subtraction operators. Aliteral character is represented inside a pair of single quotes. Allof the visible ASCII characters can be directly entered inside the quotes, such as *'a', 'z',* and *'@'*.

String Literals: String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are–Hello World", –two\nlines", –\"This is in quotes\""".

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [= *value*][, *identifier* [= *value*] ...] ;

Examples:

int a, b, c; // declares three ints, a, b, and c.

```
int d = 3, e, f = 5;    // declares three more ints, initializing d and f.
byte z = 22;            // initializes z.
double pi = 3.14159;    // declares an approximation of pi
```

Dynamic Initialization: Java allows variablesto be initialized dynamically, using any expression valid at the time the variable is declared.For example, consider a program that computes the length of the hypotenuse ofa right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
        classDynInit {
                public static void main(String args[]) {
                double a = 3.0, b = 4.0;
                double c = Math.sqrt(a * a + b * b);// c is dynamically initialized
                System.out.println("Hypotenuse is " + c);
                }
            }
```

The scope and lifetime of variables

Variables are created when their scope isentered, and destroyed when their scope is left. This means that a variable will not hold itsvalue once it has gone out of scope. Therefore, variables declared within a method will nothold their values between calls to that method. Also, a variable declared within a block willlose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

```
        // Demonstrate block scope.
        class Scope {
                public static void main(String args[]) {
                        int x; // known to all code within main
                        x = 10;
                        if(x == 10) { // start new scope
                                int y = 20; // known only to this block
                                        // x and y both known here.
                                System.out.println("x and y: " + x + " " + y);
                        x = y * 2;
                                }
        // y = 100; // Error! y not known here
        // x is still known here.
        System.out.println("x is " + x);
          }
                }
```

Simple java programs

1. **Program to add two numbers**

```
importjava.util.*;
class add{
        public static void main(String arg[]) {
        int a, b,c;
        Scanner s=new Scanner(System.in);
```

```
        System.out.println(" Enter values of a and b”);
        a=s.nextInt();
        b=s.nextInt();
        c=a+b;
        System.out.println(" sum =”+c);
                }
        }
```

2. Prgram to find the area of a circle

```
        importjava.util.*;
        classareaof{
                public static void main(String arg[]) {
                doubler.area;
                Scanner s=new Scanner(System.in);
                System.out.println(" Enter value of radius”);
                a=s.nextDouble();
                area=3.14*r*r;
        System.out.println(" area =”+area);
                }
        }
```

3. Program to compute simple interest

```
        importjava.util.*;
        classSI{
                public static void main(String arg[]) {
                floatp,r,t,si;
                Scanner s=new Scanner(System.in);
                System.out.println(" Enter vales of p,r,and t”);
                p=s.nextFloat();
                r=s.nextFloat();
                t=s.nextFlaot();
                si=p*r*t/100
        System.out.println(" Simple Interest =”+si);
                }
        }
```

4. Program to read String

```
        importjava.util.*;
        classStringdemo{
                public static void main(String arg[]) {
                String S1;
                Scanner s=new Scanner(System.in);
                System.out.println(" Enter a string”);
                S1=s.nextLine();
```

```
                System.out.println(" Entered String is"+S1);
                    }
            }
```

5. **Java Program Example - Check Leap Year or Not**

```
importjava.util.Scanner;

public class JavaProgram
{
public static void main(String args[])
   {
intyr;
      Scanner scan = new Scanner(System.in);
  System.out.print("Enter Year : ");
yr = scan.nextInt();
  if((yr%4 == 0) && (yr%100!=0))
      {
System.out.print("This is a Leap Year");
      }
else if((yr%100 == 0) && (yr%400 == 0))
      {
System.out.print("This is a Leap Year");
      }
else if(yr%400 == 0)
      {
System.out.print("This is a Leap Year");
      }
else
      {
System.out.print("This is not a Leap Year");
      }
   }
}
```

6. **Program to reverse a number**

```
  importjava.util.Scanner;
  public class JavaProgram
  {
  public static void main(String args[])
     {
  intnum, rev=0, rem;
        Scanner scan = new Scanner(System.in);
  System.out.print("Enter a Number : ");
  num = scan.nextInt();
  while(num != 0)
        {
  rem = num%10;
  rev = rev*10 + rem;
```

```
num = num/10;
        }
System.out.print("Reverse = " +rev);
    }
}
```

7. **Program to check vowel or not**

```
importjava.util.Scanner;
public class JavaProgram
{
public static void main(String args[])
    {
charch;
        Scanner scan = new Scanner(System.in);
System.out.print("Enter an Alphabet : ");
ch = scan.next().charAt(0);
if(ch=='a' || ch=='A' || ch=='e' || ch=='E' ||
ch=='i' || ch=='I' || ch=='o' || ch=='O' ||
ch=='u' || ch=='U')
        {
System.out.print("This is a Vowel");
        }
else
        {
System.out.print("This is not a Vowel");
        }
    }
}
```

Type Conversion and Casting

The conversion of data from one data type to another is called Type Casting. When one type of data is assigned to another type of variable, an ***automatic type conversion (Implicit 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.

Example: byte b=38;

```
int a=b*2;
```

Casting incompatible types/Explicit type casting/narrowing conversion

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

(*target-type*) *value*

Example:

```
class casting {
```

```
{
        public static void main(String(arg[]){
        byte a=28;
        byte b;
        inti=257;
        double d=323.142;
        b=(byte) a;  //    b=28
        b=(byte) I; // b=1 since 257%128 is 1
        b=(double) d; // b=67 since 323.142%128 is 67  }
}
```

A different type of conversion will occur when a floating-point value is assigned to aninteger 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.

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 entireexpression is promoted to **float**. If any of the operands is **double**, the result is **double**.

ARRAYS

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

One-Dimensional Arrays

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

***typearray_var*[ ];……..(1)**

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

Eg.,int months[ ];

newis a special operator that allocates memory. The general form of **new** as it applies to one-dimensional arrays appearsas follows:

***array_var*= new *type*[*size*];……(2)**

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.

Eg., months=new 9nt[12];

The statements (1) and (2) can be combined into a single statement as follows

typearray_var[]=new type[size];

Eg.,int months[]=new int[12];

Example 1:// Program to read and print an array

```
importjava.util.*;
classArraydemo{
        public static void main(Stringarg[]){
                int n, i;
                Scanner s=new Scanner(System.in);
                System.out.println("Enter the size of the array");
                n=s.nextInt();
                int a[]=new int[n];
                System.out.println("Enter the array elements");
                for(i=0; i<n; i++)
                {  a[i]=s.nextInt(); }
                for(j=0;j<n; j++)
                { System.out.println(a[j]); }
    }
        }
```

Example 2: //Explicit array initialization

```
        //program to sort an array
        importjava.util.*;
        class Arraydemo1{
                public static void main(String arg[]){
                intnum[]={101,107,108,120,99,97};
                intlen=num.length;
                for(inti=0 ; i<len; i++)
                {
                        for(int j=0 ; j<len; j++)
                        {
                        if(num[i]>num[j])
                           {    int temp= num[i];
                                num[i]=num[j];
                                num[j]=temp;  }
                        }
                }
              }
              }
```

Example 3: Program that uses Java Array Length to determine the highest value in an array.

```
public class MainClass {
public static void main(String[] args) {
int[] testArray = { 100, 50, 20, 99, 45, 55,99 };
        intmaxValue = testArray[0];

intarrayLength = testArray.length;
```

```
for (inti = 1; i<= arrayLength - 1; i++)
        {
int value = testArray[i];
if (value >maxValue) {
maxValue = value;
        }
          }
System.out.println("The max value is: "+maxValue);
  }
}
```

Multidimensional Array

In Java, *multidimensional arrays* are actually arrays of arrays. To declare a multidimensional array variable, specify each additionalindex using another set of square brackets. For example, the following declares a twodimensionalarray variable called**twoD**.

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

//program to print a unit matrix

```
class MultyArray1{
public static void main(Stringarg[]){
        int mat[][]=new int[3][3];
        mat[0][0]=1;
        mat[1][1]=1;
        mat[2][2]=1;
for(inti=0 ; i<mat.length ;i++)
        { for(int j=0; j<mat.length; j++)
         {
        System.out.println(mat[i][j]);
          }
        }
     }
 }
```

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

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

/* **Program to print the matrix as below**

```
0
1 2
3 4 5
6 7 8 9
*/
        classTwoDAgain {

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

Alternative Array Declaration Syntax

The following one dimensional array 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];
char[]twoD2[]= new char[3][4];
```

Practice the following programs

1. Write a program to read two arrays and display the sum
2. Write a java program to read and print a matrix
3. Write a program to search an element in the array.
4. Write a java program to sort the array elements in descending order.
5. Write a java program to print the matrix as follows

   1

1 1
1 1 1
1 1 1 1

6. Write a java program to add two matrices and display the sum
7. Write a java and a program to compute the product of two matrices and display the result
8. Write a program to print all real solutions to the quadratic equation $ax^2+ bx +c = 0$. Read a,b& c values. Use the formula $(–b+sqrt(b^2-4ac))/2a$.
9. Write a program to store the contact numbers of the customers. Contact numbers include residence number, mobile number and office number (Hint: Arrays)

Arithmetic Operators

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

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division— allbehave as you would expect for all numeric types. The minus operator also has a unary formthat negates its single operand. When the division operator is applied to aninteger type, there will be no fractional component attached to the result.

Example:

```
// Demonstrate the basic arithmetic operators.
        classBasicMath {
        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;
```

```
        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);
        // arithmetic using doubles
        System.out.println("\nFloating Point Arithmetic");
        double da = 1 + 1;
        doubledb = da * 3;
        double dc = db / 4;
        doubledd = 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);
        }
  }
```

The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied tofloating-point types as well as integer types. The following example program demonstratesthe **%**:

Example 1:// Demonstrate the % operator.

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

When you run this program, you will get the following output:

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

Example 2: Write a java program to generate a random number between 0 to 5

// Use the random class present in the util package and also use abs() method of Math class to get a positive number

```
import java.util.*;
public class JavaExercises{
public static void main(String[] args){
int a;
Random rn=new Random();
a=Math.abs(rn.nextInt()%6);
System.out.println("The result: "+a); }
```

```
}
```

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

```
a = a + 4;
```

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

```
a += 4;
```

This version uses the **+=** *compound assignment operator*. Both statements perform the sameaction: they increase the value of **a** by 4.

Here is another example,

```
a = a % 2;which can be expressed asa %= 2;
```

Example:

```
// Demonstrate several assignment operators.
classOpEquals {
    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
```

Increment and Decrement

The ++ and the – – are Java‘s increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

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

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

```
x++;
```

Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in ***postfix***form, where theyfollow the operand as just shown, and ***prefix***form, where they precede the operand. In the prefix form, theoperand is

incremented or decremented before the value is obtained for use in the expression.In postfix form, the previous value is obtained for use in the expression, and then the operandis modified. For example:

x = 42;
y = ++x;

In this case, **y** is set to 43, because the increment occurs *before* **x** is assignedto **y**. Thus, the line **y = ++x;** is the equivalent of these two statements:

x = x + 1;
y = x;

However, when written like this,

x = 42;
y = x++;

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

y = x;
x = x + 1;

Example:

```
// Demonstrate ++.and --
        classIncDec {
                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 = 1
c = 2
d = 1

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators

The bitwise logical operators are **&**, |, ^, and ~. The following table shows the outcome ofeach operation. The bitwise operators areapplied to each individual bit within each operand.

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

The Bitwise NOT

Also called the *bitwise complement,* the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010

becomes

11010101

after the NOT operator is applied.

The Bitwise AND

The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

00101010 42
& 00001111 15
produces
00001010 10

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

00101010 42
| 00001111 15
produces
00101111 47

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^.

```
00101010      42
^ 00001111 15
produces
00100101       37
```

Using the Bitwise Logical Operators

```
// Demonstrate the bitwise logical operators.
classBitLogic {
        public static void main(String args[]) {
        String binary[] = {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
                           "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"};
        int a = 3; // 0 + 2 + 1 or 0011 in binary
        int b = 6; // 4 + 2 + 0 or 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;
        System.out.println("        a = " + binary[a]);
        System.out.println("        b = " + binary[b]);
        System.out.println("      a|b = " + binary[c]);
        System.out.println("      a&b = " + binary[d]);
        System.out.println("      a^b = " + binary[e]);
        System.out.println("~a&b|a&~b = " + binary[f]);
        System.out.println("       ~a = " + binary[g]);
        }
}
```

The Left Shift

The left shift operator, **<<**, shifts all of the bits in a value to the left a specified number of times.
It has this general form:

value*<<*num

Here, *num*specifies the number of positions to left-shift the value in *value*. That is, the **<<**moves all of the bits in the specified value to the left by the number of bit positions specifiedby *num*. For each shift left, the high-order bit is shifted out (and lost), and a zero is broughtin on the right. Shifting a value to the left (<<) results in multiplying the value by a power of two.

Example:

```
inti=10;  // In binary form 000001010
intresukt=i<<2;  // result=40 because after shifting the result is 000101000
```

The Right Shift

The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times. Shifting a value to the right (>>) is the same as dividing it by a power of two. Its general form is

value>>num

For each right left, the lower-order bit is shifted out (and lost), and a sign bit is broughtin on the left. Here, the sign of the value is preserved.

Example:

1. The following code fragment shifts the value 32 to the right by one position, resulting in**a** being set to 16:

```
int a = 32;  // 00100000 in binary form
a = a >>1; // a now contains 16 001000000 after one right shift becomes 00010000
```

2. The following code fragment shifts the value -8 to the right by one position, resulting in**a** being set to -4:

```
int b=-8;  //  11111000 in binary form
b=b>>1; // 11111000 becomes 11111100 after one right shift.
```

The Unsigned Right Shift

When we use right shift operator i.e. ">>" it keeps sign bit intact i.e. if original number is negative then it will remain negative even after right shift i.e. first or most significant bit never lost, doesn't matter how any times you shift. On the other hand unsigned right shift operator ">>>" doesn't preserve sign of original number and fills the new place with zero, that's why it's known as unsigned right shift operator or simply *rightshift with zero fill*.

The following code fragment demonstrates the **>>>**. Here, **a** is set to –1, which sets all32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros,ignoring normal sign extension. This sets **a** to 255.

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

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111      in binary as an int -1
>>>24
00000000 00000000 00000000 11111111     in binary as an int 255
```

Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraicoperators, which combines the assignment with the bitwise operation. For example, thefollowing two statements, which shift the value in **a** right by four bits, are equivalent:

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

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

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

Example:

```
classOpBitEquals {
        public static void main(String args[]) {
                int a = 1;
```

```
        int b = 2;
        int c = 3;
        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        }
}
```

The output of this program is shown here:

```
a = 3
b = 1
c = 6
```

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. The outcome of these operations is a **boolean**value. The relational operators are most frequently used in the expressions that control the **if**statement and the various loop statements.

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

Example:

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

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

Example Program: Write Java program to allow the user to input his/her age. Then the program will show if the person is eligible to vote. A person who is eligible to vote must be older than or equal to 18 years old.

```
importjava.util.*;
public class JavaExercises
{
public static void main(String[] args)
{
int age;
Scanner sc=new Scanner(System.in);
System.out.print("What is your age?");
age=sc.nextInt();
if(age>=18)
```

```
        System.out.println("You are eligible to vote.");
        else
        System.out.println("You are not eligible to vote.");
      }
    }
```

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

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

The logical Boolean operators, **&**, |, and ^, operate on **boolean**values in the same waythat 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

Example:

```
// Demonstrate the boolean logical operators.
classBoolLogic {
        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);  // prints true
        System.out.println("       b = " + b);  // prints false
        System.out.println("     a|b = " + c);  // prints true
        System.out.println("     a&b = " + d); // prints false
        System.out.println("     a^b = " + e);  // prints true
```

```
                System.out.println("!a&b|a&!b = " + f);  // prints true
                System.out.println("       !a = " + g);  // prints false
        }
        }
```

Short Circuit Logical Operators

Short circuit logical operators evaluate second expression only if this is needed. When short-circuit AND (&&) is used, if the first value is false, second value is not evaluated as the result is false irrespective of the second value. Similarly for short-circuit OR (||), if the first value is true, then second value is not evaluated as the result is true irrespective of the second value.. These short-circuit operators will be useful when we want to control the evaluation of right hand operand.

Example:

```
classShortCircuit
{
   publicstaticvoid main(Stringarg[])
   {
      int c =0, d =100, e =50;// LINE A
      if( c ==1&& e++<100)
      {
         d =150;
      }
      System.out.println("d = "+ d );
    if( c<1 || e++<100)
      {
         d =150;
      }
      System.out.println("d = "+ d );

   }
}
```

The Assignment Operator

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

var = expression;

Here, the type of *var*must be compatible with the type of *expression.*The assignment operator allows you to create a chain of assignments. For example, consider this fragment:

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

The ? Operator (Ternary Operator)

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-elsestatements. Java ternary operator is the only conditional operator that takes three operands.The **?**has this general form:

expression1* ?*expression2* :*expression3

Here, *expression1* can be any expression that evaluates to a **boolean**value. If *expression1* is**true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?**operation is that of the

expression evaluated. Both *expression2* and *expression3* are requiredto return the same type, which can't be **void**.

Example:

```
class operator    {
        publicstaticvoid main(String[]args){
                Double number =-5.5;
                String result;
                result=(number >0.0)?"positive":"not positive";
                System.out.println(number +" is "+ result);
                }
        }
```

Operator Precedence

Table below shows the order of precedence for Java operators, from highest to lowest. The first row shows items that you may not normally think of as operators: parentheses,square brackets, and the dot operator. Technically, these are called *separators*, but they actlike operators in an expression. Parentheses are used to alter the precedence of an operation.

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

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

a >> b + 3

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

a >> (b + 3)

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

(a >> b) + 3

Adding redundant but clarifyingparentheses to complex expressions can help prevent confusion later.

a | 4 + c >> b & 7 can be clearly written as

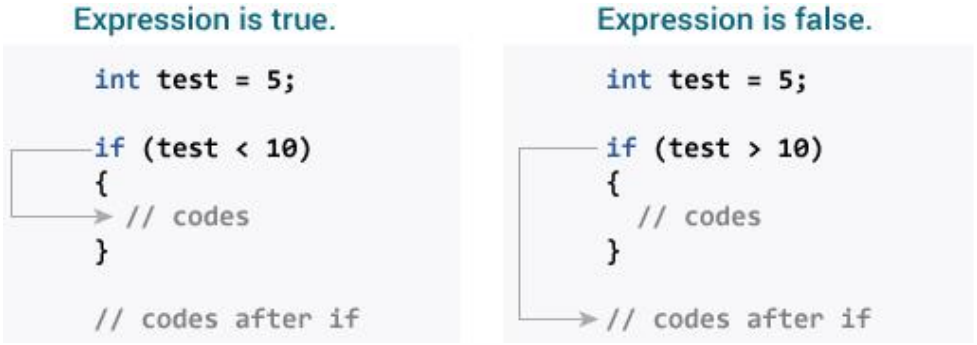
(a | (((4 + c) >> b) & 7))

Control Statements

If statement

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

How if statement works?



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

if (*condition*) *statement1*;
else *statement2*;

Here, each *statement* may be a single statement or a compound statement enclosed in curlybraces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else**clause is optional.The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise,*statement2* (if it exists) is executed. In no case will both statements be executed. For example,consider the following:

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

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

Nested ifs

A*nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very commonin programming. Consider the following example

```
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)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is notin the same block (even though it is the nearest **if** without an **else**).

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s isthe*if-else-if ladder*. The **if** statements are executed from the top down.

It looks like this:

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

```
// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String args[]) {
    int month = 4; // April
    String season;
    if(month == 12 || month == 1 || month == 2)
    season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
    season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
    season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
    season = "Autumn";
    else
    season = "Bogus Month";
    System.out.println("April is in the " + season + ".");
    }
```

Here is the output produced by the program:

```
April is in the Spring.
```

switch

The **switch** statement is Java‘s multiway branch statement. It provides an easy way to dispatchexecution to different parts of your code based on the value of an expression. It provides a better alternative than a large series of **if-else-if** statements. Here is the general formof a **switch** statement:

```
switch (expression) {
case value1:
.
.
.
// statement sequence
break;
case value2:
// statement sequence
```

```
        break;
        case valueN:
        // statement sequence
        break;
        default:
        // default statement sequence
        }
```

The *expression* must be of type **byte**, **short**, **int**, or **char.** The **switch** statement works like this: The value of the expression is compared with eachof the literal values in the **case** statements. If a match is found, the code sequence followingthat **case** statement is executed. If none of the constants matches the value of the expression,then the **default** statement is executed. The **break** statement is used inside the **switch** to terminate a statement sequence.

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

Output

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

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a*nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the**case** constants in the inner **switch** and those in the outer **switch**.

```
        switch(count) {
```

```
case 1:
        switch(target) { // nested switch
        case 0:
                System.out.println("target is zero");
        break;
        case 1: // no conflicts with outer switch
                System.out.println("target is one");
        break;
}
break;
case 2: // ...
```

Iteration Statements

Java's iteration statements 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 Java's most fundamental loop statement. It repeats a statement or blockwhile its controlling expression is true. The general form:

```
while(condition) {

// body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as longas the conditional expression is true. When *condition* becomes false, control passes to thenext line of code immediately following the loop. Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with

Example 1: // Demonstrate the while loop.

```
class While {
public static void main(String args[]) {
        int n = 4;
        while(n > 0)
        {
        System.out.println("tick " + n);
        n--;
        }
    }}
```

Output

```
tick 4
tick 3
tick 2
tick 1
```

Example 2: // **Program to print Fibonacci series using while loop.**

```
public class FibonacciWhileExample {
public static void main(String[] args){
        int maxNumber = 10, previousNumber = 0, nextNumber = 1;
        System.out.print("Fibonacci Series of "+maxNumber+" numbers:");
        int i=1;
        while(i <= maxNumber)
        {
         System.out.print(previousNumber+" ");
         int sum = previousNumber + nextNumber;
         previousNumber = nextNumber;
         nextNumber = sum;
         i++;
        }
        }
}
```

do-while

In case of while loop conditional expression controlling a **while** loop is initially false,then the body of the loop will not be executed at all. However, there are times when we would like to test the terminationexpression at the end of the loop rather than at the beginning. The **do-while** loop always executes its body at leastonce, because its conditional expression is at the bottom of the loop. Its general form is

```
do {

// body of loop

} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluatesthe conditional expression. If this expression is true, the loop will repeat. Otherwise, the loopterminates.

Example: // 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);
}
 }
```

Output

```
tick 4
tick 3
tick 2
tick 1
```

tick 0

for loop

The simplest form of the **for** loop is shown here

```
for(initialization; testexpression; update)
{
// statements
}
```

How for loop works?

1. The initialization expression is executed only once.
2. Then, the test expression is evaluated. Here, test expression is a boolean expression.
3. If the test expression is evaluated to true,
   - Codes inside the body of for loop is executed.
   - Then the update expression is executed.
   - Again, the test expression is evaluated.
   - If the test expression is true, codes inside the body of for loop is executed and update expression is executed.
   - This process goes on until the test expression is evaluated to false.
4. If the test expression is evaluated to false, for loop terminates.

Example 1:// Program to print a line 10 times

```
class Loop {
        publicstaticvoid main(String[] args){
          for(int i =1; i <=10;++i){
          System.out.println("Line "+ i);
        }
        }
}
```

Example 2: //Program to check prime number using for loop

```
import java.utin.Scanner;
publicclassPrime{
publicstaticvoid main(String[] args){
Scanner s=new Scanner(System.in);
int num = s.nextInt();
boolean flag =false;
for(int i =2; i <= num/2;++i){
        if(num % i ==0)
        {flag =true;
break; }
 }
if(!flag)
System.out.println(num +" is a prime number.")

else
                System.out.println(num +" is a prime number.")
```

```
        }
 }
```

Example 3 : Program to find factorial of a number based on user input using for loop

```
Import java.util.Scanner;
publicclassJavaExample{
        publicstaticvoid main(String[] args){
                Scanner s=new Scanner(System.in);
                int number =s.nextInt();
                 long fact =1;
                 for(int i =1; i <= number; i++)
                {
                fact = fact * i;
                }
                System.out.println("Factorial of "+number+" is: "+fact);
        }
        }
```

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop.

```
//Using Comma in Initialization
        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);
        }
        }
        }
```

In this example, the initialization portion sets the values of both **a** and **b**. The two commaseparated statements in the iteration portion areexecuted each time the loop repeats.

Theprogramgenerates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

The For-Each version of the Loop

The for-each style of **for** is also referred to as the *enhanced* **for** loop.The general form of the for-each version of the **for** is

for(data*type itr-var : collection*) *statement-block*

Here, data*type* specifies the type and *itr-var* specifies the name of an *iteration variable* that willreceive the elements from a collection, one at a time, from beginning to end. The collectionbeing cycled through is specified by *collection*. There are various types of collections thatcan be used with the **for.**

To understand the motivation behind a for-each style loop, consider the type of **for** loopthat it is designed to replace. The following fragment uses a traditional **for** loop to computethe sum of the values in an array:

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

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus,the entire array is read in strictly sequential order. This is accomplished by manuallyindexing the **nums** array by **i**, the loop control variable.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the needto establish a loop counter, specify a starting and ending value, and manually index thearray. Instead, it automatically cycles through the entire array, obtaining one element ata time, in sequence, from beginning to end. For example, here is the preceding fragmentrewritten using a for-each version of the for loop:

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

Example 1: // **Program to find the sum of the array elements using for each loop**

```
class ForEach {
        public static void main(String args[]) {
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                int sum = 0;
                // use for-each style for to display and sum the values
                for(int x : nums) {
                        System.out.println("Value is: " + x);
                        sum += x; }
                System.out.println("Summation: " + sum);
}
        }
```

Example 2: // **Program to search an element of the array using for each loop**

```
import java.util.Scanner;
class Search {
    public static void main(String args[]) {
int n, Key;
Scanner s=new Scanner(System.in);
System.out.println("Enter the size of the array");
n=s.nextInt();
int a[] = new int[n];
System.out.println("Enter the elements of the array");
for(int i=0;i<n;i++)
    a[i]=s.nextInt();
System.out.println("Enter Key to be searched");
```

```
        Key=s.nextInt();
        int found=0;
        //use for-each style for to search nums for val
        for(int x : a) {
        if(x == Key)
         {
        found = 1;
        break;
            }
      }
        if(found==1)
        System.out.println("Value found");
          else
        System.out.println("Value not found!");
                  }
                }
```

Jump Statements

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a ~~"~~civilized" form of goto. The last two uses are explained here.

Using break to Exit a Loop

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

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

Using break as a form of goto (Labeled break)

The general form of the labeled **break** statement is

break *label*;

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-aloneblock of code but it can also be a block that is the target of another statement. When this form of**break** executes, control is transferred out of the named block. The labeled block must enclosethe **break** statement, but it

does not need to be the immediately enclosing block. This means,for example, that you can use a labeled **break** statement to exit from a set of nested blocks.But you cannot use **break** to transfer control out of a block that does not enclose the **break**statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement.

Example 1:// Using break as a civilized form of goto.

```
class Break {
        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.");
                }
          }
        }
```

Example 2:// Using break to exit from nested loops

```
class BreakLoop4 {
        public static void main(String args[]) {
                outer: for(int i=0; i<3; i++) {
                for(int j=0; j<100; j++) {
                if(j == 10) break outer; // exit both loops
                System.out.print(j + " ");
                }
        System.out.println("This will not print");
}
System.out.println("Loops complete.");
}
}
```

continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continuerunning the loop but stop processing the remainder of the code in its body for this particulariteration. This is, in effect, a goto just past the body of the loop, to the loop‘s end. The **continue**statement performs such an action.

```
// Demonstrate continue statement to print ODD numbers.
class Continue {
        public static void main(String args[]) {
        for(int i=0; i<10; i++) {
```

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

Labeled continue Statement

The labeled *continue* statement is similar to the unlabelled *continue* statement in the sense that both resume the iteration. The difference with the labeled continue statement is that it resumes operation from the target label defined in the code. As soon as the labeled continue is encountered, it skips the remaining statements from the statement's body and any number of enclosing loops and jumps to the nest iteration of the enclosing labeled loop statements. Here is an example to illustrate the labeled continue statement.

```
public class LabeledContinueDemo {
   public static void main(String[] args) {
      start: for (int i = 0; i < 5; i++) {
  System.out.println();
for (int j = 0; j < 10; j++) {
System.out.print("#");
  if (j >= i)
 continue start;
   }
        System.out.println("This will never" + " be printed");
     }
   }
}
```

return

The **return** statement is used to explicitly return froma method. That is, it causes program control to transfer back to the caller of the method.

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