ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

ATME College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(ACADEMIC YEAR 2025-2026)

NOTES

Subject: OBJECT ORIENTED PROGRAMMING WITH JAVA

Subject Code: BCS306A

Semester: III

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

Department of Computer Science & Engineering

Vision of the Department

To develop highly talented individuals in Computer Science and Engineering to deal with real world challenges in industry, education, research and society.

Mission of the Department

- To inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities in the young minds & to provide a teaching environment that emphasizes depth, originality and critical thinking.
- Motivate students to put their thoughts and ideas adoptable by industry or to pursue higher studies leading to research.

Program Educational Objectives (PEO'S):

1. Empower students with a strong basis in the mathematical, scientific and engineering fundamentals to solve computational problems and to prepare them for employment, higher learning and R&D.
2. Gain technical knowledge, skills and awareness of current technologies of computer science engineering and to develop an ability to design and provide novel engineering solutions for software/hardware problems through entrepreneurial skills.
3. Exposure to emerging technologies and work in teams on interdisciplinary projects with effective communication skills and leadership qualities.
4. Ability to function ethically and responsibly in a rapidly changing environment by applying innovative ideas in the latest technology, to become effective professionals in Computer Science to bear a life-long career in related areas.

Program Specific Outcomes (PSOs)

1. Ability to apply skills in the field of algorithms, database design, web design, cloud computing and data analytics.
2. Apply knowledge in the field of computer networks for building network and internet-based applications.

MODULE 1

Syllabus:

An Overview of Java: Object-Oriented Programming, A First Simple Program, A Second ShortProgram, Two Control Statements, Using Blocks of Code, Lexical Issues, The Java Class Libraries.

Data Types, Variables, and Arrays: Java Is a Strongly Typed Language, The Primitive Types, Integers, Floating- Point Types, Characters, Booleans, A Closer Look at Literals, Variables, Type Conversion and Casting, Automatic Type Promotion in Expressions, Arrays

Operators: Arithmetic Operators, Relational Operators, Boolean Logical Operators, The Assignment Operator, The ? Operator, Operator Precedence, Using Parentheses. Control Statements: Java's Selection Statements (if, The Traditional switch), Iteration Statements (while, do-while, for, The For-Each Version of the for Loop, Local Variable Type Inference in a for Loop, Nested Loops), Jump Statements (Using break, Using continue, return).

An Overview of Java

The key features of Java are **security** and **portability (platform-independent nature)**. When we download any application from the internet, there is a chance that the downloaded code contain virus. But, downloading the Java code assures security. Java program can run on any type of system connected to internet and thus provides portability.

The Platform independent nature can be interpreted by two things:

- **Operating System Independent:** Independent of the operating system on which your source code is being run.
- **Hardware Independent:** Doesn't depend upon the hardware on which your java code is run upon i.e. it can run on any hardware configuration.

These two points make it a platform independent language. Hence, the users do not have to change the syntax of the program according to the Operating System and do not have to compile the program again and again on different Operating Systems. The meaning of this point can be understood as you read further.

C and C++ are platform dependent languages as the file which compiler of C,C++ forms is a **.exe**(executable) file which is operating system dependent. The C/C++ program is controlled by the operating system whereas, the execution of a Java program is controlled by **JVM (Java Virtual Machine).**

The JVM is the Java run-time system and is the main component of making the java a platform independent language. For building and running a java application we need JDK(Java Development Kit) which comes bundled with **Java runtime environment(JRE)** and JVM. With the help of JDK the user compiles and runs his java program. As the compilation of java program starts the **Java Bytecode** is created i.e. a **.class** file is created by JRE. Bytecode is a highly optimized set of instructions designed to be executed by JVM. Now the JVM comes into play, which is made to read and execute this bytecode. The JVM is linked with operating system and runs the bytecode to execute the code depending upon operating system. Therefore, a user can take this class file(Bytecode file) formed to any operating systemwhich is having a JVM installed and can run his programeasily without even touching the syntax of a program and without actually having the source code. The .class file which consists of bytecode is not user-understandable and can be interpreted by JVM only to build it into the machine code.

Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs. Java also has the standard data size irrespective of operating system or the processor. These features make the java as a portable (platform-independent) language.

Usually, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. To improve the performance, Java provides a **Just-in-time (JIT)** compiler for bytecode. JIT compilers alter the role of the JVM a little by directly compiling Java bytecode into native platform code, thereby relieving the JVM of its need to manually call underlying native system services. When JIT compiler is installed, instead of the JVM calling the underlying native operating system, it calls the JIT compiler. The JIT compiler in turn generates native code that can be passed on to the native operating system for execution. This makes the java program torun faster than expected.

Moreover, when a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted.

Object-Oriented Programming

Java is purely object oriented programming (OOP) language. Here, we will discuss the basics of OOPs concepts.

Two Paradigms

Every program consists of two elements viz. code and data. A program is constructed based on two paradigms: a program written around **what is happening** (known as **process-oriented model)** and a program written around **who is being affected** (known as **object-oriented model).** In process oriented model, the program is written as a series of linear (sequential) steps and it is thought of as **code acting on data.** Since this model fails to focus on real-world entities, it will create certain problems as the program grows larger.

The object-oriented model focuses on real-world data. Here, the program is organized as data and a set of well-defined interfaces to that data. Hence, it can be thought of as **data controlling access to code.** This approach helps to achieve several organizational benefits.

Abstraction

Abstraction can be thought of as **hiding the implementation details from the end-user**. A powerful way to manage abstraction is through the use of hierarchical classifications. This allows us to layer the semantics of complex systems, breaking them into more manageable pieces. For example, we consider a car as a vehicle and can be thought of as a single object. But, from inside, car is a collection of several subsystems viz. steering, brakes, sound system, engine etc. Again, each of these subsystems is a collection of individual parts (Ex. Sound system is a combination of a radio and CD/tape player). As an owner of the car, we manage it as an individual entity by achieving hierarchical abstractions.

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

OOPs Principles: Encapsulation, Inheritance and Polymorphism are the basic principles of any object oriented programming language.

Encapsulation is the mechanism to bind the data and code working on that data into a single entity. It provides the security for the data by avoiding outside manipulations. In Java, encapsulation is achieved using **classes.** A **class** is a collection of data and code. An **object** is an **instance** of a class. That is, several objects share a common structure (data) and behavior (code) defined by that class. A class is a logical entity (or prototype) and an object is a physical entity. The elements inside the class are known as**members**. Specifically, the data or variables inside the class are called as **member variables** or**instance variables** or **data members.** The code that operates on these data is referred to as **member methods** or **methods** (In C++, we term this as member function). The method operating on data will define the behavior and interface of a class.

Another purpose of the class is to hide the information from outside manipulation. Class uses **public** and **private** interfaces. The members declared as private can only be accessed by the members of that class, whereas, the public members can be accessed from outside the class.

Inheritance allows us to have code re-usability. It is a process by which one object can acquire the properties of another object. It supports the concept of hierarchical classification. For example, consider alarge group of animals having few of the abstract attributes like size, intelligence, skeletal structure etc. and having behavioral aspects like eating, breathing etc. Mammals have all the properties of Animals and also have their own specific features like type of teeth, mammary glands etc. that make them different from Reptiles. Similarly, Cats and Dogs have all the characteristics of mammals, yet with few features which are unique for themselves. Though Doberman, German-shepherd, Labrador etc. have the features of Dog class, they have their own unique individuality. This concept can be depicted using following figure.

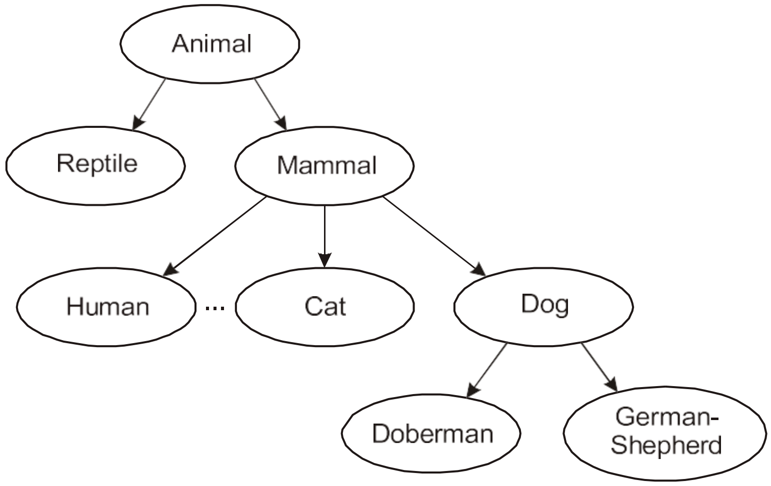


Figure 1.1 Example of Inheritance

If we apply the above concept for programming, it can be easily understood that a code written is reusable. Thus, in this mechanism, it is possible for one object to be a specific instance of a more general case. Usinginheritance, an object need only define those qualities that make it a unique object within its class. It can inherit its general attributes from its parent. Hence, through inheritance, we can achieve generalization- specialization concept. The top-most parent (or base class or **super class**) class is the generalized class and the bottom-most child (or derived class or **subclass**) class is a more specialized class with specific characteristics.

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

Polymorphism can be thought of as one interface, multiple methods. It is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature

of the situation. Consider an example of performing stack operation on three different types of data viz. integer, floating-point and characters. In a non-object oriented programming, we write functions with different names for push and pop operations though the logic is same for all the data types. But in Java, the same function names can be used with data types of the parameters being different.

A First Simple Program

Here, we will discuss the working of a Java program by taking an example –

Program 1.1 Illustration of First Java Program

```
class Prg1
{
        public static void main(String args[ ])
        {
                System.out.println(“Hello  World!!!”);
        }

}
```

Save this program as **Prg1.java**. A java program source code is a text file containing one or more class definitions is called as **compilation unit** and the extension of this file name should be **.java.**

To compile above program, use the following statement in the command prompt –

javac Prg1.java

(**Note:** You have to store the file Prg1.java in the same location as that of javac compiler or you should set the Environment PATH variable suitably.)

Now, the javac compiler creates a file **Prg1.class** containing bytecode version of the program, which can be understandable by JVM. To run the program, we have to use Java application launcher called **java**. That is, use the command –

java Prg1

The output of the program will now be displayed as –

Hello World!!!

Note: When java source code is compiled, each class in that file will be put into separate output file having the same name as of the respective class and with the extension of **.class**. To run a java code, we need a class file containing main() function (Though, we can write java program without main(), for the time-being you assume that we need a main() function!!!). Hence, it is a tradition to give the name of thejava source code file as the name of the class containing main() function.

Let us have closer look at the terminologies used in the above program now –

class is the keyword to declare a class.
Prg1 is the name of the class. You can use any valid identifier for a class name.
main() is name of the method from which the program execution starts.
public is a keyword indicating the access specifier of the method. The public members can be accessed from outside the class in which they have been declared. The main() function must be declared as public as it needs to be called from outside the class.
static The keyword static allows main() to be called without having to instantiate a particular instance of the class. This is necessary since main() is called by the Java Virtual Machine before any objects are made.
void indicates that main() method is not returning anything.
String args[ ] The main() method takes an arrayof String objects as a command-line argument.
System is a predefined class (present in java.lang package) which gives access to the system. It contains pre-defined methods and fields, which provides facilities like standard input, output, etc.
out is a static final (means not inheritable) field (ie, variable)in System class which is of the type PrintStream (a built-in class, contains methods to print the different data values). Static fields and methods must be accessed byusing the class name, so we need to use System.out.
println is a public method in PrintStream class to print the data values. After printing the data, the cursor will be pushed to the next line (or we can say that, the data is followed bya new line).

A Second Short Program
Here, we will discuss a program having variables. Variable is a named memory location which may be assigned a value in the program. A variable can be declared in the java program as –

type var_name;

Here, **type** is any built-in or user-defined data type (We will discuss various data types later in detail).
var_name is any valid name given to the variable. Consider the following example –

Program 1.2 Illustrating usage of variables

```
class Prg2
{
        public static void main(String args[])
```

```
        {
                int n;
                n=25;
                System.out.println("The value of n is: " + n);n= n*3;
                System.out.print("The current value of n is: " );
                System.out.println(n);
        }
}
```

The output will be –

```
The value of n is: 25
The current value of n is: 75
```

In the above program, we have declared an integer variable **n** and then assigned a value to it. Now, observe the statement,

```
System.out.println("The value of n is: " + n);
```

Here, we are trying to print a string value "The value of n is:" and also value of an integer n together. For this, we use + symbol. Truly speaking, the value of n is internally converted into string type and then concatenated with the string "The value of n is:". We can use + symbol as many times as we want to print several values.

The above program uses one more method System.out.print() which will keep the cursor on the same line after displaying the output. That is, no new line is not included in it.

Two Control Statements

Though control structures are discussed in Module 2, here we will glance two important structures which are needed for some of the examples in the current Module.

if Statement: When a block of code has to be executed based on the value of a condition, if statement is used. Syntax would be –

```
if(condition)
{
	//do something
}
```

Here, conditionhas to be Boolean statement (unlike C/C++, where it could be integer type). If the condition is true, the statement block will be executed, otherwise not.

To have a Boolean result from an expression, we may use relational operators like <, >, <=, == etc.

Program 1.3 Illustration of if statement

```
class IfSample
{
	public static void main(String args[])
	{
		int x, y;
```

```
                    x = 10;
                    y = 20;

                    if(x < y)
                              System.out.println("x is less than y");

                    x = x * 2;
                    if(x == y)
                              System.out.println("x now equal to y");

                    x = x * 2;

                    if(x > y)
                              System.out.println("x now greater than y");

                    if(x == y)
                              System.out.println("you won't see this");
          }
}
```

The output would be –

```
x is less than yx now
equal to y
x now greater than y
```

for Statement: Whenever a set of statements has to be executed multiple times, we will use for statement. The syntax would be –

```
for(initialization; condition; updation)
{
          //statement block
}
```

Here, initialization	contains declaring and/or initialization of one or more variables, that happens only once
condition	Must be some Boolean expression, that will be checked immediately after initialization and each time when there is an updation of variables
updation	Contains increment/decrement of variables, that will be executed after executing statement block

Program 1.4 Illustration of for statement

```
class ForTest
{
    public static void main(String args[])
    {
        int x;

        for(x = 0; x<5; x = x+1) System.out.println("This is x: " + x);
    }
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2

This is x: 3
This is x: 4
```

Using Blocks of Code
Java allows two or more statements to be grouped into blocks of code, also called code blocks. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.

For example, a block can be a target for Java's **if** and **for** statements. Consider this **if** statement:

```
if(x < y)
{ // block begins
        x = y;
        y = 0;
} // block ends here
```

The main reason for the existence of blocks of code is to create logically inseparable units of code.

Lexical Issues
Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. We will discuss the significance of each of these here.

Whitespace : In Java, whitespace is a space, tab or newline. Usually, a space is used to separate tokens; tab and newline are used for indentation.

Identifiers : Identifiers are used for class names, method names, and variable names. An identifier may be any sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number. As Java is case-sensitive, **Avg** is a different identifier than **avg**.
Examples of valid identifiers: Avg, sum1, $x, sum_sq etc.
Examples of invalid identifiers: 2sum, sum-sq, x/y etc.

Literals : A constant value in Java is created by using a literal representation of it. For example, 25 (an integer literal), 4.5 (a floating point value), 'p' (a character constant, "Hello World" (a string value).

Comments : There are three types of comments defined by Java. Two of these are well-know viz. **single-line comment** (starting with //), **multiline comment** (enclosed within /* and */). The third type of comment viz. **documentation comment** is used to produce an HTML file that documents your program. The documentation comment begins with a **/**** and ends with a ***/**.

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

Symbol	Name	Purpose
( )	Parentheses	Used to provide parameter list in method definition and to call methods. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to initialize arrays, to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types, to dereference 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 sub-packages and classes. Also used to separate a variable or method from a reference variable.

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

Abstract	assert	boolean	break	Byte	case	catch	Char	Class	Const
Continue	default	goto	do	double	else	enum	Extends	Final	Finally
Float	For	if	implements	import	instanceof	int	interface	Long	Native
New	package	private	protected	public	return	short	Static	Strictfp	Super
Switch	synchronized	this	throw	throws	transient	try	Void	While	

The keywords **const** and **goto** are reserved but are rarely used. In addition to the keywords, Java reserves the following: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes and so on.

The Java Class Libraries

The sample programs discussed in previous sections make use of two of Java's built-in methods: **println()** and **print( )**. As mentioned, these methods are members of the **System** class, which is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for windowed output. Thus, Java is a combination of the Java language itself, plus its standard classes. The class libraries provide much of the functionality that comes with Java. The standard library classes and methods are described in detail in forthcoming chapters.

Java is a Strongly Typed Language

A strongly-typed programming language is one in which each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) 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. Certain operations may be allowable only with certain data types.

In other words, every variable has a type, every expression has a type, and every type is strictly defined. And, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class. These features of Java make it a strongly typed language.

The Primitive Types

Java defines eight **primitive** (or **simple**) data types viz.

- **byte, short, int, long** : belonging to Integers group involving whole-valued signed numbers.
- **char** : belonging to Character group representing symbols in character set like alphabets, digits, special characters etc.
- **float, double** : belonging to Floating-point group involving numbers with fractional part.
- **boolean** : belonging to Boolean group, a special way to represent true/false values.

These types can be used as primitive types, derived types (arrays) and as member of user-defined types (classes). All these types have specific range of values irrespective of the platform in which the program being run. In C and C++ the size of integer may vary (2 bytes or 4 bytes) based on the platform. Becauseof platform-independent nature of Java, such variation in size of data types is not found in Java, and thus making a Java program to perform better.

Integers

Java defines four integer types viz. **byte, short, int** and **long.** All these are signed numbers and Java does not support unsigned numbers. The width of an integer type should not be thought of as the amountof storage it consumes, but rather as the behaviour it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width (in bits)	Range
long	64	-2^{63} to $+2^{63}-1$
int	32	-2^{31} to $+2^{31}-1$
short	16	-2^{15} to $+2^{15}-1$ (-32768 to +32767)
byte	8	-2^{7} to $+2^{7}-1$ (-128 to +127)

byte : This is the smallest integer type. Variables of type **byte** are especially useful when you are working with a stream of data from a network or file. They are also useful when you are working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the **byte** keyword. For example,

```
byte b, c;
```

short : It is probably the least-used Java type. Here are some examples of **short** variable declarations:short

```
s;
short t;
```

int : The most commonly used integer type is **int**. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Although you might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case.

The reason is that when **byte** and **short** values are used in an expression they are promoted to **int** when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, **int** is often the best choice when an integer is needed.

long : It is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed.

Program 1.5: Program to illustrate need for long data type

```
class Light
{
    public static void main(String args[ ])
    {
        int lightspeed;
        long days, seconds, 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.");
    }
}
```

The output will be –

In 1000 days light will travel about 16070400000000 miles.

Floating –Point Types

Floating-point (or real) numbers are used when evaluating expressions that require fractional precision. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width (in bits)	Range
double	64	4.9e–324 to 1.8e+308

float	32	1.4e–045 to 3.4e+038

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, but will become imprecise when the values are either very large or very small. Variables oftype **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing currencies, temperature etc.

Here are some example **float** variable declarations:

float hightemp, lowtemp;

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

Program 1.6 Finding area of a cirlce

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

The output would be –

Area of circle is 366.436224

Characters

In Java, **char** is the data type used to store characters. In C or C++, **char** is of 8 bits, whereas in Java it requires 16 bits. Java uses Unicode to represent characters. **Unicode** is a computing industry standard for the consistent encoding, representation and handling of text expressed in many languages of the world.
The range of a **char** is 0 to 65,536. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO- Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Though it seems to be wastage of memory,as the languages like English, German etc. can accommodate their character set in 8 bits, for a global usagepoint of view, 16-bits are necessary.

Though, **char** is designed to store Unicode characters, we can perform arithmetic operations on them. For example, we can add two characters, increment/decrement character variable etc. Consider the following example for the demonstration of characters.

Program 1.7 Demonstration of char data type

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

                ch1++;          //increment in ASCII (even Unicode) value
                System.out.println("ch1 now contains "+ch1);

                --ch2;                  //decrement in ASCII (even Unicode) value
                System.out.println("ch2 now contains "+ch2);

                /*      ch1=35;
                        ch2=30;
                        char ch3;

                        ch3=ch1+ch2;        //Error
                  */
                ch2='6'+'A';            //valid
                System.out.println("ch2 now contains "+ch2);
        }
}
```

The output would be –

```
ch1 and ch2: X Y
ch1 now contains Y
ch2 now contains X
ch2 now contains w
```

Booleans

For storing logical values (**true** and **false**), Java provides this primitive data type. Boolean is the output of any expression involving relational operators. For control structures (like if, for, while etc.) we need to give boolean type. In Java, the output of relational operators will be true or false. Consider the following program as an illustration.

Program 1.8 Demonstration of Boolean data type

```
class BoolDemo
{
```

```
        public static void main(String args[])
        {
                boolean b = false;

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

                if(b)
                        System.out.println("True block");
                b = false;
                if(b)
                  System.out.println("False Block ");
                b=(3<5);
                System.out.println("3<5 is " +b);
        }
}
```

The output would be –

```
b  is false
b is true
True block
3<5 is true
```

NOTE: Size of a Boolean data type is JVM dependent. But, when Boolean variable appears in an expression, Java uses 32-bit space (as int) for Boolean to evaluate expression.

A Closer Look at Literals

A literal is the source code representation of a fixed value. In other words, by literal we mean any number, text, or other information that represents a value. Literals are represented directly in our code without requiring computation. Here we will discuss Java literals in detail.

Integer Literals

Integers are the most commonly used type in the typical program. Any whole number value is an integer literal. For example, 1, 25, 33 etc. These are all decimal values, having a base 10. With integer literals we can use octal (base 8) and hexadecimal (base 16) also. Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, a value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range. Hexadecimal constants denoted 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. It is possible to assign an integer literal to other integer types like byte or long. When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase **L** to the literal. For example, 0x7fffffffffffffffL or 9223372036854775807L is the largest **long**. An integer can also be assigned to a **char** as long as it is within range.

Floating-Point Literals

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

Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an F or f to the constant. You can also explicitly specify a **double** literal by appending a D or d. Doing so is, of course, redundant. The default **double** type consumes 64 bits of storage, while the less-accurate **float** type requires only 32 bits.

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. In Java, they can only be assigned to variables declared as **boolean**, or used in expressions with Boolean operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as '\'' for the single-quote character itself and **'\n'** for the new-line character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, '\141' is the letter 'a'. For hexadecimal, you enter a backslash-u (**\u**), then exactly four hexadecimal digits. Following table shows the character escape sequences.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Back slash
\r	Carriage return (Enter key)
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Back space

String Literals

String literals are a sequence of characters enclosed within a pair of double quotes. Examples of string literals are

"Hello World"
"two\nlines"
"\"This is in quotes\""

Java strings must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages. In Java, strings are actually objects and are discussed later in detail.

Variables

The variable is the basic unit of storage. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

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

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

The type is any of primitive data type or class or interface. The identifier is the name of the variable. We can initialize the variable at the time of variable declaration. To declare more than one variable of the specified type, use a comma-separated list. Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b=5, c; byte z =
22; double pi = 3.1416;
char x = '$';
```

Dynamic Initialization
Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example,

```
int a=5, b=4;
int c=a*2+b;        //variable declaration and dynamic initialization
```

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

The Scope and Lifetime of Variables
A variable in Java can be declared within a block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope which determines the accessibility of variables and/or objects defined within it. It also determines the lifetime of those objects.

Java has two scopes viz. **class level scope** and **method (or function) level scope**. Class level scope is discussed later and we will discuss method scope here.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

Program 1.9 Demonstration of scope of variables

```
class Scope
{
        public static void main(String args[])
        {
                int x=10, i;                    // x and i are local to main()

                if(x == 10)
                {
                        int  y= 20;             // y is local to this block
                        System.out.println("x and y: " + x + " " + y);
                        x = y * 2;
                }

                   // y = 100;               //y cannot be accessed here
                System.out.println("x is " + x);
                System.out.println("y is " +y);

                for(i=0;i<3;i++)
                {
                        int a=3;                // a is local to this block
                        System.out.println("a  is " + a);
                        a++;
                }
        }
}
```

The output would be –

```
x and y: 10 20
x is 40
a is 3
a is 3
```

a is 3

Note that, variable **a** is declared within the scope of **for** loop. Hence, each time the loop gets executed, variable **a** is created newly and there is no effect of a++ for next iteration.

Type Conversion and Casting

It is quite common in a program to assign value of one type to a variable of another type. If two types are compatible, Java performs **implicit type conversion**. For example, int to long is always possible. But, whenever the types at two sides of an assignment operator are not compatible, then Java will not do the conversion implicitly. For that, we need to go for **explicit type conversion** or **type 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 ifyou 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, we must use a cast. A cast is simply an explicit typeconversion. It has this general form:

(target-type) value

Here, target-type specifies the desired type to convert the specified value to. For example,

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

When a floating-point value is assigned to an integer type, the fractional component is lost. And such conversion is called as **truncation** (**narrowing**)**.** 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. Following program illustrates various situations of explicit casting.

Program 1.10 Illustration of type conversion

```
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);
        }
}
```

The output would be –

```
Conversion of int to byte:      i = 257          b = 1
Conversion of double to int:   d = 323.142     i = 323
Conversion of double to byte: d = 323.142     b = 67
```

Here, 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 Expression

Apart from assignments, type conversion may happen in expressions also. In an arithmetic expression involving more than one operator, some intermediate operation may exceed the size of either of the operands. For example,

```
byte x=25, y=80, z=50;
int p= x*y/z ;
```

Here, the result of operation x*y is 4000 and it exceeds the range of both the operands i.e. byte (-128 to +127). In such a situation, Java promotes byte, short and char operands to int That is, the operation x*y is performed using int but not byte and hence, the result 4000 is valid.

On the other hand, the automatic type conversions may cause error. For example,

```
byte x=10;
byte y= x *3;        //causes error!!!
```

Here, the result of x *3 is 30, and is well within the range of byte. But, for performing this operation, the operands are automatically converted to byte and the value 30 is treated as of int type. Thus, assigningan int to byte is not possible, which generates an error. To avoid such problems, we should use type casting. That is,

```
byte x=10;
byte y=(byte) (x *3);        //results 30
```

Type Promotion Rules

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

- All **byte**, **short**, and **char** values are promoted to **int.**
- 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**.

Program 1.11 Demonstration of type promotions

```
class TypePromo
{
    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("result = " + result);
    }
}
```

The output would be –

```
result = 626.7784146484375
```

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 sub-expression, **f * b**, **b** is promoted to a **float** and the result of the sub-expression is **float**. Next, in the sub-expression **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 sub-expression 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 resultof the expression.

Arrays
Array is a collection of related items of same data type. Many items of an array share common name and are accessed using index. Array can be one dimensional or multi-dimensional.

One Dimensional Arrays
It is a list of related items. To create 1-d array, it should be declared as –

type arr_name[];

Here, **type** determines the data type of elements of **arr_name**. In Java, the above declaration will not allocate any memory. That is, there is no physical existence for the array now. To allocate memory, we should use **new** operator as follows:

arr_name=new type[size];

Here, **size** indicates number of elements in an array. The **new** keyword is used because, in Java array requires dynamic memory allocation. The above two statements can be merged as –

type arr_name[]=new type[size];

For example, following statement create an array of 10 integers –

int arr[ ]=new int[10];

Array index starts with 0 and we can assign values to arrayelements as –

arr[0]=25; arr[1]=32; and so
on.

Arrays can be initialized at the time of declaration. 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 –

int arr[ ] ={1, 2, 3, 4};

The above statement creates an integer arrayof 4 elements.

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 correctrange. If you try to access elements outside the range of the array (negative numbers or numbers greaterthan the length of the array), you will get a run-time error.

Multidimensional Arrays
Multidimensional arrays are arrays of arrays. Here, we will discuss two dimensional arrays in Java. The declaration of 2-d array is as follows –

```
type arr_name[][]=new type[row_size][col_size];
```

here, **row_size** and **col_size** indicates number of rows and columns of 2-d arrays. In other words, **row-size** indicates number of 1-d arrays and **col_size** indicates size of each of such 1-d array. Consider the following program –

Program 1.12 Demonstration of 2-d array

```
class TwoDArray
{
        public static void main(String args[])
        {
                int twoD[][]= new int[3][4];
                int i, j;

                    for(i=0; i<3; i++)
                            for(j=0; j<4; j++)
                                    twoD[i][j] = i+j;

                for(i=0; i<3; i++)
                {
                        for(j=0; j<4; j++)
                                System.out.print(twoD[i][j] + " ");

                        System.out.println();
                }
        }
}
```

The output would be –

```
0 1 2 3
1 2 3 4
2 3 4 5
```

Instead of allocating memory for 2-day as shown in the above program, we can even do it in a different way. We can first mention **row_size** and then using different statements, mention **col_size** as shown below –

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

But, above type of allocation is not having any advantage unless we need **uneven or irregular** multidimensional array. In Java, it is possible to have different number of columns for each row in a 2-d array. For example,

Program 1.13 Demonstration of irregular arrays

```
class UnevenArr
{
    public static void main(String args[])
    {
        int twoD[][] = new int[3][];
        twoD[0] = new int[3];
        twoD[1] = new int[1];
        twoD[2] = new int[5];

        int i, j, k = 0;

        for(i=0; i<3; i++)
                for(j=0; j<twoD[i].length; j++, k++)
                twoD[i][j] = k;

        for(i=0; i<3; i++)
        {
                for(j=0; j<twoD[i].length; j++)
                System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```

The output would be –

```
0 1 2
3
4 5 6 7 8
```

Here, we have declared a 2-d array with 3 rows. But, number of columns for each row varies. The first 1-d array has 3 elements, second 1-d array as a single element and the third 1-d array has 5 elements.

A 2-d array can be initialized at the time of declaration as follows –

```
int a[ ][ ]={{1,2},{3,4} };
```

We can have more than 2 dimensions as –

```
int a[ ][ ][ ]=new int[3][2][4];
```

Here, the arrayelements can be accessed using 3 indices like a[i][j][k].

Alternative Array Declaration Syntax

There is another wayof array declaration as given below –

```
type[] arr_name;
```

That is, following two declarations are same –

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

Both the declarations will create an integer array of 3 elements. Such declarations are useful when we have multiple array declarations of same type. For example,

```
int [ ] a, b, c;
```

will declare three arrays viz. a, b and c of type integer. This declaration is same as –

```
int a[ ], b[ ], c[ ];
```

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

Operators

Java provides rich set of operators, mainly divided into four groups viz. arithmetic, bitwise, relational andlogical. These operators are discussed here.

Arithmetic Operators

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

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is a subset of **int**.

Note down following few points about various operators:

- Basic arithmetic operators like +, -, * and / behave as expected for numeric data.
- The – symbol can be used as unary operator to negate a variable.
- If / is operated on two integer operands, then we will get only integral part of the result by truncating the fractional part.
- The % operator returns the remainder after division. It can be applied on integer and floating-pointtypes. For example,

```
int x=57;
double y= 32.8;
System.out.println("on integer " + x%10);        //prints 7
System.out.println("on double " + y%10);         //prints 2.8
```

- Compound assignment operators like += will perform arithmetic operation with assignment. Thatis,

  a+=2; ⟹ a=a+2;

- Increment/decrement operators (++ and --) will increase/decrease the operand by 1. That is,

  a++; ⟹ a=a+1;
  b--; ⟹ b=b-1;

- The ++ and -- operators can be used either as pre-increment/decrement or post-increment/decrement operator. For example,

  x= 5;
  y=x++; //post increment

  Now, value of x (that is 5) is assigned to y first, and x is then incremented to become 6.x= 5;
  y=++x; //pre-increment

  Now, x is incremented to 6 and then 6 is assigned to y.

NOTE that in C/C++, the % operator cannot be used on float or double and should be used only on integer variable.

Bitwise Operators

Java defines several *bitwise operators* that can be applied to **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Meaning
~	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

Since bitwise operators manipulate the bits within the integer, let us first understand the bit- representation of integer data in Java.

All of the integer types are represented by binary numbers of varying bit widths. For example, the **byte** value for 42 in binary is 00101010, where each position represents a power of two, starting with 2^0 at the rightmost bit. All of the integer types are signed integers. Java uses an encoding known as *two's complement,* which means that negative

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

Bitwise Logical Operators

The bitwise logical operators are &, |, ^ and ~. Following table shows the result of each operation.

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

Bitwise NOT

A unary NOT operator ~, also called as ***bitwise complement*** inverts all the bits of the operand. For example, the number 42, which has the following bit pattern: 00101010 becomes 11010101 after the NOT operator is applied.

Bitwise AND

As the name suggests, initially, operands are converted into binary-format. Then, the AND (&) operation is performed on the corresponding bits of operands. Consider an example –

```
int x=5, y=6,z;
z= x & y;
```

Now, this operation is carried out as –

```
x ⇨        0000 0101
y ⇨   &    0000 0110
z ⇨        0000 0100
```

Thus, z will be decimal equivalent of 0000 0100, which is 4.

Bitwise OR

Here, the OR (|) operations is performed on individual bit of operands. For example –

```
int x=5, y=6,z;
z= x | y;
```

Now, this operation is carried out as –

```
x ⇨        0000 0101
y ⇨   |    0000 0110
z ⇨        0000 0111
```

Thus, z will be decimal equivalent of 0000 0111, which is 7.

Bitwise XOR

In XOR operation, if both bits are same (either both are 1 or both 0), then the resulting bit will be 0 (false). Otherwise, the resulting bit is 1 (true). For example –

```
int x=5, y=6,z;
z= x ^ y;
```

Now, this operation is carried out as –

```
x  ⇨          0000 0101
y  ⇨   ^      0000 0110
   ____________________
z  ⇨          0000 0011
```

Thus, z will be decimal equivalent of 0000 0011, which is 3.

Left Shift

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

value << num

For each shift, one higher order bit is shifted out (or lost) and extra zero is appended as the lower order bit. Thus, for int, after 31 shifts, all the bits will be lost and result will be 0, whereas for long, after 63shifts, all bits will be lost.

Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values. As you know, **byte** and **short** values are promoted to **int** when an expression is evaluated. Furthermore, the result of such an expression is also an **int**. This means that the outcome of a left shift on a **byte** or **short** value will be an **int**, and the bits shifted left will not be lost until they shifted for 31 times. To avoid this problem, we should use type-casting as shown in the following example.

Program 2.1: Demonstration of left-shift operator

```
class ShiftDemo
{
        public static void main(String args[])
        {
                byte a = 64, b;int i;
                i = a << 2;
                b = (byte) (a << 2); System.out.println("Original value of a: " + a);
                System.out.println("i and b: " + i + " " + b);
        }
}
```

The result would be –

```
Original value of a: 64i
and b: 256        0
```

Since **a** is promoted to **int** for evaluation, left-shifting the value 64 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero.

Each left shift can be thought of as multiplying the number by 2. But, one should be careful because once the number crosses its range during left shift, it will become negative. Consider an illustration –

Program 2.2

```
class ShiftDemo1
 {
```

```
        public static void main(String args[])
        {
                int i;
                int num = 0xFFFFFFE;
                for(i=0; i<4; i++)
                {
                        num = num << 1;
                        System.out.println(num);
                }
        }
}
```

The output would be –

```
536870908
1073741816    //twice the previous value
2147483632    //twice the previous value
-32           //crosses the range of int and hence negative
```

Right Shift

The right shift operator, >> shifts all of the bits in a value to the right by a specified number of times. It has this general form:

value >> num

For each shift, one lower order bit is shifted out (or lost) and extra zero is appended as the higher orderbit. For example,

```
int a = 35;        //00100011 is the binary equivalent
a = a >> 2;        // now, a contains 8
```

Each right shift can be thought of as dividing the number by 2. When you are shifting right, the top (leftmost) bit is filled with the previous content of the top bit. This is called ***sign extension*** and is needed to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, inbinary, is

```
11111000    (–8)
>>1
11111100    (–4)
```

Unsigned Right Shift

We have seen that right shift always fills the highest order bit with the previous content of the top bit. But when we are using shift operation on non-numeric data, sign-bit has no significance. To ignore the sign- bit, we will go for unsigned right shift. The following code fragment demonstrates the >>>. Here, **a** is setto –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets **a** to 255.

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

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

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

Bitwise Operator Compound Assignment

We can use compound assignment even with bitwise operators. That is,a<<=2;

implies a=a<<2;

a^=3; implies a=a^3; and so on.

Relational Operators

The relational operators determine the relationship between two operands. Specifically, they determine equality and ordering among operands. Following table lists the relational operators supported by Java.

Operator	Meaning
==	Equal to (or comparison)
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, **==**, and the inequality test,
!=. Only numeric types can be compared using the ordering operators. That is, only integer, floating- point, and character operands may be compared to see which is greater or less than the other. For example, the following code fragment is perfectly valid:

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

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

Note that in C/C++ we can have following type of statement –

```
int flag;
…….
if(flag)
        //do something
```

In C/C++, ***true*** is any non-zero number and ***false*** is zero. But in Java, ***true*** and ***false*** are Boolean values and nothing to do with zero or non-zero. Hence, the above set of statements **will cause an error** in Java.We should write –

```
int flag;
………..
if(flag==1)
        //do some thing
```

Boolean Logical Operators

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

Operator	Meaning
&	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 truth table is given below for few operations:

A	B	A|B	A&B	A^B	!A
False	False	False	False	False	True
False	True	True	False	True	True
True	False	True	False	True	False
True	True	True	True	False	False

Program 2.3 Demonstration of 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);

        boolean h = b & (a=!a);

        System.out.println("b & (a=!a) =" +h);
        System.out.println("New a is "+a);
    }
}
```

The output would be –

a = true b
= false a|b
= true
a&b = false

a^b = true
!a&b|a&!b = true
!a = false
b & (a=!a) =false
New a is false

Note: In C/C++, the logical AND/OR operations never evaluates the second operand if the value of first operand itself can judge the result. That is, if the first operand is *false*, then second operand is not evaluated in AND operation and result will be false. Similarly, if the first operand is *true* in OR operation, without evaluating the second operand, it results *true*. But in Java, Boolean logical operators will not act so. Even if the first operand is decisive, the second operand is evaluated. This can be observed in the above program while evaluating h= b& (a= !a). Here, *b* is *false* and hence ANDed with anything results *false*. But, still the second operand (a= !a) is evaluated resulting *a* as false.

If we don't want the second operand to be evaluated, we can use *short-circuit logical* operators.

Short-Circuit Logical Operators

The short-circuit AND (&&) and OR (||) operators will not evaluate the second operand if the first is decisive. For example,

```
int x=0, n=5;
……..
if(x!=0 && n/x > 0)
        //do something
```

Here, the first operand x!= 0 is false. If we use logical AND (&) then the second operand n/x>0 will be evaluated and we will get DivisionByZero Exception. So, to avoid this problem we use && operator which will never evaluated second operand if the first operand results into false.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & e++ < 100)d =
        100;
```

Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 ornot.

The Assignment Operator

The *assignment operator* is the single equal sign, **=**. It has this general form:

var = expression;

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

```
int x, y, z;
```

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

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** isan operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which isthen assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

The ?: Operator

Java supports *ternary operator* which sometimes can be used as an alternative for *if*-then-*else* statement.The general form is –

var = *expression1* **?** *expression2* **:** *expression3;*

Here, *expression1* is evaluated first and it must return Boolean type. If it results *true*, then value of *expression2* is assigned to *var*, otherwise value of *expression3* is assigned to *var*. For example,

```
int a, b, c ;
……….
c= (a>b)?a:b;                    //c will be assigned with biggest among a and b
```

Operator Precedence

Following table describes the precedence of operators. Though parenthesis, square brackets etc. are separators, they do behave like operators in expressions. Operators at same precedence level will be evaluated from left to right, whichever comes first.

Highest

<table>
<tr><td>(), [ ], .</td></tr>
<tr><td>++, --, ~, !</td></tr>
<tr><td>*, /, %</td></tr>
<tr><td>+, -</td></tr>
<tr><td>>>, >>>, <<</td></tr>
<tr><td>>, >=, <, <=</td></tr>
<tr><td>==, !=</td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>?:</td></tr>
<tr><td>=, op=</td></tr>
</table>

Lowest

Using Parentheses

Parentheses always make the expression within them to execute first. This is necessary sometimes. For example,

```
a= b – c * d;
```

Here, c and d are multiplied first and then the result is subtracted from b. If we want subtraction first, weshould use parenthesis like

```
a= (b-c)*d;
```

Sometimes, parenthesis is useful for clarifying the meaning of an expression and for making readers tounderstand the code. For example,

```
a | 4 + c >> b & 7                can be written as (a | (((4 + c) >> b) & 7))
```

In such situations, though parenthesis seems to be redundant, it existence will not reduce theperformance of the program.

Control Statements

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: **selection, iteration,** and **jump**. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

Java's Selection Statements

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

if Statement

The general form is –

```
if (condition)
{
        //true block
}
else
{
        //false block
}
```

If the *condition* is true, then the statements written within *true block* will be executed, otherwise *falseblock* will be executed. The *condition* should result into **Boolean** type. For example,

```
int a, b, max;
…………
if(a>b)
        max=a;
else
        max=b;
```

Nested-if Statement

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. For example,

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

```
}
else
        a = d;
```

The if-else-if Statement

The general form is –

```
if(condition1)
        block1;
else if(condition2)block2;
…………..
………
…..else
        blockn
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the block associated with that **if** is executed, and the rest of the ladder is bypassed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.

switch Statement

The **switch** statement is Java's multi-way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides abetter alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression)
{
        case value1:
                // statement sequencebreak;
        case value2:
                // statement sequencebreak;
        … ............... case
        valueN:
                // statement sequencebreak;
        default:
                // default statement sequence
}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken. The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**. The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**.

NOTE:

- We can even nest switch statements one within the other.
- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value ofthe expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- A **switch** statement is usually more efficient than a set of nested **if**s.

The last point is particularly interesting because it gives insight into how the Java compiler works. When itcompiles a **switch** statement, the Java compiler will inspect each of the **case** constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-else**s. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

Iteration Statements

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

while Loop

The general form is –

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

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop.

do- while Loop

The general form is –

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

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

for Loop

The general form is –

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

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

for-each Loop

The for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-eachversion of the **for** is shown here:

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

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

Consider an example –

```
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];
```

The above set of statements can be optimized as follows –

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

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

For multi-dimensional arrays:

The for-each version also works for multi-dimensional arrays. Since a 2-d array is an array of 1-d array, the iteration variable must be a reference to 1-d array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions.

Consider the following example –

Program 2.4 Demonstration of *for-each* version of *for* loop

```
class ForEach
{
        public static void main(String args[])
        {
                int sum = 0;
                int nums[][] = new int[2][3];

                // give nums some values for(int i = 0;
                i < 2; i++)
                        for(int j=0; j < 3; j++) nums[i][j] =
                                (i+1)*(j+1);

                for(int x[ ] : nums)  //nums is a 2-d array and x is 1-d array
                {
                        for(int y : x)     // y refers elements in 1-d array x
                        {
                                System.out.println("Value is: " +y);sum += y;
                        }
                }
                System.out.println("Summation: " + sum);
        }
}
```

The output would be –

```
Value is: 1
Value is: 2
Value is: 3
Value is: 2
Value is: 4
Value is: 6
Summation: 18
```

The for-each version of **for** has several applications viz. Finding average of numbers, finding minimum and maximum of a set, checking for duplicate entry in an array, searching for an element in unsorted list etc. The following program illustrates the sequential (linear) search**.**

Program 2.5 Linear/Sequential Search

```
class SeqSearch
{
        public static void main(String args[])
        {
                int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
                int val = 5;
```

```
boolean found = false;

for(int x : nums)
{
        if(x == val)
        {
```

```
                    found = true;break;
                }
            }
            if(found)
                System.out.println("Value found!");
        }
    }
```

The output would be –

Value found !

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control toanother part of your program.

Using break

In java, ***break*** can be used in 3 different situations:

- To terminate statement sequence in **switch**
- To exit from a loop
- Can be used as a *civilized* version of ***goto***

Following is an example showing terminating a loop using break.

```
for (int i=0;i<20;i++)if(i==5)
        break;
    else
        System.out.println(" i= " + i);
```

The above code snippet prints values from 0 to 4 and when i become 5, the loop is terminated.

Using break as a form of goto

Java does not have a goto statement because it is an un-conditional jump and may end up with an infinite loop. But in some situations, goto will be useful. For example, the goto can be useful when youare exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, youcan specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems. The general form of labeled break is:

break label;

Program 2.6 Illustration of *break* statement with labels

```
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.");
                }
        }
}
```

The output would be –

```
        Before the break
        This is after second block
```

As we can see in the above program, the usage of ***break*** with a label takes the control out of the secondblock directly.

Using continue

Sometimes, we may need to proceed towards next iteration in the loop by leaving some statements. Insuch situations, we can use ***continue*** statement within ***for, while*** and ***do-while***. For example –

```
for (int i=1; i<20;i++)if (i%2 ==
        0)
                continue;
        else
                System.out.println("i = " + i);
```

The above code snippet prints only the odd numbers in the range of 1 to 20.

Using return

The ***return*** statement is used to explicitly return the method. Based on some condition, we may need togo back to the calling method sometimes. So, we can use ***return*** in such situations.

QUESTION BANK:

1. Explain key attributes of Java programming language.
2. Briefly explain JRE and JDK.
3. Explain three OOPs principles.
4. What are Keywords and Identifiers? List the rules to write an identifier.
5. Discuss various data types used in Java.
6. What is type Conversion and Casting? Explain automatic type promotion in expressions with rules and a demo program.
7. Explain scope and lifetime of variables with suitable examples.
8. "Java is a strongly typed language" - Justify this statement.
9. Write a note on
    a. Java class libraries, Literals
10. Explain array declaration and initialization in Java with suitable examples.
11. What are multi-dimensional arrays? Explain with examples.
12. What are different types of operators in Java? Explain any two of them.
13. Discuss ternary operator with examples.
14. Differentiate >> and >>> with suitable examples.
15. Briefly explain short-circuit logical operators with examples.
16. Explain different types of iteration statements with examples.
17. Discuss various selective control structures.
18. Write a note on jump statements in Java.
19. Discuss different versions of for - loop with examples.
20. Write a program to illustrate break statement with labels.

MODULE 2

Syllabus:

Introducing Classes: Class Fundamentals, Declaring Objects, Assigning Object Reference Variables, Introducing Methods, Constructors, The this Keyword, Garbage Collection, The finalize( ) Method.
Methods and Classes: Overloading Methods, Using Objects as Parameters, A Closer Look at Argument Passing, Returning Objects, Recursion, Introducing Access Control, Understanding static, Introducing final, Introducing nested and Inner classes

Class is a basis of OOP languages. It is a logical construct which defines shape and nature of an object. Entire Java is built upon classes.

Class Fundamentals

Class can be thought of as a user-defined data type. We can create variables (objects) of that data type. So, we can say that class is a ***template*** for an object and an object is an ***instance*** of a class. Most of the times, the terms *object* and *instance* are used interchangeably.

The General Form of a Class

A class contains data (member or instance variables) and the code (member methods) that operate on the data. The general form can be given as –

```
class classname
{
        type var1;
        type var2;
        …….
        type method1(para_list)
        {
                //body of method1
        }

        type method2(para_list)
        {
                //body of method2
        }
        ………..
}
```

Here, *classname* is any valid name given to the class. Variables declared within a class are called as ***instance variables*** because every instance (or object) of a class contains its own copy of these variables. The code is contained within ***methods***. Methods and instance variables collectively called as ***members*** of the class.

A Simple Class

Here we will consider a simple example for creation of class, creating objects and using members of the class. One can store the following program in a single file called ***BoxDemo.java***. (Or, two classes can be saved in two different files with the names ***Box.java*** and ***BoxDemo.java***.)

```
class Box
{
        double w, h, d;
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
                double vol;

                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                vol=b1.w*b1.h*b1.d;
                System.out.println("Volume of Box1 is " + vol);

                vol=b2.w*b2.h*b2.d;
                System.out.println("Volume of Box2 is " + vol);
        }
}
```

The output would be –

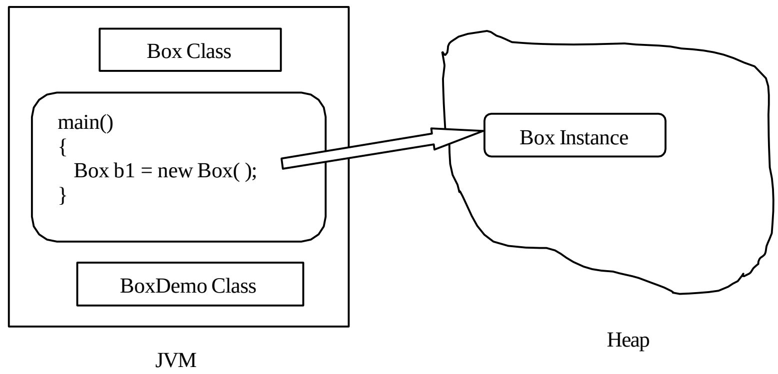
```
Volume of Box1 is 24.0
Volume of Box1 is 60.0
```

When you compile above program, two class files will be created viz. ***Box.class*** and ***BoxDemo.class***. Since *main()* method is contained in ***BoxDemo.class***, you need to execute the same.

In the above example, we have created a class ***Box*** which contains 3 instance variables *w, h, d***.**

```
Box b1=new Box();
```

The above statement creates a physical memory for one object of ***Box*** class. Every object is an instance of a class, and so, *b1* and *b2* will have their own copies of instance variables *w, h* and *d.* The memory layout for one object allocation can be shown as –


Box Class
main()
{
Box b1 = new Box( );
}
BoxDemo Class
Box Instance
JVM
Heap

Declaring Objects

Creating a class means having a user-defined data type. To have a variable of this new data type, we should create an object. Consider the following declaration:

```
Box b1;
```

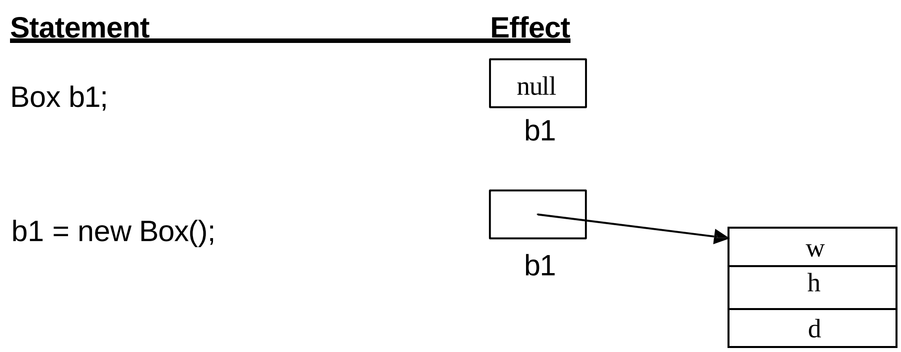
This statement will not actually create any physical object, but the object name *b1* can just ***refer*** to the actual object on the heap after memory allocation as follows –

```
b1 = new Box ();
```

We can even declare an object and allocate memory using a single statement –

```
Box b1=new Box();
```

Without the usage of *new*, the object contains **null.** Once memory is allocated dynamically, the object *b1* contains the address of real object created on the heap. The memory map is as shown in the following diagram –



Closer look at *new*

The general form for object creation is –

obj_name = new class_name();

Here, ***class_name()*** is actually a constructor call. A ***constructor*** is a special type of member function invoked automatically when the object gets created. The constructor usually contains the code needed for object initialization. If we do not provide any constructor, then Java supplies a ***default constructor.***

Java treats primitive types like *byte, short, int, long, char, float, double* and *boolean* as ordinary variables but not as an object of any class. This is to avoid extra overhead on the heap memory and also to increase the efficiency of the program. Java also provides the *class-version* of these primitive types that can be used only if necessary. We will study those types later in detail.

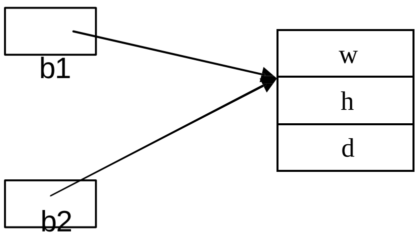
With the term *dynamic memory allocation*, we can understand that the keyword *new* allocates memory for the object during runtime. So, depending on the user's requirement memory will be utilized. This will avoid the problems with static memory allocation (either shortage or wastage of memory during runtime). If there is no enough memory in the heap when we use ***new*** for memory allocation, it will throw a run-time exception.

Assigning Object Reference Variables

When an object is assigned to another object, no separate memory will be allocated. Instead, the second object refers to the same location as that of first object. Consider the following declaration –

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

Now both b1 and b2 refer to same object on the heap. The memory representation for two objects can be shown as –



Thus, any change made for the instance variables of one object affects the other object also. Although b1 and b2 both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to b1 will simply *unhook* b1 from the original object without affecting the object or affecting b2. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

NOTE that when you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

A class can consist of instance variables and methods. We have seen declaration and usage of instance variables in Program 2.1. Now, we will discuss about methods. The general form of a method is –

```
ret_type method_name(para_list)
{
        //body of the method
        return value;
}
```

Here, ***ret_type*** specifies the data type of the variable returned by the method. It may be any primitive type or any other derived type including name of the same class. If the method does not return any value, the ***ret_type*** should be specified as ***void***.

method_name is any valid name given to the method

para_list is the list of parameters (along with their respective types) taken the method. It may be even empty also.

body of method is a code segment written to carryout some process for which the method is meant for.

return is a keyword used to send ***value*** to the calling method. This line will be absent if the ***ret_type*** is void.

Adding Methods to *Box* class

Though it is possible to have classes with only instance variables as we did for ***Box*** class of Program 2.1, it is advisable to have methods to operate on those data. Because, methods acts as interface to the classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself. Consider the following example –

```
class Box
{
        double w, h, d;

        void volume()
        {
                System.out.println("The volume is " + w*h*d);
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();

                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                b1.volume();
                b2.volume();
        }
```

```
}
```

The output would be –

```
The volume is 24.0
The volume is 60.0
```

In the above program, the ***Box*** objects *b1* and *b2* are invoking the member method *volume()* of the ***Box*** class to display the volume. To attach an object name and a method name, we use dot (**.**) operator. Once the program control enters the method *volume()*, we need not refer to object name to use the instance variables *w, h* and *d.*

Returning a value

In the previous example, we have seen a method which does not return anything. Now we will modify the above program so as to return the value of *volume* to *main()* method.

```
class Box
	{
		double w, h, d;

		double volume()
		{
			return w*h*d;
		}
	}

	class BoxDemo
	{
		public static void main(String args[])
		{
			Box b1=new Box();
			Box b2=new Box();
			double vol;

			b1.w=2;
			b1.h=4;
			b1.d=3;

			b2.w=5;
			b2.h=6;
			b2.d=2;

			vol  = b1.volume();
			System.out.println("The volume is " + vol);
			System.out.println("The volume is " + b2.volume());
		}
```

```
}
```

The output would be –

The volume is 24.0
The volume is 60.0

As one can observe from above example, we need to use a variable at the left-hand side of the assignment operator to receive the value returned by a method. On the other hand, we can directly make a method call within print statement as shown in the last line of above program.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

Adding Methods that takes Parameters

Having parameters for methods is for providing some input information to process the task. Consider the following version of ***Box*** class which has a method with parameters.

```
class Box
{
        double w, h, d;

        double volume()
        {
                return w*h*d;
        }

        void set(double wd, double ht, double dp)
        {
                w=wd;
                h=ht;
                d=dp;
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
```

```
                b1.set(2,4,3);
                b2.set(5,6,2);

                System.out.println("The volume of b1 is " + b1.volume());
                System.out.println("The volume of b2 is " + b2.volume());
        }
}
```

The output would be –

```
The volume of b1 is 24.0
The volume of b2 is 60.0
```

In the above program, the ***Box*** class contains a method ***set()*** which take 3 parameters. Note that, the variables ***wd, ht*** and ***dp*** are termed as ***formal parameters*** or just ***parameters*** for a method. The values passed like 2, 4, 3 etc. are called as ***actual arguments*** or just ***arguments*** passed to the method.

Constructors

Constructor is a special type of member method which is invoked automatically when the object gets created. Constructors are used for object initialization. They have same name as that of the class. Since they are called automatically, there is no return type for them. Constructors may or may not take parameters.

```
class Box
 {
        double w, h, d;

        double volume()
        {
                return w*h*d;
        }

        Box()                   //ordinary constructor
        {
                w=h=d=5;
        }

        Box(double wd, double ht, double dp)            //parameterized constructor
        {
                w=wd;
                h=ht;
                d=dp;
        }
}
```

```
class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
                Box b3=new Box(2,4,3);

                System.out.println("The volumeof b1 is " + b1.volume());
                System.out.println("The volumeof b2 is " + b2.volume());
                System.out.println("The volumeof b3 is " + b3.volume());
        }
}
```

The output would be –

```
The volume of b1 is 125.0
The volume of b2 is 125.0
The volume of b3 is 24.0
```

When we create two objects ***b1*** and ***b2***, the constructor with no arguments will be called and the all the instance variables ***w, h*** and ***d*** are set to 5. Hence volume of ***b1*** and ***b2*** will be same (that is 125 in this example). But, when we create the object ***b3***, the ***parameterized constructor*** will be called and hence volume will be 24.

Few points about constructors:

- Every class is provided with a ***default constructor*** which initializes all the data members to respective *default values*. (Default for numeric types is zero, for character and strings it is null and default value for Boolean type is false.)

- In the statement
  classname ob= new classname();
  the term *classname()* is actually a constructor call.
- If the programmer does not provide any constructor of his own, then the above statement will call default constructor.
- If the programmer defines any constructor, then default constructor of Java can not be used.
- So, if the programmer defines any parameterized constructor and later would like to create an object without explicit initialization, he has to provide the default constructor by his own.
  For example, the above program, if we remove ordinary constructor, the statements like
  `Box b1=new Box();`
  will generate error. To avoid the error, we should write a default constructor like –
  `Box(){ }`
  Now, all the data members will be set to their respective default values.

The *this* Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the ***this*** keyword. ***this*** can be used inside any method to refer to the ***current object***. That is, ***this*** is always a reference to the object which invokes the method call. For example, in the **Program 2.5,** the method *volume()* can be written as –

```
double volume()
{
    return this.w * this.h * this.d;
}
```

Here, usage of ***this*** is not mandatory as it is implicit. But, in some of the situations, it is useful as explained in the next section.

Instance Variable Hiding

As we know, in Java, we can not have two local variables with the same name inside the same or enclosing scopes. (Refer **Program 1.7** and a **NOTE** after that program from Chapter 1, Page 16 & 17). But we can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the **local variable *hides* the instance variable**. That is, if we write following code snippet for a constructor in **Program 2.5**, we will not get an expected output –

```
Box(double w, double h, double d)
{
    w=w;
    h=h;
    d=d;
}
```

Here note that, formal parameter names and data member names match exactly. To avoid the problem, we can use –

```
Box(double w, double h, double d)
{
    this.w=w;       //this.w refers to data member name and w refers to formal parameter
    this.h=h;
    this.d=d;
}
```

Garbage Collection

In C and C++, dynamically allocated variables/objects must be manually released using ***delete*** operator. But, in Java, this task is done automatically and is called as ***garbage collection.*** When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. Garbage collection occurs once in a while during the execution of the program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection.

The *finalize()* Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called ***finalization****.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define the ***finalize()*** method. The Java run time calls that method whenever it is about to recycle an object of that class.

The finalize( ) method has this general form:

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

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class. Note that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope. So, we can not know when ***finalize()*** method is called, or we may be sure whether it is called or not before our program termination. Therefore, if at all our program uses some resources, we should provide some other means for releasing them and must not depend on ***finalize()*** method.

Overloading Methods

Having more than one method with a same name is called as method overloading. To implement this concept, the constraints are:

- the number of arguments should be different, and/or
- Type of the arguments must be different.

NOTE that, only the return type of the method is not sufficient for overloading.

```
class Overload
{
      void test()             //method without any arguments
      {
            System.out.println("No parameters");
      }

      void test(int a)        //method with one integer argument
      {
            System.out.println("Integer a: " + a);
      }

      void test(int a, int b) //two arguments
      {
        System.out.println("With two arguments : " + a + " " + b);
      }

      void test(double a)           //one argument of double type
      {
            System.out.println("double a: " + a);
      }
}

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

            ob.test();
            ob.test(10);
            ob.test(10, 20);
            ob.test(123.25);
      }
```

```
        }
```

Overloading Constructors

One can have more than one constructor for a single class if the number and/or type of arguments are different. Consider the following code:

```
class OverloadConstruct
{
        int a, b;
        OverloadConstruct()
        {
                System.out.println("Constructor without arguments");
        }

        OverloadConstruct(int x)
        {
                a=x;
                System.out.println("Constructor with one argument:"+a);
        }

        OverloadConstruct(int x, int y)
        {
                a=x;
                b=y;
                System.out.println("Constructor with two arguments:"+ a +"\t"+ b);
        }
}

class OverloadConstructDemo
{
        public static void main(String args[])
        {
                OverloadConstruct ob1= new OverloadConstruct();
                OverloadConstruct ob2= new OverloadConstruct(10);
                OverloadConstruct ob3= new OverloadConstruct(5,12);
        }
}
```

Output:

```
        Constructor without arguments
        Constructor with one argument: 10
        Constructor with two arguments: 5        12
```

Using Objects as Parameters

Just similar to primitive types, even object of a class can also be passed as a parameter to any method. Consider the example given below –

```
class Test
{
      int  a,  b;
      Test(int i, int j)
      {

            a = i;
            b = j;
      }

      boolean equals(Test ob)
      {
            if(ob.a == this.a && ob.b == this.b)
                  return true;
            else
                  return false;
      }
}
```

```
class PassOb
{
      public static void main(String args[])
      {
            Test ob1 = new Test(100, 22);
            Test ob2 = new Test(100, 22);
            Test ob3 = new Test(-1, -1);
            System.out.println("ob1 == ob2: " + ob1.equals(ob2));
            System.out.println("ob1 == ob3: " + ob1.equals(ob3));
      }
}
```

```
Output:
      ob1 == ob2: true
      ob1 == ob3: false
```

Using one object to initialize the other:

Sometimes, we may need to have a replica of one object. The usage of following statements **will not** serve the purpose.

Box b1=new Box(2,3,4);
Box b2=b1;

In the above case, both b1 and b2 will be referring to same object, but not two different objects. So, we can write a constructor having a parameter of same class type to ***clone*** an object.

```
class Box
{
      double h, w, d;

      Box(double ht, double wd, double dp)
      {
            h=ht; w=wd; d=dp;
      }
      Box (Box bx)                  //observe this constructor
      {
            h=bx.h; w=bx.w; d=bx.d;
      }

      void vol()
      {
            System.out.println("Volume is " + h*w*d);
      }
      public static void main(String args[])
      {
```

```
                Box b1=new Box(2,3,4);
                Box b2=new Box(b1);   //initialize b2 using b1
                b1.vol();
        }       b2.vol();
            }
```

Output:

```
    Volume is 24
    Volume is 24
```

A Closer Look at Argument Passing

In Java, there are two ways of passing arguments to a method.

- **Call by value :** This approach copies the *value* of an argument into the formal parameter of the method. Therefore, changes made to the parameter of the method have no effect on the argument.
- **Call by reference:** In this approach, a reference to an argument is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

In Java, **when you pass a primitive type to a method, it is passed by value.** When you **pass an object to a method, they are passed by reference.** Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument.

```
class Test
{     int  a,  b;
      Test(int i, int j)
      {
            a = i;
            b = j;
      }
      void meth(Test o)
      {
      *= 2;
      /= 2;
      }
}
class CallByRef
{
      public static void main(String args[])
      {
```

```
        Test ob = new Test(15, 20);
        System.out.println("before call: " + ob.a + " " + ob.b);
        ob.meth(ob);
        System.out.println("after call: " + ob.a + " " + ob.b);
    }
}
```

Output:
before call: 15 20
after call: 30 10

Returning Objects

In Java, a method can return an object of user defined class.

```
class Test
{
    int a;
    Test(int i)
    {
        a = i;
    }

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

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

      Output:

  }
}
```

```
T            new Test(2); Test ob2;
e
s           ob2 = ob1.incrByTen();
t           System.out.println("ob1.a: " + ob1.a);
            System.out.println("ob2.a: " + ob2.a);
o
b           ob2 = ob2.incrByTen();
1           System.out.println("ob2.a after second increase: " + ob2.a);

=
```

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

Recursion

A method which invokes itself either directly or indirectly is called as *recursive method.* Every recursive method should satisfy following constraints:

- It should have at least one non-recursive terminating condition.
- In every step, it should be nearer to the solution (that is, problem size must be decreasing)

```
class Factorial
{
   int fact(int n)
   {
        if (n==0)
             return 1;
        return n*fact(n-1);
   }
}

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

        System.out.println("Factorial 3 is "+ f.fact(3));
        System.out.println("Factorial 8 is "+ f.fact(8));

   }
}
```

```
Output:
     Factorial of 3 is 6
     Factorial of 8 is 40320
```

Introducing Access Control

Encapsulation feature of Java provides a safety measure viz. ***access control.*** Using ***access specifiers***, we can restrict the member variables of a class from outside manipulation. Java provides following access specifiers:

- public
- private
- protected

Along with above access specifiers, Java defines a *default access level.*

Some aspects of access control are related to inheritance and package (a collection of related classes). The *protected* specifier is applied only when inheritance is involved. So, we will now discuss about only *private* and *public.*

When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class. An access specifier precedes the rest of a member's type specification. For example,

    public   int   x;
    private char ch;

Consider a program given below –

```
class Test
{     int   a;
      public int b;
      private int c;

      void setc(int i)
      {
            c = i;
      }

      int getc()
      {
            return c;
      }
}

class AccessTest
{
      public static void main(String args[])
      {
            Test ob = new Test();
            ob.a = 10;
            ob.b = 20;
            // ob.c = 100;          // inclusion of this line is Error!
            ob.setc(100);
            System.out.println("a, b, and c: " + ob.a + " " + ob.b + " "
                                                + ob.getc());
      }
}
```

Understanding *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. Instance variables declared as **static** are 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.

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.

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

	static void meth(int x)          //static method
	{
		System.out.println("x = " + x);
		System.out.println("a = " + a);
		System.out.println("b = " + b);
	}

	static     //static block
	{
		System.out.println("Static block initialized.");
		b = a * 4;
	}

	public static void main(String args[])
	{
		meth(42);
	}
}
```

Output:

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. The general form is –

classname.method();

Consider the following program:

```
class StaticDemo
{
      static int a = 42;
      static int b = 99;

      static void callme()
      {
            System.out.println("Inside static method, a = " + a);
      }
}

class StaticByName
```

```
    {
        public static void main(String args[])
        {
            StaticDemo.callme();
            System.out.println("Inside main, b = " + StaticDemo.b);
        }
    }
```

Output:

Inside static method, a = 42
Inside main, b = 99

Module 3

Inheritance

Inheritance is one of the building blocks of object oriented programming languages. It allows creation of classes with hierarchical relationship among them. Using inheritance, one can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a ***superclass***. The class that does the inheriting is called a ***subclass***. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements. Through inheritance, one can achieve re-usability of the code.

In Java, inheritance is achieved using the keyword ***extends.*** The syntax is given below:

```
class A                 //super class
{
      //members of class A
}

class B extends A       //sub class
{
      //members of B
}
```

Consider a program to understand the concept:

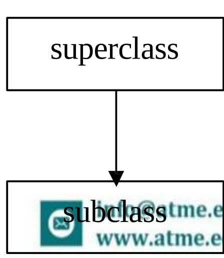
```
class A
{
      int i, j;

      void showij()
      {
            System.out.println("i and j: " + i + " " + j);
      }
}

class B extends A
{
      int k;
      void showk()
      {
            System.out.println("k: " + k);
```

```
        }
        void sum()
        {
                System.out.println("i+j+k: " + (i+j+k));
        }
}

class SimpleInheritance
{
        public static void main(String args[])
        {
                A superOb = new A();
                B subOb = new B();
                superOb.i = 10;
                superOb.j = 20;
                System.out.println("Contents of superOb: ");
                superOb.showij();

                subOb.i = 7;
                subOb.j = 8;
                subOb.k = 9;
                System.out.println("Contents of subOb: ");


                subOb.showij();
                subOb.showk();

                System.out.println("Sum of i, j and k in subOb:");
                subOb.sum();
        }
}
```

Note that, private members of the super class can not be accessed by the sub class. The subclass contains all non-private members of the super class and also it contains its own set of members to achieve specialization.

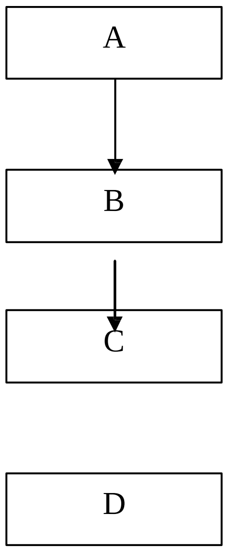
Type of Inheritance

- **Single Inheritance:** If a class is inherited from one parent class, then it is known as single inheritance. This will be of the form as shown below –



The previous program is an example of single inheritance.

- **Multilevel Inheritance:** If several classes are inherited one after the other in a hierarchical manner, it is known as multilevel inheritance, as shown below –



A Superclass variable can reference a subclass object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. Consider the following for illustration:

```
class Base
{
      void dispB()
      {
            System.out.println("Super class " );
      }
}
class Derived extends Base
{
      void dispD()
      {
            System.out.println("Sub class ");
      }
}

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

```
	{
		Base b = new Base();
		Derived d=new Derived();

		b=d;		//superclass reference is holding subclass object
		b.dispB();
		//b.dispD();	error!!
	}
}
```

Note that, the ***type of reference variable*** decides the members that can be accessed, ***but not the type of the actual object.*** That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

Using *super*

In Java, the keyword *super* can be used in following situations:

- To invoke superclass constructor within the subclass constructor
- To access superclass member (variable or method) when there is a duplicate member name in the subclass

Let us discuss each of these situations:

- **To invoke superclass constructor within the subclass constructor:** Sometimes, we may need to initialize the members of super class while creating subclass object. Writing such a code in subclass constructor may lead to redundancy in code. For example,

```
class Box
{
	double w, h, b;

	Box(double wd, double ht, double br)
	{
		w=wd; h=ht; b=br;
	}
}
class ColourBox extends Box
{
	int colour;
	ColourBox(double wd, double ht, double br, int c)
	{
		w=wd; h=ht; b=br;		//code redundancy
		colour=c;
	}
}
```

Also, if the data members of super class are private, then we can't even write such a code in subclass constructor. If we use super() to call superclass constructor, then **it must be the first statement** executed inside a subclass constructor as shown below –

```
class Box
{
      double w, h, b;
      Box(double wd, double ht, double br)
      {
            w=wd; h=ht; b=br;
      }
}

class ColourBox extends Box
{
      int colour;
      ColourBox(double wd, double ht, double br, int c)
      {
            super(wd, ht, br);          //calls superclass constructor
            colour=c;
      }
}

class Demo
{
      public static void main(String args[])
      {
            ColourBox b=new ColourBox(2,3,4, 5);
      }
}
```

Here, we are creating the object *b* of the subclass `ColourBox` . So, the constructor of this class is invoked. As the first statement within it is **`super(wd, ht, br)`**, the constructor of superclass `Box` is invoked, and then the rest of the statements in subclass constructor `ColourBox` are executed.

- **To access superclass member variable when there is a duplicate variable name in the subclass:** This form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
class A
{
      int a;
}

class B extends A
{
      int a;      //duplicate variable a

      B(int x, int y)
      {
            super.a=x;        //accessing superclass a
            a=y;              //accessing own member a
      }

      void disp()
      {
            System.out.println("super class a: "+ super.a);
            System.out.println("sub class a: "+ a);
      }
}

class SuperDemo
{
   public static void main(String args[])
   {
      B ob=new B(2,3);
      ob.disp();
   }
}
```

Creating Multilevel Hierarchy

Java supports multi-level inheritance. A sub class can access all the non-private members of all of its super classes. Consider an illustration:

```
class A
{     int a;
}
```

```
class B extends A
{     int b;
}

class C extends B
{     int c;
```

```
        C(int x, int y, int z)
        {
                a=x;  b=y;  c=z;
        }
        void disp()
        {
                System.out.println("a= "+a+ " b= "+b+" c="+c);
        }
}

class MultiLevel
{
    public static void main(String args[])
   {
        C ob=new C(2,3,4);
        ob.disp();
    }
}
```

When Constructors are called

When class hierarchy is created (multilevel inheritance), the constructors are called in the order of their derivation. That is, the top most super class constructor is called first, and then its immediate sub class and so on. If *super* is not used in the sub class constructors, then the default constructor of super class will be called.

```
class A
{
        A()
        {
                System.out.println("A's constructor.");
        }
}

class B extends A
{
        B()
        {
                System.out.println("B's constructor.");
        }
}
class C extends B
{
        C()                     {
```

```
      }
}           System.out.println("C's constructor.");
class CallingCons
{
  public static void main(String args[])
```

```
    {
      C c = new C();
    }
}
```

Output:

```
A's constructor
B's constructor
C's constructor
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to ***override*** the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

```
class A
{
      int  i,  j;
      A(int a, int b)
      {
            i = a;
            j = b;
      }
      void show()      //suppressed
      {
            System.out.println("i and j: " + i + " " + j);
      }
}
class B extends A
{
      int k;
      B(int a, int b, int c)
      {
            super(a, b);
            k = c;
      }
      void show()      //Overridden method
      {
            System.out.println("k: " + k);
      }
}
class Override
```

```
{
    public static void main(String args[])
    {
          B subOb = new B(1, 2, 3);
          subOb.show();
     }
}
```

Output:
k: 3

Note that, above program, only subclass method *show()* got called and hence only *k* got displayed. That is, the ***show()*** method of super class is suppressed. If we want superclass method also to be called, we can re-write the ***show()*** method **in subclass** as –

```
void show()
{
      super.show();    // this calls A's show()
      System.out.println("k: " + k);
}
```

Method overriding occurs *only* when the names and the type signatures of the two methods (one in superclass and the other in subclass) are identical. If two methods (one in superclass and the other in subclass) have same name, but different signature, then the two methods are simply overloaded.

Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch.* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Java implements run-time polymorphism using dynamic method dispatch. We know that, a superclass reference variable can refer to subclass object. Using this fact, Java resolves the calls to overridden methods during runtime. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
class A
{
      void callme()
      {
         System.out.println("Inside A");
      }
}
class B extends A
{
      void callme()
      {
         System.out.println("Inside B");
```

```
        }
}

class C extends A
{
        void callme()
        {
```

```
            System.out.println("Inside C");
        }
}
class Dispatch
{
   public static void main(String args[])
   {
     A a = new A();
     B b = new B();
     C c = new C();

     A r;        //Superclass reference
     r = a;      //holding subclass object
     r.callme();
     r = b;
     r.callme();
     r = c;
     r.callme();
    }
}
```

Why overridden methods?

Overridden methods are the way that Java implements the "one interface, multiple methods" aspect of polymorphism. superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses. Dynamic, run-time polymorphism is one of the most powerful mechanisms that objectoriented design brings to bear on code reuse and robustness.

Using Abstract Classes

Sometimes, the method definition will not be having any meaning in superclass. Only the subclass (specialization) may give proper meaning for such methods.In such a situation, having a definition for a method in superclass is absurd. Also, we should enforce the subclass to override such a method. A method which does not contain any definition in the superclass is termed as ***abstract method***. Such a method declaration should be preceded by the keyword ***abstract***. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass.

A class containing at least one abstract method is called as ***abstract class.*** Abstract classes can not be instantiated, that is one cannot create an object of abstract class. Whereas, a reference can be created for an abstract class.

```
abstract class A
```

```
{
      abstract void callme();
      void callmetoo()
      {
            System.out.println("This is a concrete method.");
      }
}
```

```
class B extends A
{
      void callme() //overriding abstract method
      {
            System.out.println("B's implementation of callme.");
      }
}

class AbstractDemo
{
      public static void main(String args[])
      {
            B b = new B(); //subclass object
            b.callme();     //calling abstract method
            b.callmetoo(); //calling concrete method
      }
}
```

Example: Write an abstract class *shape,* which has an abstract method *area()*. Derive three classes *Triangle, Rectangle* and *Circle* from the *shape* class and to override *area()*. Implement run-time polymorphism by creating array of references to supeclass. Compute area of different shapes and display the same.

Solution:

```
abstract class Shape
{
      final double PI= 3.1416;
      abstract double area();
}

class Triangle extends Shape
{
      int b, h;
      Triangle(int x, int y)      //constructor
      {
            b=x;
            h=y;
      }

      double area()    //method overriding
      {
            System.out.print("\nArea of Triangle is:");
            return 0.5*b*h;
```

```
        }
}

class Circle extends Shape
{
        int r;
```

```
        Circle(int rad)         //constructor
        {
              r=rad;
        }

        double area()           //overriding
        {
              System.out.print("\nArea of Circle is:");
              return PI*r*r;
        }
  }

  class Rectangle extends Shape
  {
        int a, b;
        Rectangle(int x, int y)      //constructor
        {
              a=x;
              b=y;
        }
        double area()           //overriding
        {
              System.out.print("\nArea of Rectangle is:");
              return a*b;
        }
  }

  class AbstractDemo
  {
     public static void main(String args[])
    {
       Shape r[]={new Triangle(3,4), new Rectangle(5,6),new Circle(2)};

        for(int i=0;i<3;i++)
              System.out.println(r[i].area());
     }
  }
```

Output:

    Area of Triangle is:6.0
    Area of Rectangle is:30.0
    Area of Circle is:12.5664

Note that, here we have created array *r*, which is reference to *Shape* class. But, every element in *r* is holding objects of different subclasses. That is, r[0] holds *Triangle* class object, r[1] holds *Rectangle* class object and so on. With the help of array initialization, we are achieving this, and also, we are calling respective constructors. Later, we use a *for-loop* to invoke the method *area()* defined in each of these classes.

Using *final*

The keyword *final* can be used in three situations in Java:

- To create the equivalent of a named constant.
- To prevent method overriding
- To prevent Inheritance

To create the equivalent of a named constant: 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;
```

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.

To prevent method overriding: Sometimes, we do not want a superclass method to be overridden in the subclass. Instead, the same superclass method definition has to be used by every subclass. In such situation, we can prefix a method with the keyword *final* as shown below –

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

To prevent Inheritance: As we have discussed earlier, the subclass is treated as a specialized class and superclass is most generalized class. During multi-level inheritance, the bottom most class will be with all the features of real-time and hence it should not be inherited further. In such situations, we can prevent a particular class from inheriting further, using the keyword *final*. For example –

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

Note:

- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations

The Object Class

There is one special class, ***Object***, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array. Object defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.

void wait( ) void wait(long milliseconds) void wait(long milliseconds, int nanoseconds)	Waits on another thread of execution.

The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may override the others. The **equals( )** method compares the contents of two objects. It returns **true** if the objects are equivalent, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println( )**. Many classes override this method.

Interfaces

Interface is an abstract type that can contain only the declarations of methods and constants. Interfaces are syntactically similar to classes, but they do not contain instance variables, and their methods are declared without any body. Any number of classes can implement an interface. One class may implement many interfaces. By providing the interface keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism. Interfaces are alternative means for multiple inheritance in Java.

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

```
Access interfacename
{
typefinal-varname1=value;  typefinal-varname2=value;
…………………
return-typemethod-name1(parameter-list);  return-typemethod-name2(parameter-list);
…………………
}
```

Few key-points about interface:

• When no access specifier is mentioned for an interface, then it is treated as default and the interface is only available to other members of the package in which it is declared. When an interface is declared as public, the interface can be used by any other code.

• All the methods declared are abstract methods and hence are not defined inside interface. But, a class implementing an interface should define all the methods declared inside the interface.

- Variables declared inside of interface are implicitly final and static, meaning they cannot be changed by the implementing class.
- All the variables declared inside the interface must be initialized.
- All methods and variables are implicitly public.

Implementing Interface

To implement an interface, include the implements clause in a class definition, and then create the methodsdefined by the interface.The generalform of a class that includesthe implements clause looks like this:

```
Class classname extends superclass implements interface1,interface2...
{
//class-body
}
```

Consider the following example:

```
interface Callback
{
Void callback(int param);
}

Class Client implements Callback
{
Public void callback(int p)      //note public
{
System.out.println("call back called with"+p);
}
Void test()
{
System.out.println("ordinary method");
}
}
Class TestIface
{
Public static void main(String args[])
{
Callback c = new Client();
 c.callback(42);

//  c.test()          //error!!
```

```
}
}
```

Here, the interface Callback contains declaration of one method callback(). The class Client implementing this interface is defining the method declared in interface. Note that, the method callback() is public by default inside the interface. But, the keyword public must be used while defining it inside the class. Also, the class has its own method test(). In the main() method, we are creating a reference of interface pointing to object of Client class. Through this reference, we can call interface method, but not method of the class.

The true polymorphic nature of interfaces can be found from the following example–

```
Interface Callback
{
Void callback(int param);
}

Class Client implements Callback
{
Public void callback(int p)       //notepublic
{
System.out.println("callbackcalledwith"+p);
}
}

classClient2implementsICallback
{
Public void callback(int p)
{
System.out.println("Another version of ICallBack");

 System.out.println("p squared " + p*p);
}
}

classTest Iface
{
Public static void main(String args[])
{
ICallback x[] = { newClient(),newClient2()};

for(int i=0;i<2;i++)
```

```
x[i].callback(5);
}
}
```

Output:
callback called with 5 Another version of ICallBack p squared 25

In this program, we have created array of references to interface, but they are initialized to class objects. Using the array index, we call respective implementation of callback() method.

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class all of those variable names will be in scope as constants (Similar to #define in C/C++). If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is just using a set of constants. Consider an example to illustrate the same:

```
Interface SharedConst
{
int FAIL=0;        //these are final by default int PASS=1;
}

Class Result implements SharedConst
{
double mr;

Result(double m)
{
mr = m;
}

int res()
{
if(mr<40)
return FAIL;
```

```
else return PASS;
}
}
Class Exam extends Result implements SharedConst
{
Exam(double m)
{
super(m);
}

Public static void main(String args[])
{
Exam r = new Exam(56);

switch(r.res())
{
Case FAIL:
System.out.println("Fail"); break;
Case PASS:
System.out.println("Pass"); break;
}
}
}
```

Interfaces can be extended

One interface can inherit another interface by using the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
Interface A
{
void meth1();
void meth2();
}

Interface B extends A
{
void meth3();
}
```

```
Class MyClass implements B
{
Public void meth1()
{
System.out.println("Implement meth1().");
}
Public void meth2()
{
System.out.println("Implement meth2().");
}
Public void meth3()
{
System.out.println("Implement meth3().");
}
}

Class IFExtend
{
Public static void main(String arg[])
{
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Question Bank:

1. Define class. Give syntax and example.
2. Briefly explain *static* members of the class with suitable examples.
3. Discuss method overloading. Write a program to overload a method *area()* to compute area of a triangle and a circle.
1. How do you overload a constructor? Explain with a program.
2. Define recursion. Write a recursive program to find nth Fibonacci number.
3. Write a program to implement stack operations.
4. What are different parameter passing techniques in Java? Discuss the salient features of the same.
5. What are various access specifiers in Java? List out the behaviour of each of them.
6. Create a Java class called Student with the following details as variables (USN, Name, Branch, Phone Number). Write a Java program to create n student objects and print USN, Name, Branch, and Phone number with suitable heading.
7. What is inheritance? Discuss different types of inheritance with suitable example.
8. Discuss the behavior of constructors when there is a multilevel inheritance. Give appropriate code to illustrate the process.
9. Mention and explain the uses of *super* keyword in Java.
10. How do you pass arguments to superclass constructor through the subclass constructor? Explain with a code snippet.
11. Discuss usage of *final* keyword in Java. Give suitable examples.
12. What do you mean by method overriding? Discuss with a programming example.
13. Explain abstract class and abstract method with suitable code snippet.
14. Write a note on:
    a. Use of *this* keyword
    b. Garbage Collection in Java
    c. Finalize() method
    d. Object Class
    e. Dynamic Method Dispatch
15. Create an abstract class called *Employee*. Include the members: Name, EmpID and an abstract method *cal_sal()*. Create two inherited classes SoftwareEng (with the members basic and DA) and HardwareEng (with members basic and TA). Implement runtime polymorphism (dynamic method dispatch) to display salary of different employees by creating array of references to superclass.
16. Differentiate method overloading and method overriding.