

1.0 Introduction to Object Oriented Programming

The 1960s gave birth to structured programming. This is the method of programming championed by languages such as C. With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980's. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired result in terms of bug-free, easy-to-maintain, and reusable programs. To solve these problems, a new way to program was invented, called *object-oriented programming* (OOP).

Object Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

1.1 PO Programming Vs OO Programming

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around — “*what is happening*” and others are written around — “*who is being affected*”. These are the two paradigms that govern how a program is constructed. The first way is called the *process-oriented* model or *procedural* language. This approach characterizes a program as a series of linear steps (i.e. code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success.

To manage increasing complexity, the second approach, called *object-oriented* programming, was designed. Object-oriented programming organizes a program around its data (i.e. objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

Difference between POP and OOP

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	Program is divided into small parts called <i>functions</i> .	Program is divided into parts called <i>objects</i> .
Importance	Importance is given to functions as well as sequence of actions to be done.	Importance is given to the data rather than procedures or functions because it works as a real world.
Approach	It follows Top-down approach.	It follows Bottom-up approach.
Data Moving	Data can move freely from function to function in the system.	Objects can communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	It provides an easy way to add new data and function.

Data Access	Most function uses Global data for sharing that can be accessed freely from function to function in the system.	Data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	It does not have any proper way for hiding data so it is less secure.	It supports data hiding so provides more security.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Inheritance	No such concept of inheritance in POP	Inheritance is allowed in OOP
Access Specifiers	It does not have any access specifiers.	It has access specifiers named Public, Private, Protected.
Examples	C, BASIC, FORTRAN, Pascal, COBOL.	C++, JAVA, C#, Smalltalk, Action Script.

1.2 Principles of OOP

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

- Object
- Class
- Encapsulation
- Data abstraction
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Object

Object is the basic run time entity in an object-oriented system. It may represent a person, a place, a bank account, a table of data, vectors, time and lists. Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

In OOP, A problem is analyzed in term of objects and the nature of communication between them. Objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a *structure* in C. When a program is executed, the objects interact by sending messages to one another.

For example, if “customer” and “account” are to object in a program, then the *customer* object may send a message to the *account* object requesting for the bank balance. Each object contain data, and code to manipulate data. Below figure shows two notations that are popularly used in object oriented analysis and design.

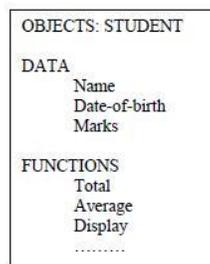


Fig. representing an object

Class

A class is a collection of similar objects that share the same properties, methods, relationships and semantics. A class can be defined as a template/blueprint that describes the behavior/state of the object.

Encapsulation

Definition: The process binding (or wrapping) code and data together into a single unit is known as Encapsulation. For example: capsule, it is wrapped with different medicines.

- Java supports the feature of Encapsulation using *classes*.
- The data in the class is not accessed by outside world.
- The functions that are defined along with the data within the same class are allowed to access the data.
- These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.
- A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the same structure and behavior defined by the class.
- The objects are referred to as instances of a class. Thus, a class is a logical construct; an object is physical reality.

Data abstraction

Definition: It a process of providing essential features without providing the background or implementation details. For example: It is not important for the user how TV is working internally, and different components are interconnected. The essential features required to the user are how to start, how to control volume, and change channels.

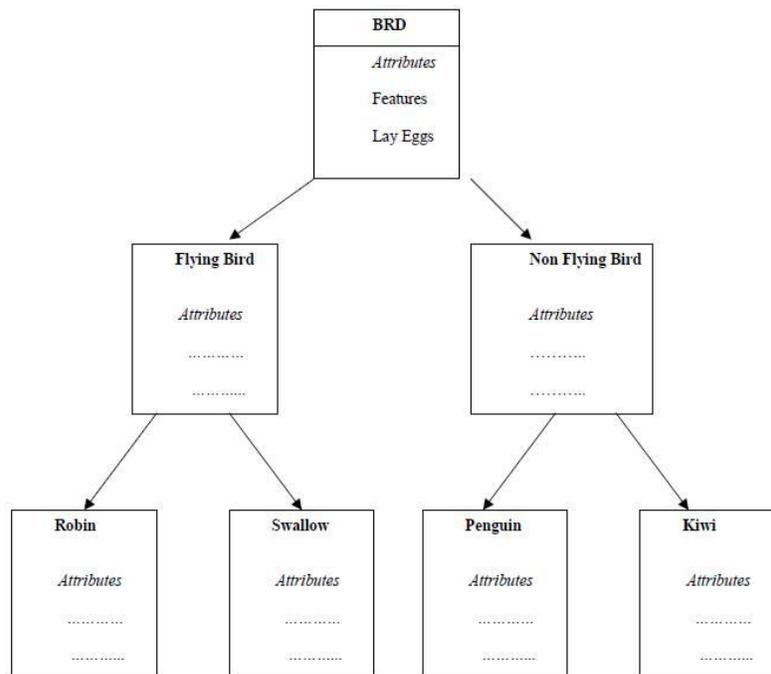
In java, we use *abstract class* and *interface* to achieve abstraction.

Inheritance

It a process by which an object of one class acquires the properties and methods of another class. It supports the concept of *hierarchical classification*. For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in below figure.

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

acquires the properties is called *subclass*. This is mainly used for *Method Overloading* and *Code reusability*.



Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation.

In Java, we use *method overloading* and *method overriding* to achieve polymorphism.

For example can be to speak something e.g. cat speaks meow, dog barks woof, duck speaks quack, etc.

Advantages of OOP

OOP offers several benefits to both the program designer and the user. Object-Oriented contributes to the solution of many problems associated with the development and quality of Software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- It presents a simple, clear and easy to maintain structure.
- Software complexity can be easily managed.
- It enhances program modularity since each object exists independently.
- New features can be easily added without disturbing the existing one.
- Objects can be reused in other program.
- It allows designing and developing safe programs using the data hiding.
- Through inheritance, we can eliminate redundant code extend the use of existing classes.
- It is easy to partition the work in a project based on objects.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.

Applications of OOP

Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as window. Hundreds of windowing systems have been developed, using the OOP techniques. The promising areas of application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented data bases
- Hypertext, Hypermedia, and Expertext
- AI and expert systems
- Neural networks and Parallel programming
- Decision support and Office automation systems
- CAM/CAD Systems

1.3 What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, secured and object-oriented programming language.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

Where it is used (Applications)?

According to Sun, 3 billion devices run java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop GUI Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javabay.com etc.
3. Enterprise Applications such as banking applications.

4. Mobile Applications
5. Embedded Systems
6. Smart Cards
7. Robotics
8. Games etc.

Types of Java Applications

There are mainly 4 types of applications that can be created using java programming:

1. **Standalone Application**

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2. **Web Application**

An application that runs on the server side and creates dynamic page, is called web application. Currently, *servlet*, *jsp*, *struts*, *jsf* etc. technologies are used for creating web applications in java.

3. **Enterprise Application**

An application that is distributed in nature such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4. **Mobile Application**

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1. **Java SE (Java Standard Edition)**

It is a java programming platform. It includes Java programming APIs such as *java.lang*, *java.io*, *java.net*, *java.util*, *java.sql*, *java.math* etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection etc.

2. **Java EE (Java Enterprise Edition)**

It is an enterprise platform which is mainly used to develop web and enterprise applications. It is built on the top of Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA etc.

3. **Java ME (Java Micro Edition)**

It is a micro platform which is mainly used to develop mobile applications.

4. **JavaFx**

It is used to develop rich internet applications. It uses light-weight user interface API.

History of Java

The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.



James Gosling



Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describe the history of java.

1. **James Gosling, Mike Sheridan, and Patrick Naughton** initiated the Java language project in June 1991 at *Sun Microsystems*. The small team of sun engineers called **Green Team**.
2. Originally designed for small, embedded systems in electronic appliances like set-top boxes.
3. Firstly, it was called "**Greentalk**" by James Gosling and file extension was **".gt"**.
4. After that, it was called **Oak** and was developed as a part of the Green project.
5. **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.
6. In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why "Java" name

7. Why had they chosen *Java* name for java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling "Java was one of the top choices along with Silk".

Since java was so unique, most of the team members preferred *Java*.

8. *Java* is an *island* of *Indonesia* where first coffee was produced (called java coffee).
9. Notice that Java is just a name not an acronym.
10. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
11. In 1995, Time magazine called Java one of the Ten Best Products of 1995.
12. JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that have been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. Initial Java Versions 1.0 and 1.1 were released in the year 1996 for Linux, Solaris, Mac and Windows.

3. J2SE 1.2 (Commonly called as Java 2) was released in the year 1998.
4. J2SE 1.3 codename Kestrel was released in the year 2000.
5. J2SE 1.4 codename Merlin was released in the year 2002.
6. J2SE 5.0 codename 'Tiger' was released in the year 2004.
7. Java SE 6 codename 'Mustang' was released in the year 2006.
8. Java SE 7 codename 'Dolphin' was released in the year 2011.
9. Java SE 8 is the current stable release which was released this year 2014.

Five Goals which were taken into consideration while developing Java

1. Keep it simple, familiar and object oriented.
2. Keep it Robust and Secure.
3. Keep it architecture-neutral and portable.
4. Executable with High Performance.
5. Interpreted, threaded and dynamic.

Why we call it Java 2, Java 5, Java 6, Java 7 and Java 8, not their actual version numbers which 1.2, 1.5, 1.6, 1.7 and 1.8?

Java 1.0 and 1.1 were Java. When Java 1.2 was released it had a lots of changes and marketers / developers wanted a new name so they called it Java 2 (**J2SE**), remove the numeric before decimal. This was not the condition when Java 1.3 and Java 1.4 were released hence they were never called Java 3 and Java 4, but they were still Java 2.

When Java 5 was released, once again it was having a lots of changes for the developer / marketers and need a new name. The next number in sequence was 3, but calling Java 1.5 as Java 3 was confusing hence a decision was made to keep the naming as per version number and till now the legacy continues.

1.4 Java Features / Buzz words

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

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



Fig. Java features

Simple

Java inherits all the best features from the programming languages like C, C++ and thus makes it really easy for any developer to learn with little programming experience. Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Secure

When Java programs are executed they don't instruct commands to the machine directly. Instead Java Virtual machine reads the program (*Byte code*) and convert it into the machine instructions. This way any program tries to get illegal access to the system will not be allowed by the JVM. Allowing Java programs to be executed by the JVM makes Java program fully secured under the control of the JVM.

Portable

Java programs are portable because of its ability to run the program on any platform and no dependency on the underlying hardware / operating system.

Object Oriented

Everything in Java is an Object. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

Robust

The multi-platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. Following features of Java make it Robust.

- Platform Independent
- Object Oriented Programming Language

- Memory management
- Exception Handling

Multithreaded

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

Architecture-neutral

The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was — *write once; run anywhere, anytime, forever*. To a great extent, this goal was accomplished.

Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic

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

1.5 Java Virtual Machine (JVM)

The key that allows Java to solve both the security and the portability problems is the byte code. The output of *Java Compiler* is not directly executable file. Rather, it contains highly optimized set of instructions. This set of instructions is called, "**byte code**". This byte code is designed to be executed by *Java Virtual Machine (JVM)*. The JVM also called as the *interpreter* for byte code.

JVM also helps to solve many problems associated with web-based programs. Translating a Java program into byte code makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given Figure 2 JVM system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to

platform, all understand the same Java byte code. Thus, the execution of byte code by the JVM is the easiest way to create truly portable programs.

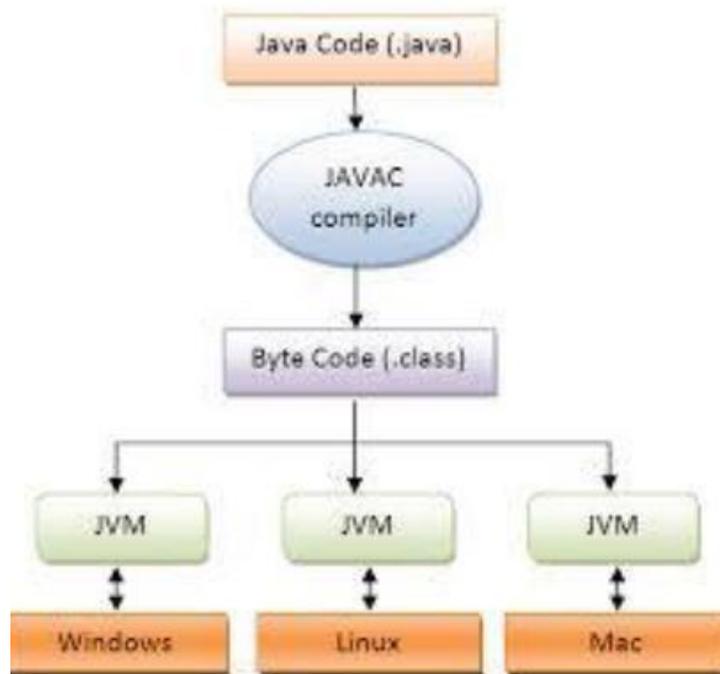


Figure 2 JVM

Internal Architecture of JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

- JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).
- It is a specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- Its implementation is known as JRE (Java Runtime Environment).
- Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

The JVM performs following operations:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

Let's understand the internal architecture of JVM. It contains class loader, memory area, execution engine etc.

1. Classloader

Classloader is a subsystem of JVM that is used to load class files.

2. Class(Method) Area

Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

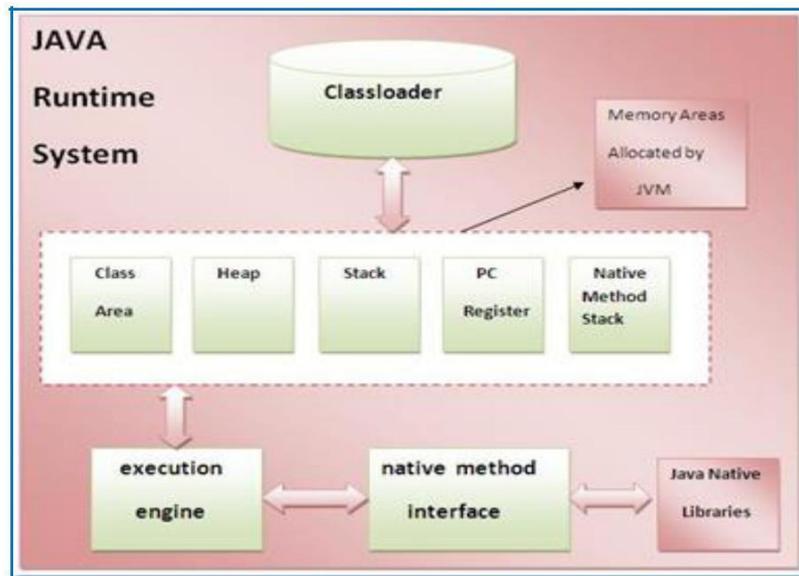


Fig. Internal architecture of JVM

3. Heap

It is the runtime data area in which objects are allocated.

4. Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5. Program Counter (PC) Register

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6. Native Method Stack

It contains all the native methods used in the application. A **native method** is a **Java method** (either an instance **method** or a class **method**) whose implementation is written in another programming language such as C.

7. Execution Engine

It contains:

- a. A virtual processor

- b. *Interpreter*: Read bytecode stream then execute the instructions.
- c. *Just-In-Time(JIT) compiler*: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.

1.6 Basic Structure of Java Program

Basically, a Java program involves the following sections:

- Documentation Section
- Package Statement
- Import Statements
- Interface Statement
- Class Definition
- Main Method Class
- Main Method Definition

Section	Description
Documentation Section	It comprises of a comment line which gives the names program, the programmer's name and some other brief details. Java provides 3 styles of comments <ol style="list-style-type: none"> 1. Single line (//) 2. Multi-line(/* */) 3. Documentation comment (/**...*/)
Package statement	The first statement allowed in Java file is the Package statement which is used to declare a package name and it informs the compiler that the classes defined within the program belong to this package. It is declared as: <i>package package_name;</i>
Import statements	The next is the number of import statements, which is equivalent to the #include statement in C++. Example: <i>import java.util.Scanner;</i>
Interface statement	Interfaces are like a class that includes a group of method declarations. This is an optional section and can be used only when programmers want to implement multiple inheritances within a program.
Class Definition	A Java program may contain multiple class definitions. Classes are the main and important elements of any Java program. These classes are used to plot the objects of the real world problem.
Main Method Class	Since every Java stand-alone program requires the main method as the starting point of the program. This class is essentially a part of Java program. A simple Java program contains only this part of the program.

To create a simple java program, you need to create a class that contains main method. Let's understand the requirement first.

Requirement for Welcome Java Example

For executing any java program, you need to

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the *jdk/bin* directory.
- Create the java program
- Compile and Run the java program

Sample Code of “Welcome” Java program

Example: **Welcome.java**

```
class Welcome
{
    public static void main(String[] args)
    {
        System.out.println("Welcome to Java");
    }
}
```

Output:

Welcome to Java

Entering the Program

- We can use any text editor such as "notepad" or "dos text editor".
- The source code is typed and is saved with ".java" as extension.
- The source code contains one or more class definitions. The program name will be same as class name in which main function is written. This is not compulsory, but by convention this is used.
- The source file is officially called as compilation unit.
- We can even use our choice of interest name for the program. If we use a different name than the class name, then compilation is done with program name, and running is done with class file name. To avoid this confusion and organize the programs well, it is suggested to put the same name for the program and class name, but not compulsory.

Compiling the Program

To compile the program, specifying the name of the source file on the command line, as shown below:

```
C:/>javac Welcome.java
```

The *javac* compiler creates the file called "*Welcome.class*" that contains the byte code version of the source code. This byte code is the intermediate representation of the source code that contains the instructions that the Java Virtual Machine (JVM) will execute. Thus the output of the *javac* is not the directly executable code.

Running the Program

To run the program, we must use Java interpreter, called "*java*". This is interpreter the "*Welcome.class*" file given as input.

```
C:/>java Welcome
```

When the program is run with java interpreter, the following output is produced: *Welcome to Java*

Description of the program

The *first line* contains the keyword *class* and class name, which actually the basic unit for encapsulation and is used to declare a class in java, in which data and methods are declared.

Second line contains "{" which indicates the beginning of the class.

Third line contains the *public static void main(String args[])*

where

- *public* keyword is an access modifier which represents visibility, it means it is visible to all.
- *static* is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- *void* is the return type of the method, it means it doesn't return any value.
- *main* represents startup of the program where execution starts.
- *String[] args* is used to read command line argument.

Fourth line contains the "{", which is the beginning of the main function.

Fifth line contains the statement

```
System.out.println("Hello World");
```

Here "*System*" is the predefined class, that provides access to the system, and *out* is the output stream that is used to connect to the console. The *println()*, is used to display string passed to it. This can even display other information to.

1.7 Lexical issues

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

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the Java program could be written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number. Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are

AvgTemp	count	a4	\$test	this_is_ok
---------	-------	----	--------	------------

Invalid identifier names include these:

High-temp	2count	4a	Not/ok
-----------	--------	----	--------

Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	'C'	"This is a test"
-----	------	-----	------------------

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string.

A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This type of 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 separators are shown in the following table:

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

The Java Keywords

- There are 50 keywords currently defined in the Java language (see below 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 identifiers. Thus, they cannot be used as names for a variable, class, or method.
- The keywords *const* and *goto* are reserved but not used.
- In addition to the keywords, Java reserves the following: **true**, **false**, and **null**. These are values defined by Java.

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

Table: Java Keywords

Operators

An operator performs an operation on one or more operands. Java provides a rich set of operators. An operator that performs an operation on one operand is called unary operator. An operator that performs an operation on two operands is called binary operator.

1.8 Variables

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

<i>type identifier [= value/literal][, identifier [= value/literal] ...] ;</i>

Here the *type* is any primitive data type, or class name. The *identifier* is the name of the variable. We can initialize the variable by specifying the equal sign and value.

Example

```
int a, b, c;           // declares three ints, a, b, and c.
int d = 3, e, f = 5;  // declares three more ints, initializing d and f.
byte z = 22;         // initializes z with 22.
double pi = 3.1415;  // declares an approximation of pi.
char c = 'x';        // variable c has the value 'x'
```

Dynamic Initialization of the variable

We can also assign a value to the variable dynamically as follow:

```
int x = 12, y=13;
float z = Math.sqrt (x+y);
```

The Scope and Lifetime of Variables

- A *scope* determines what objects are visible to parts of your program. It also determines the life time of the objects.
- Java allows, declaring a variable within any block.

- A block begins with opening curly brace({) and ended with end curly brace (}).
- Thus, each time we start new block, we create new scope.
- Many programming languages define two scopes: Local and Global
- As a general rule a variable defined within one scope, is not visible to code defined outside of the scope.
- Scopes can be also nested. The variable defined in outer scope are visible to the inner scopes, but reverse is not possible.

Example code

```

void method1( )
{
    //outer block
    int a =10;
    if(a==10)
    {
        // inner block and here a,b,c are visible to the inner scope
        int b=a*20;
        int c=a+30;
    }
    //end of inner block
    b=20*2;
    // b is not known here, which declared in inner scope
}
//end of the outer block

```

1.9 Identifiers

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

Rules for Naming Identifier:

1. The first character of an identifier must be a letter, or dollar(\$) sign.
2. The subsequent characters can be letters, an underscore, dollar sign or digit.
3. White spaces are not allowed within identifiers.
4. Identifiers are case sensitive so *VALUE* is a different identifier than *Value*

Valid Identifiers are as follows:

Average Temperature A1 Total_Score

Invalid Identifiers are as follows:

2a Area-circle Not/ok

Naming Convention for Identifiers

- *Class or Interface* - These begin with a capital letter. The first alphabet of every internal word is capitalized. Ex: class MyClass
- *Variable or Method* – These start with lower case letters. The first alphabet of every internal word is capitalized. Ex: int totalPay;

- *Constants* - These are in upper case. Underscore is used to separate the internal word.
Ex:-final double PI=3.14;
- *Package* – These consist of all lower-case letters. Ex: import java.util.*;

1.10 Data Types

Java is strongly typed language

Java is strongly typed language. The safety and robustness of the Java language is in fact provided by its strict typing. There are two reasons for this: First, every variable and expression must be defined using any one of the type. Second, the parameters to the method also should have some type and also verified for type compatibility.

Java language has 8 primitive data types: They are: *char, byte, short, int, long, float, double, boolean*. These are again categorized into 4 groups.

1. Integer Group: The integer group contains *byte, short, int, long*. These data types will need different sizes of the memory. These are assigned positive and negative values. The width and ranges of these values are as follow:

byte

- The smallest integer type is *byte*.
- This is a signed 8-bit type that has a range from –128 to 127.
- Variables of type *byte* are especially useful when you're working with a stream of data from a network or file and 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, the following declares two *byte* variables called *b* and *c*:

```
byte b, c; // where b and c are identifiers
```

short

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

Here are some examples of *short* variable declarations:

```
short s, t;
```

int

- The most commonly used integer type is *int*.
- It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- In addition to other uses, variables of type *int* are commonly employed to control loops and to index arrays.
- We can store *byte* and *short* values in an *int*.

Example: `int x=12;`

long

- *long* is a signed 64-bit type and is useful for those occasions where an *int* type is not large enough to hold the desired value.
- The range of a *long* is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

- This makes it useful when big, whole numbers are needed.

Example: long x=123456;

2. Floating-Point Group

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. These are used with operations such as square root, cosine, and sine etc. There are two types of Floating-Point numbers:

1. float
2. double.

The float type represents *single precision* and double represents *double precision*. **float**

- The type float specifies a single-precision value that uses 32 bits of storage.
- The range of a float is 1.4e-045 to 3.4e+038
- Single precision is faster on some processors and takes half as much space as double precision.
- Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.

Example

float height, price;

double

- Double precision, as denoted by the double keyword, uses 64 bits to store a value.
- The range of a *double* is 4.9e-324 to 1.8e+308
- Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.
- All the math functions, such as sin(), cos(), and sqrt(), return double values.

Example:

double area, pi;

Example program to calculate the area of a circle

```
import java.io.*;
class Circle {
    public static void main(String args[ ] ) {
        double r, area, pi;
        r=2.5;
        pi=3.14;
        area=pi*r*r;
        System.out.println("The Area of the Circle is:"+area);
    }
}
```

Output

The Area of the Circle is: 19.625

3. Characters Group

In Java, the data type used to store characters is *char*. Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.

Java char is a 16-bit type. 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.

Here is a program that demonstrates char variables:

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

Output

ch1 and ch2: X Y

4. Booleans Group

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

Here is a program that demonstrates the boolean type:

```
class BoolTest {
    public static void main(String args[])
    {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if
        statement if(b)
            System.out.println("This is executed.");
        else
            System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

Output

b is false

b is true

This is executed.

10>9 is true

1.11 Literals/Constants

A *literal* is a value that can be passed to a variable or constant in a program. Literals can be numeric, boolean, character, string or null.

Integer Literals

- Integers are the most commonly used type in the typical program.
- Any whole number value is an integer literal.
- Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number.
- Octal values are denoted in Java by a leading zero.
- Octal numbers are ranging from 0 to 7 range. A more common base for numbers used by programmers is hexadecimal,

which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant 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. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as byte or long, without causing a type mismatch error. Fortunately, such situations are easily handled. 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.

1.12 Operators

An operator is defined as a symbol that operates on operands and does something. The something may be mathematical, relational or logical operation. Java supports a lot of operators to be used in expressions. These operators can be categorized into the following groups:

1. Arithmetic operators
2. Bitwise operators
3. Relational operators and
4. Logical operators
5. Miscellaneous operators

Arithmetic Operators

- These are used to perform mathematical operations.
 - Most of them are binary operators since they operate on two operands at a time except unary minus and plus.
 - They can be applied to any integers, floating-point number or characters. •
- Java supports 5 arithmetic operators. They are +, -, *, /, %.
- The modulo (%) operator can only be applied to integer operands as well as double operands.

The following table lists the arithmetic operators:

Operator	Result
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Quotient division
%	Modulo Division
++	Increment
--	Decrement
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

Example program to perform all the arithmetic operations (Arith.java)

```
import java.io.*;
class Arith
{
    public static void main(String args[])
    {
        int a,b;
        a=5;
        b=6;
        int c=a+b;           //arithmetic addition
        int d=a-b;           //arithmetic subtraction
        int e=a/b;           //arithmetic division
        int f=a*b;           //arithmetic multiplication
        System.out.println("The Sum is :"+c);
        System.out.println("The Subtraction is :"+d);
        System.out.println("The Division is :"+e);
        System.out.println("The Multiplication is :"+f);
    }
}
```

Output

```
The Sum is : -1
The Subtraction is : -1
The Division is : 0
The Multiplication is : 30
```

The Modulus Operator (%)

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

```
// Demonstrate the % operator.
(Modulus.java) class Modulus
{
    public static void main(String args[])
    {
        int x = 42;
        double y = 6.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 2.5 = " + y % 2.5);
    }
}
```

Output

```
x mod 10 = 2
y mod 2.5 = 1.25
```

Arithmetic Compound Assignment Operators

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

$$a = a + 4;$$

In Java, you can rewrite this statement as shown here: $a += 4;$

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

Here is another example, $a = a \% 2;$ which can be expressed as $a \%= 2;$

Increment and Decrement Operators

- The $++$ and the $--$ are Java's increment and decrement operators.
- The increment operator is unary operator that increases value of its operand by 1. Similarly, the decrement operator decreases value of its operand by 1.
- These operators are unique in that they work only on variables not on constants.
- These are used in loops like while, do-while and for statements.

There are *two* ways to use increment or decrement operators in expressions. If you put the operator in front of the operand (*prefix*), it returns the new value of the operand (incremented or decremented). If you put the operator after the operand (*postfix*), it returns the original value of the operand (before the increment or decrement)

For example, this statement:

$$x = x + 1;$$

can be rewritten like this by use of the increment operator: $x++;$

Operator	Name	Value returned	Effect on variable
$x++$	Post-increment	x	Incremented
$++x$	Pre-increment	$x+1$	Incremented
$x--$	Post-decrement	x	Decrement
$--x$	Pre-decrement	$x-1$	Decrement

The following program demonstrates the increment and decrement operator. (IncDec.java)

```
class IncDec
{
    public static void main(String args[])
    {
        int x=5;
        System.out.println("x="+x++);    // post increment
        System.out.println("x="+(++x));  // pre increment
        System.out.println("x="+x--);    // post decrement
        System.out.println("x="+(--x));   // pre decrement
    }
}
```

Output

x = 5

x = 7

x = 7

x = 5

The Bitwise Operators

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

Operator	Result
~	Bitwise unary NOT (Complement)
&	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

These operators are again classified into two categories: Logical operators, and Shift operators.

The Bitwise Logical Operators

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

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

The Bitwise NOT (~)

One's complement operator (Bitwise NOT) is used to convert each 1 to 0 and 0 to 1, in the given binary pattern. It is a unary operator i.e. it takes only one operand. For example, the number 56, which has the following bit pattern:

$$\begin{array}{r} 00111000 \quad \rightarrow 56 \\ \hline \sim 11000111 \quad \rightarrow -57 \end{array}$$

The Bitwise AND (&)

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

$$\begin{array}{r} 00111000 \quad \rightarrow 56 \\ \& 00010100 \quad \rightarrow 20 \\ \hline 00010000 \quad \rightarrow 16 \end{array}$$

The Bitwise OR(|)

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

$$\begin{array}{r} 00111000 \quad \rightarrow 56 \\ | 00010100 \quad \rightarrow 20 \\ \hline 00111100 \quad \rightarrow 60 \end{array}$$

The Bitwise XOR (^)

The bitwise XOR operator combines bits such that if both operands are different then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^.

$$\begin{array}{r} 00111000 \quad \rightarrow 56 \\ ^ 00010100 \quad \rightarrow 20 \\ \hline 00101100 \quad \rightarrow 44 \end{array}$$

The following program demonstrates the bitwise logical operators:

Example: (BitwiseOp.java)

```
class BitwiseOp
{
    public static void main(String args[])
    {
        int x = 56,y=20,z;
        int a = x&y;
        int b = x|y;
        int c = x^y;
    }
}
```

```

        int d = ~x;
        System.out.println("x = " +Integer.toBinaryString(x));
        System.out.println("y = "+Integer.toBinaryString(y));
        System.out.println("x&y = "+Integer.toBinaryString(a));
        System.out.println("x|y = "+Integer.toBinaryString(b));
        System.out.println("x^y = "+Integer.toBinaryString(c));
        System.out.println("~x = "+Integer.toBinaryString(d));
    }
}

```

Output

```

x = 00111000
y = 00010100
x&y = 00010000
x|y = 00111100
x^y = 00101100
~x = 11000111

```

Bitwise Shift Operators

Java supports three bitwise shift operators. They are shift-left (<<) and shift-right (>>) and unsigned shift-right(>>>). These operations are simple and are responsible for shifting bits either to left or to the right. The syntax for shift operation can be given as:

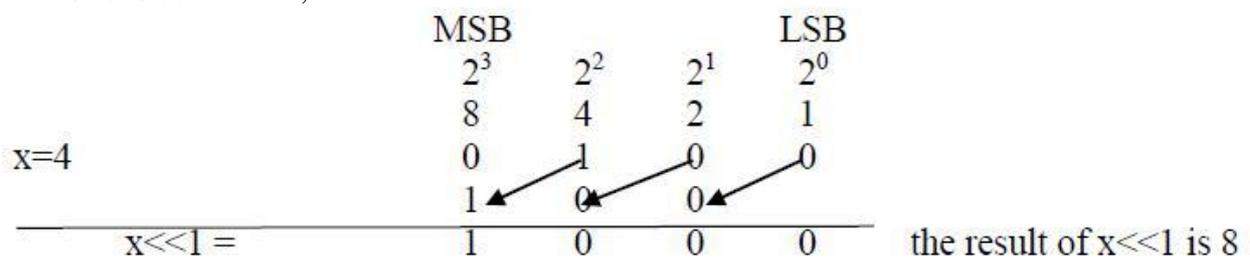
operand operator num

where the bits in *operand* are shifted left or right depending on the *operator* (<<, >>) by the number of places denoted by *num*.

The Left Shift (<<)

When we apply left-shift, every bit in x is shifted to left by one place. So, the MSB (Most Significant Bit) of x is lost, the LSB (Least Significant Bit) of x is set to 0.

Let us consider `int x=4;`

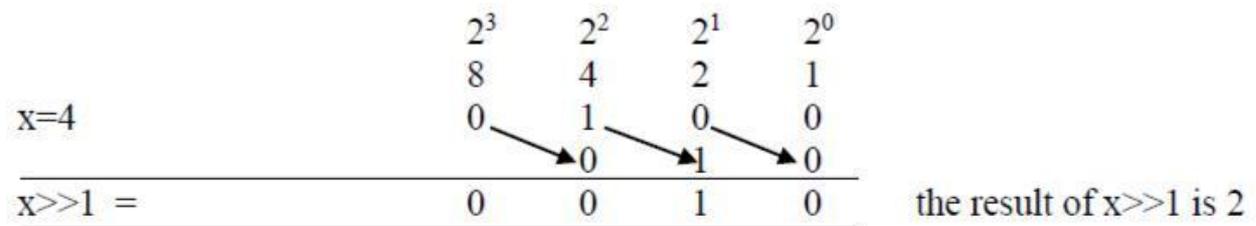


Left-shift is equals to multiplication by 2.

The Right Shift (>>)

When we apply right-shift, every bit in x is shifted to right by one place. So, the LSB (Least Significant Bit) of x is lost, the MSB (Most Significant Bit) of x is set to previous value. This preserves the sign of the value.

Let us consider `int x=4;` Now shifting the bits towards right for 1 time, will give the following result.



Right-shift is equals to division by 2.

The following program demonstrates the bitwise logical operators:

Example: (ShiftOp.java)

```
class ShiftOp
{
    public static void main(String args[])
    {
        int x = 4;
        System.out.println("x<<1 = " +x);
        System.out.println("x<<1 = " +(x<<1));
        System.out.println("x>>1 = "+(x>>1));
        x=-4;
        System.out.println("x>>>1 = "+(x>>>1));
    }
}
```

Output

```
x = 4
x<<1 = 8
x>>1 = 2
x>>>1 = 2147483646
```

Relational Operators

A relational operator, also known as a comparison operator, is an operator that compares two operands. The operands can be variables, constants or expressions. Relational operators always return either true or false depending on whether the conditional relationship between the two operands holds or not.

The outcome of these operators is a *Boolean* value. The relational operators are most frequently used in the expressions to control *if* and *loop* statements.

Java has six relational operators. The following table shows these operators along with their meanings

Operator	Meaning	Example
<	Less than	4<5 return <i>true</i>
>	Greater than	4>5 return <i>false</i>
<=	Less than or equal to	100<=100 return <i>true</i>
>=	Greater than or equal to	50>=100 return <i>false</i>
==	Equal to	4==5 return <i>false</i>
!=	Not equal to	4!=5 return <i>true</i>

The following program demonstrates the Relational operators:

Example: (RelOp.java)

```
class RelOp
{
    public static void main(String args[])
    {
        int x = 4,y=5;
        System.out.println(x + "<" + y + " = " + (x<y));
        System.out.println(x + ">" + y + " = " + (x>y));
        System.out.println(x + "<=" + y + " = " + (x<=y));
        System.out.println(x + ">=" + y + " = " + (x>=y));
        System.out.println(x + "==" + y + " = " + (x==y));
        System.out.println(x + "!=" + y + " = " + (x!=y));
    }
}
```

Output

```
4<5 = true
4>5 = false
4<=5 = true
4>=5 = false
4==5 = false
4!=5 = true
```

Logical Operators (|| and &&)

Operators which are used to combine two or more relational expressions are known as logical operators. Java supports three logical operators – logical AND(&&), logical OR(||), logical NOT(!). These are also known as *short-circuit* logical operators.

- Logical && and logical || are binary operators whereas logical ! is an unary operator.
- All of these operators when applied to expressions yield either true or false.
- When we use || operator if left hand side expression is true, then the result will be true, no matter what is the result of right hand side expression.
- In the case of && if the left hand side expression results false, then the result will be false, no matter what is the result of right hand side expression.

Example 1: (expr1 || expr2) Example2: (expr1 && expr2)

Miscellaneous Operators**The Assignment Operator**

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

var = expression;

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

```
int x, y, z;
x = y = z = 52; // set x, y, and z to 52
```

The ? Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. The ? has this general form:

$$\text{var} = \text{expr1} \ ? \ \text{expr2} \ : \ \text{expr3}$$

Here, *expr1* can be any expression that evaluates to a *boolean* value. If *expr1* is true, then *expr2* is evaluated; otherwise, *expr3* is evaluated and its value is assigned to *var*.

The following program demonstrates the Relational operators: (**Ternary.java**)

```
class Ternary
{
    public static void main(String args[])
    {
        int x=4,y=6;
        int res= (x>y)?x:y;
        System.out.println("The result is :"+res);
    }
}
```

Output

The result is : 6

1.13 Expressions

In Java programming, an expression is any legal combination of operators and operands that evaluated to produce a value. Every expression consists of at least one operand and can have one or more operators. Operands are either variables or values, whereas operators are symbols that represent particular actions.

In the expression $x + 5$; x and 5 are operands, and $+$ is an operator.

In Java programming, there are mainly two types of expressions are available. They are as follows:

1. Simple expression
2. Complex expression

Simple expression: It contains one operator and two operands or constants.

Example: $x+y$; $3+5$; $a*b$; $x-y$ etc.

Complex expression: It contains two or more operators and operands or constants.

Example: $x+y-z$; $a+b-c*d$; $2+5-3*4$; $x=6-4+5*2$ etc.

Operators provided mainly two types of properties. They are as follows:

1. Precedence
2. Associativity

Operator Precedence

It defines the order in which operators in an expression are evaluated depends on their relative precedence. Example: Let us see $x=2+2*2$

1st pass -- 2+2*2
 2nd pass -- 2+4
 3rd pass -- 6 that is x=6.

Associativity defines the order in which operators with the same order of precedence are evaluated. Let us see $x=2 / 2 * 2$

1st pass -- 2 / 2 * 2
 2nd pass -- 1*2
 3rd pass -- 2 that is x=2

Below Table shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is *left to right* (except for *assignment*, which evaluates *right to left*). The [], (), and . would have the highest precedence.

Highest(↓)							Associativity
++(postfix)	-- (postfix)						L to R
++(prefix)	-- (prefix)	~	!	+ (unary)	(-) unary	(type cast)	L to R
*	/	%					L to R
+	-						L to R
>>	>>>	<<					L to R
>	>=	<	<=	instanceof			L to R
==	!=						L to R
&							L to R
^							L to R
							L to R
&&							L to R
							L to R
?:							L to R
=	op=						R to L
Lowest							L to R

Using Parentheses

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

$$a \gg b + 3$$

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

$$a \gg (b + 3)$$

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

$$(a \gg b) + 3$$

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression.

1.14 Type Conversion and casting

Type casting is a way to convert a variable from one data type to another data type. It can be of two types: They are

1. Implicit Conversion
2. Explicit Conversion.

Implicit Conversion

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as *implicit type conversion* or *automatic type promotion*. In this, all the lower data types are converted to its next higher data type.

In the case of Java, 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,

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.

The Type Promotion Rules

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

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

Example (TypePromo.java)

```
class TypePromo
{
    public static void main(String args[])
    {
        int num=10;
        float sum,f =10;
        char ch='A';
        sum=num+ch+f;
        System.out.println("The value of sum = "+sum);
    }
}
```

Output

The value of sum = 85.0

Explicit Conversion (Type casting)

It is intentionally performed by the programmer for his requirement in a Java program. The explicit type conversion is also known as *type casting*. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit into the target type.

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

```
(target - type ) value;
```

Here the target type specifies the destination type to which the value has to be converted. Example

```
int a=1234;
byte b=(byte) a;
```

The above code converts the *int* to *byte*. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. When a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.

1.15 Flow of Control (Control Statements)

Control statements are used to control the flow of execution to advance and branch based on changes to the state of a program. Java control statements can be put into the three categories:

1. Selection
2. Iteration
3. Jump

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

1.15.1 Selection statements

They allow your program to choose different paths of execution based upon the outcome of an

expression or the state of a variable. They are also called as *Conditional or Decision Making Statements*. These include *if* and *switch*.

if..else statement

This is the Java's conditional branch statement. This is used to route the execution through two different paths. The general form of *if* statement will be as follow:

```
if (conditional expression) {
    statement1;
}
else {
    statement2;
}
```

Here the statements inside the block can be *single* statement or *composite* statements. The *conditional* expression is any expression that returns the *Boolean* value. The *else* clause is optional. The *if* works as follows: if the conditional expression is true, then *statement1* will be executed. Otherwise *statement2* will be executed. Example:

Write a java program to find whether the given number is even or odd? (**EvenOdd.java**)

```
import java.io.*;
class EvenOdd {
    public static void main(String args[]) throws IOException {
        DataInputStream dis =new DataInputStream(System.in);
        System.out.print("Enter the value of n:");
        int n=Integer.parseInt(dis.readLine());
        if(n%2==0)
            System.out.println(n+" is Even Number");
        else
            System.out.println(n+"is ODD Number");
    }
}
```

Output

```
Enter the value of n: 5
5 is ODD Number
```

Nested if

The nested if statement is *if* statement, that contains another if and else inside it. When we nest *ifs*, the else always associated with the nearest if. The general form of the nested if will be as follow:

```
if(conditional expr1) {
    if(conditional expr2) {
        statement1;
    }
else {
    statement2;
}
}
else { statement3; }
```

Example

Write a java Program to test whether a given number is positive or negative. (**Positive.java**)

```
import java.io.*;
class Positive
{
    public static void main(String args[]) throws IOException
    {
        DataInputStream dis=new DataInputStream(System.in);
        System.out.print("Enter a number to test:"); int
        n=Integer.parseInt(dis.readLine());
        if(n>-1) {
            if(n>0)
                System.out.println(n+ " is Positive");
        }
    }
}
```

```

    }
    else
        System.out.println(n+ " is Negative");
}
}

```

Output

Enter a number to test: -1
-1 is Negative

The if-else-if Ladder

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

```

if(condition1)
    statement1;
else if(condition2)
    statement2;
    .
    .
else if(conditionN)
    statementN;
else
    statement;

```

The *if* statements are executed in a sequential manner. As soon as one of the *condition* is true, the statement associated with that *if* is executed, and the rest of the ladder is bypassed. If none of the *conditions* is true, then the *final else* statement will be executed.

Example Program:

Write a Java Program to test whether a given character is Vowel or Consonant? (**Vowel.java**)

```

import java.io.*;
class Vowel {
    public static void main(String args[]) throws IOException
    {
        System.out.print("Enter character to test:");
        char ch=(char)System.in.read();
        if(ch=='a')
            System.out.println("Vowel");
        else if(ch=='e')
            System.out.println("Vowel");
        else if(ch=='i')
            System.out.println("Vowel");
        else if(ch=='o')
            System.out.println("Vowel");
        else if(ch=='u')

```

```
        System.out.println("Vowel");
    else
        System.out.println("Consonant");
    }
}
```

Output

```
Enter character to test:a
Vowel
Enter character to test:r
Consonant
```

The 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 a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch(expr)
{
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
        break;
    case valueN:
        // statement sequence
        break;
    default:
        // default statement sequence
}
```

- The expression must be of type *byte*, *short*, *int*, *char* or *string*;
- Each of the values specified in the case statements must be of a type compatible with the expression.
- Each case value must be a unique literal (that is, it must be a constant, not a variable).
- Duplicate case values are not allowed.

The switch statement works like this: The value of the *expression* is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the *default* statement is executed. However, the default statement is 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`.

Write a Java Program to test whether a given character is Vowel or Consonant using `switch`?

```
import java.io.*;
class SwitchTest
{
    public static void main(String args[]) throws IOException
    {
        System.out.print("Enter achatacter to test:");
        char ch=(char)System.in.read();
        switch(ch)
        {
            //test for small letters
            case 'a':
                System.out.println("vowel");
                break;
            case 'e':
                System.out.println("vowel");
                break;
            case 'i':
                System.out.println("vowel");
                break;
            case 'o':
                System.out.println("vowel");
                break;
            case 'u':
                System.out.println("vowel");
                break;
            default:
                System.out.println("Consonant");
        }
    }
}
```

Output

Enter character to test: i

Vowel

Enter character to test: x

Consonant

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

- The *switch* differs from the *if* in that `switch` can only test for equality, whereas `if` can evaluate any type of Boolean expression.
- No two case constants in the same `switch` can have identical values.

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

1.15. 2 Loops (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.

Advantages

- Reduce length of Code
- Take less memory space.
- Burden on the developer is reducing.
- Time consuming process to execute the program is reduced.

while

The *while* loop is a *pre-test* or *entry-controlled* loop. It uses *conditional expression* to control the loop. The while loop evaluates (checking) the test expression before every iteration of the loop, so it can execute *zero* times if the condition is initially false. The initialization of a loop control variable is generally done before the loop separately.

```
while (test expression)
{
    // body of loop
    inc/dec statement
}
```

How while loop works?

- Initially the while loop evaluates (checking condition) the test expression.
- If the test expression is true, statements inside the body of while loop is evaluated. Then, again the test expression is evaluated. The process goes on until the test expression is false.
- When the test expression is false, the while loop is terminated.

Example : Program to print 1 to 10 numbers using while loop.

```
class Test {
    public static void main(String[] args)
    {
        int n=1;
        while(n<=10)
        {
            System.out.print(n+" ");
            n++;
        }
    }
}
```

Output:

1 2 3 4 5 6 7 8 9 10

do - while statement

do-while loop is similar to *while* loop, however there is one basic difference between them. *do-while* runs at least once even if the test condition is false at first time. Syntax of do while loop is:

```
do
{
    // body of loop
    inc/dec statement
} while (test expression);
```

How do-while loop works?

- First the code block (loop body) inside the braces ({...}) is executed once.
- Then, the test expression is evaluated (checking condition). If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to false (0).
- When the test expression is false, the **do...while** loop is terminated.

Example

Write a java program to add all the number from 1 to 10. (using do-while)

class Test

```
{
    public static void main(String[] args)
    {
        int n=1;
        do
        {
            System.out.print(n+" ");
            n++;
        } while (n<=10);
    }
}
```

Output:

1 2 3 4 5 6 7 8 9 10

Comparison between do and while loops

S.No	While loop	Do-while loop
1	Condition is checked before entering into loop	Condition is checked after executing statements in loop
2	Top-tested /entry-controlled loop	Bottom-tested /exit-controlled loop
3	Minimum iterations are zero	Minimum iterations are one
4	Maximum iterations are N+1	Maximum iterations are N
5	General loop statement	General loop statement but well suited for menu-driven applications
6	Non-deterministic loop	Non-deterministic loop

for statement

It is the most general looping construct in Java. The *for loop* is commonly used when the number of iterations are exactly known. The syntax of a *for* loop is:

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

The loop header contains three parts:

- an initialization,
- a test condition, and
- incrementation(++)/decrementation(--)/update.

Initialization: This part is executed only once when we are entering into the loop first time. This part allows us to declare and initialize any loop control variables.

Condition: if it is true, the body of the loop is executed otherwise program control goes outside the for loop.

Iteration: After completion of initialization and condition steps loop body code is executed and then increment or decrements steps is execute. This statement allows to us to update any loop control variables.

How for loop works?

- First the loop initialization statement is executed.
- Then, the test expression is evaluated. If the test expression is false, for loop is terminated. But if the test expression is true, codes inside the body of for loop is executed and the update expression is executed. This process repeats until the test expression becomes false.

Note: In for loop everything is optional but mandatory to place two semicolons (; ;)

Example program: same program using the for loop

class Test

```
{
    public static void main(String[] args)
    {
        for(int i=1;i<=10;i++)
        {
            System.out.print(i+" ");
        }
    }
}
```

Output:

1 2 3 4 5 6 7 8 9 10

For each version of the for loop:

A for loop also provides another version, which is called Enhanced Version of for loop. The general form of the for loop will be as follow:

```
for ( type itr_var : collection)
{
    // body of loop
}
```

Here, *type* is the type of the iterative variable of that receives the elements from collection, one at a time, from beginning to the end. The collection is created using the array.

Example: Write a java program to print all the elements in an array?

```
class Test
{
    public static void main(String[] args)
    {
        int a[] = {12,13,14,15,16};
        for(int x:a)
        {
            System.out.print(x+" ");
        }
    }
}
```

Output

12 13 14 15 16

1.15.3 The Jump/Branching Statements

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

break statement

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

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

Using break to Exit a Loop

In Java, when *break* statement is encountered inside a loop, the loop is immediately terminated, and program control is transferred to nest statement following the loop. The *break* statement is widely used with *for loop*, *while loop*, *do-while loop* and *switch* statement. Its syntax is quite simple, just type keyword *break* followed with a semicolon.

break;

The following example illustrates the use of *break*;

```
class Test {
    public static void main(String[] args) {
        int i=1;
        while( i <= 5)
        {
            if (i==3)
                break;
        }
    }
}
```

```

        System.out.print(" "+i);
        i = i + 1;
    }
}

```

Output

```

1      2

```

Using break as a Form of Goto

For example, the *goto* can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the break statement. By using this form of break, you can, for example, break out of one or more blocks of code. The general form of the labeled break statement is shown here:

```
break label;
```

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of break executes, control is transferred out of the named block. The labeled block must enclose the break statement, but it does not need to be the immediately enclosing block.

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

Example code:

```

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.");
        }
    }
}

```

Running this program generates the following output:

Before the break.

This is after second block.

continue statement

In Java, when *continue* statement is encountered inside a loop, it stops the current iteration and places the loop in next iteration. In *while* and *do-while* loops, a *continue* statement causes control to be transferred directly to the conditional expression that controls the loop. In a *for* loop, control goes first to the *iteration* portion of the for statement and then to the *conditional expression*. For all three loops, any intermediate code is bypassed. Its syntax is quite simple, just type keyword *continue* followed with a semicolon.

continue;

```
class Test {
    public static void main(String[] args) {
        int i=1;
        while( i <= 5)
        {
            i = i + 1;
            if (i==3)
                continue;
            System.out.print(" "+i);
        }
    }
}
```

Output

2 4 5 6

return statement

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

return expr/value;

Example:

```
class Test {
    public static void main(String[] args) {           //Caller Method
        int a=3,b=4;
        int x= add(a,b);      //function call
        System.out.println("The sum is "+x);
    }
    static int add(int x,int y)           // called method
    {
        return (x+y);
    }
}
```

Output

The sum is 7

After computing the result the control is transferred to the caller method, that main in this case.

2.1 What is class?

A class is a group of objects that share the same properties and methods. It is a **template** or blueprint from which objects are created. The objects are the **instances** of class. Because an object is an instance of a class. A class is used to define new type of the data. Once defined, this new type can be used to create objects of its type. The class is the logical entity and the object is the logical and physical entity.

The general form of the class

A class is declared by use of the **class** keyword. A simplified general form of a **class** definition is shown here:

```
class classname
{
    type instance-variable1;
    type instance-variable2;
    .....
    .....
    type instance-variableN;

    type method1(parameterlist)
    {
        //body of the method1
    }
    type method2(parameterlist)
    {
        //body of the method2
    }
    .....
    .....
    type methodN(parameterlist)
    {
        //body of the methodN
    }
}
```

- The data or variables, defined within the class are called, *instance variables*.
- The methods also contain the code.
- The methods and instance variables collectively called as *members*.
- Variable declared within the methods are called *local variables*.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Student** that defines three instance variables: **sid**, **sname**, and **branch**. Currently, **Student** does not contain any methods.

```
class Student
{ //instance variables int
    sid;
    String sname; String branch;
}
```

As stated, a class defines new data type. The new data type in this example is, Box. This defines the template, but does not actually create object.

Note: By convention the First letter of every word of the class name starts with Capital letter, but not compulsory. For example student is written as “Student” The First letter of the word of the method name begins with small and remaining first letter of every word starts with Capital letter, but not compulsory. For example `-boxVolume()`

2.2 Creating the Object

There are three steps when creating an object from a class:

Declaration: A variable declaration with a variable name with an object type.

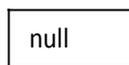
Instantiation: The 'new' key word is used to create the object.

Initialization: The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Step 1:

Student s;

Effect: s

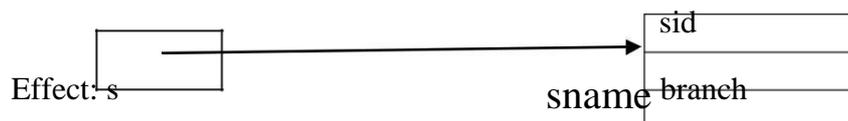


Declares the class variable. Here the class variable contains the value **null**. An attempt to access the object at this point will lead to Compile-Time error.

Step 2:

Student s=new Student();

Here *new* is the keyword used to create the object. The object name is *s*. The **new** operator allocates the memory for the object, that means for all instance variables inside the object, memory is allocated.



Student Object

Step 3:

There are many ways to initialize the object. The object contains the instance variable. The variable can be assigned values with reference of the object.

```
s.sid=1234;
s.sname="Anand";
s.branch="IT";
```

Here is a complete program that uses the **Student** class: (**StudentDemo.java**)

```
class Student
{
    int sid;
    String sname;
    String branch;
```

}

□ This class declares an object of type Student.

```
class StudentDemo
{
    public static void main(String args[])
    {
        //declaring the object (Step 1) and instantiating (Step 2) object
        Student std = new Student();

        assign values to std's instance variables (Step
        3) s.sid=1234;
        s.sname="Anand";
        s.branch="IT";

        display student details
        System.out.println(std.sid" "+std.sname+" "+std.branch);
    }
}
```

When you compile this program, you will find that two **.class** files have been created, one for **Student** and one for **StudentDemo**. The Javacompiler automatically puts each class into its own **.class** file. It is not necessary for both the **Student** and the **StudentDemo** class to actually be in the same source file.

To run this program, you must execute **StudentDemo.class**. When you do, you will see the following output:

```
1234      Anand      IT
```

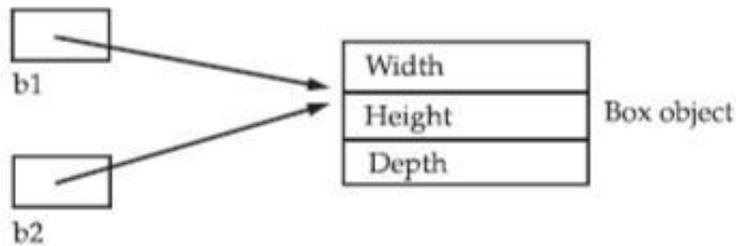
As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Student** objects, each has its own copy of **sid**, **sname**, and **branch**.

2.2.1 Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

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

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object. This situation is depicted here:



2.3 Methods

Classes usually consist of two things: *instance variables and methods*. This is the general form of a method:

```
type name(parameter-list)
{
  body of method
}
```

Here,

9. *type* specifies the type of data returned by the method. This can be any valid data type, including class types that you create.
10. If the method does not return a value, its return type must be **void**.
11. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.
12. The *parameter-list* is a sequence of type and identifier pairs separated by commas.
13. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.
14. Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

return value;

Here, *value* is the value returned.

Adding a method to the Student class (**Student.java**)

```
class Student
{
    int sid;
    String sname;
    String branch;
    show()
    {
        System.out.println("Student ID"+sid);
        System.out.println("Student Name: "+sname);
        System.out.println("Branch:"+branch);
    }
}
```

Here the method name is "show()". This methods contains some code fragment for displaying the

student details. This method can be accessed using the object as in the following code:

StudentDemo.java

```
class StudentDemo
{
    public static void main(String args[])
    {
        Student std = new Student();
        s.sid=1234;
        s.sname="Anand";
        s.branch="IT";
        std.show();           // display student details
    }
}
```

Output

```
Student ID: 1234
Student Name: Anand
Branch:IT
```

Returning a Value

A method can also return the value of specified type. The method after computing the task returns the value to the **caller** of the method.

```
class Student
{
    int sid;
    String sname;
    String branch;
    int sub1, sub2, sub3;

    public float percentage()
    {
        return (sub1+ sub2+sub3)/3;
    }
    public void show()
    {
        System.out.println("Student ID"+sid);
        System.out.println("Student Name: "+sname);
        System.out.println("Branch:"+branch);
    }
}

class StudentDemo
{
    public static void main(String args[])
    {
        Student std = new Student();
        std.sid=1234;
        std.sname="Anand";
    }
}
```

```

        std.branch="IT";
        std.sub1=89;
        std.sub2=90;
        std.sub3=91;
        float per = std.percentage();           //calling the method
        std.show();
        System.out.println("Percentage:"+per);
    }
}

```

Output

```

Student ID: 1234
Student Name: Anand
Branch: IT
Percentage: 90

```

2.3.1 Adding a method that takes the parameters

We can also pass arguments to the method through the object. The parameters separated with comma operator. The values of the actual parameters are copied to the formal parameters in the method. The computation is carried with formal arguments, the result is returned to the caller of the method, if the type is mentioned.

```

float percentage(int s1, int s2, int s3)
{
    sub1 = s1;
    sub2 = s2;
    sub3 = s3;
    return (sub1+sub2+sub3)/3;
}

```

2.3.2 Method Overloading

In Java it is possible to define **two or more methods** within the same class that share the same name, as long as their parameter declarations are different. In this case, the methods are said to be *overloaded*, and the process is referred to as **method overloading**. Method overloading is one of the ways that Java supports **polymorphism**.

5. When an overloaded method is invoked, Java looks for a match between arguments of the methods to determine which **version** of the overloaded method to actually call.
6. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
7. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
8. When an overloaded method is invoked, Method overloading supports **polymorphism** because it is one way that Java implements the —one interface, multiple methods paradigm.

Here is a simple example that illustrates method overloading:

```

class Overload
{
    void test()
    {

```

```

        System.out.println("No parameters");
    }
    5.Overload test for one integer
    parameter. void test(int a)
    {
        System.out.println("a: " + a);
    }
}
class OverloadDemo
{
    public static void main(String args[])
    {
        Overload ol = new Overload();
        double result;
        7.call all versions of test()
        ob.test();
        ob.test(10);
    }
}

```

Output

No parameters
a: 10

The *test()* method is overloaded two times, first version takes no arguments, second version takes one argument.

2.4 Constructors

A constructor is a special member function used for automatic initialization of an object. Whenever an object is created, the constructor is called automatically. Constructors can be overloaded.

Characteristics

13. Constructors have the same name as that of the class they belongs to.
14. They automatically execute whenever an object is created.
15. Constructors will not have any return type even void.
16. Constructors will not return any values.
17. The main function of constructor is to initialize objects and allocation of memory to the objects.
18. Constructors can be overloaded.
19. A constructor without any arguments is called as default constructor.

Example

```

class Student
{
    int sid;
    String sname;
    String branch;
    int sub1, sub2, sub3;

    Student()           // This is the constructor for Student
}

```

```

        {
            sid=1234;
            sname="Anand";
            branch="IT";
            sub1=89;
            sub2=90;
            sub3=91;
        }
        public float percentage()
        {
            return (sub1+ sub2+sub3)/3;
        }
        public void show()
        {
            System.out.println("Student ID"+sid);
            System.out.println("Student Name: "+sname);
            System.out.println("Branch:"+branch);
        }
    }
}
class StudentDemo
{
    public static void main(String args[])
    {
        3. declare, allocate, and initialize Student object
        Student std = new Student();
        float per = std.percentage();           //calling the method
        std.show();
        System.out.println("Percentage:"+per);
    }
}

```

Output

```

Student ID: 1234
Student Name: Anand
Branch: IT
Percentage: 90

```

Parameterized Constructors

It may be necessary to initialize the various data members of different objects with different values when they are created. This is achieved by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors*.

```

/* Here, Student uses a parameterized constructor to initialize the details of a student. */
class Student
{
    int sid;
    String sname;
    String branch;
    int sub1, sub2, sub3;
    // This is a parameterized constructor for Student
    Student(int id, String name, String b, int s1 ,int s2, int s3 )

```

```
{
    sid=id;
    sname=name;
    branch=br;
    sub1=s1;
    sub2=s2;
    sub3=s3;
}
public float percentage()
{
    return (sub1+ sub2+sub3)/3;
}
public void show()
{
    System.out.println("Student ID"+sid);
    System.out.println("Student Name: "+sname);
    System.out.println("Branch:"+branch);
}
}
class StudentDemo
{
    public static void main(String args[])
    {
        // declare, allocate, and initialize Student object
        Student std = new Student(1234, "Anand","IT",89,90,91);
        float per = std.percentage();           //getting percentage
        std.show();
        System.out.println("Percentage:"+per);
    }
}
```

Output

```
Student ID: 1234
Student Name: Anand
Branch: IT
Percentage: 90
```

2.4.1 Constructor Overloading

In Java, it is possible to define two or more class constructors that share the same name, as long as their parameter declarations are different. This is called constructor overloading.

When an overloaded constructor is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded constructor to actually call. Thus, overloaded constructors must differ in the type and/or number of their parameters.

Example: OverloadCons.java

```
class Student
{
    int sid;
    String sname;
    String branch;
```

```
int sub1, sub2, sub3;
```

```
Student()           // This is the default constructor
{
    sid=501;
    sname= "Akhil";
    branch= "CSE";
    sub1=91;
    sub2=92;
    sub3=93;
}
```

10. This is a parameterized constructor for Student

```
Student(int id, String name, String b, int s1 ,int s2, int s3 )
```

```
{
    sid=id;
    sname=name;
    branch=br;
    sub1=s1;
    sub2=s2;
    sub3=s3;
}
public float percentage()
{
    return (sub1+ sub2+sub3)/3;
}
public void show()
{
    System.out.println("Student ID"+sid);
    System.out.println("Student Name: "+sname);
    System.out.println("Branch:"+branch);
}
```

```
}
class OverloadCons
```

```
{
    public static void main(String args[])
    {
        6.create objects using the various
constructors Student std1 = new Student();
        Student std2 = new Student(1234, "Anand","IT",89,90,91);

        float per1 = std1.percentage();           //Calculating CSE student percentage
        std1.show();
        System.out.println("Percentage:"+per1);

        float per2 = std2.percentage();           //Calculating IT student percentage
        std2.show();
        System.out.println("Percentage:"+per2);
    }
}
```

Output

Student ID: 501
Student Name: Akhil
Branch: CSE
Percentage: 92

Student ID: 1234
Student Name: Anand
Branch: IT
Percentage: 90

2.5 Garbage Collection

Since objects are dynamically allocated by using the **new** operator, such objects are destroyed and their memory released for later reallocation. Java handles de-allocation for you automatically. The technique that accomplishes this is called *garbage collection*. Java has its own set of algorithms to do this as follow. There are Two Techniques:

12. **Reference Counter**
13. **Mark and Sweep.**

In the Reference Counter technique, when an object is created along with it a reference counter is maintained in the memory. When the object is referenced, the reference counter is incremented by one. If the control flow is moved from that object to some other object, then the counter value is decremented by one. When the counter reaches to zero (0), then it's memory is reclaimed.

In the Mark and Sweep technique, all the objects that are in use are *marked* and are called live objects and are moved to one end of the memory. This process we call it as compaction. The memory occupied by remaining objects is reclaimed. After these objects are deleted from the memory, the live objects are placed in side by side location in the memory. This is called copying.

It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. Garbage collection only occurs during the execution of your program. *The main job of this is to release memory for the purpose of reallocation.*

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. Inside the *finalize()* method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects

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

```
protected void finalize( )  
{
```

```

    // finalization code here
}

```

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.

2.6 static keyword

- There will be times when you will want to define a class member that will be used independently of any object of that class. To create such a member, precede its declaration with the keyword **static**.
- The *static* keyword in java is used for memory management mainly.
- We can apply java static keyword with variables, methods, blocks and nested class.
- The static keyword belongs to the class than instance of the class.
- The static can be:
 - variables (also known as class variables)
 - methods (also known as class methods)
 - block
 - nested class

The most common example of a **static** member is *main()*. **main()** is declared as **static** because it must be called before any objects exist.

Java static variable

- If you declare any variable as static, it is known static variable.
- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

3. It makes your program memory efficient (i.e it saves memory).

Example:

```

class Student
{
    int rollno;
    String name;
    String college="VVIT";
}

```

Suppose there are 3000 students in my college, now all instance data members will get memory each time when object is created. All student have its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once. Java static property is shared to all objects.

```

class Student
{
    int rollno;
    String name;
    static String college ="VVIT";
}

```

```

Student(int r, String n)
{
    rollno = r;
    name = n;
}

void show()
{
    System.out.println(rollno+" "+name+" "+college);
}

public static void main(String args[])
{
    Student s1 = new Student(1201,"Ajith");
    Student s2 = new Student8(1202,"Alekhya");
    s1.display();
    s2.display();
}
}

```

Output: 1201

```

Ajith 1202      VVIT
Alekhya        VVIT

```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```

class Counter
{
    static int count=0;    //will get memory only once and retain its value
    Counter() {
        count++;
        System.out.println(count);
    }
    public static void main(String args[])
    {
        Counter c1=new Counter();
        Counter c2=new Counter();
        Counter c3=new Counter();
    }
}

```

Output: 1 2 3

Java static method

8. If you apply *static* keyword with any method, it is known as *static method*.
9. A static method belongs to the class rather than object of a class.
10. A static method can be invoked without the need for creating an instance of a class.
11. Static method can access static data member and can change the value of it.

There are three main restrictions for the static method. They are:

- d. They cannot call **non-static** methods.
- e. They cannot access **non-static** data.
- f. They cannot refer to **this** or **super** in any way.

Example of static method that calculates cube of a given number.

class Calculate

```
{
    static int cube(int x)
    {
        return x*x*x;
    }
    public static void main(String args[])
    {
        int result = Calculate.cube(5);
        System.out.println(result);
    }
}
```

Output: 125

Another example of static method that access static data member and can change the value of it

class Student

```
{
    int rollno;
    String name;
    static String college = "BCCIT"; //static variable
    static void change() //static method
    {
        college = "VVIT";
    }

    Student(int r, String n)
    {
        rollno = r;
        name = n;
    }
    void display ()
    {
        System.out.println(rollno+" "+name+" "+college);
    }
    public static void main(String args[])
    {
        Student.change();
        Student s1 = new Student (1211,"Akshara");
        Student s2 = new Student (1222,"Chaitanya");
        Student s3 = new Student (1233,"Krishna");
        s1.display();
        s2.display();
    }
}
```

```

        s3.display();
    }
}

```

Output:

```

1211 Akshara      VVIT
1222 Chaitanya   VVIT
1233 Krishna     VVIT

```

Java static block

- It is used to initialize the static data member.
- It is executed before *main* method at the time of classloading.

Note: Static blocks are executed first, even than the static methods.

Example of static block

```

class A
{
    static int i;
    static
    {
        System.out.println("Static block is invoked");
        i=10;
    }
    public static void main(String args[])
    {
        System.out.println("Now in main");
        System.out.println("i:ow"+A.i);
    }
}

```

Output

```

static block is invoked
Now in main
I:10

```

2.7 this keyword

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines *this* keyword.
- **this** can be used inside any method to refer to the *current object*. i.e. **this** is always a reference to the object on which the method was invoked.

Usage of java this keyword

- this* can be used to refer current class instance variable.
- this* can be used to invoke current class method.
- this* can be used to invoke current class constructor.

this: To refer current class instance variable

- The *this* keyword can be used to refer current class instance variable.
- If there is ambiguity between the instance variables and parameters, *this* keyword resolves the problem of ambiguity.

Let's understand the problem with example given below:

```
class Student
{
    int sid;
    String sname;
    float fee;
    Student(int sid, String sname, float fee)
    {
        this.sid=sid;
        this.sname=sname;
        this.fee=fee;
    }
    void show()
    {
        System.out.println(sid+" "+sname+" "+fee);
    }
}
class Test
{
    public static void main(String args[])
    {
        Student std=new Student(1201,"Ankit",88000f);
        std.show();
    }
}
```

Output

```
1201 Ankit 88000.0
```

this: To invoke current class method

- You may invoke the method of the current class by using the *this* keyword.
- If you don't use the *this* keyword, compiler automatically adds *this* keyword while invoking the method.

Let's see the example

```
class A
{
    void m()
    {
        System.out.println("hello m");
    }
    void n()
    {
        System.out.println("hello n");
        this.m();
    }
}
class Test
{
    public static void main(String args[])
    {
```

```

        A a=new A();
        a.n();
    }
}

```

Output

```

hello n
hello m

```

this: To invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```

class A {
    A() {
        System.out.println("hello a");
    }
    A(int x) {
        this();
        System.out.println(x);
    }
}

class Test{
    public static void main(String args[]) {
        A a=new A(10);
    }
}

```

Output

```

hello a
10

```

2.8 Introduction to Arrays

- An Array is a collection of elements that share the same type and name.
- The elements from the array can be accessed by the index.
- All array indices start at zero.
- You can access a specific element in the array by specifying its index within square brackets.

Advantages

Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.

Random access: We can get any data located at any index position.

Disadvantages

Size Limit: We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array

To create an array, we must first create the array variable of the desired type. The general form of the One Dimensional array is as follows:

```
type array-var[] = new type[size];
```

5. Here *type* declares the base type of the array. This base type determine what type of elements that the array will hold.
6. *array-var* is the array variable that is linked to the array.
7. *size* specifies the number of elements stored in the array
8. **new** to allocate memory for an array. The elements in the array allocated by **new will automatically be initialized to zero.**

Example: `String months[] = new int[12];`

Here

- type is *String*, the variable name is *months*. All the elements in the months are *Strings*. Since, the base type is *String*.
- This example allocates 12-element array of *Strings* and links them to **months**. The elements in the array allocated by **new will automatically be initialized to null.**
-

months

Index	0	1	2	3	4	5	6	7	8	9	10	11
Element	null											

For example, this statement assigns the value “*February*” to the second element of **months**.
`months[1] = “February”;`

Index	0	1	2	3	4	5	6	7	8	9	10	11
Element	null	February	null									

Example Program: Write a Java Program to read elements into array and display them?

ArrayTest.java

```
import java.io.*;
import java.util.*;
class ArrayTest
{
    public static void main(String args[])
    {
        Scanner s=new Scanner(System.in);
        //declaring and allocating memory to array and all the elements are set
        zero int a[] = new int[5];
```

```

        //read the elements into array
        System.out.println("Enter the elements into Array:");
        for(int i=0;i<5;i++)
        {
            a[i]=s.nextInt();
        }

        //displaying the elements
        System.out.println("The elements of Array:");
        for(int i=0;i<n;i++)
        {
            System.out.print(a[i]+",");
        }
    }
}

```

Output

Enter the elements into Array :12 45 56 78 30
 The elements of Array: 12,45,56,78,30

Multidimensional Arrays

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

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

This allocates a 4 by 4 array and assigns it to **twoD**. Internally this matrix is implemented as an *array of arrays* of **int**.

	Right Index Determines the Columns			
[0,0]	[0,1]	[0,2]	[0,3]	
[1,0]	[1,1]	[1,2]	[1,3]	
[2,0]	[2,1]	[2,2]	[2,3]	
[3,0]	[3,1]	[3,2]	[3,3]	

Example Program for Matrix Addition

```

import java.io.*;
import java.util.Scanner;
class Testarray2
{
    public static void main(String args[])
    {
        int m, n, i, j;
        Scanner s=new Scanner(System.in);
        System.out.println("Enter the no. of rows and columns of matrix");
        m = s.nextInt();
        n = s.nextInt();
        int m1[][] = new int[m][n];
        int m2[][] = new int[m][n];
    }
}

```

```

int result[][] = new int[m][n];
System.out.println("Enter the elements of first
matrix:"); for ( i = 0 ; i < m ; i++ )
    for ( j = 0 ; j < n ; j++ )
        m1[i][j] = s.nextInt();
System.out.println("Enter the elements of second matrix:");
for ( i = 0 ; i < m ; i++ )
    for ( j = 0 ; j < n ; j++ )
        m2[i][j] = s.nextInt();

System.out.println("Sum of entered matrices:");
for ( i = 0 ; i < m ; i++ )
{
    for ( j = 0 ; j < n ; j++ )
    {
        result[i][j] = m1[i][j] + m2[i][j];
        System.out.print( result[i][j]+"\\t");
    }
    System.out.println();
}
}
}

```

Output

```

Enter the elements of first matrix: 1 2 3 4
Enter the elements of second matrix: 1 2 3 4
Sum of entered matrices:      2      4
                             6      8

```

2.8.1 Alternative Array Declaration Syntax

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

```
type [ ] var-name;
```

Here, the square brackets follow the type specifier, and not the name of the array variable. This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

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

creates three array variables of type **int**. It is the same as writing `int nums[], nums2[], nums3[];` // create three arrays.

2.9 Command Line arguments

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.
- They are stored as Strings in String array passed to the args in the main method.

- The first command line argument is stored in args[0], the second argument is stored in args[1], the third argument is stored in args[2], and so on.

Example program:

Write a Java program to read all the command line arguments.

```
import java.io.*;
class CLine
{
    public static void main(String args[])
    {
        for(int i=0;i<args.length();i++)
            System.out.println("args["+i+"]:"+args[i]);
    }
}
```

Output

```
C:/>javac CLine.java
C:/>java Cline Vijay Anand Chintapalli
args[0]: Vijay
args[1]: Anand
args[2]: Chintapalli
```

2.10 Nested classes

- A class defined within another class; such classes are known as *nested classes*.
- The scope of the nested class (*inner class*) is bounded by the scope of its enclosing class (*outer class*).
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class
{
    //code
    class Java_Inner_class
    {
        //code
    }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

- Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- Code Optimization: It requires less code to write.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
 - Member inner class
 - Anonymous inner class
 - Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.

Static inner class

- A static inner class is a nested class which is a static member of the outer class.
- It can be accessed without instantiating the outer class, using other static members. Just like static members.
- A static nested class does not have access to the instance variables and methods of the outer class.
- The syntax of static nested class is as follows

```

class MyOuter
{
    static class Mynested
    {}
}

```

Example: Outer.java

```

public class Outer
{
    //inner static class
    static class Nested_Demo
    {
        public void my_method()
        {
            System.out.println("This is my nested class");
        }
    }
    //body of outer class
    public static void main(String args[])
    {
        //without creating the object of outer class
        Outer.Nested_Demo nested = new Outer.Nested_Demo();
        nested.my_method();
    }
}

```

Note:

- static inner classes are rarely used in programs.**
- If the static inner class is static method, we can directly call that method in the main() function without creating the object. As follow:**
Outer.Nested_Demo.my_method();

Non-static inner class

- Creating an inner class is quite simple. You just need to write a class within a class.
- An inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

Example

```
public class Outer
{
    int outer_x=100;
    void test()
    {
        //creating the object of Inner class
        Inner inn=new Inner();
        inn.disp();
    }
    //inner non-static class
    private class Inner
    {
        public void disp()
        {
            System.out.println("The value of x is:"+outer_x);
        }
    }
    public static void main(String args[])
    {
        //creating the out class object
        Outer2 out=new Outer2();
        out.test();
    }
}
```

