JAVA

Genesis of Java

To know about genesis of java, first we need to know about C and C++.

Evolution of C: Initially C language has given a dimension to a new programming approach called structured programming. In a structured programming language, there will be top down approach, control statements and modular programming. As C provides all the three, it is referred to as structured programming language. The impact of 'C' on software development can't be ignored as it consists of the following:

- Control statements
- Modular programming
- Top down approach

It replaced the existing assembly languages successfully. When a computer language is designed, trade-offs are often made, such as the following:

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, one might ask a question like why a need for something else existed. The answer is *complexity*. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity.

Evolution of C++: C++ is a response to the above said need. Although C is one of the world's great programming languages, there is a limit to its ability to handle complexity. Once a program exceeds somewhere between 25,000 and 100,000 lines of code, it becomes so complex that it is difficult to grasp as a totality. C++ allows this barrier to be broken, and helps the programmer comprehend and manage larger programs. By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Object oriented programming has given a new dimension to the programming world the era is continuing. With the help of OOPS concept, software design has been changed. The main object oriented principles are: data abstraction, inheritance.

Creation of JAVA:

Inspite of having more features, C and C++ are unable to provide portability in a greater extent. This is where Java is different in its own way because of ***"platform independency".***

Java is purely object oriented language conceived by **James Goslings** and his team members at Sun Microsystems in 1991. The initial name of the language was "oak" and after that it was renamed as Java. The complete version of the language was released in 1995. Java has got much popularity because of its well known concept "**platform independency".**

In fact, with Java, two different types of applications:

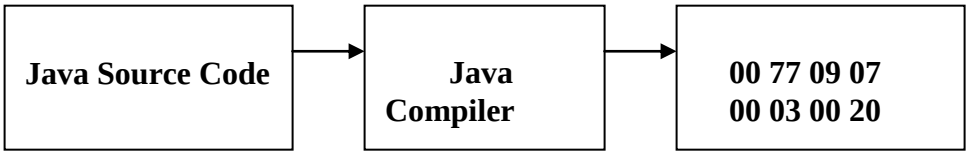
1) Stand alone applications.
2) Applets.

Stand alone applications can execute by their own where as the applets need a separate context or environment to execute themselves.

Platform Independency: Literally speaking, the concept of platform independency makes java as architecture neutral language. i.e., consider the other languages C & C++, they are designed to be compiled for any specific application. To compile any application related to C & C++, we need a full compiler that has the compatibility to the working CPU otherwise it becomes tough to manage applications. To provide better portability, Sun Microsystems concentrated on a new language and the result was Java. It can produce code that would run on a variety of CPUs under different environments. We can say that java has given a new dimension to the programming approach in terms of platform independency. Once java code is produced it can be run anywhere on any environment. Moreover, as the java code runs on any platform, it can be adapted to internet too because internet programming does require the same kind of approach. In fact, Java is no longer a language that is designed for internet programming. Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral).

"Platform independency is achieved by java through its JVM (Java Virtual Machine)"

Unlike other languages like C & C++, java compiler generates architecture-independent byte codes. A Java Virtual Machine can execute the byte codes only. Symbolically:

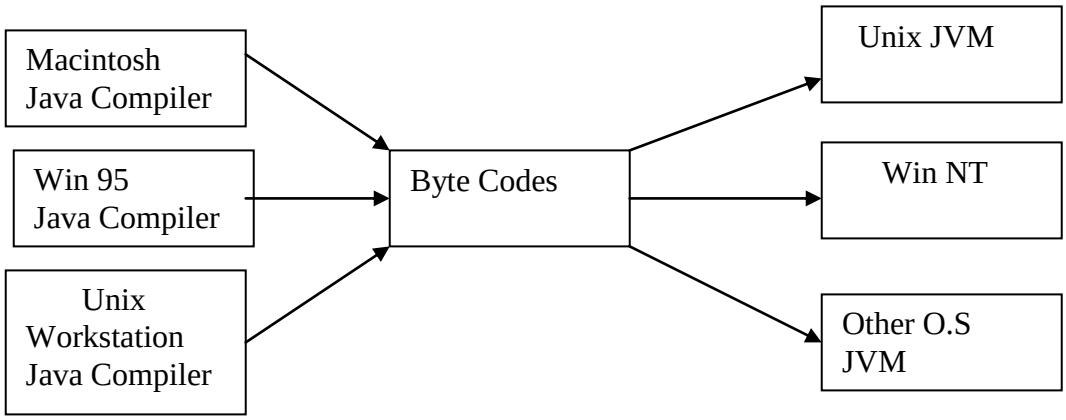


Advantages of JAVA: One obvious advantage is a runtime environment that provides platform independence: you can use the same code on Windows, Solaris, Linux, Macintosh, and so on. This is certainly necessary when programs are downloaded over the Internet to run on a variety of platforms.

Another programming advantage is that Java has syntax similar to that of C++, making it easy for C and C++ programmers to learn

JVM (Java Virtual Machine)

JVM is a tool that provides platform independency to the language. The Java compiler is machine dependent & JVM is machine dependent, it is the only byte codes that are machine independent. The following diagram depicts the same.



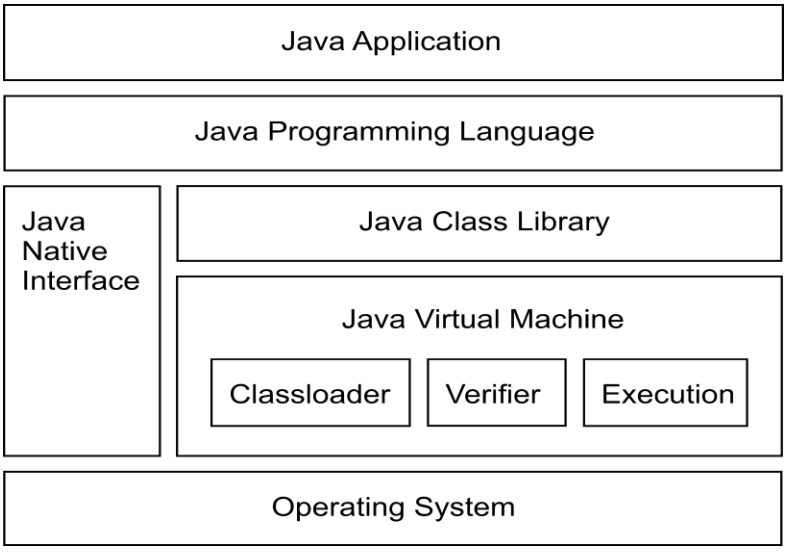
Translating a Java program into byte code helps makes it much easier to run a program in a wide variety of environments. The byte codes are the intermediate codes managed in a class file. All the components of JVM provide flexibility to the user.

A **Java virtual machine** (**JVM**) is a virtual machine capable of executing Java byte code. It is the code execution component of the Java software platform. A Java virtual machine is software that is implemented on virtual and non-virtual hardware and on standard operating systems. A JVM provides an environment in which Java byte code can be executed.

JVMs are available for many hardware and software platforms. The use of the same byte code for all JVMs on all platforms allows Java to be described as a "write once, run anywhere" programming language, as opposed to "write once, compile anywhere", which describes cross-platform compiled languages.

Java byte code is an intermediate language which is typically compiled from Java, but it can also be compiled from other programming languages.

The components of a JVM are shown in diagrammatic representation:



Java's security model has three components to look after: class loader, byte code verifier and Security Manager. The Class Loader loads all the required class files in to the disk. The byte

code verifier ensures that the Java programs have been compiled correctly, that they will obey the Virtual Machine's access restrictions, and the byte codes will not access 'private data'. The Security Manager implements a security policy for the VM. The security policy determines which activities the VM is allowed to perform, and under what circumstances.

The JVM fetches classes from a disk or from the network, and then verifies that the byte codes are safe to be executed. From the diagram let us see what these components of a JVM do. In addition the byte code verifier checks for the following:

- Access restriction violation.
- Object mismatching.
- Operands stack over or under flow.
- Incorrect byte code parameters.
- Illegal data conversion.

Java Byte Codes: Java byte code is an intermediate language which is typically compiled from Java, but it can also be compiled from other programming languages. Most programming languages compile source code directly into machine code, suitable for execution on particular microprocessor architecture. The difference with Java is that it uses byte code - a special type of machine code.

Java byte code is the form of instructions that the Java virtual machine executes. Each byte code opcode is one byte in length. Not all of the possible 256 opcodes are used. 51 are reserved for future use.

When a JVM loads a class file, it gets one stream of byte codes for each method in the class. The byte codes streams are stored in the method area of the JVM. The byte codes for a method are executed when that method is invoked during the course of running the program. They can be executed by interpretation, just-in-time compiling, or any other technique that was chosen by the designer of a particular JVM.

JDK (Java Development Kit)

JDK is a Java's Development Tool. It contains a set of command line tools with which Java programs can be compiled, debugged and interpreted. Some of the important tools are :

- Java Compiler (javac)
- Java Runtime Interpreter (java)
- Applet Viewer (appletviewer)
- Java Debugger (jdb)
- Java Achiever (jar)
- Javap (java disassembler)
- Javah (Java Header)
- API Source Code

The complete Java software/Technology comes in 2 versions:

- J2SE (Java2 Standard Edition 5.0).
- J2EE (Java2 Enterprise Edition).
- J2ME(Java2 Micro Edition)

The standard Edition is for implementing general & standard applications i.e., core and desktop applications where as the Enterprise Edition provides techniques to develop complex enterprise/server applications. There is another edition supported by Java called **"J2ME"** which concentrates more on mobile wireless applications using Java. For both the versions of Java, Sun Microsystems has given free downloads at net. The information and new updates along with software, you can obtain from the site **java.sun.com**.
Note: The other sites are javasoft.com, orielly.com

The java software was released by Sun Microsystems and its name is JDK (Java Development Kit). The early version was JDK 1.0. After that JDK 1.1, JDK 1.2 was released. At present the working version is JDK 1.4.2. Beta version for J2SE 5.0 is already released. The JDK1.2 is a major upgrade of the core and standard extension APIs of Java Development Kit. It includes version 1.1 of the Java Foundation Classes (JFC). "JFC is collection of APIs for building the GUI-related components of Java Applets and applications".

Basics of Java Environment

Java systems generally consist of several parts:

- Java Environment.
- Java Language.
- Java API (Application Programming Interface).
- Class Libraries.

Java strength is diverse from its architecture.

Phases in a Java Program:
A typical Java program gets through five different phases:

- Edit
- Compile
- Load
- Verify
- Execute

→The first phase comprises of creating & editing the user program in the disk. Any text editor like notepad is good enough to perform this. Java programs should have ". java" extension.

→The second phase is compilation stage and javac is the command line tool to compile a java program. The compiler generates class file (.class file) and a class file is collection of byte codes which a java interpreter can understand in execution stage.

→The third phase is known as loading stage. Before the program's execution, the .class file should be placed in the memory. A class loader searches for a .class file in the directory and loads it in the memory.

→The fourth phase is the verification phase. Before the program is executed, byte code verifier validates the byte codes for not violating the Java security restrictions which is of great concern in internet.

→The fifth phase is last phase, which is known to be execution stage. At this stage the java program gets executed.

Some of the Differences between Java/C++

- No support for pointers concept.
- No support for structure & Unions.
- No preprocessor directives (#define, #include , #ifdef etc.,)
- No multiple Inheritance.
- No goto.
- No operator overloading.
- No automatic coercion (casting).
- No individual functions (as Java is purely OOP language).

Features of Java (Java Buzzwords)

Simple: Java language is simple to use and easy to learn. It manages to handle all of its concepts in quite flexible manner. If the programmer is already aware of object oriented concepts, it becomes even easier. Moreover, it extracts all most all the features of C/C++. Java was designed to be easy for the professional programmer to learn and use effectively.

Secure: All the Java that provides the user is nothing but secured programming techniques. Java implements a separate Security Manager so that the user can be benefited in implementing the objects with ease of use. Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems

Robust: To provide better reliability, Java has to implement applications on variety of platforms. Hence it requires being robust language. To do so, It has to concentrate on few areas like identifying the errors i.e., error handling & memory management. In fact, Java doesn't allow you to make any mistakes. As Java is a strictly
typed language, it checks your code at compile time. However, it also checks your code at run time.

Strongly typed: Often, saying Java is a strongly typed language is absolute because it is very much particular about the type of the data. The user needs to be careful while dealing with data types.

Portability: As Java generates a byte codes (class file) as intermediate files, it can be ported to any platform without any problem. This itself is one of the major advantages of Java.

Architecture neutral: A central issue for the Java designers was that of code longevity and portability. One of the main problems of a programmer is that no guarantee exists that if you write a program today, it will run tomorrow – even on the same machine. As Java runs on JVM, this problem may not arise. Hence you can assume that Java is architecture – neutral language. The main goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished.

Object oriented: Object oriented programming was well proven technique with all of its concepts like polymorphism, inheritance. In future, it may get even more potentiality in software development. Hence implementing such concept will be an added advantage of Java. Moreover, the learner may not be in the illusion that he is learning a new language. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance non --objects.

Dynamic: Dynamic nature of Java gives more comfortness to the designer because dynamic declaration & redeclaration of data members becomes easy at runtime. This makes it possible to dynamically link the code in a safe manner.

Distributed: Java is designed for distributed environments like Internet, because it handles TCP/IP protocols. With this nature Java objects are distributed over the network and get executed remotely on demand.

Multithreaded: Java has another advantage of allowing the user to develop interactive, networked programs. To achieve this, Java supports multithreading this allows you to run many tasks simultaneously. Java provides built-in support for multithreading so that the user can design such application in a most sophisticated way.

Interpreted & High Performance: Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. the Java bytecode 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. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Data Types in Java

A data type tells about the type of the data being assigned to an identifier or data name. Java supports the following list of data types.

S.No	Data type	No.of Bytes	Range
1)	Byte	1 byte	-128 to 127
2)	short	2 bytes	-32768 to 32767
3)	int	4 bytes	-2147483648 to 2147483648
4)	long	8 bytes	-9223372036854775808 to 9223372036854775807
5)	float	4 bytes	-1.4E-45 to 3.4E+38
6)	double	8 bytes	4.9E-324 to 1.797E+308
7)	char	2 bytes	
8)	boolean	8 bits	true or false (0 / 1)

Note: Java doesn't support keyword "unsigned". So data types of unsigned type cannot be defined.

JAVA OPERATORS

Similar to all the other languages, Java supports all the operators like arithmetic operators, logical operators, relational operators and bit-wise operators.

Arithmetic Operators: To perform arithmetic operations over the variables

Java Operation	**Operator**
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Increment	++
Decrement	--
Addition Assignment	+=
Subtraction Assignment	-=
Multiplication Assignment	*=
Division Assignment	/=

Logical Operators: To evaluate expressions that result in either 'true' or 'false' value.

Java Operation	**Operator**
Logical and	&
Logical or	|
Not equal to	!=
Xor	^
Short circuit or	||
Short circuit and	&&

Relational Operators: These operators are used to test for equality of expressions. They result in either 'true' or 'false' value.

Java Operation	**Operator**
Equal to	==
Not equal to	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

Bit-wise Operators: Bit-wise operators work on individual bits of data.

Java Operation	Operator
Unary not	~
And	&
Or	|
Xor	^
Left shift	<<
Right shift	>>

Operators & their Precedence

Operators	Associativity
[] . () (method call)	left to right
! ~ ++ -- + (unary) – (unary) () (cast) new	right to left
* / %	left to right
+ -	left to right
<< >> >>>	left to right
< <= > >= instanceof	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	left to right
= += -= *= /= %= &= |= ^= <<= >>= >>>=	right to left

Java Keywords: A keyword is a word that has a special meaning in the JAVA compiler. There are 48 keywords in JAVA.

abstract	Boolean	break	byte
case	catch	char	class
continue	default	do	double
else	extends	false	final
finally	float	for	if
implements	import	instanceof	int
interface	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this

throw	throws	transient	true
try	void	volatile	while

Reserved words: *null, true, false*

Control Structures in Java

Generally, the statements in a program get executed in the order in which they are written. This is referred as **Sequential Execution.** If the programmer wants to execute some statements even before other statements by surpassing the order, he must transfer the control of the program temporarily. This can be achieved with control structures like if, for and switch etc.,

If – else – else if structures

The 'if' structure is called as a single selection structure because it selects or ignores a single action. The if ..else structure is said to be double selection structure because it selects either of two different actions.

Switch Structure

If is for single selection and if.. else is for double selection. When there is a situation where multiple conditions are to be evaluated then it is always better to use switch structure. The switch structure contains a set of case labels, and an optional default case. Each case label is terminated by a statement break.

```
switch(condition/expression)
{
 case constant:
                statements;
                break;
 case constant:
                statements;
                break;
                ……..
 default:
        statements;
        break;
}
```

Here case, break, default are the keywords.

While Structure:

While & For structures are said to be repetitive control structures or looping structures as the programmer can repeat the execution of the same statements as many times as required. The process of executing a block of statements terminates when the condition becomes false.

```
Syntax:     While(condition)
            {
                    Set of statements'
            }
```

do... while Structure :

The do... while structure is a bit different from while. In while, the condition is checked first and then the statements in the loop are evaluated where as in do...While structure, the statements in the loop are executed first and then the condition is evaluated.

Syntax:

```
do
{
        Set of statements;
}
While(condition);
```

For Structure:

The 'for' structure is also a looping structure that performs a statement / a block statements for a specified no. of times. In other words, for control structure may be used if the no. of iterations is definite

Syntax:

```
for(initialization; condition; incrementation/decrementation)
{
        Statements
}
```

In JDK1.5, a new version of for control structure was introduced. The syntax follows:

```
for (identifier  in Range)
{
  Set of statements;
}
```

Object Oriented Concepts

Object Oriented Programming is a style of programming. As a definition, **"**OOP is a method of implementation in which programs are organized as a cooperative collection of objects, each of which represents an instance of the class**"**. In other words OOP encapsulates data (attributes) and methods (behavior) into objects.

OOP is an advanced version of all the other programming techniques like assembly language programming, procedural and structured programming. In all the specified programming techniques, the following drawbacks can be identified that are solved in OOPS to a greater extent:

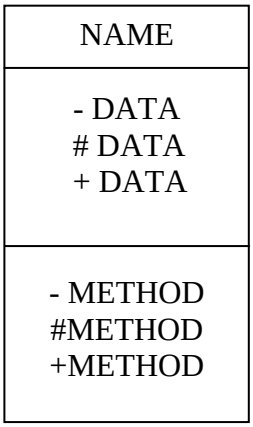
- Emphasis is more on the process rather than data.
- Functions are more interdependent and difficult to separate.
- Modifications of one function could lead to recompilation of entire application.
- No concept of true code reusability.
- Dynamic behavior is not easy to implement.
- Software development / maintenance is difficult to achieve and moreover expensive.
- Complex code, hard to write, debug and maintain.

OOPS Features:

1. Classes and objects
2. Data abstraction
3. Data encapsulation
4. Message Passing
5. Polymorphism
6. Inheritance
7. Extensibility

Class

A class is a blue print for generating various objects (or) a class is a blue print that defines the variables and methods common to all objects of a certain kind (or) a class is a prototype based on which objects can be derived (or) a class is a collection of data and it's corresponding methods. Or a class is a description of several objects.



Object

An object is an instance of a class. In other words an object is a physical construct that occupies certain amount of space in memory.

As a class is a logical construct, it doesn't occupy any memory where as an object is a physical construct and it occupies certain space.

The fundamental concept for OOP is an 'object ', which is an entity that has existence. An object fundamentally consists of three characteristics:

- A state
- A behavior
- An identity

Data abstraction: Abstraction is the process of exhibiting only the essential characteristics of an object depending on programmers view. Abstraction is of two types:

- Data abstraction
- Functional Abstraction

Ex :

Data Abstraction	- Empno, Ename, Desg, salary
Functional Abstraction	- getDetails( ), dispDetails( )

Data Encapsulation: It is a process of wrapping up of data and methods in to a single entity.

Message Passing: It is nothing but invoking a method on an object.

Polymorphism: It is the ability to provide multiple definitions to the same method signature. It allows providing one interface multiple methods.

Inheritance: It is the ability to extract the features of one class to another. It is one of the striking features of inheritance. Java supports inheritance extensively as the main objective of inheritance is code reusability.

Extendibility: As we use object oriented approach, it is one of advantages. The code independence can be achieved.

<u>Advantages of OOPs:</u>

- **Modularity:** All classes and objects can be treated as separate modules. This makes the designing simpler.
- **Ease of Maintenance:** Code is easier to maintain. The concept of encapsulation localizes the errors. I.e., debugging is made easy.
- **Reusability and Extensibility:** Through the concept of inheritance, we can easily extend existing classes. We can alter the behavior and also add new features. Code reusability is the great advantage.

- **Powerful modeling paradigm:** The system based on OOP is close to real world and hence simple to understand.

Structure of a Java Program

```
Import section
Package declaration section // if needed
Interface declaration & definition section  // if needed
Public class class-name
{
  static variable declaration; // if needed
   public static void main( String s [ ] )
   {
     body of main()
   }
  other static function definitions  //if needed
} //end of main class
other class definitions.  // if needed
```

Guidelines to design a class: -

- Always keep data as private.
- Always initialize data.
- Don't use too many basic types in a class.
- Use a standard form for class definition.
- Use descriptive names to the class and its members.

A simple class in Java:

```
public class demo
{
  public static void main( String s[ ] )
  {
          --------
            --------  // body of  main( )
  }
}
```

Declaration of Objects

As Java is an object oriented language, any no. of classes can be defined in a program. If a class is defined, user has to create an object for that class to make use of the data. In fact it is a two step process.

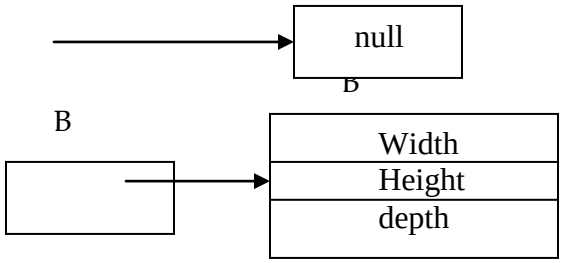
First you must declare a variable of the class type. This variable doesn't define an object. Instead, it is simply an object reference.

Second, we create a physical copy for the class and assign the address to the reference. To do this, use "new" keyword. The new keyword dynamically allocates memory for an object and returns the address.

Ex :

Box B; //creating a reference

B = new Box( );



Example:

```
Public class demo
{
 public static void main(String s[ ])
 {
   test  t  = new test( ); // instance for class test is created
   t . display( );
 }
} // end of  main class
class test
{
 int a ;
 public void display( )
  {
    a  = 101;
    System.out.println( "the value of a is ….."+a);
  }
} // end of other class.
```

Note:

1) If a program execution is over, the corresponding objects created will automatically be destroyed by the built-in garbage collector of JVM which will be invoked once the objects go beyond the scope of the programmer.

2) If we do not initialize the instance variables then they automatically assigned certain values.

- For numeric data, a zero is assigned.
- For objects & strings, a null is assigned.
- For Boolean data, a false is assigned.

Static Variables & Static Methods

A static variable is a variable that can be shared by all the objects of same class. A static variable once initialized cannot be redeclared. Static keyword must be specified before the variable

name. It can be accessed by the class name. The user needs not to create an object for the class in which it is defined.
Similarly a static method is one whose implementation is exactly the same for all the objects of particular class. A static method will have access to static data only.

Properties of Static Variables:

1) They share the same value for all the objects of the class.
2) They share the same memory location for all the objects.
3) They can be accessed without objects.

Properties of Static methods:

1) Static methods can be accessed without creating an object for the class.
2) Static methods have access to only static variables.
3) Static methods can be called directly from static methods.

Ex :

```
Public class demo
{
  public static void main(String s[ ])
  {
  test.display( );
  }
} // end of  main class

class test
{
  public void display( )
  {
    System.out.println( "demo for static methods" );
   }
} // end of other class.
```

Constructor

A constructor is a member function that initializes an object (or) a constructor is a member function or method that gets invoked without making an explicit call to it (or) a constructor is a method that gets called at the time of creation of an object for a class. It is the only method that gets invoked first without the knowledge of the user.

Points about a constructor:

A constructor:

- ✓ must have the same name as the class name.
- ✓ must be defined as public.
- ✓ must return nothing. Even "void" should not be specified.
- ✓ Except initialization a constructor does nothing.

Types of Constructors:

- Default constructor
- Parameterized constructor

Example:

```
Public class demo
{
  public static void main(String s[ ])
  {
    test  t  = new test( ); // instance for class test is created
    t . display( );
  }
} // end of  main class

class test
{
 int a ;
  public test( )     // constructor definition.
   {
     a  = 101;
   }
 public void display( )
  {
     System.out.println( "The value of a is ….."+a);
   }
} // end of other class.
```

Parameterized Constructor:

The compiler creates a default constructor if we do not specify any constructor. A constructor can also have certain parameters or arguments; such a constructor is called as parameterized constructor.

Ex :

```
public class demo
{
  public static void main(String s[ ])
  {
    test  t  = new test(12,11.45f ); // instance for class test is created
    t . display( );
  }
} // end of  main class
class test
```

```
{
  int x ;   float y;
  public test( int a, float b)     // constructor definition.
  {
    x = a; y  = b;
  }
  public void display( )
  {
    System.out.println(  "The values of x & y  are ….." + x +"   "+ y );
  }
} // end of other class.
```

'this' keyword:

"this" is a keyword that refers to the current object in use. It is used in the method of a class. In other words, this keyword refers to the object on which the method was invoked.

Ex1:

```
class test
{
 int a, float b, char c;
 public test ( int a, float f, char c )
 {
   this. a = a;
   this . b = b;
   this . c = c;
 }
}
```

Ex2:

```
class Complex
{
 int r,i;
 public Complex(int r, int i)
 {
  this.r = r;
  this.i = i;
 }
 public void show()
 {
   System.out.println(r+" "+i+"i");
 }
 public Complex add(Complex x)
 {
```

```
    r += x.r;
    i += x.i;
    return this; // it returns current object i.e., object 'a'.
  }
}

public class ComplexDemo
{
 public static void main(String args[])
 {
   Complex a = new Complex(3,4);
   Complex b = new Complex(2,3);

   a.show();
   b.show();
   Complex c = a.add(b);
   c.show();
  }
```

finalize( )

In fact in Java, objects once created can be destroyed automatically. Java provides built-in garbage collector to gather all the unused objects for destruction. So, the programmer needs not to implement any destructor method explicitly. However, sometimes it is necessary to perform certain actions before destroying the object. In such cases implementation of such actions is done in a method called finalize( ).

POLYMORPHISM:

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class. There are two types of polymorphisms:

i. Compile Time polymorphism
ii. Runtime Polymorphism

Method Overloading: Method overloading is a process of providing multiple definitions to the same method. In other words, we can say method overloading provides "one interface multiple methods". In other words, more than one definition can be given to the same method signature. Here method signature is given below: Method signature doesn't include return type.

return type **method name(arguments list)**;
method signature

All the methods are differentiated by means of: types of arguments & number of arguments.
For ex: No. of arguments:

- Show()
- Show(int)
- Show(int,int);

Types of arguments:

- Show(int);
- Show(double)
- Show(long)
- Show(String)

The following signatures are invalid:

- int show()
- String show()
- long show()

Method overloading is commonly used to create several methods with unique name to perform similar kind of tasks, but with different data. i.e., it is possible to define two or more methods within the same class that share the same name, as long as their parameters declarations are different. If this is the case, the whole process is said to be method overloading. It is also known as compile time polymorphism or early binding or compile time binding. Here binding means associating method call with the method definition.
Ex:

```
class demo
{
 int   a , b;
 public void get_data( )
 {
   a = 10;
   b = 20;
 }
 public void get_data( int a )
 {
   this . a  = a;
   b = 30;
 }
 public void get_data( int a  , int  b)
 {
   this . a = a;
   this . b = b;
 }
 public void display( )
```

```
  {
   System.out.println("the values are …….”+a+”   “+b);
  }
}
public class test
{
 public static void main( String s[ ]  )
 {
  demo d = new demo( );
  d.get_data( );
  d. display( );
  demo d = new demo( );
  d.get_data(12 );
  d. display( );
  demo d = new demo( 44,55);
  d.get_data( );
  d. display( );
}
```

Constructor Overloading

An interesting feature of a constructor is that a class can have multiple constructors. This is called as constructor overloading. All the constructors have the same name as the corresponding class, and they differ only in terms of their signature (i.e., in terms of no. of arguments or data types of their arguments or both). However, as a constructor is also a method of a class, it can also be overloaded.

Ex:

```
class demo
{
 int   a , b;

 public demo( )
 {
  a = 10;  b = 20;
  System . out . println("the values are …….”+a+”   “+b);
 }

 public demo( int a )
 {
  this . a  = a;  b = 30;
  System.out.println("the values are …….”+a+”   “+b);
 }
```

```
 public demo( int a  , int  b)
 {
   this . a = a;  this . b = b;
   System.out.println("the values are …….."+a+"   "+b);
 }
}

public class test
{
 public static void main( String s[ ]  )
 {
   demo d1  =  new demo( );
   demo d2  =  new demo( 12);
   demo d3  =  new demo( 12,13);
 }
}
```

INHERITANCE

One of the most striking features of an Object Oriented language is nothing but inheritance. Inheritance is a process of achieving code reusability. With inheritance code duplication is avoided by reusing the code (attributes and methods) of one class in another class. In other words it is a process of extraction of the features of one class into another.

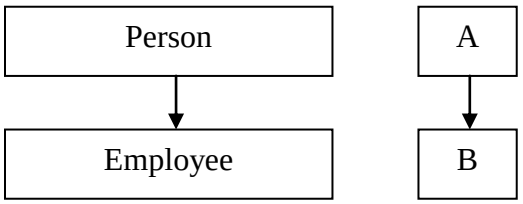
A class that inherits another is said to be a subclass/child class and a class that is being inherited by other is referred as super class/ parent class. To achieve this, use **"extends"** keyword while defining the subclass.

Example: Person → Student → Engineer → Computer Scientist

There are several types of inheritances in OOPs and the list follows:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance: In Single Inheritance, one class inherits the features of only one class. i.e., there exists only one super class & one sub class.



Super class/ base class/parent class

Sub class/derived class/child class

Super Class: a class that is being inherited by some other class is a super class
Subclass: a class that inherits other classes is the sub class.

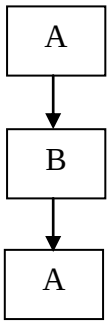
In fact, if class B inherits a class called A, all the methods of A can be invoked through the object of B as if they are defined in class B. The user need not to create an object for the super class to access it's methods as you have an object of it's sub class. Hence we say that all the attributes and methods of public type in super class become public in corresponding sub classes. The following example illustrates the same.

Ex: class A

```
{
  public void  show1( )
  {
   System.out.println( "super class method 1");
  }
  public void show2( )
  {
   System.out.println(" super class method 2 ");
  }
}
class B extends A
{
  public void show3( )
  {
   System.out.println( "sub class method 1 ");
  }
  public void show4( )
  {
   System.out.println( "sub class method 2 ");
  }
}

public class demo
{
  public static void main(String s[ ] )
   {
    B b = new B( );
    b. show1( );
    b. show2( );
    b. show3( );
    b. show4( );
 }
}
```

Multilevel Inheritance: In this inheritance, there will be multiple super classes and sub classes. For example, class A is inherited by class B and class B is inherited by class C and class C is inherited by another class called D and the process will go on. This is referred as multilevel inheritance.

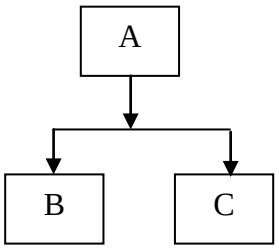


Super class/ base class/parent class

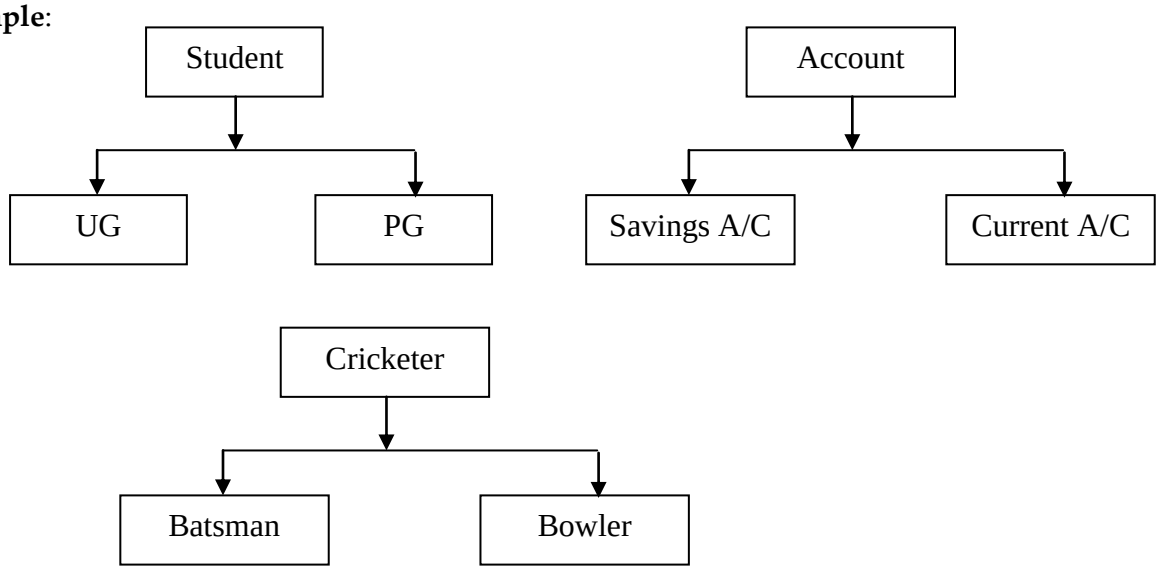
Intermediate super class

Sub class/derived class/child class

Hierarchical Inheritance: This inheritance follows a tree structure. Multiple classes may inherit one class. Hence this category is referred as hierarchical inheritance. i.e., one super class and multiple subclasses.



Example:



This model of inheritance is also known as **IS-A relationship.**
i.e., we can say, student IS-A UG or student IS-A PG. Likewise, an Account IS-A Savings Account or an Account IS-A Current Account. Cricketer IS-A bowler or Cricketer IS-A Batsman.

Method Overriding

In a class hierarchy, when a method in sub class has the same name and type signature as a method in its super class, then the method in the sub class is said to override the method in the super class. The whole process is said to be method overriding. Moreover, When a method in the sub class is called then always the method defined in the sub class only gets invoked.

Points about Method Overriding: **(in** comparison with method overloading)

- ✓ An overriding method replaces the method it overrides.
- ✓ Each parent class method can be overridden at most once in any one subclass.
- ✓ Overriding methods must have identical arguments list that is identical type and order, otherwise they will simply be treated as method overloading.
- ✓ The return type must also be identical.

Some more rules to be observed in Method Overriding

- ✓ An overriding method must not be less accessible than the method it overrides.
- ✓ An overriding method must not throw any checked exceptions that are not declared by the overriding method.

Ex :

```
class   A
{
  public void  show( )
  {
    System.out.println( " super class method ");
  }
}
class  B   extends   A
{
   public void show( )
    {
     System.out.println (" sub class method info ");
    }
}
public class demo
{
  public static void main(String s [ ] )
   {
     B   b  = new  B ( );
     b.show ( );
   }
}
```

Super Keyword

Super is a Java keyword that used to reference no-static methods or variables of that parent class that may be hidden by the current class. There are two different situations where "super" keyword comes in to picture.

In fact, if we define a parameterized constructor in both super and sub classes then it becomes difficult to send arguments to the super class constructor as we do not create an object to it. In such cases, through the sub class constructor one can send/pass arguments to the corresponding super class. To achieve this, we take the help of "**super"** keyword.

Ex :

```
 class superclass
{
  int a , b;
  public superclass( int a, int b )
  {
        this. a = a;  this. b = b;
  }
 public void show ( )
 {
   System.out.println( " the values passed from " + " the sub class are …." +a+ "   " +b);
  }
}
class subclass extends superclass
{
 public subclass(int a, int b)
 {
  super ( a , b);
  System.out.println("arguments are passed successfully");
 }
}
public class superdemo
{
 public static void main(String st[ ])
 {
   subclass s  = new subclass(22 , 33);
   s. show ( );
 }
}
```

Note: While passing arguments from sub class to super class, the first statement must be the super keyword otherwise the compilation error raises.

Abstract Classes

So far the classes you have created and practiced are called concrete classes for the reason that we can create objects from the classes. But some classes do exist where you cannot create objects from the objects. These classes are called **abstract** classes. In some other way we can say that sometimes we need such a super class which doesn't implement methods, instead, it creates a generalized form for a method that will be shared by all of its subclasses. Now such super class is referred as abstract class.
Literally, an abstract class is an incomplete class whose methods are defined in the corresponding subclasses.

Generalized form of abstract method :

abstract return type method _name (arguments list);

Some Points about Abstract Class

- ✓ Any class that contains one or more methods must be declared as abstract.
- ✓ "abstract" is the keyword to be used while defining a class as abstract.
- ✓ No object is created to an abstract class.
- ✓ No abstract constructors & static methods of abstract type can be defined.
- ✓ Along with abstract methods, complete methods can also be defined.

Ex :

```
abstract class Area
{
  public void area ( ) ;
}
class Square extends Area
{
   int a;
   int result;
  public void area ( )
   {
     result = a * a;
      System.out.println ( "the area of a square is……"+ result );
   }
}
```

```
class Rectangle extends Area
{
  int b, h;
   public void area ( )
   {
    b = 5; h = 6;
    result  =  b * h;
    System.out.println ( "the area of a rectangle is ::::::::::"+ result );
   }
}
public class AbstractDemo
{
 public static void main( String s [ ] )
 {
  Square s = new Square ( );
  Rectangle r = new Rectangle( );
  s .area( );
  r . area( );
}
}
```

Dynamic Method Dispatch

The other advantage of Method Overriding is nothing but Dynamic Method Dispatch. It is a mechanism by which a call to an overridden method is resolved at runtime, rather than compile time. The concept o dynamic method dispatch is vital because this is how Java implements or achieves runtime polymorphism.

In Dynamic Method Dispatch, a general principle of holding or referring a subclass object through the super class object reference variable is implemented.

For example, there is a super class A being inherited by two subclasses B & C and all the three classes are implementing overridden methods. Now, Java determines which version of those methods to be executed at run time. In some other way, "it is a type of the object being referred to" that determines which version of an overridden method is executed.

Ex :

```
abstract class  Super
{
 abstract public void display( );
}

class Subclass1 extends Super
{
```

```
 public void display( )
{
 System.out.println( " this is a sub class method1");
}
}  // end of subclass 1

class Subclass2 extends Super
{
 public void display( )
{
 System.out.println( " this is a sub class method2");
}
} // end of subclass 2

public class DynamicMethodDispatch
{
 public static void main(String s[ ])
 {

   Super  ss;  // object reference of  class "Super "

  int a;
  a  =  Integer . parseInt (s[0]);
        if ( a == 1)
          ss  = new Subclass1();  // obtaining the reference of subclass 1
        else
        if ( a == 2)
          ss =  new Subclass2(); ;  //obtaining the reference of subclass 2
   ss . display ( ) ;
  }
}
```

final keyword

Java has given a new way of approach to prevent method overriding and inheritance as well. In certain situations, the user may not want to override a method. To achieve this, use **final** keyword. Any method declared as final in a super class can not be overridden in the corresponding sub class.

Points of final keyword

- ✓ final variable cannot be reassigned.
- ✓ final method cannot be overridden.

- ✓ final class cannot be inherited.
- ✓ Every method in final class is implicitly final.
- ✓ static & private methods are implicitly final.

Interfaces

An interface is a similar to a class but it consists of all incomplete or abstract methods. An interface serves the purpose of implementation of multiple inheritances. The keyword **"interfaces "**is to be used to define an interface.

An interface is inherited by a class with the help of a keyword **"implements"**. A class can extend an interface, an interface can inherit another interface but an interface cannot inherit a class. In order to inherit an interface with another interface by means of **extends** keyword only.

Generalized form of an interface

```
interface interface_name
{
  return type methodname(arg.list);
  return type methodname(arg.list);
  - - - - - - - -
}
```

Ex:

```
interface interface1
{
 public void show1( );
 public void show2( );
}
class Myclass implements interface1
{
  public void show1( )
  {
    System.out.println( "this is the first method implementation of interface " );
   }

  public void show2( )
  {
    System.out.println( "The second method implementation  of interface " );
   }
}
public class interfacedemo
{
```

```
 public static void main(String s[ ] )
  {
    Myclass m = new Myclass( );
    m.show1( );
    m.show2( );
  }
}
```

Points to be noted about interfaces:

- All the fields (also known as variables) in a given interface are defined automatically as static and final even if we do not specify.
- All the methods in a given interface are automatically abstract & public even if we do not specify.
- No method must be defined.
- A class may inherit any no. of interfaces.
- All the methods in an interface must be declared as public while defining in the corresponding subclasses.
- We can create reference to an interface just like an abstract class.
- One interface can extend another interface.
- An interface can't inherit a class.

Note: While inheriting an interface, a class must use "implements" keyword and while inheriting an interface to another interface, use "extends" keyword.

Relationships:

The relationship between any two classes can be as follows:

- **IS-A** Relationship (through Inheritance)
- **HAS-A** Relationship (object as member)

IS-A Relationship is nothing but Inheritance that too Hierarchical Inheritance.

HAS-A Relationship is given below: It represents a class that holds the object of another class. i.e., a class HAS An object of some other class. In such situation the first class is the container class. Suppose class A holds object of class B as a member then A is the container class.

Ex:

```
class B {
 public void show()
 {
        System.out.println("Hello");
 }
```

```
}

class A
{
 B x;
 public void display()
 {
        x = new B();
        x.show();
        System.out.println("Hai");
 }
}

public class TDemo
{
 public static void main(String s[])
 {
    A m = new A();
    m.display();
 }
}
```

PACKAGES

A package is a container that holds a collection of classes & interfaces. or a package is an association of several classes & interfaces. In some other way, related classes & interfaces that are grouped together in a directory on disk are said to be packages. Package provides a convenient approach for

- Maintaining group of classes and interfaces.
- Avoiding naming conflicts. We can keep the methods with the same signature in different packages.

Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality.

It helps **Organize** your classes into a folder structure and make it easy to locate and use them. More importantly, It helps improve **re-usability.**

In general all the user defined java classes get stored in a default package. Packages & interfaces are the basic elements for a Java program. All the predefined classes & interfaces of Java are existed in terms of predefined packages. For example, java.lang, java.util, java.net, java.awt etc., together the packages are said to be Java API (Application Programming Interface).

A package is to be included in the user application so that the predefined classes can be used. To achieve this, one can use a keyword called **"import"**. Packages are of two types: predefined & user defined. The following is the list of predefined packages:

- java.lang- essential classes like thread, object and string
- java.awt – graphical user interface components
- java.io - input/output streams and files
- java.net - networking related classes
- java.applet – applet creation
- java.util - utility classes like data and classes for data structures
- java.rmi - classes for remote method invocation
- java.sql - classes for interacting with a database

For example, the statement **import java.lang.*;** includes all the predefined Java language classes in the current application. However, java.lang is a package that is by default imported.

User Defined Packages:

To define user packages, use a keyword "package". A user defined package may be defined if there exists more no. of classes in the application to be developed. General form of package definition is :

package *pkg name* ;

This statement must be the first statement to place the class in the specified package. A hierarchy of packages can be defined as follows :

Package *pkg1 [. pkg2 [. pkg3*]] ;

Here *pkg2* is the sub package of *pkg1* and *pkg3* is the sub package of *pkg2*.
All the Java predefined packages are existed in the **"CLASSPATH"** set by the environmental variable.

Creating packages: Packages are defined using the package statement as under:

```
package Mypkg;
public class MyClass
{
        // body of the class;
}
```

Creating the package involves the following steps:

2. Declare th package at the beginning of a file using th form :*package <packagename>.*
3. Define the class that is to be put in the package and declare it public.
4. Create the subdirectory under the directory where the main source files are stored.
5. Store the listing as the class name java file in the subdirectory created.

6. Compile the file. This creates the class file in the sub-directory.

Classpath: The java compiler and the java interpreter, for locating packages, use the classpath environment variable. To the existing value of the of this variable the path to the newly created package should be added under.

set CLASSPATH=*path1;path2*

or

set classpath=%classpath%; c:\Mypkg;

Importing classes in Package

The classes and interfaces defined in a package can be made use of in any program file with the import statement. Following are the two forms of the import statement:

import <packagename>.<classname>;
import pacakgename.*;

example:

```
package p;
public class Complex
{
 int r,i;
 public Complex(int r, int i)
 {
  this.r = r;
  this.i = i;
 }
 public void show()
 {
   System.out.println(r+" "+i+"i");
 }
 public Complex add(Complex x)
 {
    r += x.r;
    i += x.i;
    return this;
 }
}
```

Save the above program as 'Complex.java' in the current working directory. Compile it as follows:

F:\jpro> javac –d . Complex.java

It creates a folder with the name 'p' in **F:\jpro** and places the Complex.class file in '**F:\jpro\p'**. Here '–d ' is a switch represents directory. And dot(.) represents the current working directory.
After this, create another file with the name DemoComplex.java and edit the following code in the file:

```
public class DemoComplex
{
        public static void main(String args[])
        {
                p.Complex a = new p.Complex(3,4); // creating object for Complex
                                                  // Class in package p
                p.Complex b = new p.Complex(2,3);

                a.show();
                b.show();
                p.Complex c = a.add(b);
                c.show();
        }
}
```

(Or)

```
import p.Complex;

public class DemoComplex
{
 public static void main(String args[])
 {
   Complex a = new Complex(3,4);
   Complex b = new Complex(2,3);

   a.show();
   b.show();
   Complex c = a.add(b);
   c.show();
  }
}
```

Save and compile the above & we get the expected results.

Access Control methodology

Access to variables and methods is done through access modifiers in Java. Access modifiers specify the levels of access between class members and outside world (i.e., objects of other classes). Java provides support for four types access modifiers:

- private.
- public.
- protected.
- default (when no modifier is specified).

Private: If a variable or method is defined as private, no class outside of the current class(including subclass) has access to it. It is the most restrictive access modifier.

public: It has global visibility. It is assumed as most generous access modifier as it provides global accessibility to the variables and methods irrespective of class.

Protected: It works sometimes as public and sometimes as private. i.e., in all the subclasses, the methods and variables of a class work as public and in all the non-subclasses, they work as private.

default: If we do not specify any modifier, Java considers the method or variable as default. All the variables and method that are defined as default are public if the class that wants to have an access to these variables & methods is existed in the same package where the original class is existed. i.e., if a variable or method is not specified any access modifier, then only classes (including subclasses) in the same package can have access.

	private	**no modifier**	**protected**	**public**
same class	Yes	Yes	Yes	Yes
same package subclass	No	Yes	Yes	Yes
same package non- subclass	No	Yes	Yes	Yes
different pack-age subclass	No	No	Yes	Yes
different pack-age non subclass	No	No	No	Yes

Method Overloading Vs Method Overriding:

S. No	Method overloading	Method Overriding
1	Overloading occurs when two or more methods in the same class share the same name.	Overriding is completely different. If a derived class requires a different definition for an inherited method, then that method can be redefined in the derived class. This would be considered overriding.

2	Ex:
	
	void changeDate(year)
	void changeDate(month,year)
	void changeDate(day,month,year)
		Ex:
		Class A {
		public void method() {
		……
		}
		}
		Class B extends A {
		public void method() {
		……
		}
		}

3	Overloading doesn't require inheritance	Overriding requires inheritance
4	Constructors can be overloaded	Constructors can't be overridden
5	Here all overloaded methods perform same operation.	Here overridden methods can perform different operations
6	Method signatures are different. They vary in arguments.	Including arguments method signatures are identical.
7	Return type is not considered	Return type of the methods must be identical.

Restrictions of access modifiers in Method Overriding:

- ✓ A private method can be overridden by a private, default, protected and public methods.
- ✓ A default method can be overridden by a default, protected and public methods.
- ✓ A protected method can be overridden by a protected and public method.
- ✓ A public method can be overridden by only a public method.

Container Class: A class that holds an object of some other class is nothing but a container class. A container holds one or more child components. A container has a layout that determines how the child components are arranged within the container.

class B {

```
 public void show()
 {
        System.out.println("Hello");
 }
}

class A
{
 B x;
 public void display()
 {
        x = new B();
        x.show();
        System.out.println("Hai");
 }
}

public class ContainerDemo
{
 public static void main(String s[])
 {
    A m = new A();
    m.display();
 }
}
```

Inner Classes:

An inner class is a class that is defined inside another class. There are four reasons:

- An object of an inner class can access the implementation of the object that created it—including data that would otherwise be private.
- Inner classes can be hidden from other classes in the same package.
- Anonymous inner classes are handy when you want to define callbacks on the fly.
- Inner classes are very convenient when you are writing event-driven programs

Ex:

```
class Outer
{
  private int a;
```

```
  public Outer()
  {
     a = 10;
  }

  private class Inner
  {
     private int b;

     public void disp()
     {
      b = 20;
      System.out.println("Outer class data: "+a+"  "+"Inner class Data: "+b);
     }

  }

  public void show()
  {
     Inner  y = new Inner();
     y.disp();
     System.out.println(a);
  }

}

public class InnerDemo
{
 public static void main(String s[])
 {
    Outer x = new Outer();
    x.show();
  }
}
```

Inner classes are a major addition to the language. Inner classes are anything but simple. The syntax is complex. Inner classes are translated into regular class files with $ (dollar signs) delimiting outer and inner class names and the virtual machine does not have any special knowledge about them.

Sometimes inner classes are used only once in a method, in such cases we define an inner class locally. Such classes are referred to as local inner classes.

Local classes are never declared with an access specifier (that is, public or private). Their scope is always restricted to the block in which they are declared.

Local classes have a great advantage—they are completely hidden from the outside world.

Local classes have another advantage over other inner classes. Not only can they access the fields of their outer classes, they can even access local variables! However, those local variables must be declared final.

When using local inner classes, you can often go a step further. If you want to make only a single object of this class, you don't even need to give the class a name. Such a class is called ***anonymous inner class***.

Local Class:

A *local class* is declared locally within a block of Java code, rather than as a member of a class. Typically, a local class is defined within a method, but it can also be defined within a static initializer or instance initializer of a class. Because all blocks of Java code appear within class definitions, all local classes are nested within containing classes. For this reason, local classes share many of the features of member classes.

The defining characteristic of a local class is that it is local to a block of code. Like a local variable, a local class is valid only within the scope defined by its enclosing block.

Features of Local Classes: Local classes have the following interesting features:

- Like member classes, local classes are associated with a containing instance, and can access any members, including private members, of the containing class.
- In addition to accessing fields defined by the containing class, local classes can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition and declared final.

Restrictions on Local Classes: Local classes are subject to the following restrictions:

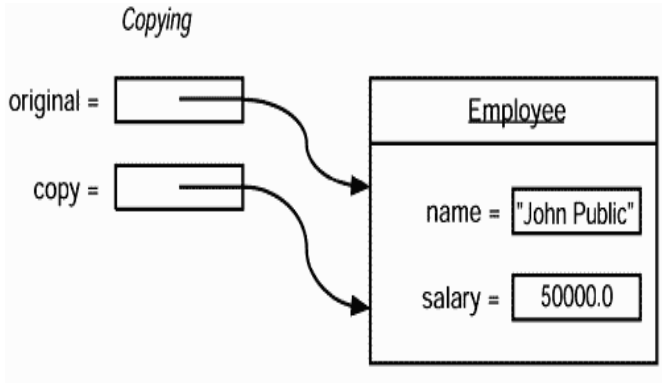
- A local class is visible only within the block that defines it; it can never be used outside that block.

- Local classes cannot be declared public, protected, private, or static. These modifiers are for members of classes; they are not allowed with local variable declarations or local class declarations.
- Like member classes, and for the same reasons, local classes cannot contain static fields, methods, or classes. The only exception is for constants that are declared both static and final.
- Interfaces cannot be defined locally.
- A local class, like a member class, cannot have the same name as any of its enclosing classes.

OBJECT CLONING:

When you make a copy of a variable, the original and the copy are references to the same object. This means a change to either variable also affects the other.

```
Employee original = new Employee ("John Public", 50000);
Employee copy = original;
```




```
Employee copy = (Employee)original.clone();
  // must cast—clone returns an Object
```