UNIT –II

Classes, Arrays, Strings and Vectors: Classes, Objects and Methods: Introduction, Defining a Class, Adding Variables, Adding Methods, Creating Objects, Accessing Class Members, Constructors, Methods Overloading, Static Members, Nesting of Methods, Inheritance: Extending a Class Overriding Methods, Final Variables and Methods, Finalizer methods, Abstract Methods and Classes, Visibility Control. Arrays, Strings and Vectors: Arrays, One-dimensional Arrays, Creating an Array, Two -Dimensional Arrays, Creating an Array, Two – dimensional Arrays, Strings, Vectors, Wrapper Classes.

1. **Define class in java?**

Class:-

A class is a keyword in java.
Definition: A class is blue print of an object.

OR

A class is a group of objects, data members & its associated member functions.
A class is also known as user defined data type.

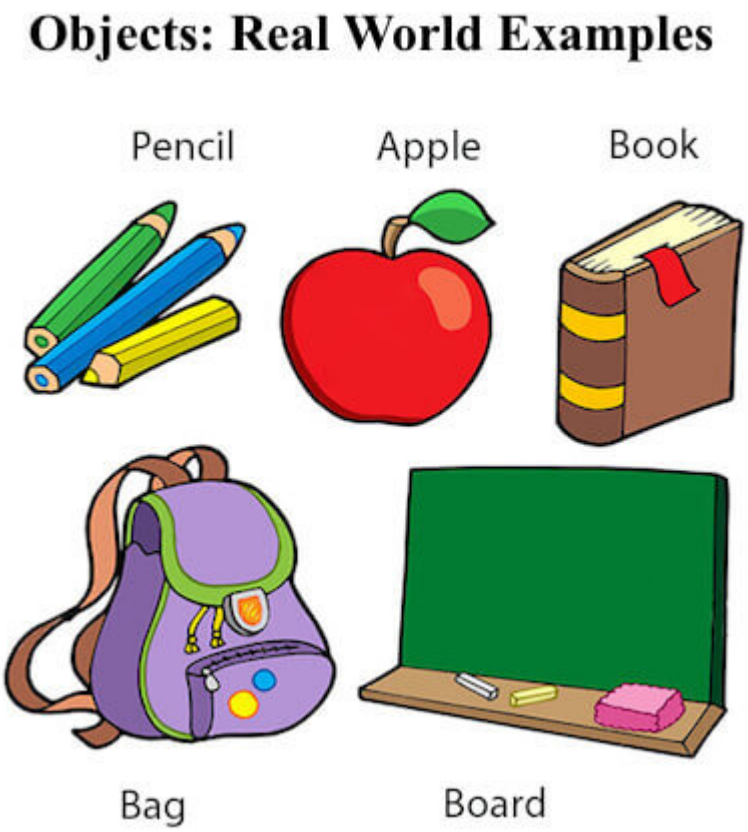
Syntax:

```
class<classname>
{
  //collection of objects
}
```

Example:

```
class A
{

public static void main(String[] arg)
{
 System.out.println("welcome");
}
}
```

2. Define object in java?



An entity that has state and behavior is known as an object

Example:

chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible).

The example of an intangible object is the banking system.

Syntax:

classname<objectname> =new classname();

in java objects can create by using new operator/keyboard.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.

- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Object Definitions:

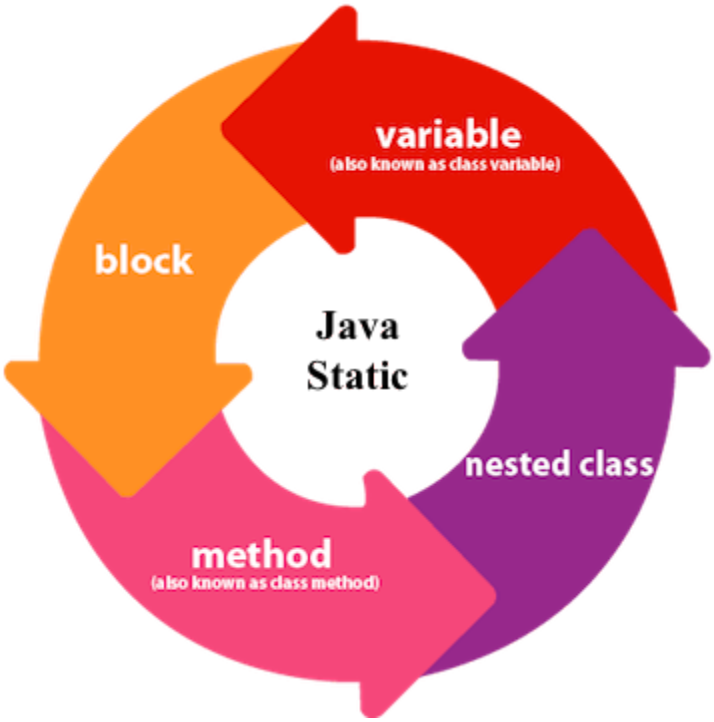
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

Java static keyword

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

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

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

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student
{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n)
{
   rollno = r;
   name = n;
   }
   //method to display the values
   void display ()
{
System.out.println(rollno+" "+name+" "+college);
}
}
//Test class to show the values of objects
public class TestStaticVariable1
{
 public static void main(String args[])
{
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

Example:

```
class Student
{

int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[])
{
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Define constructor?

- Constructor is a special member function, which contains same name as a class name.

- Constructor don’t a have return type (not even void)
- Constructor is a non-static method.
- Constructors will execute for every object request.
- Constructors can’t be inheriting from one class to another class.
- Constructors can be overload but it will not be override.
- Duplicate constructors are not possible in java.

Syntax:

```
class Sample
{
   Sample()
  {
     //constructor block
}
}
```

Example:

```
class Sample
{
   Sample()
  {
    System.out.println(“I’m constructor”);
}
public static void main(String[] args)
{
 Sample s1=new Sample ();
}
}
```

Example 1:

```
package app;

public class Constructor
{
	/* constructor:
	 * 1. constructor is a special member function.
	 * 2. because it contains same name as a class name.
	 * (the class name & constructor name should be same).
	 * 3. constructor is a non-static(method) definition block.
	 * 4. constructors will execute while creating an object.
	 * 5. constructors will execute for every object request.
	 * 6. constructors will overload but it will not override.
	 * 7. constructors d'not have any return type not even (void).
	 * 8. constructors will not inherited from one class to another class.
```

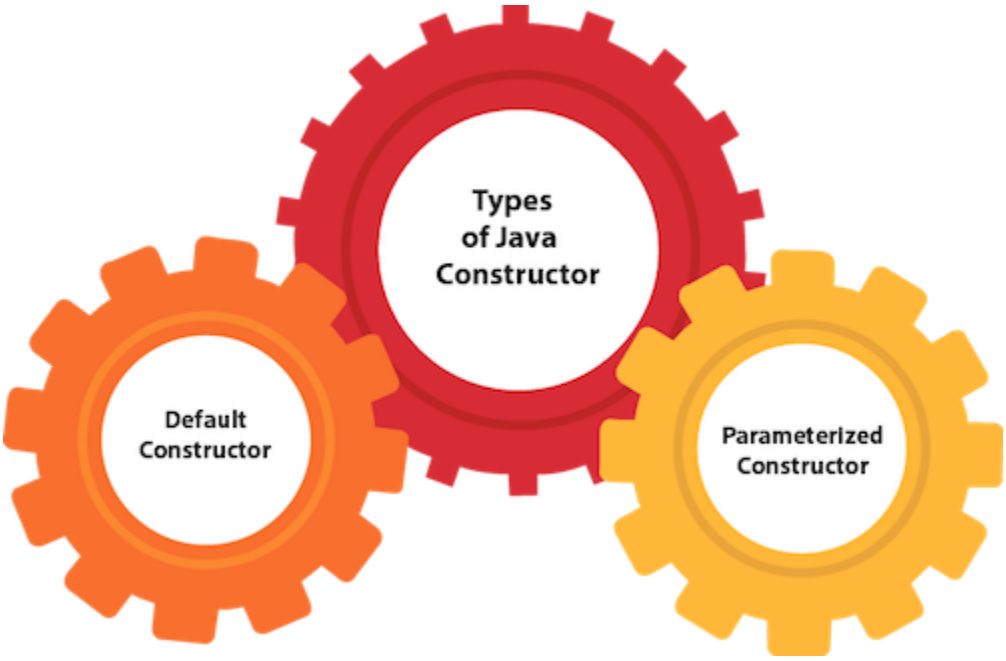
```
        */

           Constructor()
           {
                  System.out.println("I'm a constructor-block");
           }

       public static void main(String[] args)
       {
              Constructor c1=new Constructor();
              Constructor c2=new Constructor();

       }

}
```

Explain different types of constructors in java?

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

<class_name>()

{

//constructor body;

}

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class.

It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Java
{
//creating a default constructor
Java()
{
System.out.println("Bike is created");
}
//main method
public static void main(String args[])
{
//calling a default constructor
Java j=new Java();
}
}
```

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

we have created the constructor of Student class that have two parameters.

We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4
{
   int id;
   String name;
   //creating a parameterized constructor
   Student4 (int i, String n)
{
   id = i;
   name = n;
   }
   //method to display the values
   void display()
  {
```

```
System.out.println (id+" "+name);
}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4 (111,"Karan");
    Student4 s2 = new Student4 (222,"Aryan");
    //calling method to display the values of object
    s1.display ();
    s2.display ();
   }
}
```

What is constructor overloading? Explain constructor overloading with an example?

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.

They are arranged in a way that each constructor performs a different task.

They are differentiated by the compiler by the number of parameters in the list and their types.

Example:

```
/*Constructor Overload:
 * 1. def: two or more constructors are having same name as a class name
 * with different list of parameters is known as constructor overloading.
 *
 * Syntax:

 * class A
 * {
 *     A()
 *     {
 *       //con-1
 *     }
 *     A(int i)
 *     {
 *       //con-2
 *     }
 *     A(float f)
 *     {
 *       //con-3
```

```
 *     }
 *      constructor signature:
 *
 *      1. name of the constructor
 *      2. list of parameters
 *      3. order of the parameters.
 *
 */
```

Example:

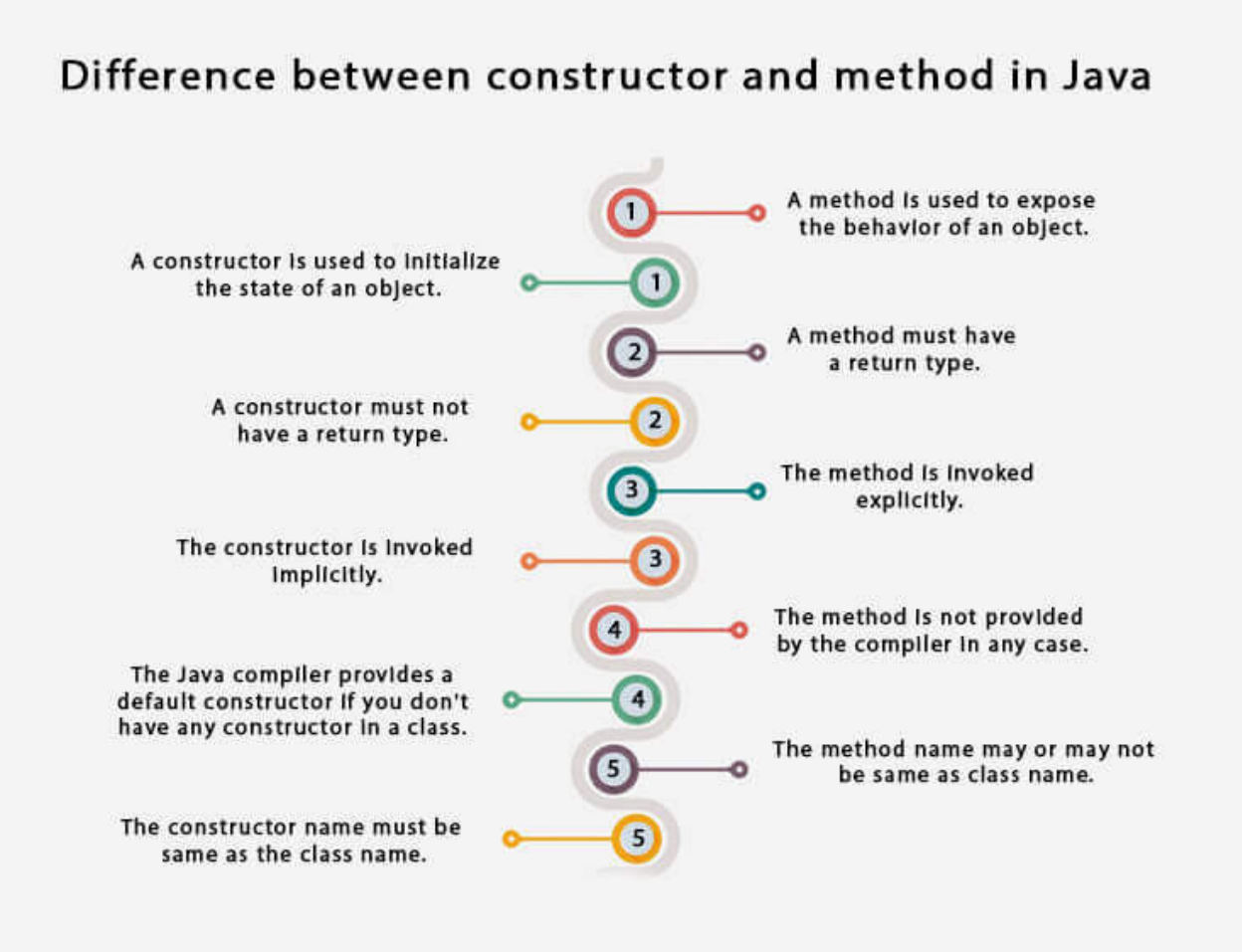
```
public class ConstructorOverload
{
	ConstructorOverload ()
	{
		System.out.println (" no list of parameters : ");
	}
	ConstructorOverload (int i)
	{
		System.out.println (" one list of parameters:");
	}

	ConstructorOverload (int i, int j)
	{
		System.out.println (" two list of parameters:");
	}

	public static void main (String[] args)
	{
		ConstructorOverload c1=new ConstructorOverload();
		ConstructorOverload c2=new ConstructorOverload (200);
		ConstructorOverload c3=new ConstructorOverload (10, 20);


	}

}
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.

The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.



Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class
- /Java program to initialize the values from one object to another object.
-

```
class Student6
{
   int id;
   String name;
   //constructor to initialize integer and string
   Student6 (int i, String n)
{
   id = i;
   name = n;
   }
   //constructor to initialize another object
   Student6 (Student6 s)
{
   id = s.id;
   name =s.name;
   }
   void display()
{
System.out.println (id+" "+name);
}

   public static void main(String args[]){
   Student6 s1 = new Student6 (111,"Karan");
   Student6 s2 = new Student6 (s1);
   s1.display ();
   s2.display ();
  }
}
```

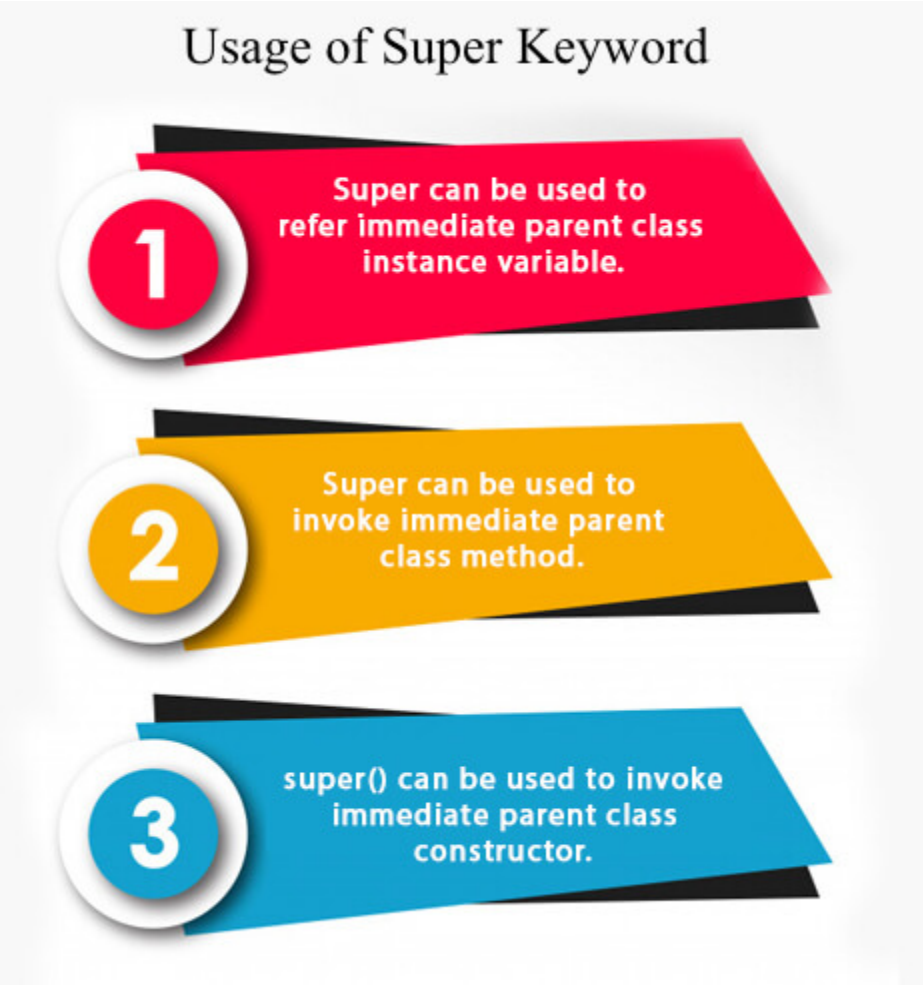
Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.



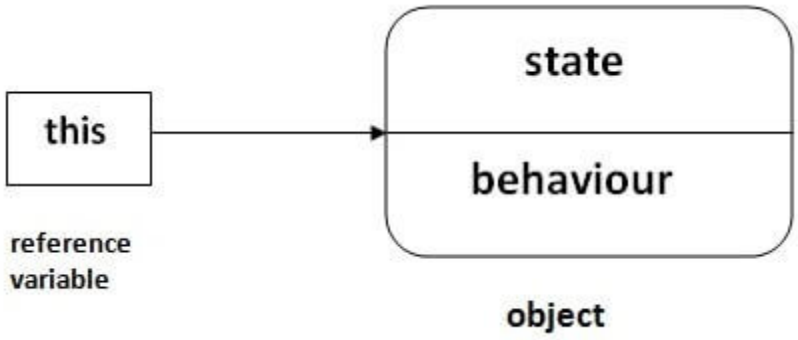
1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal
{
String color="white";
}
class Dog extends Animal
{
String color="black";
void printColor()
{
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1
{
public static void main(String args[])
{
Dog d=new Dog();
d.printColor();
}
}
```

this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

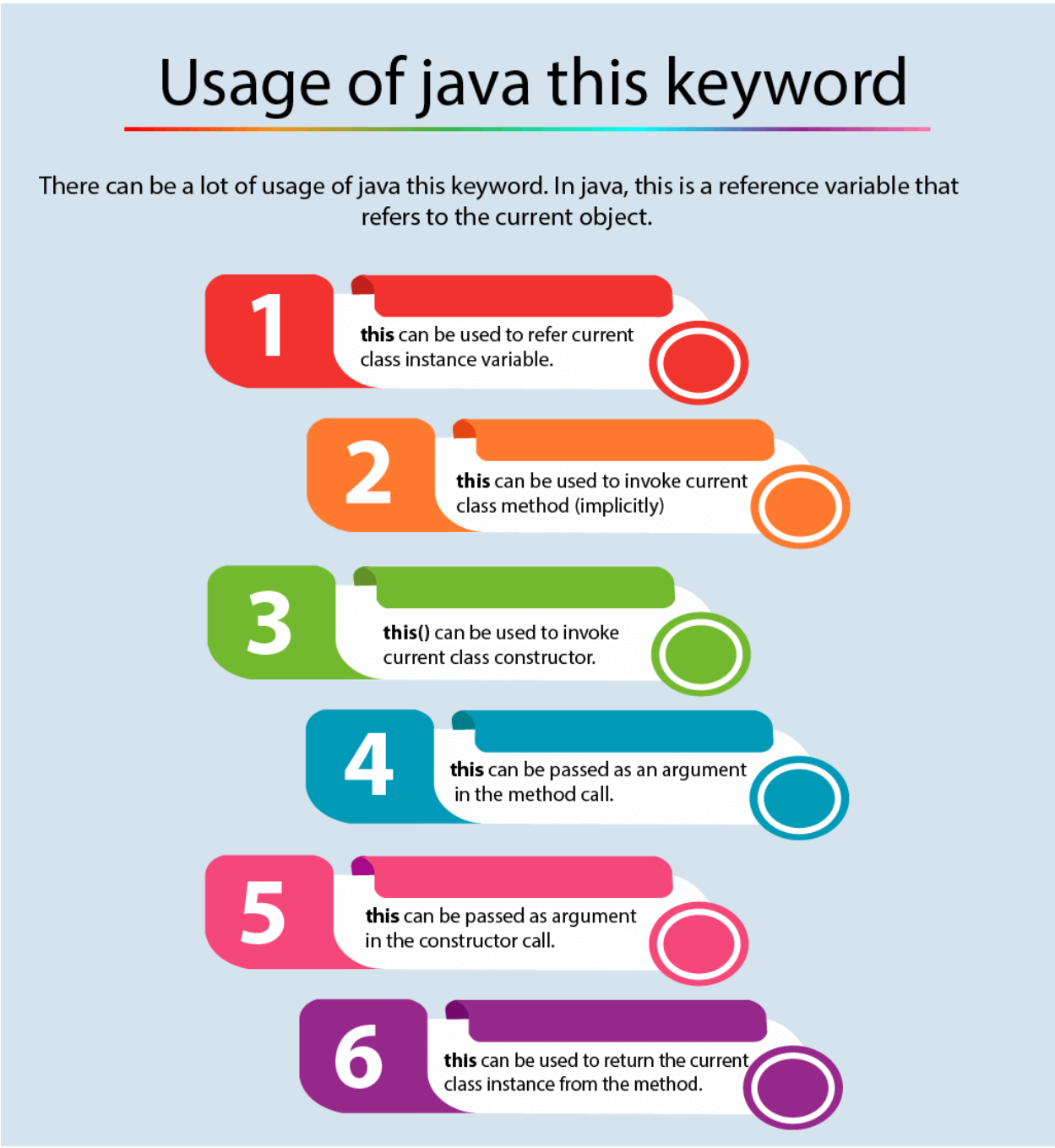


Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Suggestion: If you are beginner to java, lookup only three usage of this keyword.




```
package app;

public class this1
{
   int i,j;
       public this1(int i, int j)
       {
              this.i=i;
              this.j=j;
              System.out.println(i);
```

```
            System.out.println(j);
        }
        public static void main(String[] args)
        {
           this1 t1=new this1(10,20);

        }

}
```

What is method overloading? Explain method overloading with an example?

Method Overloading in Java

Definition:

Two or more methods should have a same method name with different list of parameters in a single class is known as Method Overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

```
   package app;

public class MethodOverload
{
```

```
/*
 * Method Overloading:
 *
 * Definition::
 *
```

Two or more methods should have a same method name with different list of parameters in a single class is known as Method Overloading.

```
 *
 * Note:
 *
 * 1. Both static & non-static methods will be overload.
 * 2. if a method is static object is optional.
 * 3. if a method is non-static we must create an object.
 *
 * Imp note:
 *
 * Method overloading is mainly using to avoid duplication.
 * the duplication can be avoid by using method signature.
 *
 * the method signature contains:
 *   1. Name of the Method
 *   2. List of Parameters.
 *   3. Order of the Parameters.
 *
 *
 *   Method Overloading is mainly used to perform polymorphism.
 *
 *
 */


 //non-static method overloading:



  int water()
 {
       System.out.println ("liquid state");
       return 0;
 }

  int water(int i)
 {
       System.out.println (" solid state");
       return 0;

 }

  int water(long l)
 {
       System.out.println ("Gas State");
       return 0;
 }
```

```
	public static void main(String[] args)
	{
		MethodOverload m1=new MethodOverload ();
		m1.water ();
		m1.water (25);
		m1.water (2);
	}

}
```

What is Method Overriding? Explain Method Overriding with an example?

Definition:

Two or more methods should contains the same method name with same list of parameters in different classes is known as Method Overriding.

```
package app;

class PU
{
	void exam()
	{
		System.out.println(" PU examination");
	}
}

class UG extends PU
{
	void exam()
	{
		System.out.println(" UG Examination");
	}
}

class PG extends UG
{
	void exam()
	{
		System.out.println(" PG Examination:");
	}
}

public class MethodOverriding
{
	/*
	 * Method OverRiding:
	 *
	 * Def:
```

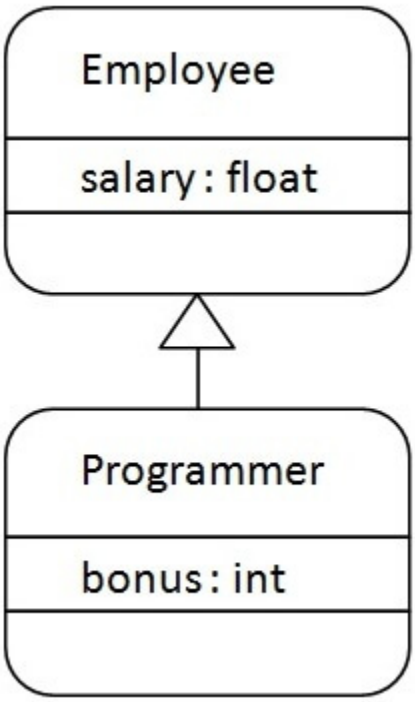
```
         *
         * Two or more methods should contains the same method name with same
          list of parameters in different classes is known as Method Overriding.
         *
         * Note:
         * 1. only non-static methods will override.
         * 2. when method is non-static we must create object.
         *
         * syntax:
         *
         * class PU
         * {
         *    void exam()
         *    {
         *        // PU exam
         *    }
         *  }
         *
         *  class UG extends PU
         *  {
         *     void exam()
         *     {
         *        // UG exam
         *     }
         *  }
         *
         */

        public static void main(String[] args)
        {
               //object creation

                PU p=new PU ();
                  p.exam ();
                  UG u=new UG ();
                  u.exam ();
                  PG p1=new PG ();
                  p1.exam ();

        }

}
```

Difference between method overloading and method overriding in java

No.	Method Overloading	Method Overriding

1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Inheritance:



Inheritance:
1. Inheritance is a process of creating a new class from an existed class.
2. The newly created class should have at least one property of its parent class.
3. The newly created classes are known as child class/ derived class/sub class.
4. The existed classes are known as parent class/ base class /super class.

Note:

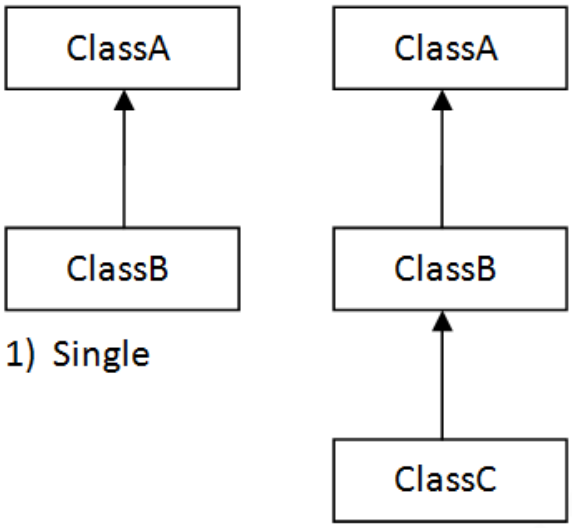
extends:
1. extends is a keyword in java.
2. It is used to make a relation b/w parent and child class.
(Super and subclass) or (base class & derived class).

Syntax:

```
class A extends Object
{
  // base class
}
```

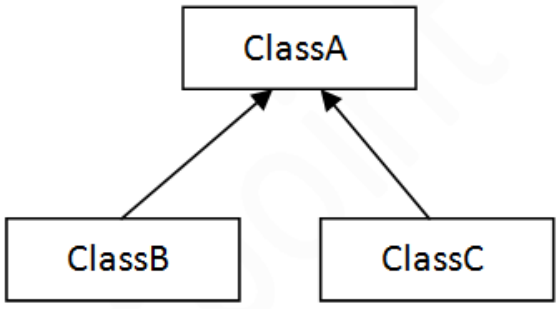
```
   public static void main(String[] args)
   {
          System.out.println (“Inheritance :”);
   }
}
```

Types of Inheritance:



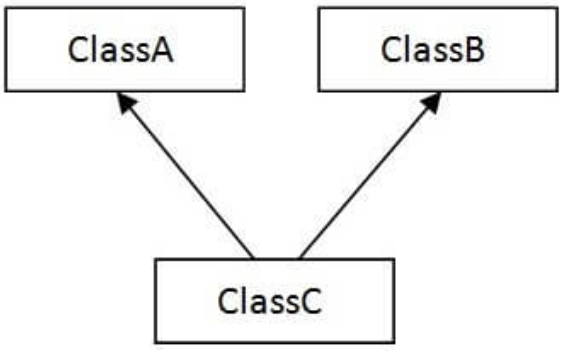
1) Single

2) Multilevel

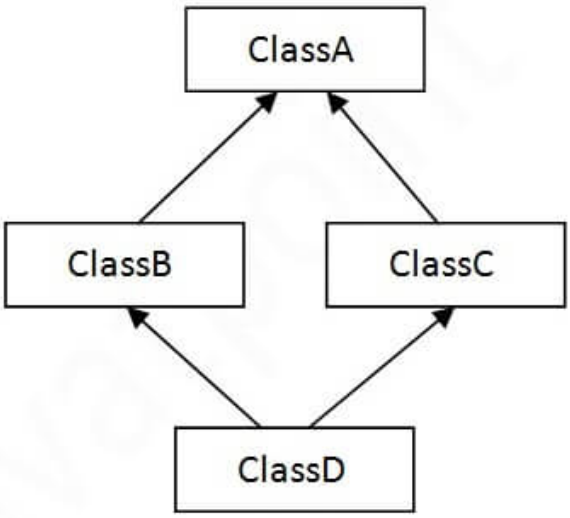


3) Hierarchical

Note: Multiple inheritance is not supported in Java through class.



4) Multiple



5) Hybrid

Single Inheritance:-

Single inheritance is an inheritance, which contains only one parent class and only one child class.

Syntax:

```
class A
 {
    // parent class
 }
 class B extends A
 {
    //
 }
```

```
package app;

public class SingleInheritance extends Object
{

	void display()
	{
		System.out.println (" it is a non-static method");
	}
	public static void main(String[] args)
	{
		SingleInheritance s1=new SingleInheritance ();
		s1.display ();
	}

}
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*.

When a class extends a class, which extends another class then this is called **multilevel inheritance**. ... So in this case class C is implicitly inheriting the properties and methods of class A along with class B that's what is called **multilevel inheritance**.

```
package app;

class MCA1
{
	void exam()
	{
		System.out.println(" MCA examination:");

	}
```

```
}

class MBA extends MCA1
{
	void exam()
	{
		System.out.println("MBA examination");
	}
}
class MCOM extends MBA
{
	void exam()
	{
		System.out.println(" MCOM Examination:");
	}
}



public class MultiLevel
{


	public static void main(String[] args)
	{
		MCA1 m1=new MCA1();
		m1.exam();
		MBA m2=new MBA();
		 m2.exam();
		 MCOM m3=new MCOM();
		   m3.exam();
	}

}
```

Hierarchical Inheritance

When two or more classes inherit a single class, it is known as *hierarchical inheritance*.

```
class Animal
{
void eat()
{
System.out.println ("eating...");
}
}
class Dog extends Animal
{
void bark()
```

```
{
System.out.println ("barking...");
}
}
class Cat extends Animal{
void meow()
{
System.out.println ("meowing...");
}
}
class TestInheritance3
{
public static void main(String args[])
{
Cat c=new Cat ();
c.meow ();
c.eat();
//c.bark();//C.T.Error
}
}
```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

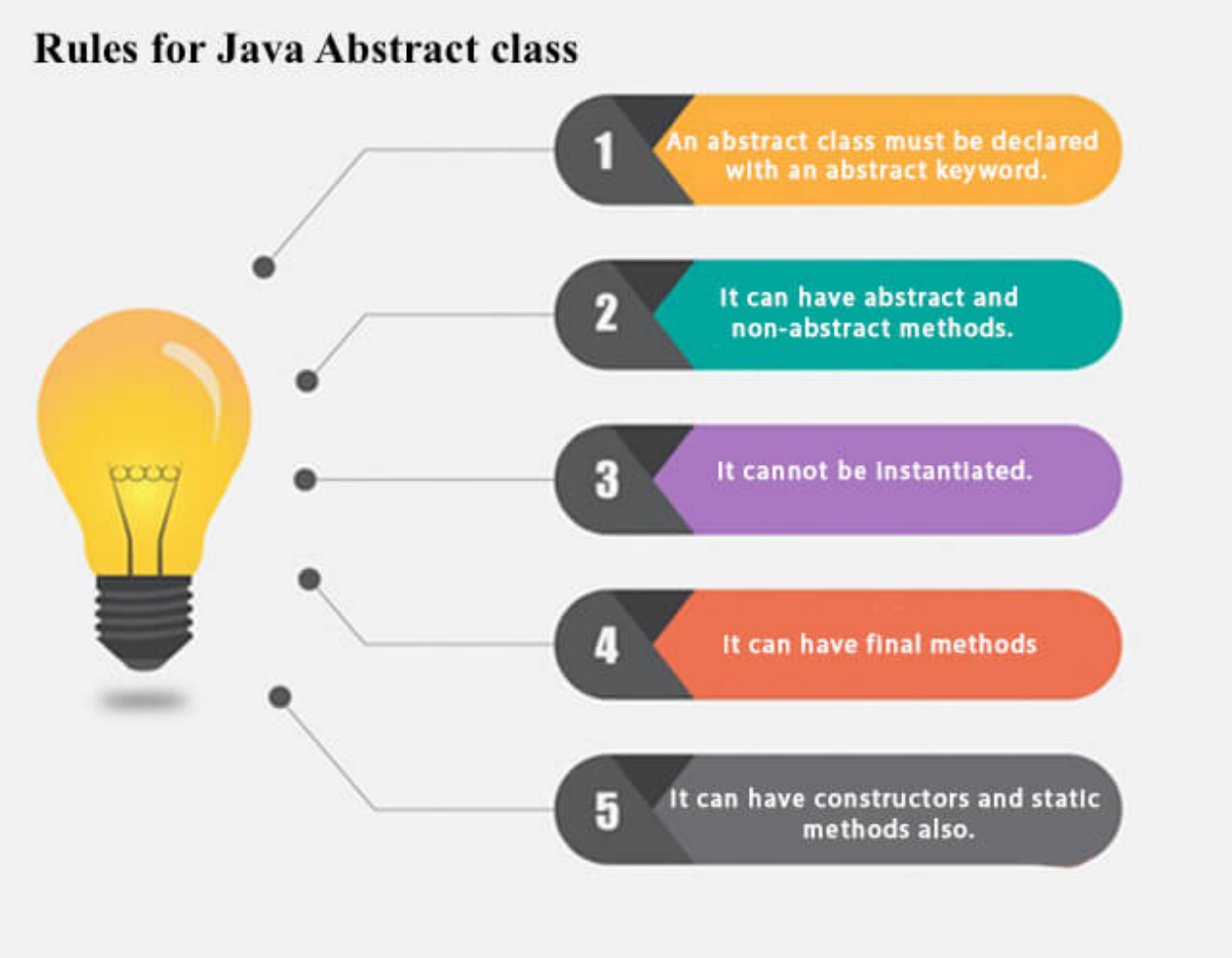
```
class A
{
void msg()
{
```

```
System.out.println ("Hello");
}
}
class B
{
void msg()
{
System.out.println ("Welcome");
}
}
class C extends A,B
{   //suppose if it were

 public static void main(String args[])
{
   C obj=new C();
   obj.msg (); //Now which msg() method would be invoked?
}
}
```

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.



Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

```
package app;

abstract class MCA
```

```
    {
        abstract void read();
        abstract void result();
        //abstract void read(int i);
        void eat()
        {
               System.out.println(" concrete method");
        }
    }

    class MSC extends MCA
    {
        public void read()
        {
               System.out.println(" read method got implemented");
        }
        public void result()
        {
               System.out.println(" result method got implemented");
        }
    }
    public class Abstract3 {

        public static void main(String[] args)
        {
               MSC m1=new MSC();
               m1.read();
               m1.result();
               m1.eat();

        }

    }


package app;

abstract class PU12
{
       abstract void puexam();
       void result()
       {
              System.out.println("all pass");
       }
}
  abstract class UG12 extends PU12
  {
         abstract void ugexam();
         void result()
         {
                System.out.println("all pass-concrete");
         }
  }

class PG12 extends UG12
```

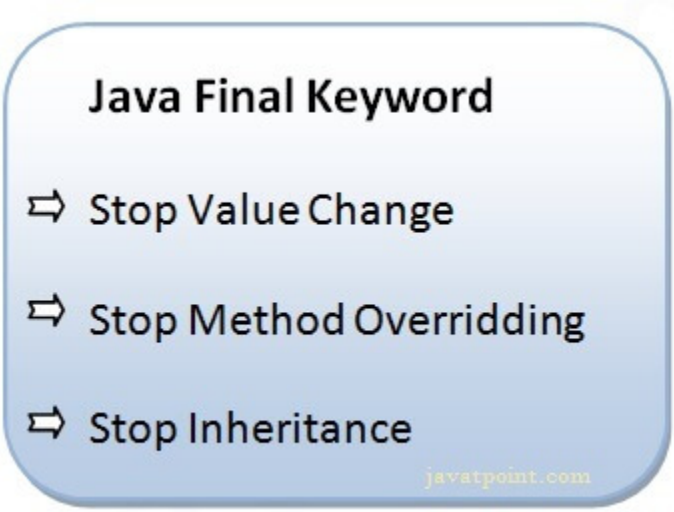
```
{
     public void puexam()
     {
        System.out.println(" pu examination:");
     }
     public void ugexam()
     {
        System.out.println(" UG Examination: ");
     }

}
public class Abstract2
{

       public static void main(String[] args)
       {
          PG12 p=new PG12();
          p.puexam();
          p.ugexam();
          p.result();
       }

}
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

```
package app;

public class Final_Variable
{

   public static void main(String[] args)
   {
         final int i=10;
         System.out.println(i);
         System.out.println(i+10);


         /*
          * final :
          *
          * 1. final is a keyword in java.
          * 2. final keyword is applicable to classes, methods & variables.
          * 3. final variables can be as global (static / instance) &
          * local variable
          *
          * 4. final variables must be declared and initialized with values.
          * if not we will get CTE
          *
          * 5. final variables are not eligible to perform unary operators.
          *
          *
          *
          *
          */
```

```
    }
}
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike
{
  final void run()
{
System.out.println ("running");
}
}

class Honda extends Bike{
   void run()
{
System.out.println ("running safely with 100kmph");
}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
```

```
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

String:

Java String

In Java, string is basically an object that represents sequence of char values.

An array of characters works same as Java string. For example:

String s=new String(ch);

```
package app;

public class String1 {

	public static void main(String[] args)
	{
		/*
		 * 1. String:
		 *     A string is a predefined class in java.
		 *     all string objects are immutable.
		 *     each and every content should be enclosed by " "
		 *     way of creating string objects in java
		 *
		 *     1. syntax:
		 *
		 *     String s1="abc";  constant string pool
		 *
		 *     2. syntax:
		 *
		 *     String s2=new String("abc");  // non constant string pool
		 *
		 *
		 *     String Operations:
		 *     1. concat()
		 *     2. length()
		 *     3. Indexof()
```

```
     *      4.lastIndexof()
     *      5. charAt()
     *      6.toupperCase()
     *      7.tolowerCase()
     *      8. equals()
     *      9. equalsIgnore()
     *
     *
     *
     */


        String s1="abc";  // constant string pool
        String s2="xyz";
        System.out.println(s1);
        System.out.println(s2);


        String a1=new String("christ");   //non constant string pool
        String a2=new String("college");   //non constant string pool

        System.out.println(a1);
        System.out.println(a2);
        System.out.println(a1.concat(a2));
    }

}
```

Java String charAt()

The **java string charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is length of the string. It returns **StringIndexOutOfBoundsException** if given index number is greater than or equal to this string length or a negative number.

```
public class CharAtExample
{
public static void main(String args[])
{
String name="javatpoint";
char ch=name.charAt(4);//returns the char value at the 4th index
System.out.println(ch);
}
}
```

Java String concat

The **java string concat()** method *combines specified string at the end of this string*. It returns combined string. It is like appending another string.

Java String concat() method example

```
public class ConcatExample
{
public static void main(String args[])
{
String s1="java string";
s1.concat("is immutable");
System.out.println(s1);
s1=s1.concat(" is immutable so assign it explicitly");
System.out.println(s1);
}

}
```

Java String equals()

The **java string equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of Object class.

```
public class EqualsExample
{
public static void main(String args[])
{
String s1="javatpoint";
String s2="javatpoint";
String s3="JAVATPOINT";
String s4="python";
System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
}

}
```

Java String indexOf()

The **java string indexOf()** method returns index of given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

```
public class IndexOfExample
{
public static void main(String args[])
{
String s1="this is index of example";
//passing substring
int index1=s1.indexOf("is");//returns the index of is substring
int index2=s1.indexOf("index");//returns the index of index substring
System.out.println(index1+"  "+index2);//2 8

//passing substring with from index
int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
System.out.println(index3);//5 i.e. the index of another is

//passing char value
int index4=s1.indexOf('s');//returns the index of s char value
System.out.println(index4);//3
}

}
```

Java String lastIndexOf()

The **java string lastIndexOf()** method returns last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

```
public class LastIndexOfExample
{
public static void main(String args[])
{
String s1="this is index of example";//there are 2 's' characters in this sentence
int index1=s1.lastIndexOf('s');//returns last index of 's' char value
System.out.println(index1);//6
}
```

```
}
```

Java String length()

The **java string length()** method length of the string. It returns count of total number of characters. The length of java string is same as the unicode code units of the string.

```
public class LengthExample
{
public static void main(String args[])
{
String s1="javatpoint";
String s2="python";
System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string
System.out.println("string length is: "+s2.length());//6 is the length of python string
}

}
```

Java String toLowerCase()

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale.

```
public class StringLowerExample
{
public static void main(String args[])
{
String s1="JAVATPOINT HELLO stRIng";
String s1lower=s1.toLowerCase();
System.out.println(s1lower);
}

}
```

Java String toUpperCase()

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

```
public class StringUpperExample
{
public static void main(String args[])
{
String s1="hello string";
String s1upper=s1.toUpperCase();
System.out.println(s1upper);
}
}
```

Java String trim()

The **java string trim()** method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

```
public class StringTrimExample
{
public static void main(String args[])
{
String s1="  hello string   ";
System.out.println(s1+"javatpoint");//without trim()
System.out.println(s1.trim()+"javatpoint");//with trim()
}

}
```

Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

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

```
{
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

```
package app;

public class SBuffer {

	public static void main(String[] args)
	{
		/* StringBuffer is a predefined class in java
		 * its available in java.lang.*;
		 * all stringBuffer objects are mutable.
		 *
		 * syntax:
		 * StringBuffer sb=new StringBuffer("abc");

		 *
		 */

		StringBuffer sb1=new StringBuffer("abc");
		System.out.println(sb1);

		//append------>concat()

		StringBuffer sb2=new StringBuffer("xyz");
		System.out.println(sb1.append(sb2));
		StringBuffer sb3=new StringBuffer("pqr");

		System.out.println(sb3.reverse());
	}

}
```

Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

- **The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:**

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float

double	Double

```
package app;

public class Wrapper
{

	public static void main(String[] args)
	{
		/*
		 * wrapper class:
		 * jdk1.5
		 * primitive datatypes in java // predefine datatype
		 * total:
		 *
		 * primitive datetype	wrapper class
		 * byte				Byte
		 * short			Short
		 * int				Integer
		 * float			Float
		 * long				Long
		 * double			Double
		 * char				Character/Char
		 * boolean			Boolean
		 *
		 */

		/*
		 * Def: Each & every primitive datatypes have an equivalent
		 * classes those classes are known as wrapper classes.
		 *
		 * wrapper classes are mainly using to perform boxing and unboxing
		 * operations.
		 *
		 * Boxing:
		 *
		 * the process of conversion from primitive type to object type is
		 * known as boxing.
		 *
		 * int i=10;
		 * Integer i1=new Integer(i);
		 *
		 *
		 *
		 * unboxing:
		 * the process of conversion from objective type to primitive type
		 * is known as unboxing.
		 *
```

```
             * int i2=i1.intvalue();
             *
             */


            double d=100d;
            Double d1=new Double(d);
            double d2=d1.doubleValue();
            System.out.println(d2);

      }



}
```

Java Arrays

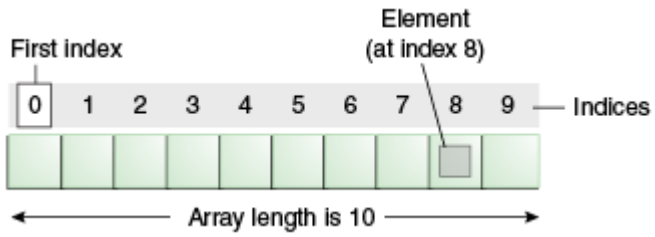
Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the 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 which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array
- dataType[] arr; (or)
- dataType []arr; (or)
- dataType arr[];

Instantiation of an Array in Java

arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
class Testarray
{
public static void main(String args[])
{
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
```

```
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}

}
```

Declaration, Instantiation and Initialization of Java Array

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1
{
public static void main(String args[])
{
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}

}
```

Java Vector

Vector is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-

- Vector is synchronized.
- Java Vector contains many legacy methods that are not the part of a collections framework.

Java Vector class Declaration

1. **public class** Vector<E>
2. **extends** Object<E>
3. **implements** List<E>, Cloneable, Serializable

```
package app;

import java.util.Vector;

public class Vector1 {

	public static void main(String[] args)
	{
		Vector v1=new Vector();
		v1.add("ramesh");
		v1.add(20);
		v1.add(false);
		v1.add(2f);
		System.out.println(v1);
		System.out.println(v1.remove(3));

	}

}
```

Java Vector Example

```
import java.util.*;
public class VectorExample
{
    public static void main(String args[])
{
      //Create a vector
      Vector<String> vec = new Vector<String>();
      //Adding elements using add() method of List
```

```
        vec.add("Tiger");
        vec.add("Lion");
        vec.add("Dog");
        vec.add("Elephant");
        //Adding elements using addElement() method of Vector
        vec.addElement("Rat");
        vec.addElement("Cat");
        vec.addElement("Deer");

        System.out.println("Elements are: "+vec);
    }
}
```