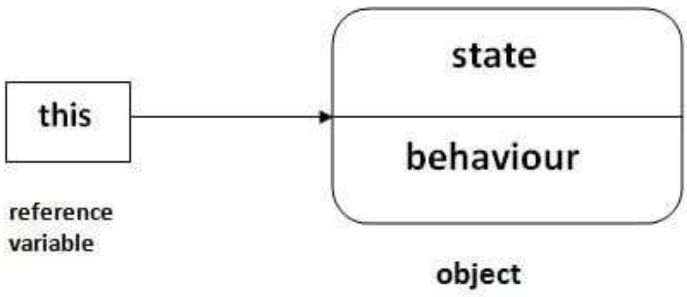
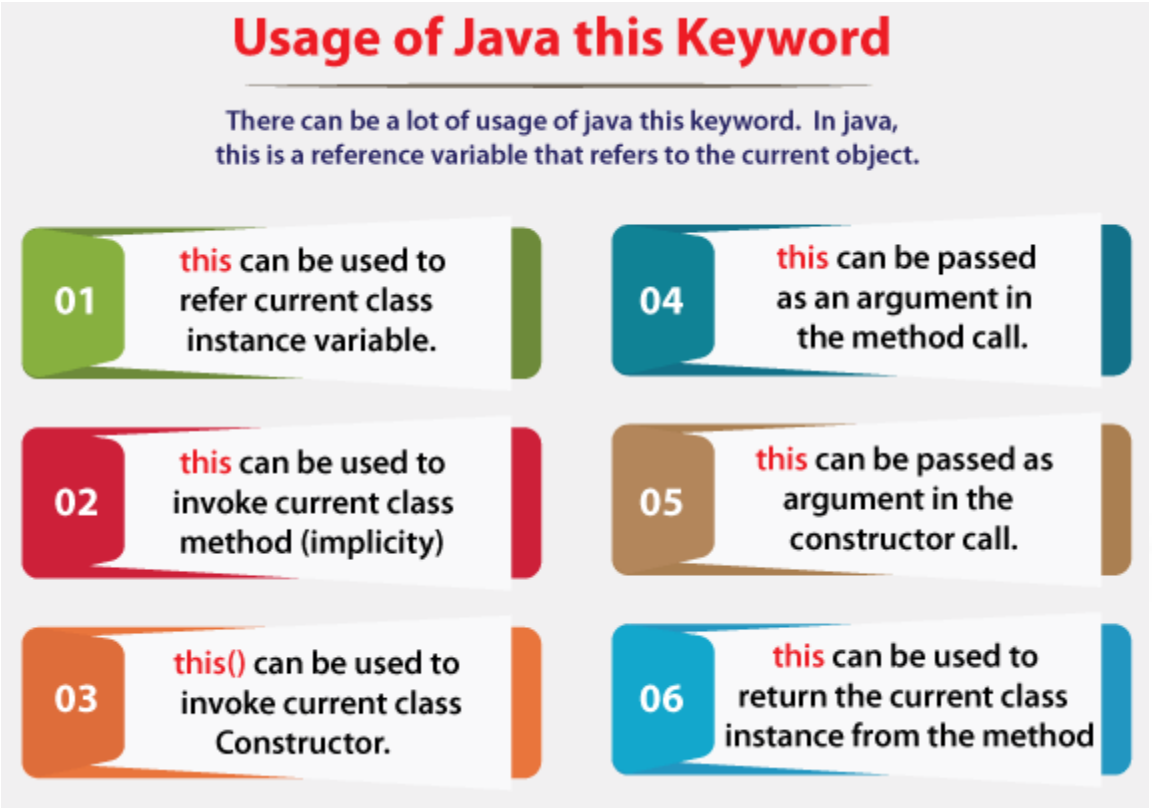
Keywords in JAVA

1. **this:** Java this keyword can be used to refer the current object in a method or constructor.
2. **super:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
3. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
4. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.

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





By Trainer – Rajul Soni

1) this: to refer current class instance variable

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

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. class TestThis1{
13. public static void main(String args[]){
14. Student s1=new Student(111,"ankit",5000f);
15. Student s2=new Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}
```

Output:

```
19.  0 null 0.0
20.  0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. this.rollno=rollno;
```

```
7.  this.name=name;
8.  this.fee=fee;
9.  }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis2{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}
```

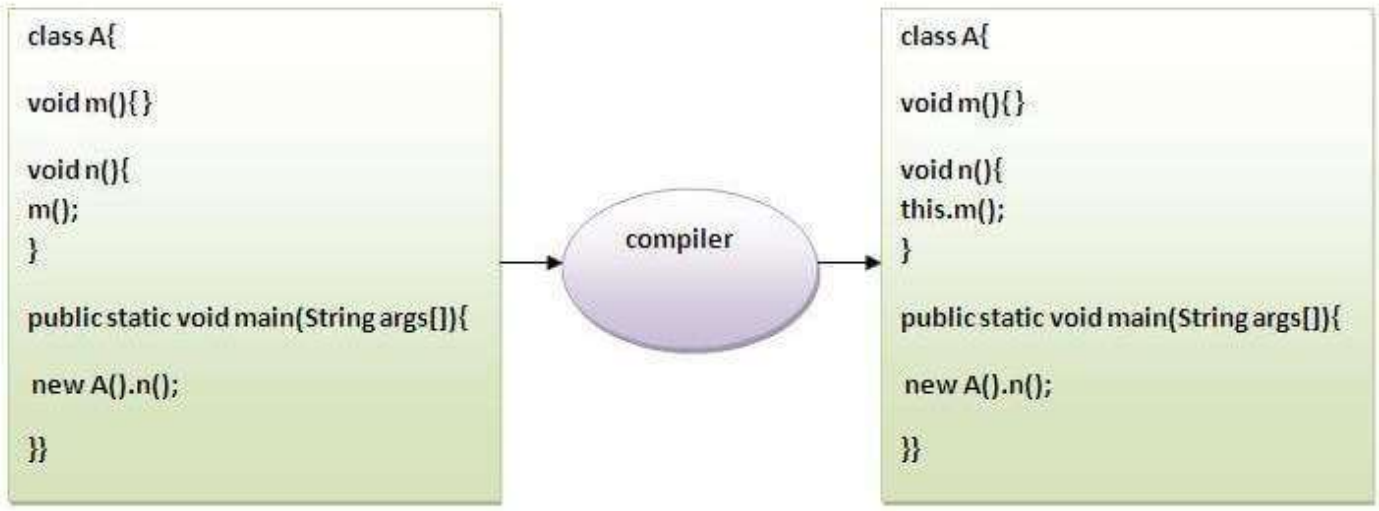
Output:

```
20.  111 ankit 5000.0
21.  112 sumit 6000.0
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

2) this: to invoke current class method

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




```
1.  class A{
2.  void m(){System.out.println("hello m");}
3.  void n(){
4.  System.out.println("hello n");
```

```
5. //m();//same as this.m()
6. this.m();
7. }
8. }
9. class TestThis4{
10. public static void main(String args[]){
11. A a=new A();
12. a.n();
13. }}
```

Output:

```
hello n
hello m
```

3) this() : to invoke current class constructor

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

Calling default constructor from parameterized constructor:

```
1. class A{
2. A(){System.out.println("hello a");}
3. A(int x){
4. this();
5. System.out.println(x);
6. }
7. }
8. class TestThis5{
9. public static void main(String args[]){
10. A a=new A(10);
11. }}
```

Output:

```
hello a
10
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
1. class Student{
2. int rollno;
3. String name,course;
4. float fee;
5. Student(int rollno,String name,String course){
6. this.rollno=rollno;
7. this.name=name;
8. this.course=course;
9. }
10. Student(int rollno,String name,String course,float fee){
11. this(rollno,name,course);//reusing constructor
12. this.fee=fee;
13. }
14. void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. class TestThis7{
17. public static void main(String args[]){
18. Student s1=new Student(111,"ankit","java");
19. Student s2=new Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}
```

Output:

```
23.  111 ankit java 0.0
24.  112 sumit java 6000.0
```

Rule: Call to this() must be the first statement in constructor.

```
1. class Student{
2. int rollno;
3. String name,course;
4. float fee;
5. Student(int rollno,String name,String course){
```

```
6. this.rollno=rollno;
7. this.name=name;
8. this.course=course;
9. }
10. Student(int rollno,String name,String course,float fee){
11. this.fee=fee;
12. this(rollno,name,course);//C.T.Error
13. }
14. void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. class TestThis8{
17. public static void main(String args[]){
18. Student s1=new Student(111,"ankit","java");
19. Student s2=new Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}
```

Output:

```
Compile Time Error: Call to this must be first statement in constructor
```

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
1. class S2{
2.   void m(S2 obj){
3.   System.out.println("method is invoked");
4.   }
5.   void p(){
6.   m(this);
7.   }
8.   public static void main(String args[]){
9.   S2 s1 = new S2();
10.  s1.p();
11.  }
12. }
```

Output:

```
13.  method is invoked
```

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
1.  class B{
2.    A4 obj;
3.    B(A4 obj){
4.      this.obj=obj;
5.    }
6.    void display(){
7.      System.out.println(obj.data);//using data member of A4 class
8.    }
9.  }
10.
11. class A4{
12.   int data=10;
13.   A4(){
14.    B b=new B(this);
15.    b.display();
16.   }
17.   public static void main(String args[]){
18.    A4 a=new A4();
19.   }
20. }
21. Output:10
```

Output – 10

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1.  return_type method_name(){
2.  return this;
3.  }
```

Example of this keyword that you return as a statement from the method

```
1. class A{
2. A getA(){
3. return this;
4. }
5. void msg(){System.out.println("Hello java");}
6. }
7. class Test1{
8. public static void main(String args[]){
9. new A().getA().msg();
10. }
11. }
```

Output:

```
Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
1. class A5{
2. void m(){
3. System.out.println(this);//prints same reference ID
4. }
5. public static void main(String args[]){
6. A5 obj=new A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }
```

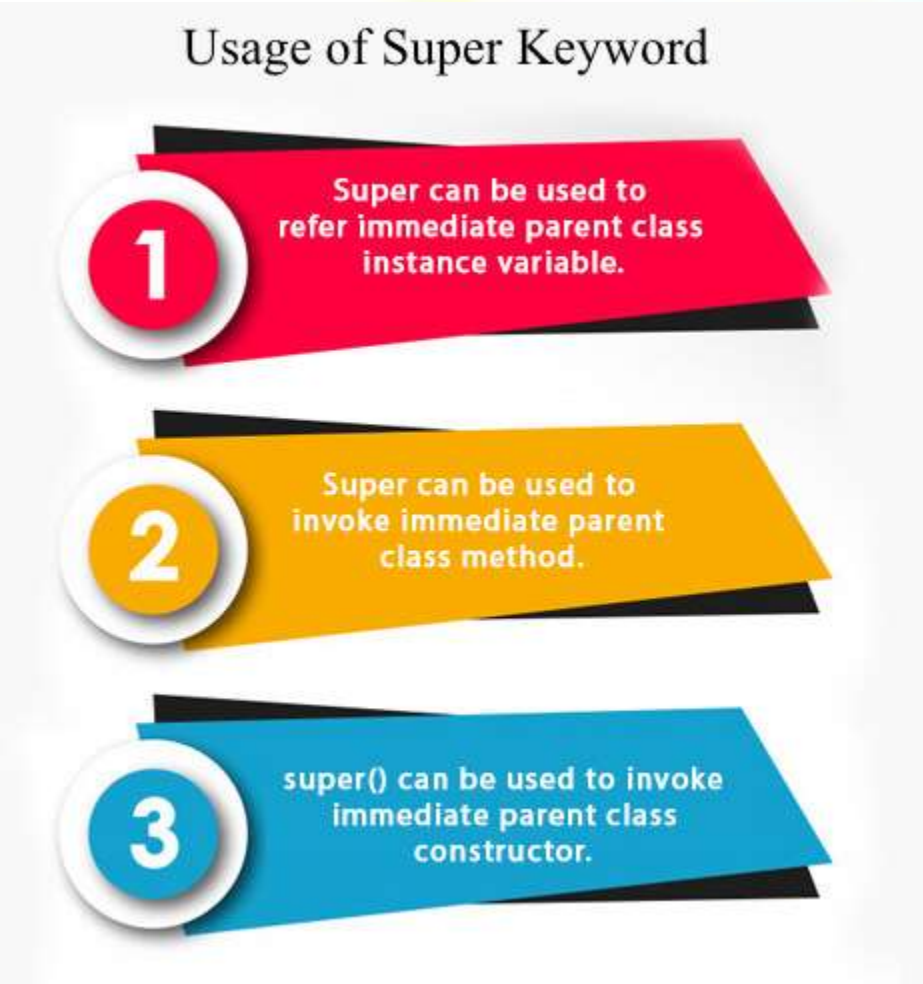
Output – A5@22b3ea59

A5@22b3ea59

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.



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.

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
```

```
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10. }
11. class TestSuper1{
12. public static void main(String args[]){
13. Dog d=new Dog();
14. d.printColor();
15. }}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10. }
11. }
12. class TestSuper2{
13. public static void main(String args[]){
14. Dog d=new Dog();
```

```
15. d.work();
16. }}
```

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

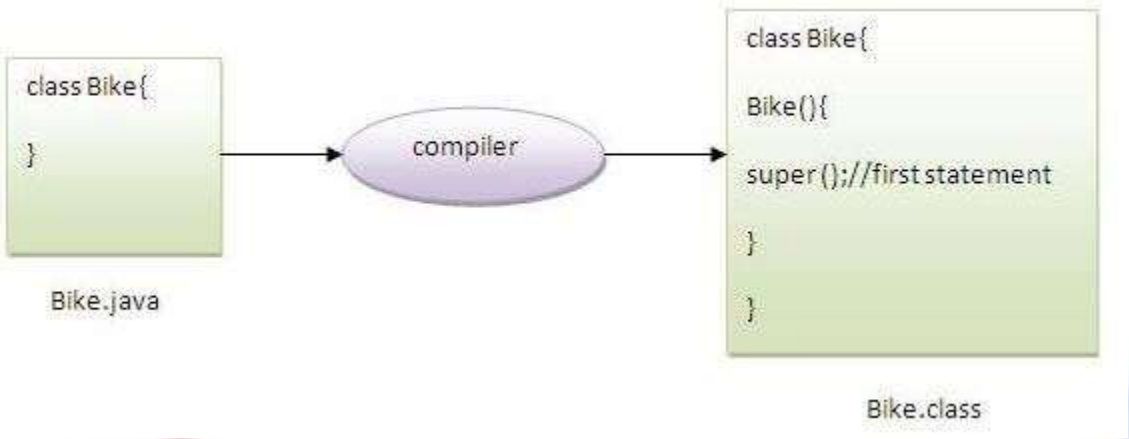
The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. }
9. }
10. class TestSuper3{
11. public static void main(String args[]){
12. Dog d=new Dog();
13. }}
```

Output:

```
animal is created
dog is created
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
1. class Person{
2. int id;
3. String name;
4. Person(int id,String name){
5. this.id=id;
6. this.name=name;
7. }
8. }
9. class Emp extends Person{
10. float salary;
11. Emp(int id,String name,float salary){
12. super(id,name);//reusing parent constructor
13. this.salary=salary;
14. }
15. void display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. class TestSuper5{
```

```
18. public static void main(String[] args){
19. Emp e1=new Emp(1,"ankit",45000f);
20. e1.display();
21. }}
```

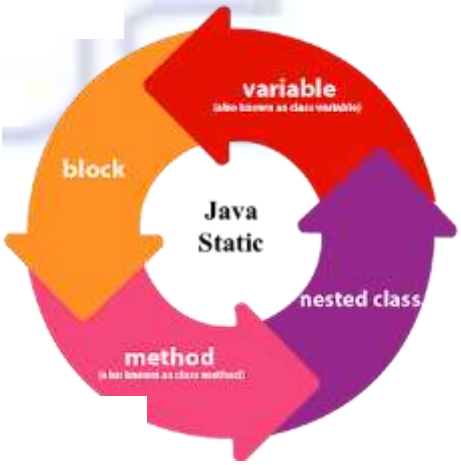
Output:

```
1 ankit 45000
```

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).

Understanding the problem without static variable

```
1. class Student{
2.     int rollno;
3.     String name;
4.     String college="ITS";
5. }
```

By Trainer – Rajul Soni

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

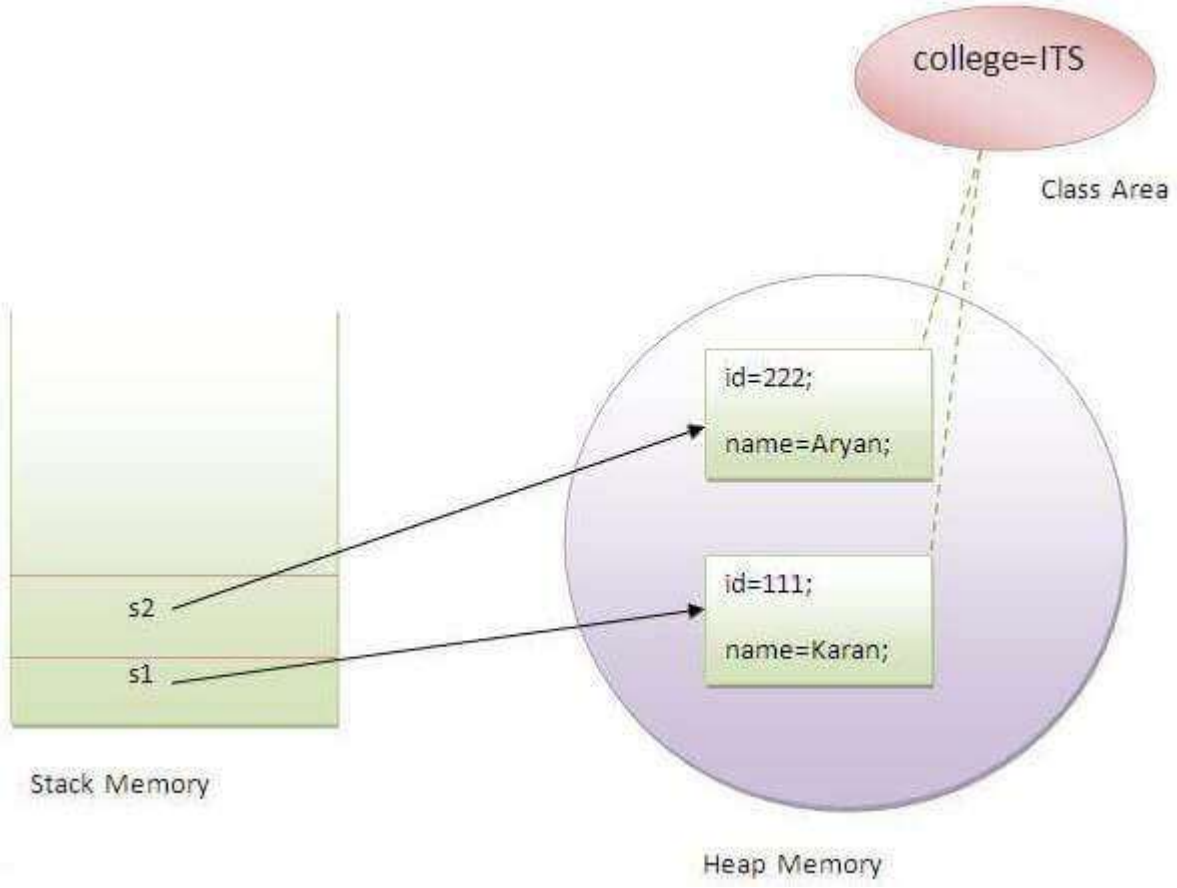
Java static property is shared to all objects.

Example of static variable

```
1. //Java Program to demonstrate the use of static variable
2. class Student{
3.    int rollno;//instance variable
4.    String name;
5.    static String college ="ITS";//static variable
6.    //constructor
7.    Student(int r, String n){
8.    rollno = r;
9.    name = n;
10.   }
11.   //method to display the values
12.   void display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. public class TestStaticVariable1{
16.  public static void main(String args[]){
17.  Student s1 = new Student(111,"Karan");
18.  Student s2 = new Student(222,"Aryan");
19.  //we can change the college of all objects by the single line of code
20.  //Student.college="BBDIT";
21.  s1.display();
22.  s2.display();
23.  }
24. }
```

25. Output:

```
26.  111 Karan ITS
27.  222 Aryan ITS
```



Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
1. class Counter{
2. int count=0;//will get memory each time when the instance is created
3.
4. Counter(){
5. count++;//incrementing value
6. System.out.println(count);
7. }
8.
9. public static void main(String args[]){
10. //Creating objects
11. Counter c1=new Counter();
12. Counter c2=new Counter();
13. Counter c3=new Counter();
14. }
15. }
```

By Trainer – Rajul Soni

16. Output:

```
17.  1
18.  1
19.  1
```

Program of counter by static variable

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

```
1.  class Counter2{
2.  static int count=0;//will get memory only once and retain its value
3.
4.  Counter2(){
5.  count++;//incrementing the value of static variable
6.  System.out.println(count);
7.  }
8.
9.  public static void main(String args[]){
10. //creating objects
11. Counter2 c1=new Counter2();
12. Counter2 c2=new Counter2();
13. Counter2 c3=new Counter2();
14. }
15. }
```

16. Output:

```
17.  1
18.  2
19.  3
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
1. class Student{
2.     int rollno;
3.     String name;
4.     static String college = "ITS";
5.     //static method to change the value of static variable
6.     static void change(){
7.     college = "BBDIT";
8.     }
9.     //constructor to initialize the variable
10.    Student(int r, String n){
11.    rollno = r;
12.    name = n;
13.    }
14.    //method to display values
15.    void display(){System.out.println(rollno+" "+name+" "+college);}
16. }
17. //Test class to create and display the values of object
18. public class TestStaticMethod{
19.    public static void main(String args[]){
20.    Student.change();//calling change method
21.    //creating objects
22.    Student s1 = new Student(111,"Karan");
23.    Student s2 = new Student(222,"Aryan");
24.    Student s3 = new Student(333,"Sonoo");
25.    //calling display method
26.    s1.display();
27.    s2.display();
28.    s3.display();
29.    }
```

```
30. }
```

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

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

Example of static block

```
1. class A2{
2.   static{System.out.println("static block is invoked");}
3.   public static void main(String args[]){
4.    System.out.println("Hello main");
5.   }
6. }
```

Output:static block is invoked

Hello main

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

```
1. class A3{
2.   static{
3.   System.out.println("static block is invoked");
4.   System.exit(0);
```

```
5.    }
6.  }
```

Output:

```
static block is invoked
```

Since JDK 1.7 and above, output would be:

```
Error: Main method not found in class A3, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

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).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1. class Bike9{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
7.  Bike9 obj=new  Bike9();
8.  obj.run();
9.  }
10. }//end of class
```

Output:Compile Time Error

2) Java final method

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

Example of final method

```
1. class Bike{
2.   final void run(){System.out.println("running");}
3. }
4.
5. class Honda extends Bike{
6.    void run(){System.out.println("running safely with 100kmph");}
7.
8.    public static void main(String args[]){
9.    Honda honda= new Honda();
10.   honda.run();
```

By Trainer – Rajul Soni

```
11.    }
12. }
```

Output:Compile Time Error

3) Java final class

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

Example of final class

```
1.  final class Bike{}
2.
3.  class Honda1 extends Bike{
4.    void run(){System.out.println("running safely with 100kmph");}
5.
6.    public static void main(String args[]){
7.    Honda1 honda= new Honda1();
8.    honda.run();
9.    }
10. }
```

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1.  class Bike{
2.    final void run(){System.out.println("running...");}
3.  }
4.  class Honda2 extends Bike{
5.     public static void main(String args[]){
6.      new Honda2().run();
7.     }
8.  }
```

Output:running...

By Trainer – Rajul Soni

Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

   There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7.  public static void main(String args[]){
8.    A obj=new A();
9.    System.out.println(obj.data);//Compile Time Error
10.   obj.msg();//Compile Time Error
11.   }
12. }
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
1. class A{
2. private A(){}//private constructor
3. void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6.  public static void main(String args[]){
7.    A obj=new A();//Compile Time Error
8.  }
```

```
9. }
```

Note: A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1. package pack;
2. class A{
3.   void msg(){System.out.println("Hello");}
4. }
```

```
1. package mypack;
2. import pack.*;
3. class B{
4.   public static void main(String args[]){
5.    A obj = new A();//Compile Time Error
6.    obj.msg();//Compile Time Error
7.   }
8. }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1. package pack;
2. public class A{
3. protected void msg(){System.out.println("Hello");}
4. }
```

```
1. package mypack;
2. import pack.*;
3.
4. class B extends A{
5.   public static void main(String args[]){
6.    B obj = new B();
7.    obj.msg();
8.   }
9. }
```

Output:Hello

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
1. package pack;
2. public class A{
3. public void msg(){System.out.println("Hello");}
4. }
```

```
1. package mypack;
2. import pack.*;
3.
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();
```

```
7.      obj.msg();
8.    }
9.  }
```

Output – Hello

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

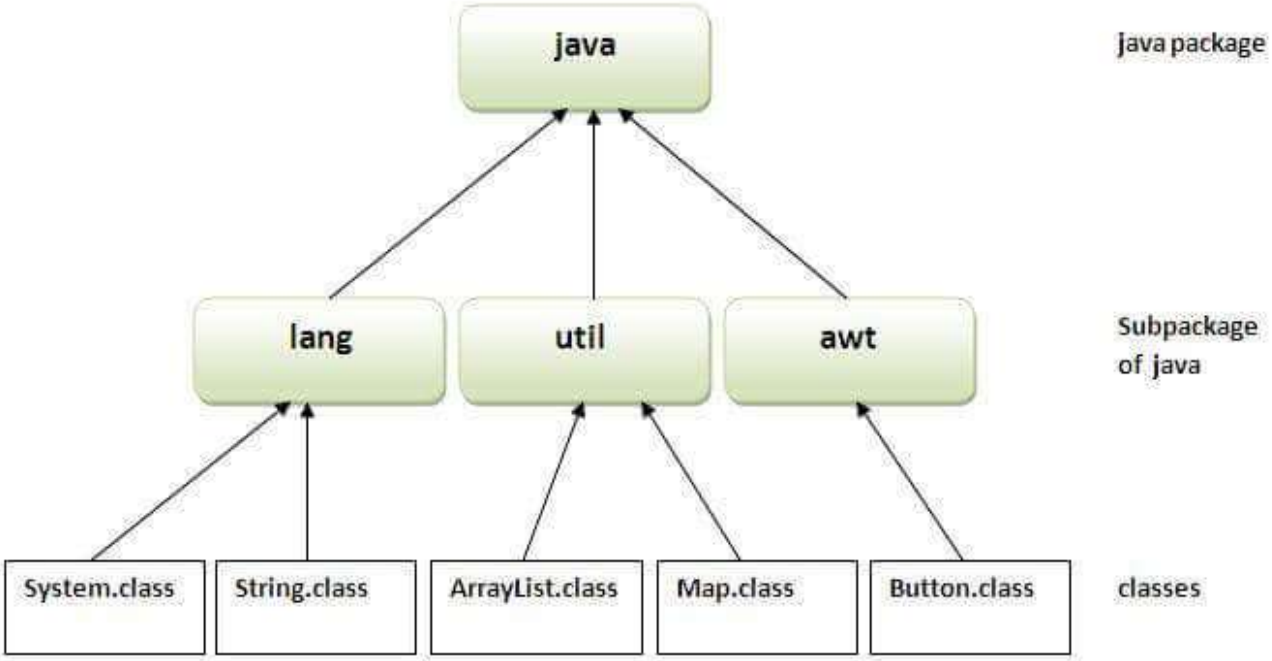
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
1. package mypack;
2. public class Simple{
3.  public static void main(String args[]){
4.     System.out.println("Welcome to package");
5.    }
6. }
```

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
1. package pack;
2. public class A{
3.   public void msg(){System.out.println("Hello");}
4. }
```

```
1. package mypack;
2. import pack.*;
3.
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();
7.    obj.msg();
8.   }
9. }
```

Output – Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1. package pack;
2. public class A{
3.   public void msg(){System.out.println("Hello");}
4. }
```

```
1. package mypack;
2. import pack.A;
3.
```

```
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();
7.    obj.msg();
8.   }
9. }
```

OUTPUT – Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
1. package pack;
2. public class A{
3.   public void msg(){System.out.println("Hello");}
4. }
```

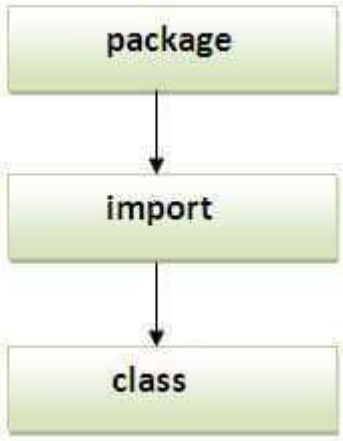
```
1. package mypack;
2. class B{
3.   public static void main(String args[]){
4.    pack.A obj = new pack.A();//using fully qualified name
5.    obj.msg();
6.   }
7. }
```

Output – Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

> *The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.*

Example of Subpackage

```
1. package com.javatpoint.core;
2. class Simple{
3.   public static void main(String args[]){
4.    System.out.println("Hello subpackage");
5.   }
6. }
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

```
Output:Hello subpackage
```