Core Java:

Declarations and Access Control, Object Orientation, Operators, Flow Control, Exceptions, Gradle Fundamentals, TDD with Junit 5, Strings, I/O, Formatting, and Parsing, Generics and Collections, Threads.

Declarations and Variables:

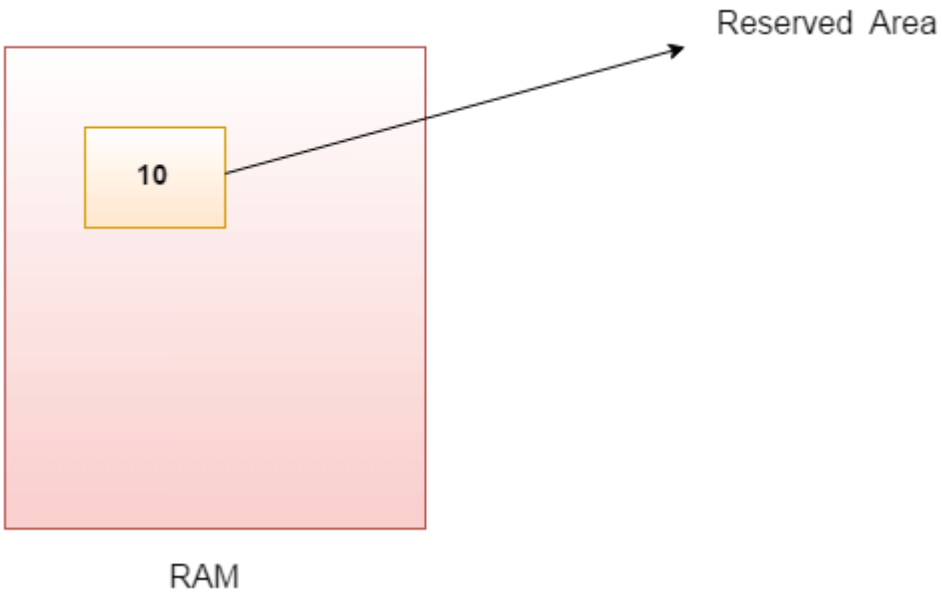
A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

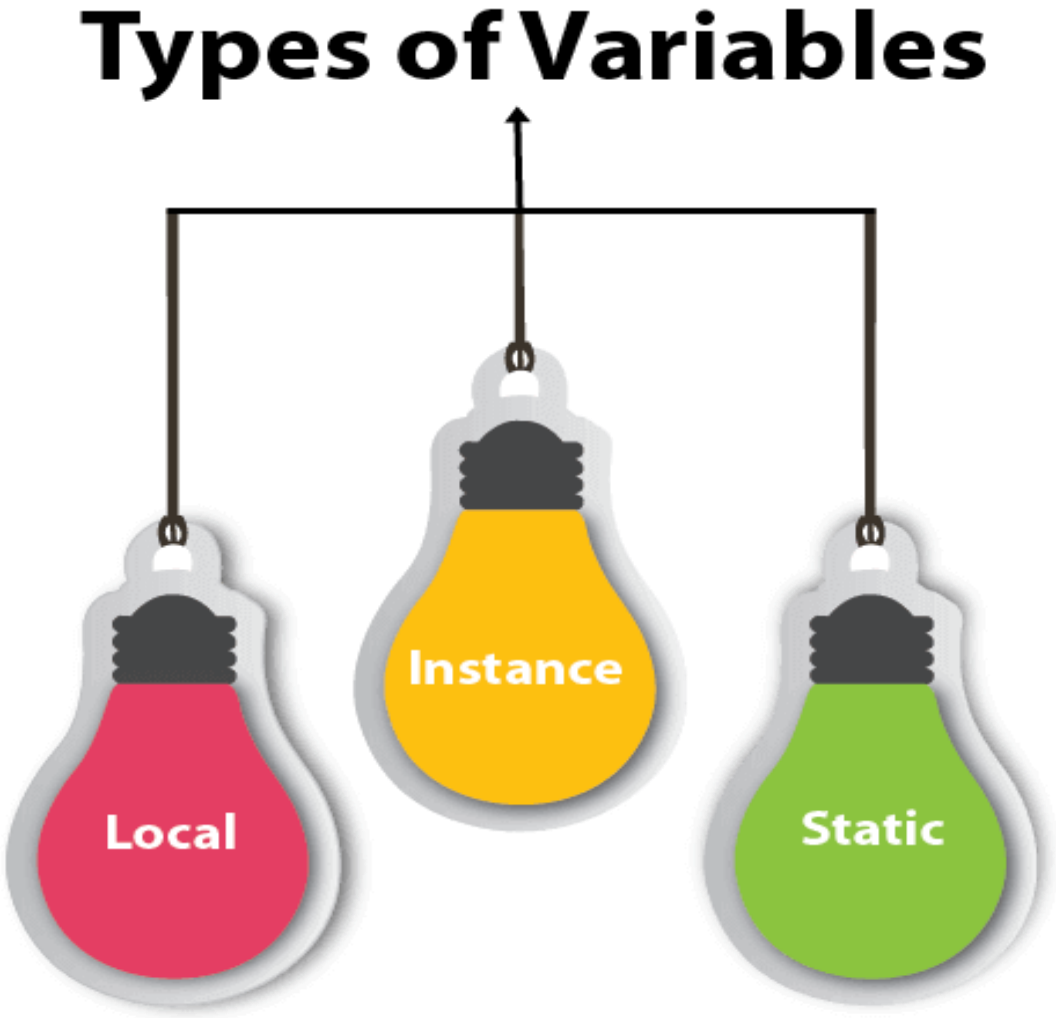


1. `int data=50;//Here data is variable`

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable.
- static variable



1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
1. public class A
2. {
3.    static int m=100;//static variable
4.    void method ()
5.    {
6.       int n=90;//local variable
7.    }
8.    public static void main(String args[])
9.    {
10.      int data=50;//instance variable
11.   }
12. }//end of class
```

Java Variable Example: Add Two Numbers

```
1. public class Simple{
2. public static void main(String[] args){
3. int a=10;
4. int b=10;
5. int c=a+b;
6. System.out.println(c);
7. }
8. }
```

Output:

```
20
```

Java Variable Example: Widening

```
1. public class Simple{
2. public static void main(String[] args){
3. int a=10;
4. float f=a;
5. System.out.println(a);
6. System.out.println(f);
7. }}
```

Output:

```
10
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
1. public class Simple{
2. public static void main(String[] args){
3. float f=10.5f;
4. //int a=f;//Compile time error
5. int a=(int)f;
6. System.out.println(f);
7. System.out.println(a);
8. }}
```

Output:

```
10.5
10
```

Java Variable Example: Overflow

```
1. class Simple{
2. public static void main(String[] args){
3. //Overflow
4. int a=130;
5. byte b=(byte)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}
```

Output:

```
130
-126
```

Java Variable Example: Adding Lower Type

```
1. class Simple{
2. public static void main(String[] args){
3. byte a=10;
4. byte b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. byte c=(byte)(a+b);
7. System.out.println(c);
8. }}
```

Output:

```
20
```

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

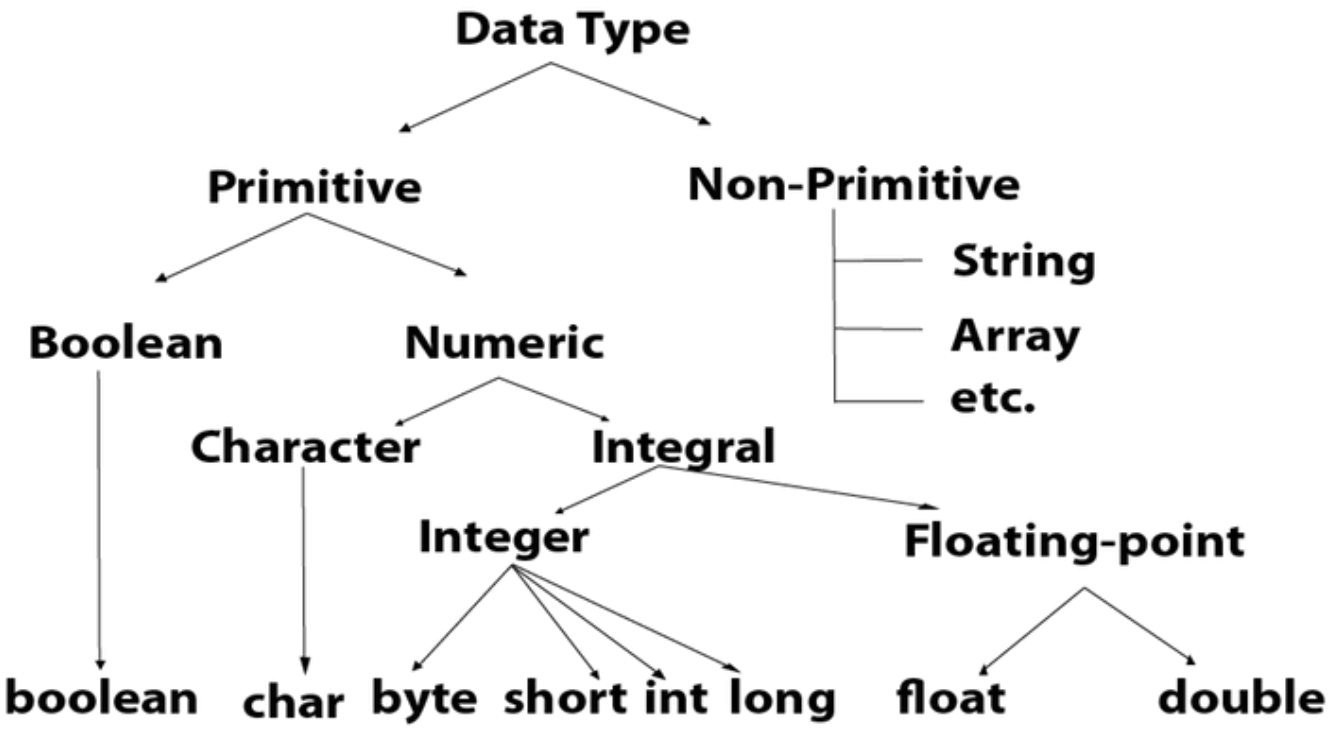
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

1. Boolean one = **false**

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

1. **byte** a = 10, **byte** b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

```
1. short s = 10000, short r = -5000
2.
```

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

```
1. int a = 100000, int b = -200000
```

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is -9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example:

```
1. long a = 100000L, long b = -200000L
```

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

```
1. float f1 = 234.5f
```

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
1. double d1 = 12.3
```

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

```
1. char letterA = 'A'
```

Access Modifiers in Java

1. Private access modifier
2. Role of private constructor
3. Default access modifier
4. Protected access modifier
5. Public access modifier
6. Access Modifier with Method Overriding

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.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

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:

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

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.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

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.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

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

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
```

```
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output:Hello

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
class A{
protected void msg(){System.out.println("Hello java");}
}

public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

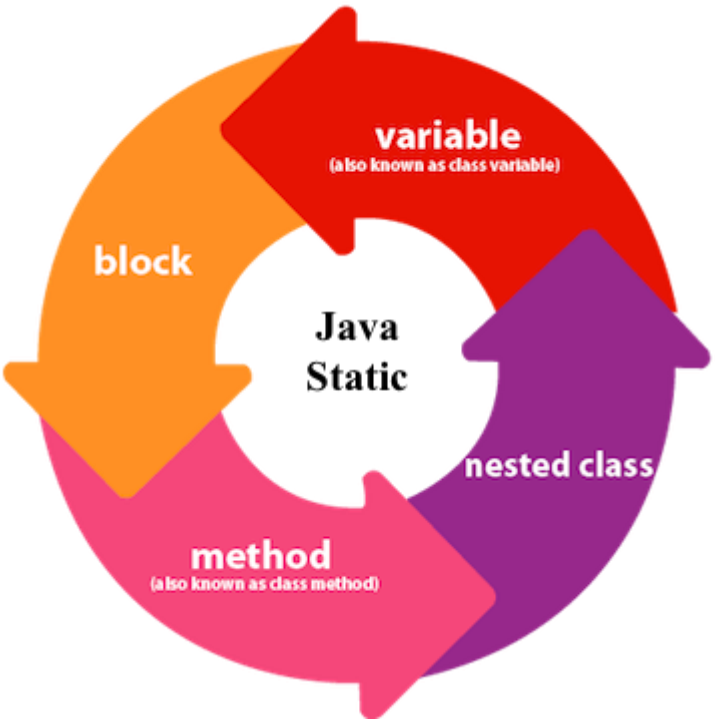
The default modifier is more restrictive than protected. That is why, there is a compile-time error.

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

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

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.

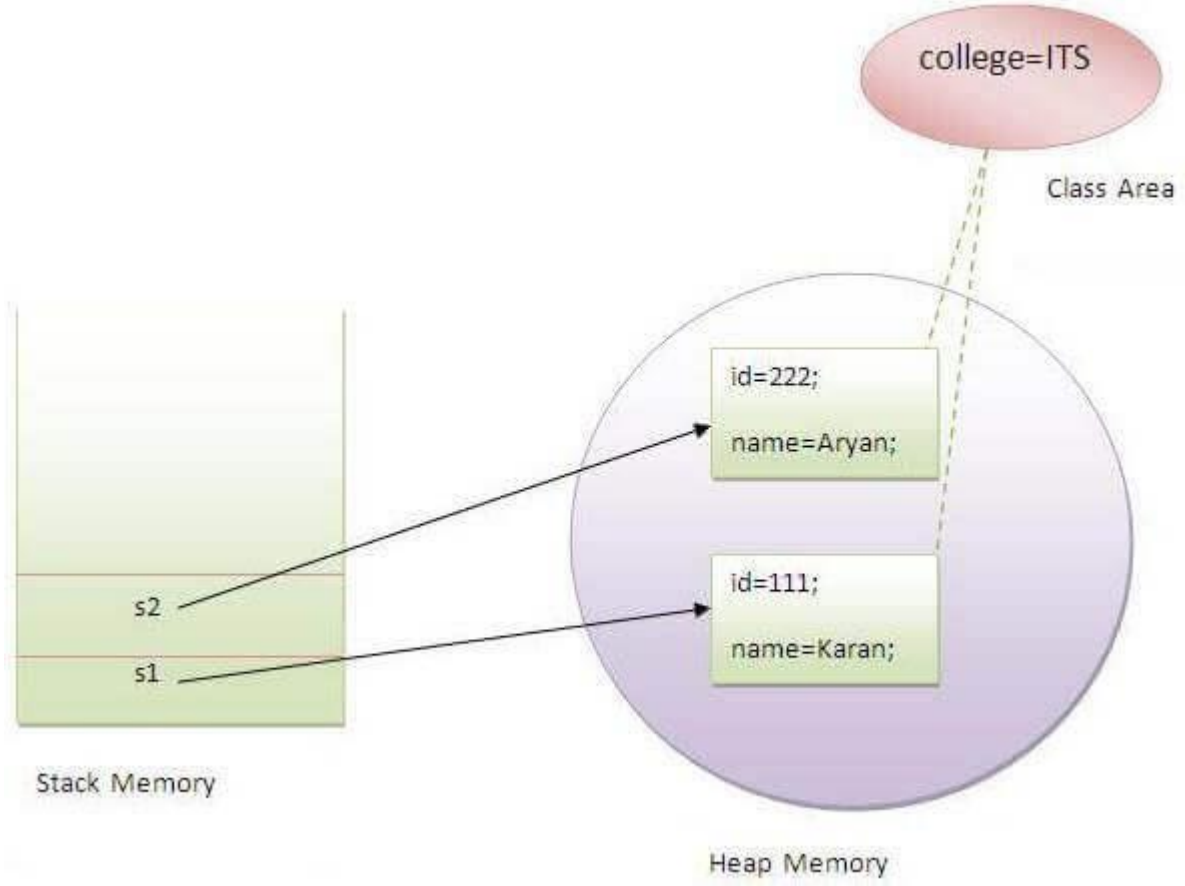
Example of static variable

```
//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();
 }
}
```

Test it Now

Output:

```
111 Karan ITS
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. //Java Program to demonstrate the use of an instance variable
2. //which get memory each time when we create an object of the class.
3. class Counter{
4. int count=0;//will get memory each time when the instance is created
5.
6. Counter(){
7. count++;//incrementing value
8. System.out.println(count);
9. }
10.
11. public static void main(String args[]){
12. //Creating objects
13. Counter c1=new Counter();
14. Counter c2=new Counter();
15. Counter c3=new Counter();
16. }
```

```
17. }
```
Test it Now

Output:

```
1
1
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. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. class Counter2{
4. static int count=0;//will get memory only once and retain its value
5.
6. Counter2(){
7. count++;//incrementing the value of static variable
8. System.out.println(count);
9. }
10.
11. public static void main(String args[]){
12. //creating objects
13. Counter2 c1=new Counter2();
14. Counter2 c2=new Counter2();
15. Counter2 c3=new Counter2();
16. }
17. }
```
Test it Now

Output:

```
1
2
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. //Java Program to demonstrate the use of a static method.
2. class Student{
3.     int rollno;
4.     String name;
5.     static String college = "ITS";
6.     //static method to change the value of static variable
7.     static void change(){
8.     college = "BBDIT";
9.     }
10.    //constructor to initialize the variable
11.    Student(int r, String n){
12.    rollno = r;
13.    name = n;
14.    }
15.    //method to display values
16.    void display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. public class TestStaticMethod{
20.   public static void main(String args[]){
21.   Student.change();//calling change method
22.   //creating objects
23.   Student s1 = new Student(111,"Karan");
24.   Student s2 = new Student(222,"Aryan");
25.   Student s3 = new Student(333,"Sonoo");
26.   //calling display method
27.   s1.display();
28.   s2.display();
29.   s3.display();
30.   }
31. }
```

Test it Now

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

```
1. //Java Program to get the cube of a given number using the static method
2.
3. class Calculate{
4.   static int cube(int x){
```

```
5.    return x*x*x;
6.    }
7.
8.    public static void main(String args[]){
9.    int result=Calculate.cube(5);
10.   System.out.println(result);
11.   }
12. }
```

Test it Now

Output:125

Restrictions for the static method

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

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
1.  class A{
2.   int a=40;//non static
3.
4.   public static void main(String args[]){
5.    System.out.println(a);
6.   }
7.  }
```

Test it Now

Output:Compile Time Error

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.  }
```

Test it Now

```
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.  }
```

Test it Now

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
```