**INDEX**

|  |  |
| --- | --- |
| **Java Topic** | **Page No** |
| **Post Increment Operator** | **2** |
| **Pre Increment Operator** | **3-4** |
| **Global Variables** | **5-6** |
| **SIB, Static Variables** | **7-9** |
| **Non static Members** | **10-14** |
| **Constructor** | **15-20** |
| **Super and This Keyword** | **21-25** |
| **Final Keyword** | **26-33** |
| **For Each Loop** | **34-36** |
| **Class and Instance Variables** | **37-42** |

**Keywords in Java:**

#### **Keywords used for data types: eight(8) primitive types**

boolean byte char short int long float double

**Keywords used for access control are:**

private protected public

These keywords control the visibility of variables and methods inside the class.

**Keywords used for modifiers are**:

abstract final static private protected public

synchronized transient native volatile strictfp

**Keywords used for exception handling are:**

try catch finally throw throws

**Keywords used for loops or decision-makers are:**

break continue do while for

switch case default if else

**Keywords used for class functions are:**

package import class extends implements interface

new return instanceof this super void

**Keywords used for assigned values are:**

true false null

**Outdated keywords are:**

const goto

**Java Modifier Summary**

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Used on** | **Meaning** |
| abstract | class  interface  method | Contains unimplemented methods and cannot be instantiated.  All interfaces are abstract. Optional in declarations  No body, only signature. The enclosing class is abstract |
| final | class  method  field  variable | Cannot be subclassed  Cannot be overridden and dynamically looked up  Cannot change its value. static final fields are compile-time constants.  Cannot change its value. |
| native | method | Platform-dependent. No body, only signature |
| none(package) | class  interface  member | Accessible only in its package  Accessible only in its package  Accessible only in its package |
| private | member | Accessible only in its class(which defines it). |
| protected | member | Accessible only within its package and its subclasses |
| public | class  interface  member | Accessible anywhere  Accessible anywhere  Accessible anywhere its class is. |
| strictfp | class  method | All methods in the class are implicitly strictfp.  All floating-point computation done is strictly conforms to  the IEEE 754 standard. All values including intermediate results must be expressed as IEEE float or double values.  It is rarely used. |
| static | class  method  field   initializer | Make an inner class top-level class  A class method, invoked through the class name.  A class field, invoked through the class name  one instance, regardless of class instances created.  Run when the class is loaded, rather than when an instance is created. |
| synchronized | method | For a static method, a lock for the class is acquired before executing the method. For a non-static method, a lock for the specific object instance is acquired. |
| transient | field | Not be serialized with the object, used with object serializations. |
| volatile | field | Accessible by unsynchronized threads, very rarely used. |

**All Possible Combinations of Features and Modifiers**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Modifier** | **Class** | **Variable** | **Method** | **Constructor** | **Free-Floating Block** |
| public | yes | yes | yes | yes | no |
| protected | no | yes | yes | yes | no |
| none or package or default | yes | yes | yes | yes | yes |
| private | no | yes | yes | yes | no |
| final | yes | yes | yes | no | no |
| abstract | yes | no | yes | no | no |
| static | no | yes | yes | no | yes |
| native | no | no | yes | no | no |
| transient | no | yes | no | no | no |
| volatile | no | yes | no | no | no |
| synchronized | no | no | yes | no | yes |
| strictfp | yes | no | yes | yes | no |

**Declaring a Class:**

[access modifiers] class className

[extends superClass]

[implements interface1, interface2,..]{

//fields

//methods

//member classes

//member interfaces

//constructors

//instance initializers

//static initializers

//inherited members if a superclass exists.

}

**Important Points**:

1)If a class is declared public, then it can be referred to from other packages

2)A class declared without any modifier, especially without any access modifier can only be accessed in its own package

**Declare an abstract class**

If a class is incompletely implemented, it must be declared abstract, such a class cannot be instantiated, but can be extended by subclasses. Under the following conditions, a class must be declared abstract.

* If a class contains an abstract method
* If a class has not implemented an abstract method which declares in any of its superclass
* If a class neither declares nor inherits a method in one of its direct superinterfaces

**In the example:**

abstract class Employee {

String name, address, phone;

int companyID = 888;

int getCompanyID() {

return companyID;

}

abstract double calculateSalary(double rate, double hours);

}

**SalesDepartment.java**

abstract class SalesDepartment extends Employee {

double sales;

}

**FrontDeskEmployee.java**

class FrontDeskEmployee extends Employee {

double calculateSalary(double rate, double hours) {

return xx;

}

}

Explanation of above example:

a class Employee is declared that must be declared abstract, because it contains a declaration of an abstract method named calculateSalary

The subclass of Employee named SalesDepartment inherits the abstract method calculateSalary, but not provide any implementation, so it must also be declared abstract

On the other hand, the subclass of Employee named FrontDeskEmployee provides an implementation of calculateSalary, so it needs not be declared abstract.

**A compile-time error** occurs if an attempt is made to create an instance of an abstract class like:

Employee em = new Employee(); //wrong

or

Employee em = new SalesDepartment();//wrong

would result in a compile-time error because both Employee and SalesDepartment are abstract classes. The correct declaration would be

Employee em = new FrontDeskEmployee()

**For a class implementing one or more interfaces,**

interface A {

void amethod();

}

interface B {

void bmethod();

}

abstract class C implements A, B {

amethod() {}

}

class D implements A, B {

amethod() {}

bmethod() {}

}

**Explanation:**

class C should be declared abstract because it doesn't provide implementation for bmethod of B interface.

class D should not be declared abstract because it provides implementations for amethod and bmethod in both interfaces A and B which class D implemented.

**Important Points:**

An abstract class may have constructors or a main(). You can execute a compiled abstract class if it has a correctly formed main() method.

If you want your class to be subclassed and to complete the implementation in the subclass, you should declare such class abstract.

If you don't want a class to be subclassed, you should declare it final.

If you don't want a class to be instantiated, you may declare the class with a private default constructor. An example is class Math.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**JDK** stands for **Java Development Kit** or sometimes it is also referred as **Java Standard Edition Development Kit**. JDK is a development environment to develop wide range of applications such as desktop applications, web applications or mobile applications using Java programming language.

**JRE**– Java Runtime Environment, is also part of JDK. JRE provides the minimum runtime requirements for executing a java application. It consists of Java Virtual Machine(JVM) executables, core classes and some supporting files.

**main** is a method name. Method name must be in lowercase. **main** method is a special method because execution of a java program starts from **main** method. This method takes one argument of type String array. Remember **main** is not a keyword.

**String** is a final class from *java.lang* package.

**System** is also a final class from *java.lang* package. **out** is a static member of **System** class of type *PrintStream*. **println** is a method of *PrintStream* class.

**Post Increment Operator:**

**package** incrementConcept;

**public** **class** PostIncrement {

**public** **static** **void** main(String[] args) {

**int** i=0;

System.***out***.println(i++);

}

}

If you are thinking that output will be 1 then you are wrong. If you run this program output will be 0 but not 1. Because the operator used is post increment operator.

**According to definition of post increment operator, first, value of the variable is used and then incremented** i.e. first, value of i(0) is printed and then i is incremented to 1. So, here usage value(used value) of i is 0 and storage value(value stored in the memory) is 1.

Another Program:

**package** incrementConcept;

**public** **class** PostIncrement {

**public** **static** **void** main(String[] args) {

**int** i;

**for**(i=0;i<10;i++) {

System.***out***.println(i++);

}

}

}

Output:

0

2

4

6

8

**Pre Increment Operator:**

**package** incrementConcept;

**public** **class** PreIncrement {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**int** i=0;

System.***out***.println(++i);

}

}

Here, output will be 1, The operator used is pre increment operator**. When you use pre increment operator, first, value is incremented and then used**. In the above program, first, value of i is incremented to 1 then it is used. So, usage value and storage value both are same.

**Another Example for Pre increment Operator:**

**package** incrementConcept;

**public** **class** PreIncrement {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**int** i=0;

**for**(i=0;i<10;++i)

{

System.***out***.println(++i);

}

}

}

Output:

1

3

5

7

9

**Post Decrement Operator:**

**package** incrementConcept;

**public** **class** PostDecrement {

**public** **static** **void** main(String[] args) {

**int** i = 0;

System.***out***.println(i--);

}

}

Here, the operator used is post decrement operator. It operates in the same manner as post increment operator but here the value is decremented. The output of this program will be 0 not -1, because first, value is used and then decremented. So, here usage value is 0 and storage value is -1.

Pre increment Operator:

**package** incrementConcept;

**public** **class** PreDecrement {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**int** i = 0;

System.***out***.println(--i);

}

}

Here, the operator used is pre decrement operator. It is also operates in the same manner as pre increment operator but here the value is decremented. If you run this program, output will be -1. That means first, value is decremented and then used. So, usage value is -1 and storage value is also -1.

**Can we use local variables before they are initializes?**

**package** variables;

**public** **class** GlobalAndLocalVariables {

**static** **int** *globalVariable*;

**static** **void** methodOne() {

**int** localToMethodOne=0;

System.***out***.println(*globalVariable*);

System.***out***.println(localToMethodOne);

}

**public** **static** **void** main(String[] args) {

**int** localToMain=0;

System.***out***.println(localToMain);

System.***out***.println(*globalVariable*);

}

}

In the above program, ‘globalVariable’ declared in the Line 3 is a global variable. **Global Variable** has to be declared anywhere in the class body but not inside any method or block. If a variable is declared as global, it can be used anywhere in the class. For example, see the above program, ‘globalVariable’ declared in the Line 3 is used inside the methodOne() (Line 6) and also inside the main() method (Line 12). So, Global variables are available for all methods and blocks of that class.

If the variable is declared inside a method or block, it is called **local variable**. Local variable is available only to method or block in which it is declared. For example, in the above program, ‘localToMethodOne’ is a local variable of methodOne() and it is accessible only in methodOne()and not available outside the methodOne(). If you use outside methodOne(), you will get compile time error. Variable ‘localToMain’ declared is also local variable. It is available only inside main() method.

**Default Values of Global Variables:**

If you don’t initialize global variables, they take default values of declared type. For example, If global variable is int type and it is not initialized explicitly, it will take default value of int type i.e 0. Below is the list of some data types and their default values.

|  |  |
| --- | --- |
| DataType | DefaultValue |
| Int | 0 |
| Boolean | false |
| Byte | 0 |
| Short | 0 |
| Long | 0 |
| Float | 0.0 |
| Double | 0.0 |
| All derived Data types | null |

**Important Note : If the local variables are not initialized explicitly, they don’t take default values. They remain uninitialized until you initialize them explicitly.**

Now, come to our question, **can we use local variables before they are initialized?**. Consider following program.

**package** variables;

**public** **class** LocalVariableBehaviour {

**public** **static** **void** main(String[] args) {

**int** i;

System.***out***.println(i);

i=10;

}

}

If you try to compile above program, you will get a **compile time error :** The local variable i may not have been initialized**.** Because, any variable, global or local, should have some value before they are used. If you don’t initialize global variables explicitly, they take default values. But, If you don’t initialize local variables explicitly, they don’t take default values. They remain uninitialized until you initialize them explicitly. Therefore, local variables will not be having any value until they are initialized explicitly. Therefore, when you use local variables before they are initialized, you get compile time error. That’s why we can’t use local variables before they are initialized.

In The above program, local variable i is used(Line 5) before it is initialized(Line 6).

To make the above program error free, put i=10 before System.out.println(i).

**SIB – Static Initialization Block, Static Variables And Static Methods**

Static variables, Static Initialization Block and Static Methods – these all are static components or static members of a class. These static members are stored inside the Class Memory. To access static members, you need not to create objects. Directly you can access them with class name.

Static Initialization Block is used to initialize only static variables. It is a block without a name. It contains set of statements enclosed within { }. The syntax of SIB looks like this,

1. static
2. {
3. //Set Of Statements
4. }

Consider the following program.

1. class StaticComponents
2. {
3. static **int** staticVariable;
4. static
5. {
6. System.out.println("StaticComponents SIB");
7. staticVariable = 10;
8. }
9. static void **staticMethod**()
10. {
11. System.out.println("From StaticMethod");
12. System.***out***.println("Printing staticVariable from Static Method: " + *staticVariable*);
13. }
14. }
15. public class MainClass
16. {
17. static
18. {
19. System.out.println("MainClass SIB");
20. }
21. public static void **main**(**String**[] args)
22. {
23. //Static Members directly accessed with Class Name
24. StaticComponents.staticVariable = 20;
25. StaticComponents.staticMethod();
26. }
27. }

Output:

Main class static initialization

static components SIB

From Static Method

Printing staticVariable from Static Method: 20

Let us discuss execution of above program step by step.

**Step 1:**

When you trigger >java MainClass, java command divides allocated memory into two parts – Stack and Heap. First, java command enters stack memory for execution. First, it checks whether **MainClass** is loaded into heap memory or not. If it is not loaded, loading operation of MainClass starts. Randomly some memory space is allocated to MainClass. It is called **Class memory**. All static members are loaded into this class memory. There is only one satic member in MainClass – main() method. It is loaded into class memory of MainClass.

**Step 2:**

After loading all static members, SIB – Static initialization Blocks are executed. Remember, **SIBs are not stored in the heap memory. They just come to stack, execute their tasks and leaves the memory**. So, after loading main() method, SIB of MainClass enters stack for execution. There is only one statement (Line 22) in SIB. it is executed. It prints “MainClass SIB” on console. After executing this statement, SIB leaves the stack memory.

**Step 3:**

Now, java command calls main() method for execution. main() method enters the stack. First statement (Line 28) is executed first. First, It checks whether class StaticComponents is loaded into memory. If it is not loaded, loading operation of StaticComponents takes place. Randomly, some memory is allocated to Class StaticComponents, then all static members of StaticComponents – ‘staticVariable’ and ‘staticMethod()’ are loaded into that class memory. ‘staticVariable’ is a global variable. So, first it is initialized with default value i.e 0.

**Step 4 :**

After loading all static members of StaticComponents, SIB blocks are executed. So, SIB of class StaticComponents enters the stack for execution. First Statement (Line 7) is executed. It prints “StaticComponents SIB” on the console. In the second statement, value 10 is assigned to ‘staticVariable’. There are no other statements left for execution, so it leaves stack memory.

**Step 5 :**

Now control comes back to main() method. The remaining part of first statement i.e value 20 is assigned to ‘staticVariable’ of class StaticComponents, is executed. In the second statement (Line 29), it calls staticMethod() of class StaticComponents for execution.

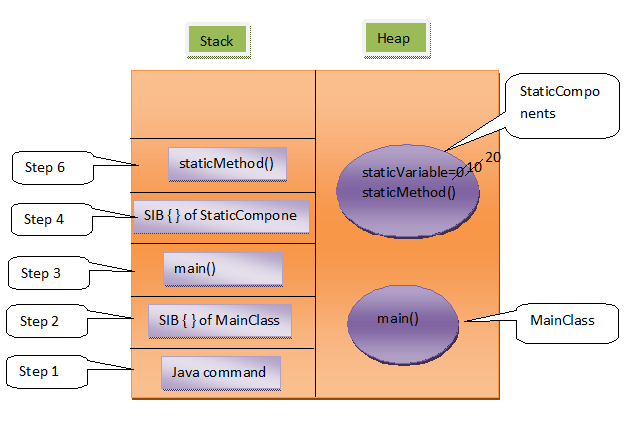
**Step 6:**

staticMethod() of StaticComponents enters stack for execution.  First statement (Line 13) is executed first. It prints “From staticMethod” on the console. In the second statement (Line 14), it prints the value of staticVariable i.e 20 on the console. There are no statements left. so, it leaves the stack.

**Step 7:**

Again, control comes back to main() method. There are no other statements left in main() method. so, it also leaves stack. java command also leaves the stack.

Diagramatic representation of memory allocation of above program looks like this.



**Non Static Members and their Memory Management in Java:**

**Key Points:**

Class : Class is the model/template/blueprint for the objects to be created of its type.

Object : It is an instance of a class. It is the real-time copy of class.

If you don’t understand with the definitions, read out this example. A class is like a blue print of a house. With this blueprint, you can build any number of houses. Each house build with this blueprint is an object or an instance of that blue print.

Non-Static variables and Non-Static methods are non-static components of a class. These are also called instance components of a class. Non-static components are stored inside the object memory. Each object will have their own copy of non-static components. But,  static components are common to all objects of that class.

**Facts about Non Static Members:**

* 1. **you can’t refer a non-static members through a class name. Because, non-static members are stored inside the object memory. You have to refer them through objects only.**
  2. **directly** **you can’t use non-static member inside a static method. Because, non-static members are stored inside the object memory. You have to create objects to use them. You have to refer them through objects only.**
  3. **All static members are loaded into the object memory.**
  4. **You can refer a static member of a class through object of that class like in Line 32. Whenever you refer a static member through a object, compiller replaces object name with its class name like a1.staticVariable is treated as A.staticVariable by the compiler.**
  5. **changes made to static components through one object is reflected in another object also. Because, the same copy of static components is available to all the objects of that class.**

1. class A
2. {
3. **int** nonStaticVariable;
4. static **int** staticVariable;
5. static void **staticMethod**()
6. {
7. System.out.println(staticVariable);
8. // System.out.println(nonStaticVariable);
9. }
10. void **nonStaticMethod**()
11. {
12. System.out.println(staticVariable);
13. System.out.println(nonStaticVariable);
14. }
15. }
16. class MainClass
17. {
18. public static void **main**(**String**[] args)
19. {
20. A.staticVariable = 10;
21. // A.nonStaticVariable = 10;
22. A.staticMethod();
23. // A.nonStaticMethod();
24. A a1 = **new** **A**();
25. A a2 = **new** **A**();
26. System.out.println(a1.nonStaticVariable);
27. System.out.println(a1.staticVariable);
28. a1.nonStaticMethod();
29. a1.staticMethod();
30. System.out.println(a2.staticVariable);
31. a1.staticVariable = 20;
32. System.out.println(a2.staticVariable);
33. }
34. }

**Output :**

10  
0  
10  
10  
0  
10  
10  
20

Let’s discuss memory allocation of above example step by step.

**Step 1 :**

When you trigger >java MainClass, java command divides allocated memory into two parts – stack and heap. First java command enters stack for execution. First it loads class **MainClass**into heap memory. Randomly some memory is allocated to MainClass. All static members are loaded into this class memory. There is only one static member in MainClass i.e main() method. It is loaded into class memory. After loading static members, SIBs are executed. But there is no SIBs in MainClass. So, directly java command calls main() method for execution.

**Step 2 :**

main() method enters stack for execution. First statement (Line 23) refers to class A. First it checks whether class A is loaded into heap memory or not. If it is not loaded, it loads class A into heap memory. Randomly some memory is allocated to class A. All static members of class A , ‘staticVariable’ and ‘staticMethod()’ , are loaded into this memory. ‘staticVariable’ is first initialized with default value 0. No SIBs in Class A. So, after loading static members, main() method assigns value 10 to ‘staticVariable’ of class A.

Second statement (Line 24) of main() method is commented. **Because, you can’t refer a non-static members through a class name. Because, non-static members are stored inside the object memory. You have to refer them through objects only.**

**Step 3 :**

In Line 25, it calls staticMethod() of class A. staticMethod() comes to stack for execution. First statement(Line 8) prints value of ‘staticVariable’ i. e 10 on the console.

Second statement(Line 9) is commented. Because, **directly** **you can’t use non-static member inside a static method. Because, non-static members are stored inside the object memory. You have to create objects to use them. You have to refer them through objects only.**

No statements left in staticMethod(). So, it leaves the stack memory.

**Step 4 :**

Control comes back to main() method. The next statement (Line 26) is also commented. **Because, You can’t refer non-static member through a class name.** In the next statement (Line 28), an object of class A type is created. Randomly, some memory is allocated to object. **All non-static members, ‘nonStaticVariable’ and ‘nonStaticMethod()’,  of class A are loaded into this object memory.** ‘nonStaticVariable’ is a global variable, so it is first initialized with default value 0. A reference variable of type class A  **‘a1’** is created in main() method. It points to this newly created object.

In the same manner, object ‘a2’ is also created (Line 29). In the next statement (Line 31), value of ‘nonStaticVariable’ of ‘a1’ i.e 0 is printed. In the next statement (Line 32), value of ‘staticVariable’ of class A i.e 10 is printed.

**You can refer a static member of a class through object of that class like in Line 32. Whenever you refer a static member through a object, compiller replaces object name with its class name like a1.staticVariable is treated as A.staticVariable by the compiler.**

In the next statement (Line 33), it calls ‘nonStaticMethod()’ of a1.

**Step 5 :**

‘nonStaticMethod()’ of a1 comes to the stack for execution. First statement (Line 14) prints value of  ‘staticVariable’ of class A i.e 10 on the console. Second statement (Line 15) prints the value of ‘nonStaticVariable’ of a1 i.e 0. There are no other statements left in ‘nonStaticMethod()’ , so it leaves the stack.

**Step 6 :**

Control comes back to Line 34 of main() method. It calls staticMethod() of class A. ‘staticMethod()’ enters the stack for execution. First statment (Line 8) prints value of  ‘staticVariable’  i.e 10 on the console. It leaves the memory after executing this statement.

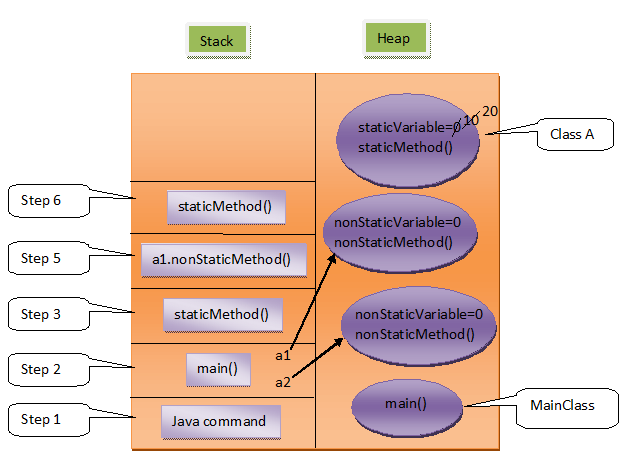
**Step 7 :**

Control comes back to the main() method. Line 36 prints value of ‘staticVariable’ i.e 10 on the console through object a2. In the next statement it changes value of ‘staticVariable’ to 20 through a1. In the next statement, again it prints the value of ‘staticVariable’ through a2. This time 20 is printed on the console.

**This means changes made to static components through one object is reflected in another object also. Because, the same copy of static components is available to all the objects of that class.**

As all statements are executed, first main() method then java command leaves the stack memory.

Diagramatic representation of memory allocation of above program looks like this,



**Constructors in Java**

**Rules of a Constructor:**

1) **Name of the constructor must be same as that of class name**. if you another name, it will give compile time error. If you give another name, it is neither a method because of no return type, nor constructor because name is different from class name.

2) **Constructors must not have a return type**. If you keep return type for the constructor, it will be treated as another method.But compiler gives a warning saying that this method has a constructor name. That means, it is legal to have method name same as constructor name or same as class name but it is not recommended.

3) **Every class should have at least one constructor**. If you don’t write constructor for your class, compiler will give default constructor. **Default constructor is always public and it has no arguments (No-Arg Constructor).**

4) **Constructor can be declared as private.** If you declare constructor as private, you can’t use it outside that class.

5) **One class can have more than one constructors. It is called Constructor Overloading.**Through constructor overloading, you can have multiple ways to create objects.

6) **Duplicate Constructors not allowed.** If you keep duplicate constructors, you will get compile time error.

7) **Multiple arguments of the constructors can’t have same name.** If the two arguments have the same name, you will get compile time error

**8) Only public, protected and private keywords are allowed before a constructor name.** If you keep any other keyword before a constructor name, it gives compile time error.

9**) First statement in a constructor must be either super() or this().** If you put any other statements you will get compile time error.If you don’t include these statements, by default compiler will keep super() calling statement. **super() – It is a calling statement to default constructor of super class. this()- it is a calling statement to constructor of the same class.**

10) Recursive constructor calling is not allowed.

11) No Cylic calling of constructors.

1)**Name of the constructor must be same as that of class name**. if you another name, it will give compile time error. If you give another name, it is neither a method because of no return type, nor constructor because name is different from class name.

1. class A
2. {
3. **A**()
4. {
5. // Constructor of Class A
6. }
7. **A1**()
8. {
9. // Compile time error, It is neither a constructor nor a method
10. }
11. }

2) **Constructors must not have a return type**. If you keep return type for the constructor, it will be treated as another method.But compiler gives a warning saying that this method has a constructor name. That means, it is legal to have method name same as constructor name or same as class name but it is not recommended.

1. class A
2. {
3. **A**()
4. {
5. // Constructor of Class A, not having any return type.
6. }
7. void **A**()
8. {
9. // constructor having a return type, It will be treated as method but with a warning.
10. }
11. }

3) Every class should have at least one constructor. If you don’t write constructor for your class, compiler will give default constructor. Default constructor is always public and it has no arguments (No-Arg Constructor).

1. class A
2. {
3. // No Constructors written
4. }

Compiler will treat the above code as,

1. class A
2. {
3. public **A**()
4. {
5. // Constructor provided by the compiler.
6. }

4) Constructor can be declared as private. If you declare constructor as private, you can’t use it outside that class.

1. class A
2. {
3. private **A**()
4. {
5. // Private Constructor
6. }
7. void **methodOne**()
8. {
9. //You can use private constructor inside the class
10. A a1 = **new** **A**();
11. }
12. }
13. class MainClass
14. {
15. public static void **main**(**String**[] args)
16. {
17. //You can't use private constructor ouside the class like this
18. // A a1 = new A();
19. }
20. }

5) One class can have more than one constructors. It is called **Constructor Overloading.**Through constructor overloading, you can have multiple ways to create objects.

1. class A
2. {
3. **A**()
4. {
5. // First Constructor
6. }
7. **A**(**int** i)
8. {
9. // Second Constructor
10. }
11. **A**(**int** i, **int** j)
12. {
13. // Third Constructor
14. }
15. }

you can create the objects to the above class in three ways like below,

1. class MainClass
2. {
3. public static void **main**(**String**[] args)
4. {
5. A a1 = **new** **A**(); //Using First Constructor
6. A a2 = **new** **A**(10); // Using Second Constructor
7. A a3 = **new** **A**(10, 20); // Using Third Constructor
8. }
9. }

6)Duplicate Constructors not allowed. If you keep duplicate constructors, you will get compile time error.

1. class A
2. {
3. **A**(**int** i)
4. {
5. // Duplicate Constructor
6. }
7. **A**(**int** i)
8. {
9. // Duplicate Constructor
10. }

7) Multiple arguments of the constructors can’t have same name. If the two arguments have the same name, you will get compile time error.

1. class A
2. {
3. **A**(**int** i, **int** i)
4. {
5. // Duplicate Arguments Passed. It gives compile time error
6. }
7. }

8) Only public, protected and private keywords are allowed before a constructor name. If you keep any other keyword before a constructor name, it gives compile time error.

1. class A
2. {
3. **final** **A**()
4. {
5. **//Constructor can not be final**
6. }
7. static **A**()
8. {
9. **//Constructor can not be static**
10. }
11. abstract **A**()
12. {
13. **//Constructors can not be abstract**
14. }
15. }

9) First statement in a constructor must be either super() or this(). If you put any other statements you will get compile time error.If you don’t include these statements, by default compiler will keep super() calling statement. **super() – It is a calling statement to default constructor of super class. this()- it is a calling statement to constructor of the same class.**

**package** constructors;

**public** **class** ConstructorWithSuperThisKeyWord {

ConstructorWithSuperThisKeyWord(){

//By Default, Compile will keep super() calling statement here.

System.***out***.println(" First Constructor");

}

ConstructorWithSuperThisKeyWord(**int** i){

//Compiler will not keep any statement here

**super**();

System.***out***.println(" Second Constructor");

}

ConstructorWithSuperThisKeyWord(**int** i, **int** j){

//Compiler will not keep any statement here

**this**();

System.***out***.println(" Third Constructor");

}

ConstructorWithSuperThisKeyWord(**int** i, **int** j, **int** k)

{

**this**();

System.***out***.println("Fourth Constructor");

// super(); It will give error if you keep super() here

}

**public** **static** **void** main(String[] args) {

//ConstructorWithSuperThisKeyWord A1=new ConstructorWithSuperThisKeyWord(); --- output : first constructor

//ConstructorWithSuperThisKeyWord A2=new ConstructorWithSuperThisKeyWord(10); --- output : second constructor

//ConstructorWithSuperThisKeyWord A3=new ConstructorWithSuperThisKeyWord(10,20); --- output : first constructor , third constructor

ConstructorWithSuperThisKeyWord A4=**new** ConstructorWithSuperThisKeyWord(10,20,30); //--- output : first constructor, fourth constructor

}

}

10) Recursive constructor calling is not allowed.

1. class A
2. {
3. **A**()
4. {
5. **this**();
6. // It gives compile time error
7. }
8. }

11) No Cylic calling of constructors.

1. class A
2. {
3. **A**()
4. {
5. **this**(10);
6. // It gives compile time error
7. }
8. **A**(**int** i)
9. {
10. **this**();
11. // It gives compile time error
12. }
13. }

**Super Keyword:**

super keyword is used to access super class members inside the sub class. Using super keyword, we can access super class methods, super class fields and super class constructors in the sub classes.

For example, following program calls super class constructor, field and method from sub class.

**SuperClass.java**

**package** superThisConcept;

**public** **class** SuperClass {

**int** i=10;

**public** SuperClass(**int** j) {

System.***out***.println("Super class constructor");

}

**void** methodOfSuperClass() {

System.***out***.println("From Method of Super class constructor");

}

}

**SubClass.Java**

**package** superThisConcept;

**public** **class** SubClass **extends** SuperClass{

**public** SubClass() {

**super**(10);

// **TODO** Auto-generated constructor stub

}

**void** methodOfSubClass() {

System.***out***.println(**super**.i);

**super**.methodOfSuperClass();

System.***out***.println("From Method of Sub class");

}

**public** **static** **void** main(String[] args) {

SubClass sc=**new** SubClass();

sc.methodOfSubClass();

}

}

**Output:**

Super class constructor

10

From Method of Super class constructor

From Method of Sub class

super class constructor is called by **super()** calling statement.You can’t use super() calling statement outside the constructor. By default, super() calling statement is the first statement in any constructor.

**When to Use super keyword?.**

1. If you want same implementation as that of super class method in the sub class, but want to add some more extra statements to it, in such cases, super keyword will be very useful. class SuperClass
2. {
3. void **methodOfSuperClass**()
4. {
5. //Some task
6. }
7. }
8. class SubClass extends SuperClass
9. {
10. void **methodOfSubClass**()
11. {
12. **super**.methodOfSuperClass(); // super class method is called
13. //add some other extra statements fulfilling the requirements
14. }
15. //you can implement same task by overriding super class method also
16. void **methodOfSuperClass**()
17. {
18. //super class method is overrided.
19. **super**.methodOfSuperClass();
20. //add some other extra statements fulfilling the requirements
21. }
22. }

**This Keyword:**

this keyword is used to access other members of the same class. Using this keyword, you can access methods, fields and constructors of the same class within the class. this refers to current instance of the class.

**package** superThisConcept;

**public** **class** ThisClass {

**int** i;

**public** ThisClass() {

System.***out***.println("Thisclass constructor ");

}

ThisClass(**int** j){

**this**(); //calling statement to thisclass constructor

System.***out***.println("This class second constructor");

}

**void** methodOne()

{

System.***out***.println("From method one");

}

**void** methodTwo()

{

System.***out***.println(**this**.i); //Accessing same class field

**this**.methodOne(); //Accessing same class method

}

**public** **static** **void** main(String[] args) {

ThisClass tc=**new** ThisClass(10);

tc.methodOne();

tc.methodTwo();

}

}

this() is the calling statement to same class constructor. It must be used within constructor only. If it is used, it must be the first statement in the constructor.

**Important Notes:**

* **You can’t use super and this keywords in a static method and in a static initialization block even though you are referring static members.**
* class SuperClassOne
* {
* **int** i; //Non-Static member
* static void **methodOne**()
* {
* //static method
* System.out.println("From Super Class");
* }
* }
* class SubClassOne extends SuperClassOne
* {
* static
* {
* System.out.println(**super**.i);
* **this**.methodTwo();
* //Above statements give compile time error
* //You can't use super and this keywords inside SIB
* }
* static void **methodTwo**()
* {
* **super**.methodOne();
* **this**.methodOne();
* //These also give compile time error
* //You can't use super and this keywords inside static method
* //even though you are accessing static methods
* }
* }
* **You should call super() and this() calling statements inside the constructors only and they must be first statement in the constructors.**
* class SuperClassOne
* {
* void **methodOne**()
* {
* System.out.println("From Super Class");
* }
* }
* class SubClassOne extends SuperClassOne
* {
* public **SubClassOne**()
* {
* System.out.println("constructors");
* **super**();
* //compile time error
* //super() calling statement must be first statement in constructor
* }
* void **methodTwo**()
* {
* **super**();
* **this**();
* //compile time error
* //you should call super() and this()
* //calling statements only in constructors.
* }
* }

**Final Keyword in Java:**

A **final keyword in java** can be used with a class, with a variable and with a method. final keyword restricts the further modification. When you use final keyword with an entity (class or variable or method), it gets the meaning that entity is complete and can not be modified further.

**Final Class in Java:**

We can’t create a subclass to the class or we can’t extend a class or we can’t modify a class which is declared as **final**.

1. **final** class FinalClass
2. {
3. //some statements
4. }
5. class SubClass extends FinalClass
6. {
7. //compile time error
8. //Can't create sub class to the final class
9. }

**Final Method in Java:**

We can’t override a method or we can’t modify a method in the sub class which is declared as **final** in the super class.

1. class SuperClass
2. {
3. **final** void **methodOne**()
4. {
5. //some statements
6. }
7. }
8. class SubClass extends SuperClass
9. {
10. @Override
11. void **methodOne**()
12. {
13. //Compile time error
14. //can not override final method
15. }
16. }

**Final Variable in Java:**

The value of a final variable can not be changed in the whole execution once it got initialized.

1. class AnyClass
2. {
3. **final** **int** i = 10;
4. void **methodOne**()
5. {
6. i = 20; //compile time error
7. //final field can not be re-assigned
8. }
9. }

**Key Points to remember for Final :**

**1)Any class or any method can be either abstract or final but not both. abstract and final are totally opposite. Because, abstract class or abstract method must be implemented or modified in the sub classes but final does not allow this. This creates an ambiguity.**

1. //The following class gives compile time error
2. **final** abstract class AnyClass
3. {
4. //Any class can not be final and abstract
5. **final** abstract void **methodOne**();
6. //method can not be final and abstract at a time
7. }

**2)final method can be overloaded and that overloaded method can be overridden in the sub class.**

1. class SuperClass
2. {
3. **final** void **methodOne**()
4. {
5. //final method
6. }
7. void **methodOne**(**int** i)
8. {
9. //final method can be overloaded
10. }
11. }
12. class SubClass extends SuperClass
13. {
14. @Override
15. void **methodOne**(**int** i)
16. {
17. //Overloaded method can be overridden
18. }
19. }

**3)** **final variable can not be re-initialized but final variable can be used to initialize other variables.**

1. class AnyClassOne
2. {
3. **final** **int** i = 10;
4. void **methodOne**()
5. {
6. i++;
7. //above statement gives Compile time error.
8. //value of final variable can not be changed
9. **int** j = i; //final variable can be used to initialize other variables.
10. System.out.println(i); //final variable can be used
11. }
12. }

**4)** **When an array reference variable is declared as final, only variable itself is final but not the array elements.**

1. public class UseOfFinalKeyword
2. {
3. public static void **main**(**String**[] args)
4. {
5. **final** **int** X[] = **new** **int**[10]; //final array variable
6. X[2] = 10;
7. X[2] = 20; //Array element can be re-assigned
8. X = **new** **int**[30]; //compile time error
9. //can't re-assign new array object to final array variable
10. }
11. }

**5)When a reference variable is declared as final, you can’t re-assign a new object to it once it is referring to an object. But, you can change the state of an object to which final reference variable is referring.**

1. class A
2. {
3. **int** i = 10;
4. }
5. public class UseOfFinalKeyword
6. {
7. public static void **main**(**String**[] args)
8. {
9. **final** A a = **new** **A**(); //final reference variable
10. a.i = 50;
11. //you can change the state of an object to which final reference variable is pointing
12. a = **new** **A**(); //compile time error
13. //you can't re-assign a new object to final reference variable
14. }
15. }

**6)**Static variables, non-static variables and local variables all can be final. once the final variables are initialized, even you can’t re-assign the same value.

1. class A
2. {
3. static **final** **int** i = 10; //final static variable
4. **final** **int** j = 20; //final non-static variable
5. void **methodOne**(**final** **int** k)
6. {
7. //k is final local variable
8. k = 20; //compile time error
9. }
10. }
11. public class UseOfFinalKeyword
12. {
13. public static void **main**(**String**[] args)
14. {
15. A a = **new** ();
16. a.i = 10; //Compile time error
17. a.j = 20; //even you can't assign same value to final variables
18. a.methodOne(20);
19. }
20. }

**7)**If the global variables are not initialized explicitly, they get default value at the time of object creation. But final global variables don’t get default value and they must be explicitly initialized at the time of object creation. Uninitialized final field is called **Blank Final Field**.

1. class A
2. {
3. **int** i; //Non-final global variable, no need to initialize them
4. **final** **int** j; //Blank Final Field
5. **A**()
6. {
7. j=20;
8. //final global variable must get a value at the time of object creation.
9. }
10. }
11. public class UseOfFinalKeyword
12. {
13. public static void **main**(**String**[] args)
14. {
15. A a = **new** **A**();
16. }
17. }

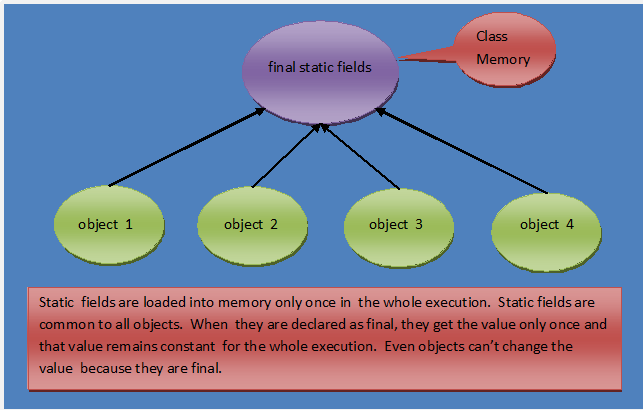
**8)** final non-static global variable must be initialized at the time of declaration or in all constructors or in any one of IIBs – Instance Initialization Blocks.

1. class A
2. {
3. **final** **int** i; //Final non-static global variable may be initialized here OR
4. //may be initialized in any one of IIB's,
5. // because while object creation, all IIBs are called. OR
6. {
7. i = 30;
8. }
9. {
10. //i = 40;
11. }
12. //must be initialized in all constructors.
13. //because while object creation, only one constructor is called
14. **A**()
15. {
16. //i=20;
17. }
18. **A**(**int** j)
19. {
20. // i=j;
21. }
22. **A**(**int** j, **int** k)
23. {
24. // i = 50;
25. }
26. }

**9)** final static global variable must be initialized at the time of declaration or in any one of SIBs – Static Initialization Blocks. (final static global variable can’t be initialized in constructors)

1. class A
2. {
3. static **final** **int** i; //final static global variable may be initialized here OR
4. //may be initialized in any one of SIBs.
5. static
6. {
7. i = 30;
8. }
9. static
10. {
11. //i = 40;
12. }
13. //final static global variable can not be initialized in constructors
14. **A**()
15. {
16. //i=20;
17. }
18. **A**(**int** j)
19. {
20. //i=j;
21. }
22. **A**(**int** j, **int** k)
23. {
24. //i = 50;
25. }
26. }

**10)** The global variable which is declared as final and static remains unchanged for the whole execution. Because, Static members are stored in the class memory and they are loaded only once in the whole execution. They are common to all objects of the class. If you declare static variables as final, any of the objects can’t change their value as it is final. Therefore, variables declared as final and static are sometimes referred to as **Constants**. All fields of interfaces are referred as constants, because they are final and static by default.



**For Each Loop in Java**

for-each loop in java is the enhanced version of for loop. It is introduced from JDK 5. **It is used to iterate all elements of an array or Collection.**

## Syntax Of for-each Loop :

The syntax of for-each loop is as follows,

1. **for**(Data\_Type variable : array or collection)
2. {
3. }

Where Data\_Type specifies type and variable specifies iteration variable.

## How for-each loop works :

The iteration variable in the for-each loop receives every element of an array or collection one at a time starting from first element to last element. i.e In the first iteration, it gets the first element. In the second iteration, it gets the second element and so on. Thus it iterates all elements of an array or the collection. The type of iteration variable must be compatible with the type of array or collection.

## Example of for-each loop which iterates an array :

1. public class ForEachLoop
2. {
3. public static void **main**(**String**[] args)
4. {
5. //An array of strings
6. **String**[] str = {"First", "Second", "Third", "Fourth", "Fifth"};
7. //iterating every element of str using for-each loop
8. **for** (**String** s : str)
9. {
10. System.out.println(s);
11. }
12. }
13. }

## Example of for-each loop which iterates the collection :

1. public class ForEachLoop
2. {
3. public static void **main**(**String**[] args)
4. {
5. //An ArrayList of strings
6. ArrayList<**String**> list = **new** ArrayList<**String**>();
7. //Adding elements to ArrayList
8. list.add("First");
9. list.add("Second");
10. list.add("Third");
11. list.add("Fourth");
12. //iterating every element of list using for-each loop
13. **for** (**String** s : list)
14. {
15. System.out.println(s);
16. }
17. }
18. }

## Nested for-each Loop :

for-each loop can be nested like normal for loop. Here is the example for Nested for-each loop which iterates two dimensional array.

1. public class ForEachLoop
2. {
3. public static void **main**(**String**[] args)
4. {
5. //Two Dimensional Array
6. **int**[][] twoDArray = { {1, 2, 3, 4}, {5, 6, 7, 8} };
7. //iterating every element of twoDArray using for-each loop
8. **for** (**int**[] oneDArray : twoDArray)
9. {
10. System.out.print("[");
11. //iterating every element of oneDArray using for-each loop
12. **for** (**int** i : oneDArray)
13. {
14. System.out.print(i+"\t");
15. }
16. System.out.println("]");
17. }
18. }
19. }

**Output :**  
[1 2 3 4 ]  
[5 6 7 8 ]

**Class Variables :**

While methods represent the behavior of an object, variables represent the state of an object. Variables may be of two types – ***Class variables* and *Instance Variables***. Class variables are common to all instances of that class where as instance variables are specific to an object. Let’s discuss class variables and instance variables in java and difference between them in detail.

1)Class variables, also called as static variables, are declared with the keyword *static*.

class StaticVariables

{

    static int i;        //Static Variable

    static String s;     //Static Variable

}

2)Class variables are common to all instances of that class i.e these variables will be shared by all objects of that class. Hence, changes made to these variables through one object will reflect in all objects.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | class ClassVariables  {      static int i = 10;        //Static Variable        static String s = "STATIC";     //Static Variable  }    public class MainClass  {      public static void main(String[] args)      {          ClassVariables obj1 = new ClassVariables();            ClassVariables obj2 = new ClassVariables();            //accessing class variables through obj1            System.out.println(obj1.i);      //Output : 10            System.out.println(obj1.s);      //Output : STATIC            //accessing class variables through obj2            System.out.println(obj2.i);      //Output : 10            System.out.println(obj2.s);      //Output : STATIC            //Making changes to class variables through obj2            obj2.i = 20;            obj2.s = "STATIC - STATIC";            //accessing class variables through obj1            System.out.println(obj1.i);      //Output : 20            System.out.println(obj1.s);      //Output : STATIC - STATIC            //accessing class variables through obj2            System.out.println(obj2.i);      //Output : 20            System.out.println(obj2.s);      //Output : STATIC - STATIC      }  }  3) Class variables can be referred through class name as well as object reference.   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      static int i = 100;        //Class Variable  }    public class MainClass  {      public static void main(String[] args)      {          //Referring class variable through class name            System.out.println(A.i);            A a = new A();            //Referring class variable through object reference            System.out.println(a.i);      }  } |   Instance Variables In Java :  1) Instance variables, also called as non-static variables are declared without*static* keyword.   |  |  | | --- | --- | | 1  2  3  4  5  6 | class InstanceVariables  {      int i;        //Instance Variable        String s;     //Instance Variable  } |   2) Instance variables are not common to all instances of a class. Each object will maintain it’s own copy of instance variables. Hence, changes made to instance variables through one object will not reflect in another object.   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | class InstanceVariables  {      int i = 10;        //Instance Variable        String s = "NON-STATIC";     //Instance Variable  }    public class MainClass  {      public static void main(String[] args)      {          InstanceVariables obj1 = new InstanceVariables();            InstanceVariables obj2 = new InstanceVariables();            //obj1 instance variables            System.out.println(obj1.i);       //Output : 10            System.out.println(obj1.s);       //Output : NON-STATIC            //obj2 instance variables            System.out.println(obj2.i);       //Output : 10            System.out.println(obj2.s);       //Output : NON-STATIC            //changing obj1 instance variables            obj1.i = 20;            obj1.s = "INSTANCE VARIABLE";            //obj1 instance variables            System.out.println(obj1.i);       //Output : 20            System.out.println(obj1.s);       //Output : INSTANCE VARIABLE            //obj2 instance variables            System.out.println(obj2.i);       //Output : 10            System.out.println(obj2.s);       //Output : NON-STATIC      }  } |   3) Instance variables can be referred only through object reference.   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      int i = 100;        //Instance Variable  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();            //Referring instance variable through object reference            System.out.println(a.i);            //You can't refer instance variable through class name, you will get compile time error            //System.out.println(A.i);      }  } |   Difference Between Class Variables And Instance Variables In Java :   |  |  | | --- | --- | | Class Variables | Instance Variables | | Class variables are declared with keyword *static*. | Instance variables are declared without *static*keyword. | | Class variables are common to all instances of a class. These variables are shared between the objects of a class. | Instance variables are not shared between the objects of a class.  Each instance will have their own copy of instance variables. | | As class variables are common to all objects of a class, changes made to these variables through one object will reflect in another. | As each object will have its own copy of instance variables, changes  made to these variables through one object will not reflect in another  object. | | Class variables can be accessed using either class name or object reference. | Instance variables can be accessed only through object reference. | |