**Hello World:**

In this post, we will see how to develop simple Hello world java program. Type the following java program in a notepad and save it in a **bin** folder of **JDK** installation directory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Hello  {       public static void main(String args[])       {            System.out.println("Hello World");       }  } |

In the above program,

1) **class** is a keyword. Every keyword in java should be in lowercase.

2) **Hello** is our class name. First letter of class name should be in uppercase. It is not a condition but it is just a convention.

3) **public**, **static**and **void** all are keywords.

4) **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.

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

6) **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.

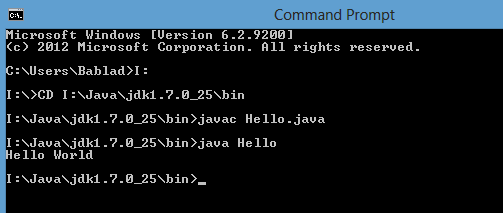
7) You can explore the source code of both **System** class and **String** class. Go to JDK installation directory and extract the ‘**src**‘ zip file. Then go to **src –> java –> lang**. In *lang* folder, you will find both **System** and **String** Java files.

8) Above program prints “**Hello World**” on the console

**Compiling And Running A Java Program:**

|  |  |
| --- | --- |
| 2  3  4  5  6  7 | class Hello  {       public static void main(String args[])       {            System.out.println("Hello World");       }  } |

Type the above java program in notepad and save with a class name in the **bin** folder of JDK installation directory like “**Hello.java**“. File name should be same as **class name** and extension should be **.java.**

Open the Command Prompt and go to bin folder of JDK installation directory. (In my system, it is I:\Java\jdk1.7.0\_25\bin)  
  
Use CD command to go to bin folder of JDK installation directory.

**>CD I:\Java\jdk1.7.0\_25\bin**

Then trigger **javac** command. **javac command is used to compile any java files.**Pass file name as an argument to the javac command. Don’t forget to add .java extension to the file name like below.

**I:\Java\jdk1.7.0\_25\bin>javac Hello.java**

If the java file contains any compile time errors, compilation will be failed and list of errors will be displayed on the console. If  the java file does not contain any compile time errors, compilation will be successful and **.class** file will be generated in the same folder.

Now run this generated **.class** file to get the desired output. **java command is used to run .class files.**Trigger java command, you will get “Hello World” as output. Just pass name of generated .class file as an argument to java command. No need to add extension just like below.

**I:\Java\jdk1.7.0\_25\bin>java Hello**

Both javac and java commands are in bin folder of JDK installation directory. We have saved our file in the same folder. It is easy to compile and run java files saved in the same folder as that of javac and java command.

**Setting Path Environement Variable:**

we will see how to compile and run java programs saved in different folders other than bin folder of JDK installation directory and also setting path environment variable.

Let us consider following java program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Sample  {       public static void main(String[] args)       {            System.out.println("I am saved in different folder");       }  } |

Type the above program in notepad and save it in different folder other than bin folder of JDK installation directory as **“Sample.java”**. I will save it in **I:\Sources**in my system**.**The path of bin folder in my system is **I:\Java\jdk1.7.0\_25\bin.**

Open the Command Prompt and go to the folder where you have saved your java file, using CD command.

**C:\…\…>I:**

**I:\>CD Sources                                                         ——->(In My System)**

If you try to trigger javac command from this location, you will get an exception saying that **javac command is not recognised as internal or external command.**To run javac or java command from this location, **you need to set path variable to bin folder of JDK installation directory.**

**I:\Sources> set path=I:\Java\jdk1.7.0\_25\bin**

To see whether the path has been set or not use **echo** command like below.

**I:\Sources> echo %path%                   ——–> It shows value of path variable**

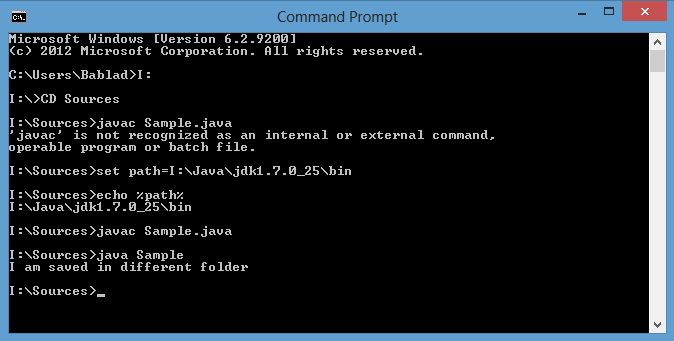
Setting the path variable is like telling command prompt that where it should search for javac and java commands.

Now trigger javac and java command to compile and run the program.

**I:\Sources> javac Sample.java**

**I:\Sources> java Sample**

This is how it looked in my system.



This setting of path variable is only applicable to this instance of command prompt. If you close this command prompt and open another command prompt, this path settings are not available. Again you have to set the path variable for that instance of command prompt.

To avoid the setting the path variable each time you open the command prompt, what you need to do is set Path Environment Variable for whole system.

**To set Path Environment Variable for the whole system**, Right click on My Computer icon on your desktop, then Go to

System Properties —> Advanced —> Environment Variables —> choose to set for either user or for whole system —> If the path variable already exist click on Edit otherwise click on New —> If you have clicked on New, enter variable name as path and variable value as path to bin folder of JDK installation directory.  If you clicked on Edit, append “;” and path of bin folder of JDK installation directory to alraedy existed value. —> Click on OK

**Usage of javac command with -d option and java command with -classpath option:**

how to set classpath and how to use javac command with -d option.

Let us consider following java program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class ProgramOne  {       public static void main(String[] args)       {            System.out.println("Setting Classpath Environment Variable");       }  } |

Type the above program in notepad and save it as **“ProgramOne.java”**. I will save it in **I:\Sources** in my system.

Till now, we have seen, when you trigger javac command, **.class** file is generated in the same folder as that of .java file. How to make this .class file should be generated in another folder say **I:\Classes**(in my system)? This can be achieved using **-d option** of javac command.

Open the command prompt and go to the location of .java file using CD command.

**C:\…\….> I:**

**I:\> CD Sources                                                             ——-> (In My System)**

Now trigger the javac command with -d option. Syntax of javac command with -d option is,

**>javac  -d  (Specify the path where to save generated .class files)  FileName.java**

We have already set the path environment variable for the whole system, so no need to set it in the command prompt. If you have not set the path environment variable, go through our previous concept to see how to set path environment variable.

Directly run javac command like below.

**I:\Sources> javac -d  I:\Classes ProgramOne.java**

This will save generated .class file in **I:\Classes.**

If you run java command from this location, you will get an error saying **could not find or load main class**. Because .class file is saved in another folder. To run .class file, you need to go to that location, again using CD command. This will be the time consuming. The easy and best way to run .class files saved in another folder is use  **-classpath** option of java command. Syntax of java command with -classpath option is,

**>java  -classpath  (path of generated .class files)  ClassName**

In our example it looks like,

**I:\Sources> java -classpath I:\Classes ProgramOne**

You can also avoid use of -classpath option each time you run java command by setting the classpath variable by using **set classpath** command.

**I:\Sources> set classpath=I:\Classes**

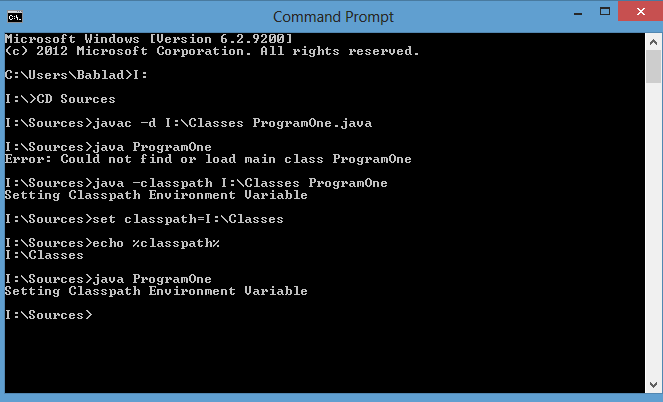
You can also check whether the classpath has been correctly set or not using **echo** command.

**I:\Sources> echo %classpath%       ——-> It will display the value of classpath variable.**

Now you can run java command without using -classpath option.

**I:\Sources> java ProgramOne**

This is how it looked in my system.



This classpath setting is available only for this instance of command prompt. You can also avoid setting classpath each time you open command prompt by **setting classpath in Environment variables section** of your system. The procedure to set classpath variable is same as that setting path environment variable as we discussed in previous concept

# Naming, Compiling and Running Java Files Containing More Than One Class Definitions:

we will see how to name, compile and running java files containing more than one class definitions.

1). Consider the following program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ClassOne  {       public static void main(String[] args)       {           ClassTwo.methodOfClassTwo();       }  }    class ClassTwo  {       static void methodOfClassTwo()       {           System.out.println("From Class Two");       }  } |

Naming : You can save this program with any name. It can be **ClassOne.java** or it can be **ClassTwo.java** or it can be **anything.java.**

Compile : You have to compile this program with the name you have given like **>javac** **ClassOne.java**or**>javac ClassTwo.java**or**>javac anything.java.**

When you compile a java file, the number of .class files generated will be equal to number of class definitions in it. i.e If a java file has one class definition, one .class file will be generated. If it has two class definitions, two .class file will be generated and so on.

Running : That means for the above program, two .class files will be generated. Then which one to run? is it **>java ClassOne** or is it **>java ClassTwo…..**It must be **>java ClassOne, because execution of any java program start with main() method.** If you try to run **>java ClassTwo,**you will get an **error: Main method not found in class ClassTwo, please define the main method as public static void main(String[] args).**

2). Now consider same example with litte modification, just declare ClassOne as **public**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ClassOne  {       public static void main(String[] args)       {            ClassTwo.methodOfClassTwo();       }  }    class ClassTwo  {       static void methodOfClassTwo()       {            System.out.println("From Class Two");       }  } |

Naming : The name of above java file must be and only “ClassOne.java”. You can’t give any other name. If you give any other name you will get compile time error : **class ClassOne is public, should be declared in a file named ClassOne.java.**

Compile : Only one name is allowed here so you have to compile with that name i.e **>javac ClassOne.java.**

Running : That must be **>java ClassOne.** because this is only class that has main() method.

3). Now make little more modifications to the program. Declare ClassTwo as public and ClassOne as default.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ClassOne  {       public static void main(String[] args)       {            ClassTwo.methodOfClassTwo();       }  }    public class ClassTwo  {       static void methodOfClassTwo()       {            System.out.println("From Class Two");       }  } |

Naming : You have to save this file with name as **“ClassTwo.java”**. If you give any other name you will get compile time error becuase ClassTwo is public.

Compile : It must be **>javac ClassTwo.java.**

Running : You must name this program as **ClassTwo.java**, you must compile this program as **>javac ClassTwo.java**but you have to run it as **>java ClassOne not as >java ClassTwo.**Because only ClassOne has main() method. ClassTwo doesn’t have main() method. If you run it as **>java ClassTwo,**you will get run time error**: Main method not found in class ClassTwo, please define the main method as public static void main(String[] args).**

4). Now make little more modifications to the program. Declare both the classes as public.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ClassOne  {       public static void main(String[] args)       {            ClassTwo.methodOfClassTwo();       }  }    public class ClassTwo  {       static void methodOfClassTwo()       {            System.out.println("From Class Two");       }  } |

Naming : Whatever you give the name, whether it is ClassOne.java or ClassTwo.java or anything.java,  you will get compile time error. Because **One java file should contain only one or zero public class. It should not contain more than one public class.**

5) Look at the following program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ClassOne  {       public static void main(String[] args)       {            System.out.println("From Class One");       }  }    class ClassTwo  {       public static void main(String[] args)       {            System.out.println("From Class Two");       }  } |

Naming : You can save this program with any name. It can be **ClassOne.java** or it can be **ClassTwo.java** or it can be **anything.java**as there is no public class**.**

Compile : You have to compile this program with the name you have given i.e **>javac** **ClassOne.java**or**>javac ClassTwo.java**or**>javac anything.java.**

Running : Notice that both the classes have main() method. You can run both the classes with their name. i.e  If you trigger **>java ClassOne**, you will get **From Class One** as output. If you trigger **>java ClassTwo**, you will get **From Class Two**as output.

# Increment And Decrement Operators:

**1). Post Increment Operator**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Sample  {       public static void main(String[] args)       {            int i = 0;            System.out.println(i++);       }  } |

Let us have a look at above program. If you are thinking that output will be 1 then you are wrong. If you run this program output will be 0 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.

**2). Pre Increment Operator**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Sample  {       public static void main(String[] args)       {            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.

**3). Post Decrement Operator**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Sample  {       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.

**4). Pre Decrement Operator**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Sample  {       public static void main(String[] args)       {            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 initialized?:

we will discuss about global and local variables, their behavior and default values and you will also get an answer to the above question.

Let us start with one simple program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class GlobalAndLocalVariables  {       static int globalVariable;         static void methodOne()       {            int localToMethodOne = 0;            System.out.println(localToMethodOne);            System.out.println(globalVariable);      //    System.out.println(localToMain);       }         public static void main(String[] args)       {            int localToMain = 0;            System.out.println(localToMain);            System.out.println(globalVariable);      //    System.out.println(localToMethodOne);       }  } |

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 9) and also inside the main() method (Line 17). 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() (Line 8) and not available outside the methodOne() (Line 18). If you use outside methodOne(), you will get compile time error. Variable ‘localToMain’ declared in Line 15 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.

|  |  |
| --- | --- |
| Data Type | Default value |
| Int | 0 |
| boolean | false |
| byte | 0 |
| short | 0 |
| long | 0 |
| float | 0.0 |
| double | 0.0 |
| All Derived 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class LocalVariableBehavior  {       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 : i may not have been initialised.** 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 6) before it is initialized(Line 7).

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

# 50 Java Keywords With Examples:

**1) abstract**

abstract keyword is used to implement the abstraction in java. A method which doesn’t have method definition must be declared as abstract and the class containing it must be declared as abstract. You can’t instantiate abstract classes. Abstract methods must be implemented in the sub classes. You can’t use abstract keyword with variables and constructors. [[***See more***](https://javaconceptoftheday.com/abstraction-in-java/)]

|  |  |
| --- | --- |
| 1  2  3  4 | abstract class AbstractClass  {      abstract void abstractMethod();  } |

**2) assert**

assert keyword is used in the assertion statements. These statements will enable you to test your assumptions about a program. Assertion statements provide the best way to detect and correct the programming errors. Assertion statements take one boolean expression as input and assumes that this will be always true. If the boolean expression returns false, AssertionError will be thrown.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MainClass  {      public static void main(String[] args)      {          System.out.println("Enter your marks");            Scanner sc = new Scanner(System.in);            int marks = sc.nextInt();            assert marks > 35 : "FAIL";      }  } |

**3) boolean**

boolean keyword is used to define boolean type variables. boolean type variables can hold only two values – either true or false.

|  |  |
| --- | --- |
| 1 | boolean isActive = true; |

**4) break**

The break keyword is used to stop the execution of a loop(for, while, switch-case) based on some condition.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class MainClass  {      public static void main(String[] args)      {          for (int i = 0; i < 100; i++)          {              System.out.println(i);                if(i == 50)              {                  break;              }          }      }  } |

**5) byte**

byte keyword is used to declare byte type of variables. A byte variable can hold a numeric value in the range from -128 to 127.

|  |  |
| --- | --- |
| 1 | byte b = 50; |

**6) switch       7) case**

Both switch and case keywords are used in the switch-case statement.

|  |  |
| --- | --- |
| 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 | public class MainClass  {      public static void main(String[] args)      {          Scanner sc = new Scanner(System.in);            System.out.println("Enter Day :");            int day = sc.nextInt();            switch (day)          {              case 1 : System.out.println("SUNDAY");              break;                case 2 : System.out.println("MONDAY");              break;                case 3 : System.out.println("TUESDAY");              break;                case 4 : System.out.println("WEDNESDAY");              break;                case 5 : System.out.println("THURSDAY");              break;                case 6 : System.out.println("FRIDAY");              break;                case 7 : System.out.println("SATURDAY");              break;                default: System.out.println("Invalid");              break;          }      }  } |

**8) try     9) catch     10) finally**

try, catch and finally keywords are used to handle the exceptions in java. The statements which are to be monitored for exceptions are kept in the try block. The exceptions thrown by the try block are caught in the catch block. finally block is always executed. [[***See more***](https://javaconceptoftheday.com/try-catch-finally-blocks-java/)]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class MainClass  {      public static void main(String[] args)      {          try          {              int i = Integer.parseInt("abc");          }          catch(NumberFormatException ex)          {              System.out.println(ex);          }          finally          {              System.out.println("This will be always executed");          }      }  } |

**11) char**

char keyword is used to declare primitive char type variables. char represents the characters in java.

|  |  |
| --- | --- |
| 1  2  3  4  5 | char a = 'A';    char b = 'B';    char c = 'C'; |

**12) class**

class keyword is used to define the classes in java.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class MyClass  {      class MyInnerClass      {          //Inner Class      }  } |

**13) continue**

continue keyword is used to stop the execution of current iteration and start the execution of next iteration in a loop.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class MainClass  {      public static void main(String[] args)      {          for (int i = 0; i <= 100; i++)          {              if(i % 5 != 0)              {                  continue;              }                System.out.println(i);          }      }  } |

**14) default**

default keyword is used to define the default methods in an interface (From Java 8). default keyword is also used in the switch-case statements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | interface MyInterface  {      public default void myDefaultMethod()      {          System.out.println("Default Method");      }  } |

**15) do**

do keyword is used in a do–while loop. do-while loop is used to execute one or more statements repetitively until a condition returns false.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class MainClass  {      public static void main(String[] args)      {          int a = 10;            int b = 20;            do          {              a = a + b;                b = b + 10;                System.out.println("a = "+a);                System.out.println("b = "+b);            } while (a <= 100);      }  } |

**16) double**

double keyword is used to declare primitive double type of variables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MainClass  {      public static void main(String[] args)      {          double d1 = 23.56;            double d2 = 56.23;            double d3 = d1 + d2;            System.out.println(d3);      }  } |

**17) if         18) else**

if and else keywords are used in if-else block.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class MainClass  {      public static void main(String[] args)      {          Scanner sc = new Scanner(System.in);            System.out.println("Enter a string :");            String input = sc.next();            if(input.equalsIgnoreCase("JAVA"))          {              System.out.println("It's JAVA");          }          else          {              System.out.println("It's not JAVA");          }      }  } |

**19) enum**

enum keyword is used to define enum types.

|  |  |
| --- | --- |
| 1  2  3  4 | enum MyEnums  {      A, B, C, D;  } |

**20) extends**

extends keyword is used in inheritance. It is used when a class extends another class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class SuperClass  {      //Super Class  }    class SubClass extends SuperClass  {      //Sub Class  } |

**21) final**

final keyword is used when a class or a method or a field doesn’t need further modifications. final class can’t be extended, final method can’t be overridden and the value of a final field can’t be changed. [[***See more***](https://javaconceptoftheday.com/final-keyword-in-java/)]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | final class FinalClass  {      final int finalVariable = 10;        final void finalMethod()      {          //final method      }  } |

**22) float**

float keyword indicates primitive float type of variables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MainClass  {      public static void main(String[] args)      {          float f1 = 45.26f;            float f2 = 84.25f;            float f3 = f2 - f1;            System.out.println(f3);      }  } |

**23) for**

for loop is used to execute the set of statements until a condition is true.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class MainClass  {      public static void main(String[] args)      {          for (int i = 0; i <= 10; i++)          {              System.out.println(i);          }      }  } |

**24) implements**

implements keyword is used while implementing an interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface MyInterface  {      void myMethod();  }    class MyClass implements MyInterface  {      public void myMethod()      {          System.out.println("My Method");      }  } |

**25) import**

import keyword is used to import the members of a particular package into current java file. [[***See more***](https://javaconceptoftheday.com/static-import-java/)]

|  |  |
| --- | --- |
| 1  2  3 | import java.sql.\*;  import java.util.Arrays;  import java.util.Scanner; |

**26) instanceOf**

instanceOf is used to check whether an object is of specified type. The syntax for using instanceOf keyword is “**Object\_Reference instanceOf Type**“.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class A  {    }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();            if(a instanceof A)          {              System.out.println("a is of type A");          }      }  } |

**27) int**

int keyword is used to declare primitive integer type of variables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MainClass  {      public static void main(String[] args)      {          int i1 = 10;            int i2 = 20;            int i3 = i1 \*  i2;            System.out.println(i3);      }  } |

**28) interface**

interface keyword is used to define the interfaces in java.

|  |  |
| --- | --- |
| 1  2  3  4 | interface MyInterface  {      void myMethod();  } |

**29) long**

long is used to define the primitive long type variables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MainClass  {      public static void main(String[] args)      {          long l1 = 101;            long l2 = 202;            long l3 = l1 +  l2;            System.out.println(l3);      }  } |

**30) native**

native keyword is used with a method to indicate that a particular method is implemented in native code using Java Native Interfaces(JNI).

|  |  |
| --- | --- |
| 1  2  3  4 | class AnyClass  {      public native void anyMethod(int i, double d);  } |

**31) new**

new keyword is used while creating the instances of a class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class A  {    }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      }  } |

**32) package**

package keyword is used to specify a package to which the current file belongs to.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package pack1;    class A  {    } |

**33) private**

private keyword is used to declare a member of a class as private. private methods and fields are visible within the class in which they are defined.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class A  {      private int i = 111;   //private field        private void method()      {          //private method      }  } |

**34) protected**

protected keyword is used to declare a member of a class as protected. protected members of a class are visible within the package only, but they can be inherited to any sub classes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class A  {      protected int i = 111;   //protected field        protected void method()      {          //protected method      }  } |

**35) public**

public keyword is used to declare the members of a class or class itself as public. public members of a class are visible from anywhere and they can be inherited to any sub classes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class A  {      public int i = 222;   //public field        public A()      {          //public constructor      }        public void method()      {          //public method      }  } |

**36) return**

return keyword is used to return the control back to the caller from the method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class A  {      int method(int i)      {          return i\*i;     //method returning a value      }  } |

**37) short**

short keyword is used to declare primitive short type variables.

|  |  |
| --- | --- |
| 1  2  3 | short s1 = 11;    short s2 = 22; |

**38) static**

***static*** keyword is used to define the class level members of a class. static members of a class are stored in the class memory and you can access them directly through class name. No need to instantiate a class. [[See more](https://javaconceptoftheday.com/static-members-java/)]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class A  {      static int staticField = 555;    //Static Field        static void staticMethod()      {          //Static method      }  }    public class MainClass  {      public static void main(String[] args)      {          System.out.println(A.staticField);    //Accessing staticField via class name            A.staticMethod();     //Accessing staticMethod via class name      }  } |

**39) strictfp**

strictfp keyword is used to implement the strict precision of floating point calculations on different platforms. strictfp can be used with classes, interfaces and methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | strictfp interface I  {      //strictfp applied on interface  }    strictfp class C  {      //strictfp applied on class  }    class A  {      strictfp void method()      {          //strictfp applied on method      }  } |

**40) super**

super keyword is used to access super class members inside a sub class.

|  |  |
| --- | --- |
| 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 | class A  {      int i;        public A(int i)      {          this.i = i;      }        void methodA()      {          System.out.println(i);      }  }    class B extends A  {      public B()      {          super(10);    //Calling super class constructor      }        void methodB()      {          System.out.println(super.i);    //accessing super class field            super.methodA();    //Calling super class method      }  } |

**41) synchronized**

synchronized keyword is used to implement the synchronization in java. only one thread can enter into a method or a block which is declared as synchronized. Any thread which wants to enter synchronized method or block must acquire object lock of those methods or blocks. [[***See more***](https://javaconceptoftheday.com/synchronization-in-java/)]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class AnyClass  {      synchronized void synchronizedMethod()      {          //Synchronized method      }        void anyMethod()      {          synchronized (this)          {              //Synchronized block          }      }  } |

**42) this**

this keyword is used to access other members of the same class.

|  |  |
| --- | --- |
| 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 | class AnyClass  {      int i;        AnyClass()      {          System.out.println("First Constructor");      }        AnyClass(int j)      {          this();    //calling statement to First Constructor          System.out.println("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      }  } |

**43) throw**

throw keyword is used to throw the exceptions manually. [[***See more***](https://javaconceptoftheday.com/throwing-rethrowing-exception-java/)]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class MainClass  {      public static void main(String[] args)      {          try          {              //throwing NumberFormatException manually                throw new NumberFormatException();          }          catch(Exception ex)          {              System.out.println(ex);          }      }  } |

**44) throws**

throws keyword is used to specify the exceptions which the current method may throw. [[***See more***](https://javaconceptoftheday.com/throws-keyword-java/)]

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class A  {      void method() throws NumberFormatException      {          int i = Integer.parseInt("abc");      }  } |

**45) transient**

transient keyword is used in serialization. A variable which is declared as transient will not be eligible for serialization.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class MyClass implements Serializable  {      int a;        transient String s;   //This will not be serialized        double d;  } |

**46) void**

void keyword is used to indicate that method returns nothing.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class A  {      void methodReturnsNothing()      {          //Method returns no value      }  } |

**47) volatile**

volatile keyword is used in the concurrent programming. The value of a variable which is declared as volatile will be written into or read from the main memory.

|  |  |
| --- | --- |
| 1  2  3  4 | class A  {      public volatile int counter = 0;  } |

**48) while**

while keyword is used in the while loop.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class MainClass  {      public static void main(String[] args)      {          int i = 10;            while (i <= 100)          {              System.out.println(i);                i = i + 10;          }      }  } |

**49) goto        50) const**

Both goto and const are reserved words in java but they are currently not used.

**Note** : true, false and null are not the keywords. They are literals in java.

# Final:

# 10 Points Every Java Programmer Should Know About 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.

In this post, we will discuss some about 10 important points about final keyword which every java programmer should know. Let’s start with some simple basic things about final keyword in java.

## **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  2  3  4  5  6  7  8  9  10 | final class FinalClass  {      //some statements  }    class SubClass extends FinalClass  {      //compile time error      //Can't create sub class to the final class  } |

## **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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class SuperClass  {      final void methodOne()      {          //some statements      }  }    class SubClass extends SuperClass  {      @Override      void methodOne()      {          //Compile time error          //can not override final method      }  } |

## **final variable in java :**

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

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

## **10 Points Every Java Programmer Should Know About final Keyword In Java :**

**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  2  3  4  5  6  7  8  9 | //The following class gives compile time error    final abstract class AnyClass  {      //Any class can not be final and abstract        final abstract void methodOne();      //method can not be final and abstract at a time  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class SuperClass  {      final void methodOne()      {          //final method      }        void methodOne(int i)      {          //final method can be overloaded      }  }    class SubClass extends SuperClass  {      @Override      void methodOne(int i)      {          //Overloaded method can be overridden      }  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class AnyClassOne  {      final int i = 10;        void methodOne()      {          i++;          //above statement gives Compile time error.          //value of final variable can not be changed            int j = i;        //final variable can be used to initialize other variables.            System.out.println(i);  //final variable can be used      }  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class UseOfFinalKeyword  {      public static void main(String[] args)      {          final int X[] = new int[10];     //final array variable            X[2] = 10;          X[2] = 20;     //Array element can be re-assigned            X = new int[30];  //compile time error          //can't re-assign new array object to final array variable      }  } |

**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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class A  {      int i = 10;  }    public class UseOfFinalKeyword  {      public static void main(String[] args)      {          final A a = new A();  //final reference variable            a.i = 50;          //you can change the state of an object to which final reference variable is pointing            a = new A();  //compile time error            //you can't re-assign a new object to final reference variable      }  } |

**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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class A  {      static final int i = 10;   //final static variable      final int j = 20;          //final non-static variable        void methodOne(final int k)      {          //k is final local variable          k = 20;   //compile time error      }  }    public class UseOfFinalKeyword  {      public static void main(String[] args)      {          A a = new ();            a.i = 10;     //Compile time error          a.j = 20;     //even you can't assign same value to final variables            a.methodOne(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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class A  {      int i;   //Non-final global variable, no need to initialize them        final int j;         //Blank Final Field        A()      {          j=20;            //final global variable must get a value at the time of object creation.      }  }    public class UseOfFinalKeyword  {      public static void main(String[] args)      {          A a = new A();      }  } |

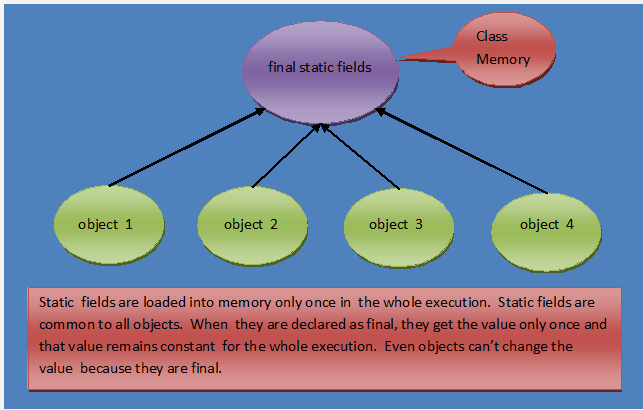
**8)** final non-static global variable must be initialized at the time of declaration or in all constructors or in any one of IIBs – Instance Initialization Blocks.

|  |  |
| --- | --- |
| 1  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 | class A  {      final int i;  //Final non-static global variable may be initialized here  OR        //may be initialized in any one of IIB's,      // because while object creation, all IIBs are called.  OR        {          i = 30;      }        {          //i = 40;      }        //must be initialized in all constructors.      //because while object creation, only one constructor is called        A()      {          //i=20;      }        A(int j)      {         // i=j;      }        A(int j, int k)      {         // i = 50;      }  } |

**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  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 | class A  {      static final int i;   //final static global variable may be initialized here OR        //may be initialized in any one of SIBs.        static      {          i = 30;      }        static      {          //i = 40;      }        //final static global variable can not be initialized in constructors        A()      {          //i=20;      }        A(int j)      {          //i=j;      }        A(int j, int k)      {          //i = 50;      }  } |

**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.



# 2. Java Memory Management:

# Memory Management In Java – Stack And Heap:

Whenever you trigger a java command, it divides allocated memory into two parts – **Stack and Heap.**Stack is used only for execution purpose. Heap is used for storage purpose. consider the following program and we will see how it uses stack and heap memory through diagram.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class StackAndHeapMemory  {       static void methodOne()       {            System.out.println("From Method One");            methodTwo();       }         static void methodTwo()       {            System.out.println("From Method Two");       }         public static void main(String[] args)       {            System.out.println("Main Method Started");            methodOne();            System.out.println("Main Method Ended");       }  } |

When you trigger >java StackAndHeapMemory, java command divides allocated memory into two parts. one is stack and another one is heap. As already said, stack is used for execution purpose and heap is used for storage purpose. Here is the steps involved in execution of above program.

**Step 1 :**

First java command enters the stack memory for execution. First it checks whether the class StackAndHeapMemory is loaded in heap memory or not. If it is not loaded, loading operation of class StackAndHeapMemory starts.

**Step 2 :**

Randomly one object is created in the heap memory. It is also called **Class Memory**.  After object creation, all static members are loaded into class memory. You know that execution of every java program start with main() method. So, java commands calls main() method for execution.

**Step 3 :**

main() method enters stack memory for execution. First statement in main() method (Line 16) is executed. It prints “Main Method Started” on the console. In the second statement (Line 17), it calls methodOne() for execution.

**Step 4 :**

methodOne() enters the stack for execution. First statement (Line 5) of methodOne() is executed first. It prints “From Method One” on the console. In the second statement (Line 6), it calls methodTwo() for execution.

**Step 5 :**

methodTwo() enters the stack for execution. In the methodTwo(), there is only one statement (Line 11). This statement is executed. It prints “From Method Two” on the console. There is no other statements in methodTwo(). So, methodTwo() leaves stack memory.

**Step 6 :**

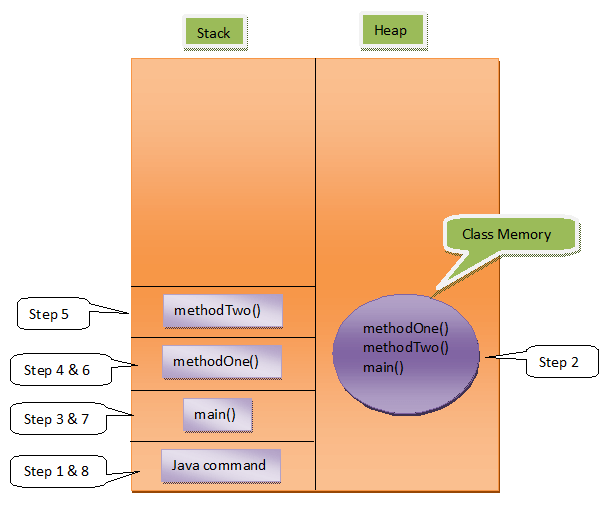
Now, again control comes back to methodOne(). Second statement (Line 6) of methodOne() is already executed in step 4. There are no other statements left in methodOne(). So, methodOne() also leaves stack.

**Step 7 :**

Now, control comes back to main() method. second statement (Line 17) of main() is already executed is in step 3. So, now third statement (Line 18) is executed. It prints “Main Method Ended” on the console. As there are no other statements left in main() method, it also leaves stack after clearing the heap memory.

**Step 8 :**

java command also leaves stack memory and gives back the allocated memory to OS.

Here is the diagrammatic representation of memory allocation of the above program.  


**Output :**

Main Method Started  
From Method One  
From Method Two  
Main Method Ended

# 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  2  3  4 | static  {       //Set Of Statements  } |

Consider the following program.

|  |  |
| --- | --- |
| 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 | class StaticComponents  {       static int staticVariable;         static       {            System.out.println("StaticComponents SIB");            staticVariable = 10;       }         static void staticMethod()       {            System.out.println("From StaticMethod");            System.out.println(staticVariable);       }  }    public class MainClass  {       static       {            System.out.println("MainClass SIB");       }         public static void main(String[] args)       {           //Static Members directly accessed with Class Name            StaticComponents.staticVariable = 20;            StaticComponents.staticMethod();       }  } |

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.

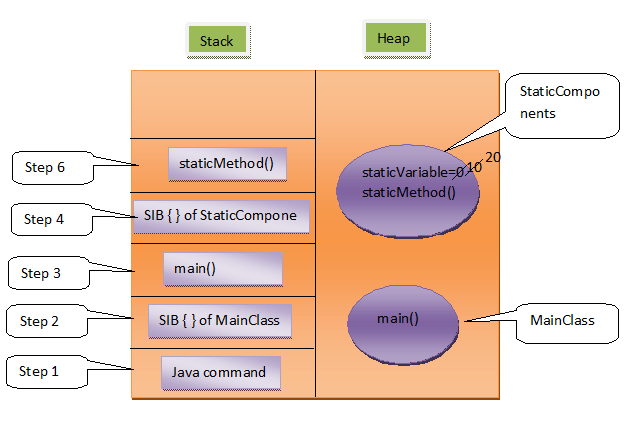
**Step 6:**

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.



**Output :**

Main Class SIB  
StaticComponents SIB  
From StaticMethod  
20

# Non-Static Members And Their Memory Management In Java:

we will see non-static components of a class.

Let’s start with simple definitions of class and object.

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.

Let’s have a look at this example.

|  |  |
| --- | --- |
| 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 | class A  {       int nonStaticVariable;       static int staticVariable;         static void staticMethod()       {            System.out.println(staticVariable);       //   System.out.println(nonStaticVariable);       }         void nonStaticMethod()       {            System.out.println(staticVariable);            System.out.println(nonStaticVariable);       }  }    class MainClass  {       public static void main(String[] args)       {            A.staticVariable = 10;       //   A.nonStaticVariable = 10;            A.staticMethod();      //    A.nonStaticMethod();              A a1 = new A();            A a2 = new A();              System.out.println(a1.nonStaticVariable);            System.out.println(a1.staticVariable);            a1.nonStaticMethod();            a1.staticMethod();              System.out.println(a2.staticVariable);            a1.staticVariable = 20;            System.out.println(a2.staticVariable);       }  } |

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.

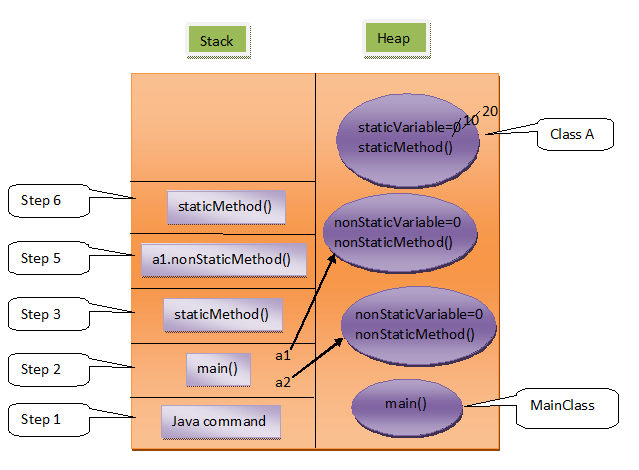
**Step 7 :**

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,



**Output :**

10  
0  
10  
10  
0  
10  
10  
20

# How The Strings Are Stored In The Memory?:

In Java, strings are special. Java gives some special attention to string types that no other types enjoy such attention. For example, to create the string objects you need not to use ‘**new**‘ keyword. Where as to create other type of objects you have to use ‘new’ keyword. Like this, strings enjoy some special attention by the java. This attention is worth the while, because the strings are used almost everywhere while developing any kind of applications.

While storing the string objects in the memory also, they are specially treated by the Java. After reading this article, you will come to know how they are specially treated in the memory.

We all know that JVM divides the allocated memory to a Java program into two parts. one is **Stack** and another one is **heap**. Stack is used for execution purpose and heap is used for storage purpose. In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called **String Constant Pool**.

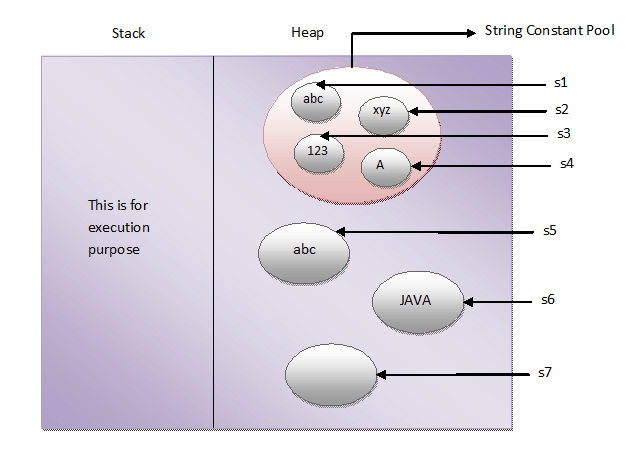
Whenever you create a string object using string literal, that object is stored in the **string constant pool** and whenever you create a string object using new keyword, such object is stored in the heap memory.

For example, when you create string objects like below, they will be stored in the String Constant Pool.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | String s1 = "abc";    String s2 = "xyz";    String s3 = "123";    String s4 = "A"; |

And when you create string objects using new keyword like below, they will be stored in the heap memory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | String s5 = new String("abc");    char[] c = {'J', 'A', 'V', 'A'};    String s6 = new String(c);    String s7 = new String(new StringBuffer()); |

This is how String Constant Pool looks like in the memory.  


One more interesting thing about String Constant Pool is that, **pool space is allocated to an object depending upon it’s content**. There will be no two objects in the pool having the same content.

This is what happens when you create string objects using string literal,

**“When you create a string object using string literal, JVM first checks the content of to be created object. If there exist an object in the pool with the same content, then it returns the reference of that object. It doesn’t create new object. If the content is different from the existing objects then only it creates new object.”**

But, when you create string objects using new keyword, a new object is created whether the content is same or not.

This can be proved by using “==” operator. As “==” operator returns true if two objects have same physical address in the memory otherwise it will return false. In the below example, s1 and s2 are created using string literal “abc”. So, s1 == s2 returns true. Where as s3 and s4 are created using new operator having the same content. But, s3 == s4 returns false.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class StringExamples  {      public static void main(String[] args)      {          //Creating string objects using literals            String s1 = "abc";            String s2 = "abc";            System.out.println(s1 == s2);        //Output : true            //Creating string objects using new operator            String s3 = new String("abc");            String s4 = new String("abc");            System.out.println(s3 == s4);        //Output : false      }  } |

**In simple words, there can not be two string objects with same content in the string constant pool. But, there can be two string objects with the same content in the heap memory.**

# How Are the Arrays stored in the memory?

Internally, Arrays are like objects. That’s why they are created using new operator. Array objects can hold two types of data.

1) primitive type of data

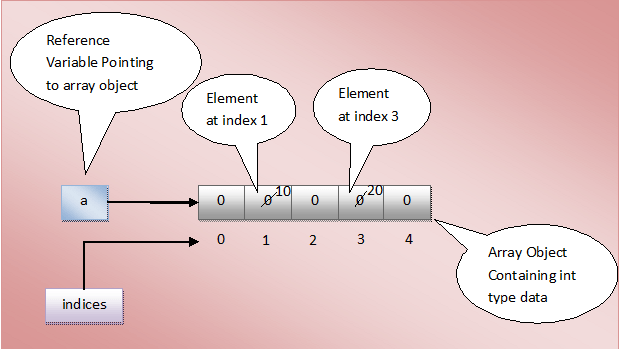
2) References to derived type of data.

Let’s discuss how the memory is allocated to an array containing primitive data or references to other objects.

## **Array Containing Primitive Type Of Data :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a;           //Declaring int type array reference variable which will be referring to int type array object            a = new int[5];    //Creating an int type Array Object containing 5 elements of int type            a[1] = 10;         //Changing Value of element at index 1            a[3] = 20;         //Changing Value of element at index 3      }  } |

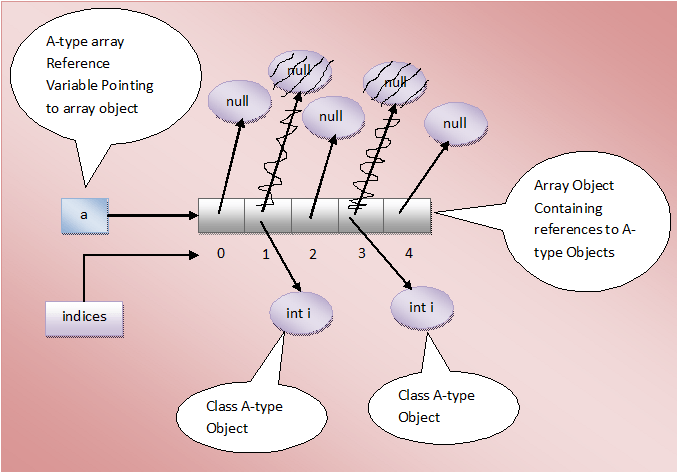
In the above example, First statement of main method (Line 5) declares an array reference variable of int type. In the next statement, an array object containing 5 elements of int type is created and assigned to already declared array reference variable. Initially, all elements of array are initialized with default values (i.e 0 in this example). In the next statement (Line 9), Value of element at index 1 is changed to 10. In the next statement, value of element at index 3 is changed to 20. This can be well explained with the diagram like below.



## **Array Containing References To Derived Type Of Data :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class A  {      int i;  }    public class ArraysInJava  {      public static void main(String[] args)      {          A[] a;           //Declaring array reference variable of A-type which will be referring to an array object containing references to A-type objects            a = new A[5];    //Creating an Array Object containing 5 references to A-type objects            a[1] = new A();         //Creating an object of A-type and assigning it to element at index 1            a[3] = new A();         //Creating an object of A-type and assigning it to element at index 3      }  } |

In the above example, first statement of main method (Line 10) declares an array reference variable of class A-type. This array reference variable must refer to an array object holding references to class A-type objects. In the next statement, one array object which can hold 5 references to class A-type objects is created. Please remember that, in this statement only an array object is created not 5 objects of class A-type are created . Initially, all elements of array will be pointing to null. In the next two statements (Line 14 and 16), two objects of class A-type are created and assigned to elements at index 1 and 3 respectively. It can be represented using diagram like below.



# Garbage Collection And finalize() method In Java:

You all know that an object is created in the memory using **new** operator. Constructor is used to initialize the properties of that object. When an object is no more required, it must be removed from the memory so that that memory can be reused for other objects. Removing unwanted objects or **abandoned objects** from the memory is called **garbage collection** (GC). In the languages like C++, GC is performed manually using destructors.

But, there is no destructors in java. In java, there exist better mechanism to handle the garbage collection. You need not to delete unwanted objects explicitly. JVM does this for you. JVM implicitly sweeps out abandoned objects from the memory.

Before moving on to Garbage Collection in java, let’s have a look at the finalize() method of Object class.

## **finalize() method In Java:**

finalize() method is a protected and non-static method of **java.lang.Object** class. This method will be available in all objects you create in java. This method is used to perform some final operations or clean up operations on an object before it is removed from the memory.  you can override the finalize() method to keep those operations you want to perform before an object is destroyed. Here is the general form of finalize() method.

|  |  |
| --- | --- |
| 1  2  3  4 | protected void finalize() throws Throwable  {      //Keep some resource closing operations here  } |

## **Garbage Collection In Java :**

Whenever you run a java program, JVM creates three threads. 1) main thread   2) Thread Scheduler   3) Garbage Collector Thread. In these three threads, main thread is a user thread and remaining two are daemon threads which run in background.

The task of main thread is to execute the main() method. The task of thread scheduler is to schedule the threads. The task of garbage collector thread is to sweep out abandoned objects from the heap memory. Abandoned objects or dead objects are those objects which does not have live references. Garbage collector thread before sweeping out an abandoned object, it calls finalize() method of that object. After finalize() method is executed, object is destroyed from the memory. That means clean up operations which you have kept in the finalize() method are executed before an object is destroyed from the memory.

Garbage collector thread does not come to heap memory whenever an object becomes abandoned. It comes once in a while to the heap memory and at that time if it sees any abandoned objects, it sweeps out those objects after calling finalize() method on them. Garbage collector thread calls finalize() method only once for one object.

Let’s discuss some interesting points about garbage collection and finalize() method.

## **Some Interesting Points About Garbage Collection And finalize() method In Java :**

***1)*** In some scenarios, finalize() method is not at all called by the garbage collector thread. For example, When I executed the below program in my system, finalize() method of Class A is not at all executed.

|  |  |
| --- | --- |
| 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 | class A  {      int i = 50;        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method");      }  }    public class Test  {     public static void main(String[] args)     {        //Creating two instances of class A          A a1 = new A();          A a2 = new A();          //Assigning a2 to a1          a1 = a2;          //Now both a1 and a2 will be pointing to same object          //An object earlier referred by a1 will become abandoned          System.out.println("done");     }  } |

***2)*** You can make finalize() method to be executed forcefully using either **Runtime.getRuntime().runFinalization()** OR **Runtime.runFinalizersOnExit(true)**. But, both the methods have disadvantages. Runtime.getRuntime().runFinalization() makes the just best effort to execute finalize() method. It is not gauranteed that it will execute finalize() method. Runtime.runFinalizersOnExit(true) is deprecated in JDK because some times it runs finalize() method on live objects also.

|  |  |
| --- | --- |
| 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 | class A  {      int i = 50;        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method");      }  }    public class Test  {     public static void main(String[] args)     {        //Creating two instances of class A          A a1 = new A();          A a2 = new A();          //Assigning a2 to a1          a1 = a2;          //Making finalize() method to execute forcefully          Runtime.getRuntime().runFinalization();          System.out.println("done");     }  } |

***3)*** you can call garbage collector explicitly using **System.gc()** or **RunTime.getRunTime().gc()**. Again it is just a request to garbage collector not a command. It is up to garbage collector to honour this request.

|  |  |
| --- | --- |
| 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 | class A  {      int i;        public A(int i)      {          this.i = i;      }        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method, i = "+i);      }  }    public class Test  {     public static void main(String[] args)     {         //Creating two instances of class A           A a1 = new A(10);           A a2 = new A(20);           //Assigning a2 to a1           a1 = a2;           //Now both a1 and a2 will be pointing same object           //An object earlier referred by a1 will become abandoned               //Calling garbage collector thread explicitly           System.gc();              //OR call Runtime.getRuntime().gc();           System.out.println("done");     }  } |

***4)*** finalize() methods are not chained like constructors.i.e there is no calling statement to super class finalize() method inside the finalize() method of sub class. You need to explicitly call super class finalize() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | protected void finalize() throws Throwable  {      System.out.println("From Finalize Method");        //Calling super class finalize() method explicitly        super.finalize();  } |

***5)*** Exceptions occurred in finalize() method are not propagated. They are ignored by the garbage collector.

**6)** You can call finalize() method explicitly on an object before it is abandoned. When you call, only operations kept in finalize() method are performed on an object. Object will not be destroyed from the memory.

|  |  |
| --- | --- |
| 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  47  48  49  50  51 | class A  {      int i;        public A(int i)      {          this.i = i;      }        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method, i = "+i);            //Calling super class finalize() method explicitly            super.finalize();      }  }    public class Test  {     public static void main(String[] args)     {         //Creating two instances of class A           A a1 = new A(10);           A a2 = new A(20);           //Calling finalize() method of a1 before it is abandoned         try         {             a1.finalize();         }         catch (Throwable e)         {             e.printStackTrace();         }           //Assigning a2 to a1           a1 = a2;           //Now both a1 and a2 will be pointing same object           //An object earlier referred by a1 will become abandoned           System.out.println("done");     }  } |

***7)*** finalize() method on an abandoned object is called only once by the garbage collector thread. GC ignores finalize() method called on an object by the developer.

# Types Of References In Java : Strong, Soft, Weak And Phantom:

One of the beauty of the Java language is that it doesn’t put burden of memory management on the programmers. Java automatically manages the memory on the behalf of the programmers. Java programmers need not to worry about freeing the memory after the objects are no more required. **Garbage Collector Thread** does this for you. This thread is responsible for sweeping out unwanted objects from the memory. But, you have no control over garbage collector thread. You can’t make it to run whenever you want. It is up to JVM which decides when to run garbage collector thread. But, with the introduction of **java.lang.ref** classes, you can have little control over when your objects will be garbage collected.

Depending upon how objects are garbage collected, references to those objects in java are grouped into 4 types. They are,

1) Strong References

2) Soft References

3) Weak References

4) Phantom References

Let’s discuss these reference types in detail.

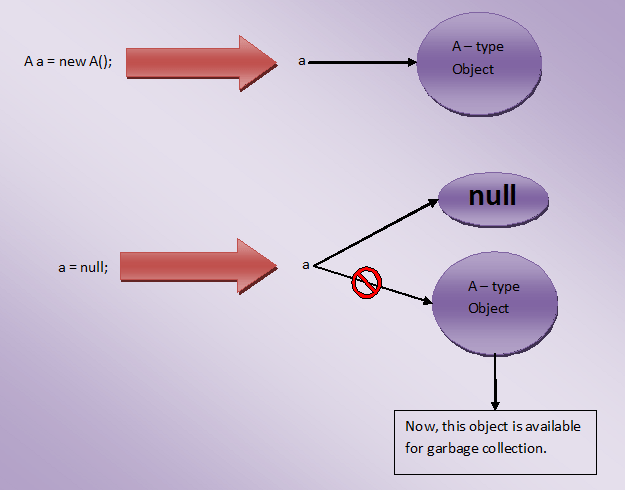
## **1) Strong References**

These type of references we use daily while writing the code. Any object in the memory which has active **strong reference** is not eligible for garbage collection. For example, in the below program, reference variable **‘a’** is a strong reference which is pointing to class A-type object. At this point of time, this object can’t be garbage collected as it has strong reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class A  {      //Class A  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            a = null;    //Now, object to which 'a' is pointing earlier is eligible for garbage collection.      }  } |

If you make reference **‘a’** to point to null like in Line 12, then, object to which ‘a’ is pointing earlier will become eligible for garbage collection. Because, it will have no active references pointing to it. This object is most likely to be garbage collected when garbage collector decides to run.

Look at the below picture for more precise understanding.



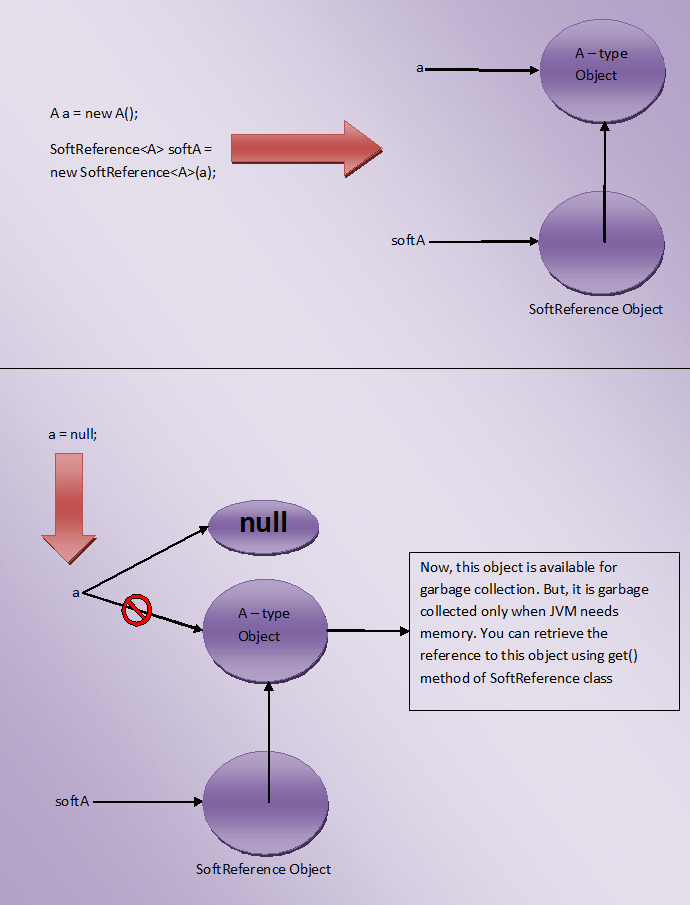
## **2) Soft References**

The objects which are softly referenced will not be garbage collected (even though they are available for garbage collection) until JVM badly needs memory. These objects will be cleared from the memory only if JVM runs out of memory. You can create a soft reference to an existing object by using  **java.lang.ref.SoftReference** class. Below is the code example on how to create a soft reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Soft Reference to A-type object to which 'a' is also pointing            SoftReference<A> softA = new SoftReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is eligible for garbage collection. But, it will be garbage collected only when JVM needs memory.            a = softA.get();    //You can retrieve back the object which has been softly referenced      }  } |

In the above example, you create two strong references – ‘**a**‘ and ‘**softA**‘. ‘a’ is pointing to A-type object and ‘softA’ is pointing to SoftReference type object. This SoftReference type object is internally referring to A-type object to which ‘a’ is also pointing. When ‘a’ is made to point to null, object to which ‘a’ is pointing earlier becomes eligible for garbage collection. But, it will be garbage collected only when JVM needs memory. Because, it is softly referenced by ‘softA’ object.

Look at the below picture for more clarity.



One more use of SoftReference class is that you can retrieve back the object which has been softly referenced. It will be done by using **get()** method. This method returns reference to the object if object is not cleared from the memory. If object is cleared from the memory, it will return null.

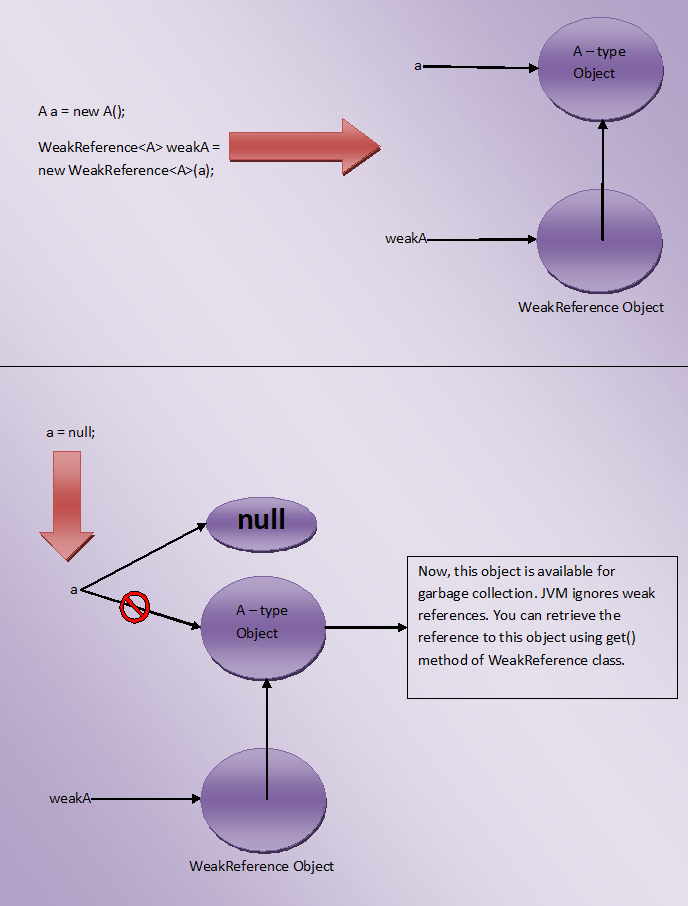
## **3) Weak References**

JVM ignores the **weak references**. That means objects which has only week references are eligible for garbage collection. They are likely to be garbage collected when JVM runs garbage collector thread. JVM doesn’t show any regard for weak references.

Below is the code which shows how to create weak references.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Weak Reference to A-type object to which 'a' is also pointing.            WeakReference<A> weakA = new WeakReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection.            a = weakA.get();    //You can retrieve back the object which has been weakly referenced.      }  } |

Look at the below picture for more clear understanding.



You may think that what is the use of creating weak references if they are ignored by the JVM, Use of weak reference is that you can retrieve back the weakly referenced object if it is not yet removed from the memory. This is done using get() method of WeakReference class. It will return reference to the object if object is not yet removed from the memory.

## **4) Phantom References**

The objects which are being referenced by **phantom references** are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called **‘reference queue’**. They are put in a reference queue after calling finalize() method on them. You can’t retrieve back the objects which are being phantom referenced. That means calling get() method on phantom reference always returns null.

Below example shows how to create Phantom References.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating ReferenceQueue            ReferenceQueue<A> refQueue = new ReferenceQueue<A>();            //Creating Phantom Reference to A-type object to which 'a' is also pointing            PhantomReference<A> phantomA = new PhantomReference<A>(a, refQueue);            a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection. But, this object is kept in 'refQueue' before removing it from the memory.            a = phantomA.get();    //it always returns null      }  } |

# 3. String:

# Introduction To Strings:

**String** represents sequence of characters enclosed within the double quotes.  **“abc”**, **“JAVA”**, **“123”**, **“A”** are some examples of strings. In many languages, strings are treated as character arrays. But In java, strings are treated as **objects**. To create and manipulate the strings, Java provides three classes.

**1) java.lang.String                     (From JDK 1.0)**

**2) java.lang.StringBuffer            (From JDK 1.5)**

**3) java.lang.StringBuilder           (From JDK 1.5)**

Let’s discuss some introductory points about these three classes.

1) All these three classes are members of **java.lang** package and they are final classes. That means you can’t create subclasses to these three classes.

2) All three classes implement **Serializable** and **CharSequence** interface.

3) **java.lang.String** objects are **immutable** in java. That is, once you create String objects, you can’t modify them. Whenever you try to modify the existing String object, a new String object is created with modifications. Existing object is not at all altered. Where as **java.lang.StringBuffer** and **java.lang.StringBuilder** objects are **mutable**. That means, you can perform modifications to existing objects.

4) Only **String** and **StringBuffer** objects are thread safe. **StringBuilder** objects are not thread safe. So whenever you want immutable and thread safe string objects, use java.lang.String class and whenever you want mutable as well as thread safe string objects then use java.lang.StringBuffer class.

5) In all three classes, **toString()** method is overrided. So. whenever you use reference variables of these three types, they will return contents of the objects not physical address of the objects.

6) **hashCode()** and **equals()** methods are overrided only in java.lang.String class but not in java.lang.StringBuffer and java.lang.StringBuilder classes.

7) There is no **reverse()** and **delete()** methods in String class. But, StringBuffer and StringBuilder have reverse() and delete() methods.

8) In case of String class, you can create the objects without **new** operator. But in case of StringBuffer and StringBuilder class, you have to use new operator to create the objects.

# Exploring java.lang.String Class : Constructors Of String Class:

we have seen that **java.lang.String** class is a final class. It’s objects are immutable. As they are immutable, they are also thread safety. In this article, we will discuss some of the methods and constructors of the String class which are used to create and manipulate the string objects.

## **Constructors Of java.lang.String Class :**

There are total **13 constructors** in java.lang.String class. It provides many ways to create the string objects. You can refer all constructors of String class [here](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html).

Here are some mostly used constructors of String class.

If you want to create an empty string object, then use **no-arg constructor** of String class.

|  |  |
| --- | --- |
| 1 | String s = new String();     //It creates a string object without characters in it. |

Below constructor takes **character array as an argument**.

|  |  |
| --- | --- |
| 1  2 | char[] chars = {'J', 'A', 'V', 'A'};     //Character Array  String s = new String(chars);    //Creating a String object by passing character array as an argument |

Below constructor takes **string as an argument**.

|  |  |
| --- | --- |
| 1 | String s = new String("JAVA");   //Creating a string object by passing string as an argument |

This constructor takes **StringBuffer type as an argument.**

|  |  |
| --- | --- |
| 1  2 | StringBuffer strBuff = new StringBuffer("abc");  String s = new String(strBuff);   //Creating a string object by passing StringBuffer type as an argument |

This constructor takes **StringBuilder type as an argument**.

|  |  |
| --- | --- |
| 1  2 | StringBuilder strBldr = new StringBuilder("abc");  String s = new String(strBldr);   //Creating a string object by passing StringBuilder type as an argument. |

## **String Literals :**

In Java, all string literals like “java”, “abc”, “123” are treated as objects of java.lang.String class. That means, all methods of String class are also applicable to string literals.  
You can also create the objects of String class without using new operator. This can be done by assigning a string literal to reference variable of type java.lang.String class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class StringExamples  {      public static void main(String[] args)      {          //Creating String objects without using new operator            String s1 = "abc";            String s2 = "abc"+"def";            String s3 = "123"+"A"+"B";            System.out.println(s1);     //Output : abc            System.out.println(s2);     //Output : abcdef            System.out.println(s3);     //Output : 123AB      }  } |

## **Finding The Length Of The String :**

length() method of String class is used to find the length of the string. The length of the string is the number of characters in it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class StringExamples  {      public static void main(String[] args)      {          String s = new String();                    //Creating an empty string object          System.out.println(s.length());            //Output : 0            char[] chars = {'J', 'A', 'V', 'A'};          String s1 = new String(chars);            //Creating string object of 4 characters          System.out.println(s1.length());         //Output : 4            String s2 = new String(s1+"J2EE");       //Creating string object of 8 characters          System.out.println(s2.length());        //Output : 8            //Using String.length() method on string literals            System.out.println("abc".length());        //Output : 3            System.out.println("123456".length());     //Output : 6            System.out.println("A".length());          //Output : 1      }  } |

# Exploring java.lang.String Class : Concatenation Of Strings:

## **Concatenation Of Strings :**

* The “**+**” operator is used to concatenate two or more string objects or string literals.

|  |  |
| --- | --- |
| 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 | public class StringExamples  {      public static void main(String[] args)      {          //Concatenating five string literals using "+" operator            String s = "Java"+"Concept"+"Of"+"The"+"Day";            System.out.println(s);       //Output : JavaConceptOfTheDay            //Creating five string objects            String s1 = new String("Java");            String s2 = new String("Concept");            String s3 = new String("Of");            String s4 = new String("The");            String s5 = new String("Day");            //Concatenating five string objects using "+" operator            System.out.println(s1+s2+s3+s4+s5);      //Output : JavaConceptOfTheDay      }  } |

* You can concatenate a string object with other data types like int, double, long, char etc using “+” operator. There is a one rule of “+” operator which states that  “**If any one operand of ‘+’ operator is a string, then it will be string concatenation otherwise it will be a normal addition”.**The same rule applies here also.

|  |  |
| --- | --- |
| 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 | public class StringExamples  {      public static void main(String[] args)      {          //Concatenating a string object with int type            int i = 1000;            String s = "Java"+i;            System.out.println(s);        //Output : Java1000            //Concatenating a string object with double type            double d = 523.69;            String s1 = "Java"+d;            System.out.println(s1);      //Output : Java523.69            //Concatenating a string object with char type            char c = 'A';            String s2 = "Java"+c;            System.out.println(s2);       //Output : JavaA            //Concatenating a string object with boolean type            boolean b = true;            String s3 = "Java"+b;            System.out.println(s3);      //Output : Javatrue      }  } |

* You can concatenate a string object not only with primitive types but also with the derived types. When you use derived type in the string concatenation, the string returned by the toString() method of that derived type is used.

|  |  |
| --- | --- |
| 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  47  48  49  50  51  52 | package strings;    class A  {      int i;        public A(int i)      {          this.i = i;      }        //toString() method is not overrided.      //So, it will return physical address of the object  }    class B  {      int i;        public B(int i)      {          this.i = i;      }        //Overriding toString() method        @Override      public String toString()      {          return "i = "+i;      }  }    public class StringExamples  {      public static void main(String[] args)      {          A a = new A(50);            String s = "Java";            //Concatenating a string object with A-type            System.out.println(s+a);      //Output : Javastrings.A@42719c            B b = new B(100);            //Concatenating string object with B-type            System.out.println(s+b);     //Output : Javai = 100      }  } |

* When you are adding two or more objects of different types using “+” operator, addition of the objects takes place from left to right. While adding, if any one operand is string then it will be string concatenation otherwise it will be normal addition.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class StringExamples  {      public static void main(String[] args)      {          int i = 5000;            double d = 6000.0006;            String s = "Java";            System.out.println(i+d+s);      //Output : 11000.0006Java            System.out.println(s+i+d);      //Output : Java50006000.0006            System.out.println(i+s+d);      //Output : 5000Java6000.0006      }  } |

* **Can we concatenate the string objects without using “+” operator?.**  
  **Yes**, we can concatenate string objects without using “+” operator. This can be done using **concat()** method of java.lang.String class. But using concat() method, we can concatenate only two string objects. It is not possible to concatenate more than two string objects using concat() method. And also using concat() method we can’t concatenate a string object with other type of object. Because, concat() method takes only String type as an argument.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class StringExamples  {      public static void main(String[] args)      {          String s1 = "JAVA";            String s2 = "J2EE";            System.out.println(s1.concat(s2));      //Output : JAVAJ2EE      }  } |

* Here is one special example of String concatenation. You can add two null objects referred by two String type reference variables (Like in Line 9 in the below example) , but you can’t add two hard coded null objects (Like in Line 15 in the below example). It gives compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class StringExamples  {      public static void main(String[] args)      {          String s1 = null;            String s2 = null;            System.out.println(s1+s2);       //Output : nullnull            System.out.println("null"+"null");    //Output : nullnull            System.out.println(s1+"JAVA"+s2);     //Output : nullJAVAnull        //  System.out.println(null+null);     //Compile Time Error      }  } |

# Exploring java.lang.String Class : Character Extraction:

**java.lang.String** class provides many methods to extract the characters from a string object. The characters in the string object are not stored like character array where each character is indexed. But, many string methods use indexes to address the characters in the string object. Like array, The index of string also starts from 0 (Zero).

Below are some methods which are used to extract characters from a string object.

## **1) charAt() Method :**

This method returns character at the specified index. Here is signature of this method.

**public char charAt(int index)**

Where index must be between 0 and length() – 1. This method will throw **StringIndexOutOfBoundsException if**index passed is negative or not less than the length of the string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class StringExamples  {      public static void main(String[] args)      {          String s = "Java Concept Of The Day";            System.out.println(s.charAt(5));      //Output : C            System.out.println(s.charAt(10));     //Output : p            System.out.println(s.charAt(25));     //This statement will throw StringIndexOutOfBoundsException      }  } |

## **2) getChars() Method**

This method copies the set of characters from the string into specified character array. Here is the signature of this method.

**public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**

This method copies characters of a string object starting from **‘srcBegin’** to **‘srcEnd’** into character array **‘dst’** at the index **‘dstBegin’**. This method will also throw **StringIndexOutOfBoundsException**if ‘srcBegin’ or ‘srcEnd’ are not between 0 and length() – 1 or if characters extracted does not fit into destination array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class StringExamples  {      public static void main(String[] args)      {          String s = "Java Concept Of The Day";            //Defining destination char array            char[] dst = new char[10];            //Copying the set of characters from s into dst.            s.getChars(5, 11, dst, 2);            for (char c : dst)          {              System.out.print(c);       //Output : --Concep--          }      }  } |

## **3) toCharArray() Method :**

This method converts whole string into a character array. Below is the signature of this method.

**public char[] toCharArray()**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class StringExamples  {      public static void main(String[] args)      {          String s = "Java Concept Of The Day";            //Converting string 's' into character array.            char[] dst = s.toCharArray();            for (char c : dst)          {              System.out.print(c);     //Output : Java Concept Of The Day          }      }  } |

## **4) subString() Method**

This method returns a sub string of the specified string. This method has two forms.

**public String substring(int beginIndex)** –> This form returns sub string starting from **‘beginIndex’** to the end of the specified string.

**public String substring(int beginIndex, int endIndex)** –> This form returns sub string starting from **‘beginIndex’** to **‘endIndex’** of the specified string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class StringExamples  {      public static void main(String[] args)      {          String s = "Java Concept Of The Day";            String subString1 = s.substring(11);            System.out.println(subString1);           //Output : t Of The Day            String subString2 = s.substring(5, 15);            System.out.println(subString2);         //Output : Concept Of      }  } |

# How The Strings Are Stored In The Memory?

In Java, strings are special. Java gives some special attention to string types that no other types enjoy such attention. For example, to create the string objects you need not to use ‘**new**‘ keyword. Where as to create other type of objects you have to use ‘new’ keyword. Like this, strings enjoy some special attention by the java. This attention is worth the while, because the strings are used almost everywhere while developing any kind of applications.

While storing the string objects in the memory also, they are specially treated by the Java. After reading this article, you will come to know how they are specially treated in the memory.

We all know that JVM divides the allocated memory to a Java program into two parts. one is **Stack** and another one is **heap**. Stack is used for execution purpose and heap is used for storage purpose. In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called **String Constant Pool**.

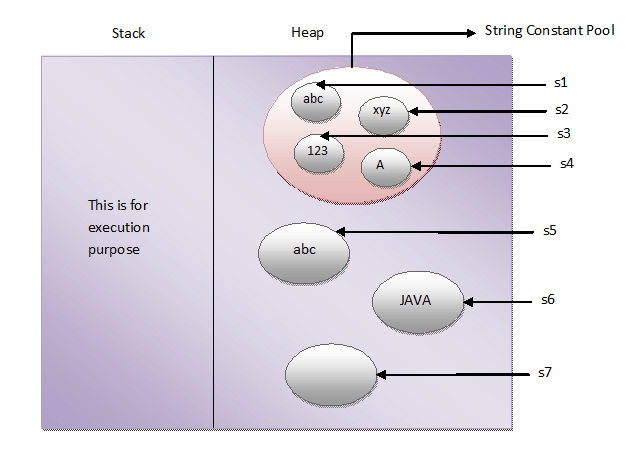
Whenever you create a string object using string literal, that object is stored in the **string constant pool** and whenever you create a string object using new keyword, such object is stored in the heap memory.

For example, when you create string objects like below, they will be stored in the String Constant Pool.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | String s1 = "abc";    String s2 = "xyz";    String s3 = "123";    String s4 = "A"; |

And when you create string objects using new keyword like below, they will be stored in the heap memory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | String s5 = new String("abc");    char[] c = {'J', 'A', 'V', 'A'};    String s6 = new String(c);    String s7 = new String(new StringBuffer()); |

This is how String Constant Pool looks like in the memory.  


One more interesting thing about String Constant Pool is that, **pool space is allocated to an object depending upon it’s content**. There will be no two objects in the pool having the same content.

This is what happens when you create string objects using string literal,

**“When you create a string object using string literal, JVM first checks the content of to be created object. If there exist an object in the pool with the same content, then it returns the reference of that object. It doesn’t create new object. If the content is different from the existing objects then only it creates new object.”**

But, when you create string objects using new keyword, a new object is created whether the content is same or not.

This can be proved by using “==” operator. As “==” operator returns true if two objects have same physical address in the memory otherwise it will return false. In the below example, s1 and s2 are created using string literal “abc”. So, s1 == s2 returns true. Where as s3 and s4 are created using new operator having the same content. But, s3 == s4 returns false.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class StringExamples  {      public static void main(String[] args)      {          //Creating string objects using literals            String s1 = "abc";            String s2 = "abc";            System.out.println(s1 == s2);        //Output : true            //Creating string objects using new operator            String s3 = new String("abc");            String s4 = new String("abc");            System.out.println(s3 == s4);        //Output : false      }  } |

**In simple words, there can not be two string objects with same content in the string constant pool. But, there can be two string objects with the same content in the heap memory.**

# An Example To Prove Strings Are Immutable:

One more interesting thing about String objects in java is that they are **immutable**. That means once you create a string object, you can’t modify the contents of that object. If you try to modify the contents of string object, a new string object is created with modified content.

In this article, We will discuss the examples which prove that strings are immutable.

## **An Example To Prove Strings Are Immutable :**

First, create one string object ‘s1’ using string literal “JAVA”.

|  |  |
| --- | --- |
| 1 | String s1 = "JAVA"; |

Create one more string object ‘s2’ using the same string literal “JAVA”.

|  |  |
| --- | --- |
| 1 | String s2 = "JAVA"; |

We have seen in the previous [article](https://javaconceptoftheday.com/how-the-strings-are-stored-in-the-memory/) that string objects created using string literal are stored in the **String Constant Pool** and any two objects in the pool can’t have same content. Here s1 and s2 are created using same literal. Therefore, they will be pointing to same object in the pool. Then s1 == s2 should return true.

|  |  |
| --- | --- |
| 1 | System.out.println(s1 == s2);       //Output : true |

Now, I want to make little modification to this object through ‘s1’ reference. I want to append “J2EE” at end of this string through ‘s1’. That can be done like below,

|  |  |
| --- | --- |
| 1 | s1 =s1 + "J2EE"; |

This statement appends “J2EE” to the object to which s1 is pointing and re-assigns reference of that object back to s1.

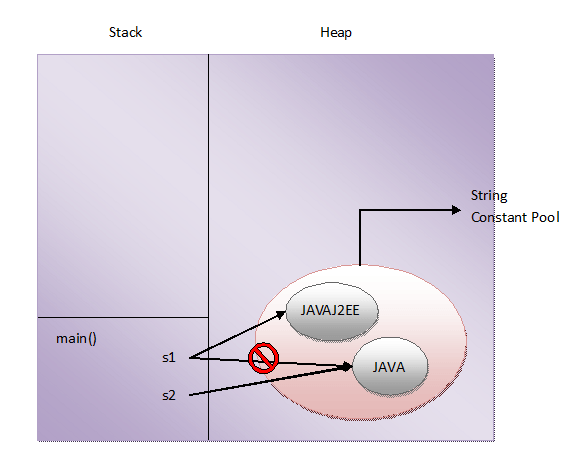
Now, compare physical address of s1 and s2 using “==” operator. This time it will return false.

|  |  |
| --- | --- |
| 1 | System.out.println(s1 == s2);       //Output : false |

That means now both s1 and s2 are pointing to two different objects in the pool. Before modifications they are pointing to same object. Once we tried to change the content of the object using ‘s1’, a new object is created in the pool with “JAVAJ2EE” as it’s content and it’s reference is assigned to s1. If the strings are mutable, both s1 and s2 should point to same object even after modification. That never happened here. That proves the string objects are immutable in java.

The whole program can be written like this,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class StringExamples  {      public static void main(String[] args)      {          String s1 = "JAVA";            String s2 = "JAVA";            System.out.println(s1 == s2);         //Output : true            s1 = s1 + "J2EE";            System.out.println(s1 == s2);         //Output : false      }  } |

This is how it looks like in the memory.  


## **is new String() also immutable?**

After seeing the above example, one more question may left in your mind. **Are string objects created using new operator also immutable? The answer is Yes**. String objects created using new operator are also immutable although they are stored in the heap memory. This can be also proved with help of an example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class StringExamples  {      public static void main(String[] args)      {          String s1 = new String("JAVA");            System.out.println(s1);         //Output : JAVA            s1.concat("J2EE");            System.out.println(s1);         //Output : JAVA      }  } |

In this example, a string object is created with “JAVA” as it’s content using new operator and it’s reference is assigned to s1. I have tried to change the contents of this object using concat() method. But, these changes are not reflected in the object as seen in Line 11. Even after the concatenation, content of the object is same as before. This is because the strings are immutable. Once I tried to concatenate “J2EE” to an existing string “JAVA”, a new string object is created with “JAVAJ2EE” as it’s content. But we don’t have reference to that object in this program.

## **Conclusion :**

**Immutability is the fundamental property of string objects. In whatever way you create the string objects, either using string literals or using new operator, they are immutable.**

# When To Use “==”, equals() And hashCode() On Strings:

**“==” operator**, **equals() method** and **hashcode() method**s are used to check the equality of any type of objects in Java. In this article, we will discuss which is the better way to check the equality of two string objects.

**“==” operator** compares the two objects on their physical address. That means if two references are pointing to same object in the memory, then comparing those two references using “==” operator will return true. For example, if s1 and s2 are two references pointing to same object in the memory, then invoking **s1 == s2** will return true. This type of comparison is called **“Shallow Comparison”**.

**equals() method**, if not overrided, will perform same comparison as “==” operator does i.e comparing the objects on their physical address. So, it is always recommended that you should override equals() method in your class so that it provides field by field comparison of two objects. This type of comparison is called **“Deep Comparison”**.

In java.lang.String class, equals() method is overrided to provide the comparison of two string objects based on their contents. That means, any two string objects having same content will be equal according to equals() method. For example, if s1 and s2 are two string objects having the same content, then invoking **s1.equals(s2)** will return true.

**hashCode() method** returns hash code value of an object in the Integer form. It is recommended that whenever you override equals() method, you should also override hashCode() method so that two equal objects according to equals() method must return same hash code values. This is the general contract between equals() and hashCode() methods that must be maintained all the time.

In java.lang.String class, hashCode() method is also overrided so that two equal string objects according to equals() method will return same hash code values. That means, if s1 and s2 are two equal string objects according to equals() method, then invoking **s1.hashCode() == s2.hashCode()** will return true.

Let’s apply these three methods on string objects and try to analyse their output.

Define two string objects like below,

|  |  |
| --- | --- |
| 1  2  3 | String s1 = "JAVA";    String s2 = "JAVA"; |

Now apply above methods on these two objects.

**s1 == s2** —> will return true as both are pointing to same object in the constant pool.  
**s1.equals(s2)** —> will also return true as both are referring to same object.  
**s1.hashCode() == s2.hashCode()** —> It also returns true.

This type of comparison is straight forward. There is no speculation about this comparison. Let’s define the string objects like below,

|  |  |
| --- | --- |
| 1  2  3 | String s1 = new String("JAVA");    String s2 = new String("JAVA"); |

**s1 == s2** —> will return false because s1 and s2 are referring to two different objects in the memory.  
**s1.equals(s2)** —> will return true as both the objects have same content.  
**s1.hashCode() == s2.hashCode()** —> It will also return true because two equals string objects according to equals() method will have same hash code values.

Comparing the string objects defined like below will also give same result as the above.

|  |  |
| --- | --- |
| 1  2  3 | String s1 = "JAVA";    String s2 = new String("JAVA"); |

**s1 == s2** —> will return false because s1 and s2 are referring to two different objects in the memory.  
**s1.equals(s2)** —> will return true as both the objects have same content.  
**s1.hashCode() == s2.hashCode()** —> It will also return true.

Now, you may conclude that If there is a requirement of comparing two string objects on their physical address, then use “==” operator and if there is a requirement of comparing two string objects on their contents, then use equals() method or hashCode() method.

Hold on…. Before jumping onto conclusion, compare these two string objects.

|  |  |
| --- | --- |
| 1  2  3 | String s1 = "0-42L";    String s2 = "0-43-"; |

**s1 == s2** —> will return false as s1 and s2 are referring to two different objects in the memory. **(Expected…)**  
**s1.equals(s2)** —> It will also return false as both the objects have different content. **(Expected…)**  
**s1.hashCode() == s2.hashCode()** —> It will return true. **(???….)**

This is because, **two unequal string objects according to equals() method may have same hash code values**. Therefore, it is recommended not to use hashCode() method to compare two string objects. You may not get expected result.

## **Conclusion :**

When you want to check the equality of two string objects on their physical existence in the memory, then use “==” operator. If you want to check the equality of two string objects depending upon their contents, then use equals() method. It is recommended not to use hashCode() method to check the equality of two string objects. You may get unexpected result.

# How StringBuffer And StringBuilder Differ From String Class:

String objects created using **java.lang.String** class are immutable. Once they are created, they can not be modified. If you try to modify them, a new string object will be created with modified content. This property of String class may cause some memory issues for applications which need frequent modification of string objects. To overcome this behavior of String class, two more classes are introduced in Java to represent the strings. They are **StringBuffer** and **StringBuilder**. Both these classes are also members of  **java.lang**package same as String class.

In this article, I have tried to figure out how these two classes differ from String class.

## **Immutability :**

This is main reason why StringBuffer and StringBuilder are introduced. As objects of String class are immutable, objects of StringBuffer and StringBuilder class are **mutable**. You can change the contents of StringBuffer and StringBuider objects at any time of execution. When you change the content, new objects are not created. Instead of that the changes are applied to existing object. Thus solving memory issues may caused by String class.

## **Object Creation :**

You have to use ‘**new**‘ operator to create objects to StringBuffer and StringBuilder classes. You can’t use string literals to create objects to these classes. For example, you can’t write **StringBuffer sb = “JAVA”** or **StringBuilder sb = “JAVA”**. It gives compile time error. But, you can use both string literals and new operator to create objects to String class.

## **Storage Area :**

As objects of StringBuffer and StringBuilder are created using only new operator, they are stored in **heap memory**. Where as objects of String class are created using both string literals and new operator, they are stored in string constant pool as well as heap memory.

## **Thread Safety :**

Any immutable object in java is thread safety. Because they are unchangeable once they are created. Any type of thread can’t change the content of immutable object. This applies to objects of String class also. Of the StringBuffer and StringBuilder objects, only StringBuffer objects are thread safety. All necessary methods in StringBuffer class are synchronized so that only one thread can enter into it’s object at any point of time. Where as StringBuilder objects are not thread safety.

## **Performance :**

Because of thread safety property of String and StringBuffer classes, they reduces the performance of multithreaded applications. Because, multiple threads can’t enter into objects of these classes simultaneously. One thread has to wait until another thread is finished with them. But, you will not find performance problems if you use StringBuilder class. Becuase, multiple threads can enter into objects of this class. But, be aware that StringBuilder is not thread safety.

## **String Concatenation :**

There will be serious performance issues when you are performing lots of string concatenation using String class. This is because, each time you perform string concatenation using string class, a new object will be created with the concatenated string. This slows down an application. But, if you use either StringBuffer or StringBuilder instead of String class, your application will perform better. Below program shows time taken by all three classes to perform string concatenation 10000 times.

|  |  |
| --- | --- |
| 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 | public class StringExamples  {      public static void main(String[] args)      {          String s = "JAVA";            long startTime = System.currentTimeMillis();            for(int i = 0; i <= 10000; i++)          {              s = s + "J2EE";          }            long endTime = System.currentTimeMillis();            System.out.println("Time taken by String class : "+(endTime - startTime)+" ms");            StringBuffer sb = new StringBuffer("JAVA");            startTime = System.currentTimeMillis();            for(int i = 0; i <= 10000; i++)          {              sb.append("J2EE");          }            endTime = System.currentTimeMillis();            System.out.println("Time taken by StringBuffer class : "+(endTime - startTime)+" ms");            StringBuilder sb1 = new StringBuilder("JAVA");            startTime = System.currentTimeMillis();            for(int i = 0; i <= 10000; i++)          {              sb1.append("J2EE");          }            endTime = System.currentTimeMillis();            System.out.println("Time taken by StringBuilder class : "+(endTime - startTime)+" ms");      }  } |

Output :  
Time taken by String class : 429 ms  
Time taken by StringBuffer class : 2 ms  
Time taken by StringBuilder class : 0 ms

Therefore, when you are performing lots of string concatenation in your application, it is better to use StringBuffer class (if you need thread safety) or StringBuilder class (If you don’t need thread safety).

## **equals() and hashCode() Methods :**

In StringBuffer and StringBuilder classes, equals() and hashCode methods are not overrided. Where as in String class they are overrided.

## **toString() Method :**

toString() method is overrided in all three classes. You can also convert StringBuffer and StringBuilder objects to String type by calling toString() method on them.

# What Is String Intern In Java?

String objects in java are stored in two places in memory. One is **String Constant Pool** and another one is **Heap Memory**. String objects created using string literals are stored in String Constant Pool where as string objects created using new operator are stored in heap memory.

## **Why You Need String Constant Pool? :**

String objects are most used objects in the development of any kind of applications. Therefore, there has to be a special arrangement to store these objects. String Constant Pool is one such special arrangement. In string constant pool, there will be no two objects with the same content. Heap memory can have any number of objects with same content.

Just imagine creating 1000 string objects with same content in heap memory and one string object with that content in String Constant Pool. Which one saves the memory?. which one will save the time?. Which one will be accessed faster?. It is, of course, String Constant Pool. That’s why you need String Constant Pool.

## **What Is String Intern? :**

**String intern** or simply **intern** refers to string object in the String Constant Pool. **Interning** is the process of creating a string object in String Constant Pool which will be exact copy of string object in heap memory.

## **intern() Method :**

**intern()**methodof java.lang.String class is used to perform interning i.e creating an exact copy of heap string object in string constant pool. When you call this method on a string object, first it checks whether there exist an object with the same content in the String Constant Pool. If object does not exist in the pool, it will create an object with the same content in the string constant pool and returns the reference of that object. If object exist in the pool than it returns reference of that object without creating a new object.

Look at the below example. Object ‘s1’ will be created in heap memory as we are using new operator to create it. When we call intern() method on s1, it creates a new string object in the string constant pool with “JAVA” as it’s content and assigns it’s reference to s2. So, **s1 == s2** will return false because they are two different objects in the memory and s1.equals(s2) will return true because they have same content.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class StringExamples  {      public static void main(String[] args)      {          String s1 = new String("JAVA");            String s2 = s1.intern();       //Creating String Intern            System.out.println(s1 == s2);       //Output : false            System.out.println(s1.equals(s2));    //Output : true      }  } |

Look at this example. Object s1 will be created in string constant pool as we are using string literal to create it and object s2 will be created in heap memory as we are using new operator to create it. When you call intern() method on s2, it returns reference of object to which s1 is pointing as it’s content is same as s2. It does not create a new object in the pool. So, **S1 == s3** will return true as both are pointing to same object in the pool.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class StringExamples  {      public static void main(String[] args)      {          String s1 = "JAVA";            String s2 = new String("JAVA");            String s3 = s2.intern();       //Creating String Intern            System.out.println(s1 == s3);       //Output : true      }  } |

## **String Literals Are Automatically Interned :**

When you call intern() on the string object created using string literals it returns reference of itself. Because, you can’t have two string objects in the pool with same content. That means string literals are automatically interned in java.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class StringExamples  {      public static void main(String[] args)      {          String s1 = "JAVA";            String s2 = s1.intern();       //Creating String Intern            System.out.println(s1 == s2);       //Output : true      }  } |

## **What is the use of interning the string?**

**To Save The memory Space :**

Using interned string, you can save the memory space. If you are using lots of string objects with same content in your code, than it is better to create an intern of that string in the pool. Use that intern string whenever you need it instead of creating a new object in the heap. It saves the memory space.

**For Faster Comparison :**

Assume that there are two string objects s1 and s2 in heap memory and you need to perform comparison of these two objects more often in your code. Then using s1.intern() == s2.intern() will be more fast then s1.equals(s2). Because, equals() method performs character by character comparison where as “==” operator just compares references of objects.

# 20 Things You Should Know About Strings In Java:

**Strings in Java** are most used data types while developing any kind of applications. Hence, strings are treated as very special in Java. This article contains 20 important points about strings in Java. These points are also most discussed ones in the Java interviews.

**1)** In Java, you can create string objects in two ways. One is using **new** operator and another one is using string **literals**.

|  |  |
| --- | --- |
| 1  2  3 | String s1 = "abc";        //Creating string object using string literal    String s2 = new String("abc");          //Creating string object using new operator |

**2)** String objects created using string literals are stored in **String Constant Pool** and string objects created using new operator are stored in the **heap memory**. Click [here](https://javaconceptoftheday.com/how-the-strings-are-stored-in-the-memory/) to see how strings are stored in the memory.

**3)** What Is String Constant Pool?

String objects are most used data objects in Java. Hence, Java has a special arrangement to store the string objects. String Constant Pool is one such arrangement. String Constant Pool is the memory space in the heap memory specially allocated to store the string objects created using string literals. In String Constant Pool, there will be no two string objects having the same content.

Whenever you create a string object using string literal, JVM first checks the content of the object to be created. If there exist an object in the string constant pool with the same content, then it returns the reference of that object. It doesn’t create a new object. If the content is different from the existing objects then only it creates new object.

**4)** String is a **derived type**, not a primitive type like int, double etc. Strings are objects in Java.

**5)** String objects in Java are **immutable**. That means, once you create String objects, you can’t modify them. If you try to modify them, a new object will be created with modifications.

**6)** To overcome the immutability of String objects, two more classes are introduced in Java. They are **StringBuffer** and **StringBuilder** classes. Objects of StringBuffer and StringBuilder class are mutable.

**7)** All three classes – String, StringBuffer and StringBuilder are **final**. That means you can’t extend them. All three classes are members of **java.lang** package.

**8)** In all three classes – String, StringBuffer and StringBuilder, **toString()** method is overridden. That means, whenever you use references to objects of these classes, actual content of those objects will be retrieved.

**9)** **equals()** and **hashCode()** methods are overridden in String class but they are not overridden in StringBuffer and StringBuilder classes.

**10)** String and StringBuffer objects are **thread safety** where as StringBuilder objects are not thread safety.

**11)** Using “**==**“, **equals()** and **hashCode()** on String objects.

All three – “==”, equals() and hashCode() are used to check the equality of two string objects. If you want to check the equality of two string objects based on their physical address, then use “==” operator. If you want to check the equality of two string objects based on their content, then use equals() method. It is recommended not to use hashCode() method to compare the string objects. You may get unexpected results. Click [here](https://javaconceptoftheday.com/when-to-use-equals-hashcode-on-strings/) to see when to use “==”, equals() and hashcode() on strings.

**12)** Strings in Java are backed by **character array**. You can retrieve this array using toCharArray() method of String class.

**13)** If you are performing lots of string concatenation in your code, then use either StringBuffer or StringBuilder classes. These two classes give better performance than String class. Click [here](https://javaconceptoftheday.com/stringbuffer-stringbuilder-string-class/) to see the differences between String, StringBuffer and StringBuilder classes.

**14)** Java doesn’t support operator overloading except ‘**+**‘ operator. ‘**+**‘ can be used for number addition as well as to concatenate two string objects. This is the special treatment given by the Java to string objects.

**15)** Java provides 4 methods to compare the strings.

1) **equals()** – This method returns true if contents of two string objects are same.  
2) **equalsIgnoreCase()** – This method compares two string objects but ignores the case of the characters when comparing.  
3) **compareTo()** – This method compares one string with another and returns an integer if the string is smaller or equal or greater than the other string.  
4) **compareToIgnoreCase()** – This method is same as compareTo() but ignores the case of the characters when comparing.

**16)** You need not to create objects to access the String class methods. You can do so using **string literals** also. Look at the below example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class MainClass  {      public static void main(String[] args)      {          System.out.println("abc".charAt(0));          //Output : a            System.out.println("abc".equalsIgnoreCase("ABC"));      //Output : true            System.out.println("abc".compareTo("abc"));         //Output : 0            System.out.println("abc".indexOf('c'));        //Output : 2      }  } |

**17)** What Is String Intern?

String object in the string constant pool is called as **String Intern**. You can create an exact copy of heap memory string object in the string constant pool. This process of creating an exact copy of heap memory string object in string constant pool is called **interning**. **intern()** method is used for interning. Click [here](https://javaconceptoftheday.com/string-intern-java/) to see more about string intern in Java.

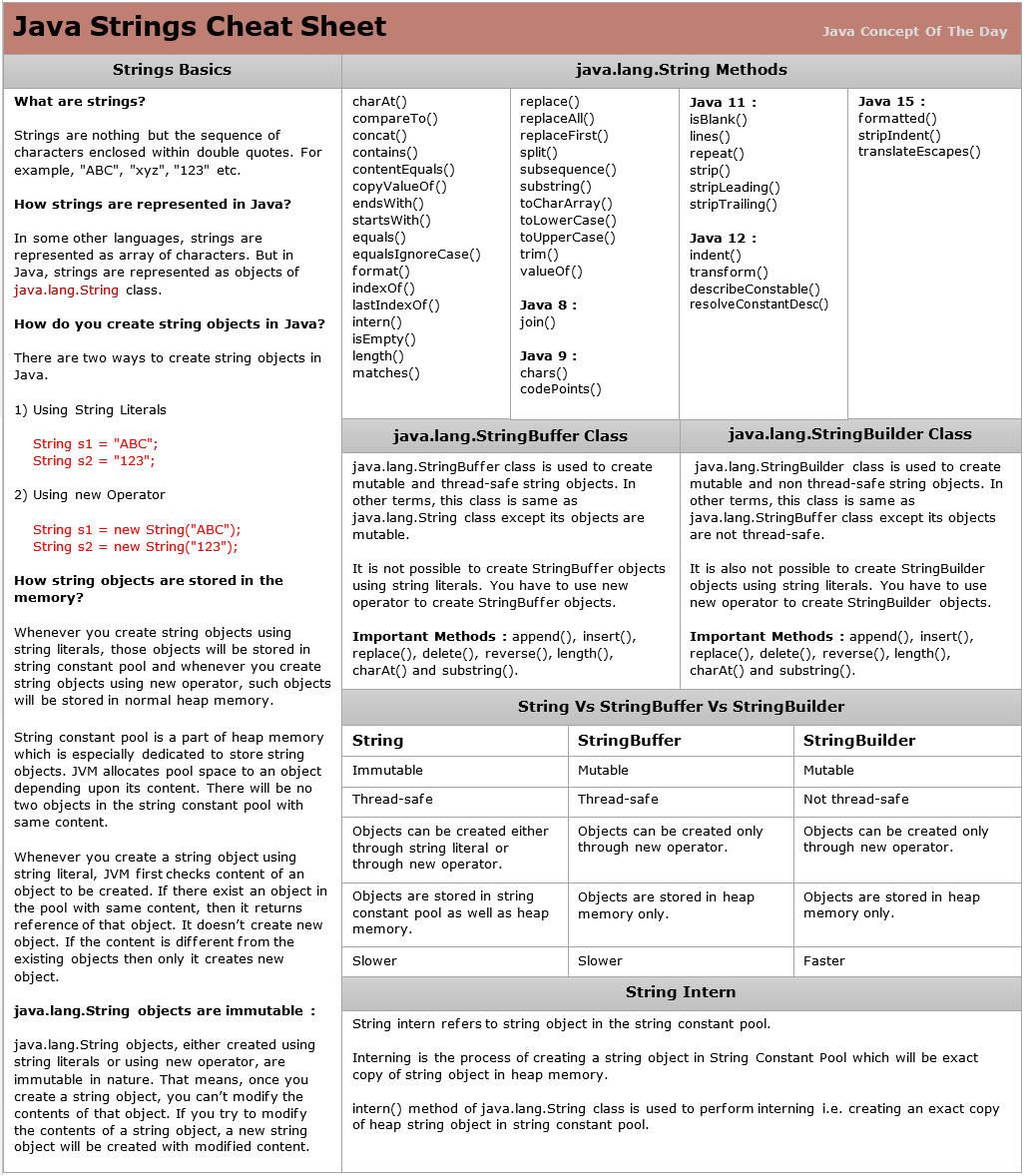
**18)** **indexOf(),** **lastIndexOf()** and **matches(String regex)** are the methods to perform search within a string.

**19)** Unlike in C and C++, Strings in Java are not terminated with **null** character. Strings are treated as objects in Java.

**20)** Java provides lots of in built methods to manipulate the string objects. click [here](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) to see the documentation of String class

# Java Strings Cheat Sheet:

Below is the Java strings cheat sheet. You can use it as quick reference guide for Java strings while preparing for the Java interviews.

[](https://i0.wp.com/javaconceptoftheday.com/wp-content/uploads/2023/06/Java_Strings_Cheat_Sheet.png?ssl=1)

#### **Strings Basics :**

**What are strings?**

Strings are nothing but the sequence of characters enclosed within double quotes. For example, “ABC”, “xyz”, “123” etc.

**How strings are represented in Java?**

In some other languages, strings are represented as array of characters. But in Java, strings are represented as objects of java.lang.String class.

**How do you create string objects in Java?**

There are two ways to create string objects in Java.

1) Using String Literals

String s1 = “ABC”;

String s2 = “123”;

2) Using new Operator

String s1 = new String(“ABC”);

String s2 = new String(“123”);

**How string objects are stored in the memory?**

Whenever you create string objects using string literals, those objects will be stored in string constant pool and whenever you create string objects using new operator, such objects will be stored in normal heap memory.

String constant pool is a part of heap memory which is specially dedicated to store string objects. JVM allocates pool space to an object depending upon it’s content. There will be no two objects in the string constant pool with same content.

Whenever you create a string object using string literal, JVM first checks content of an object to be created. If there exist an object in the pool with same content, then it returns reference of that object. It doesn’t create new object. If the content is different from the existing objects then only it creates new object.

**String objects are immutable :**

String objects, either created using string literals or using new operator, are immutable in nature. That means, once you create a string object, you can’t modify the contents of that object. If you try to modify the contents of a string object, a new string object is created with modified content.

#### **java.lang.StringBuffer Class :**

java.lang.StringBuffer class is used to create mutable and thread-safe string objects. In other terms, this class is same as java.lang.String class except it’s objects are mutable.

It is not possible to create StringBuffer objects using string literals. You have to use new operator to create StringBuffer objects.

**Important Methods :** append(), insert(), replace(), delete(), reverse(), length(), charAt() and substring().

#### **java.lang.StringBuilder Class :**

java.lang.StringBuilder class is used to create mutable and non thread-safe string objects. In other terms, this class is same as java.lang.StringBuffer class except it’s objects are not thread-safe.

It is also not possible to create StringBuilder objects using string literals. You have to use new operator to create StringBuilder objects.

**Important Methods :** append(), insert(), replace(), delete(), reverse(), length(), charAt() and substring().

#### **Difference Between String, StringBuffer and StringBuilder :**

|  |  |  |
| --- | --- | --- |
| **String** | **StringBuffer** | **StringBuilder** |
| Immutable | Mutable | Mutable |
| Thread-safe | Thread-safe | Not thread-safe |
| Objects can be created either through string literal or through new operator | Objects can be created only through new operator | Objects can be created only through new operator |
| Objects are stored in string constant pool as well as heap memory | Objects are stored in heap memory only. | Objects are stored in heap memory only. |
| Slower | Slower | Faster |

#### **String Intern :**

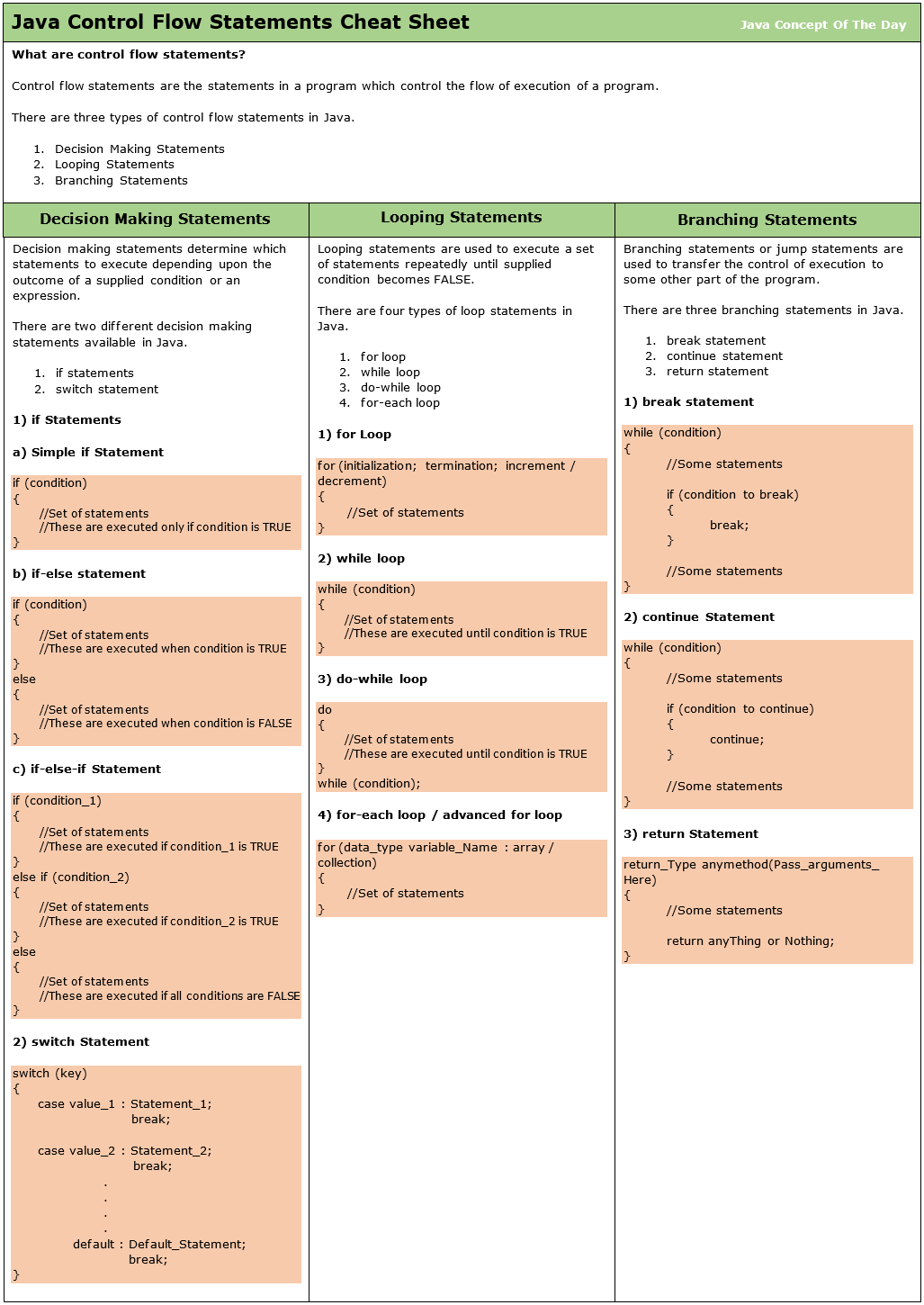
String intern refers to string object in the string constant pool.

**Interning** is the process of creating a string object in String Constant Pool which will be exact copy of string object in heap memory.

**intern()**methodof java.lang.String class is used to perform interning i.e creating an exact copy of heap string object in string constant pool.

# 4. Java Control Flow Statements Cheat Sheet:

Here is the quick reference guide for control flow statements in Java. Download the Java control flow statements cheat sheet below and refer it whenever required.

[](https://i0.wp.com/javaconceptoftheday.com/wp-content/uploads/2023/07/Java_Control_Flow_Statements_Cheat_Sheet.png?ssl=1)

### **What are control flow statements?**

Control flow statements are the statements in a program which control the flow of execution of a program.

There are three types of control flow statements in Java.

1. Decision Making Statements
2. Looping Statements
3. Branching Statements

### **Decision Making Statements :**

Decision making statements determine which statements to execute depending upon the outcome of a supplied condition or an expression.

There are two different decision making statements available in Java.

1. if statements
2. switch statement

**1) if Statements**

There are three different versions of if statements available in Java.

**a) Simple if Statement**

Simple if statement is the most basic of all control flow statements. It is used when you want a certain block of code or set of statements are to be executed only when a supplied condition evaluates to TRUE.

|  |  |
| --- | --- |
| 1  2  3  4 | if (condition)  {      //Set of statements : These are executed only if condition is TRUE  } |

**b) if-else statement**

It is an extension to simple if statement. It has two blocks. One is if-block and another one is else-block. if-block is executed if condition evaluates to TRUE and else-block is executed if condition is evaluated to FALSE.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | if (condition)  {      //Set of statements : These are executed when condition is TRUE  }  else  {      //Set of statements : These are executed when condition is FALSE  } |

**c) if-else-if Statement**

if-else-if statement is used when more than one conditions are to be evaluated. If a particular condition evaluates to TRUE, block associated with that condition will be executed. If all the conditions are evaluated to FALSE, else-block will be executed at the end.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | if (condition\_1)  {      //Set of statements : These are executed if condition\_1 is TRUE  }  else if (condition\_2)  {      //Set of statements : These are executed if condition\_2 is TRUE  }  else  {      //Set of statements : These are executed if all conditions are FALSE  } |

**2) Switch Statement**

if statements evaluate only boolean expressions whose possible outcomes are either TRUE or FALSE. switch statement can be used to evaluate expressions who have multiple outcomes and of different types like char, byte, short, int, enum types, string literals and some wrapper classes (Character, Byte, Short and Integer).

switch statement is used when you have multiple outcomes of an expression. The result of an expression is matched with case label and statements associated with that matching case label are executed. Thus, switch statement will have multiple possible execution paths depending upon supplied key or the outcome of an expression.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | switch (key)  {      case value\_1 : Statement\_1;                     break;        case value\_2 : Statement\_2;                     break;                   .                   .                   .                   .      default :      Default\_Statement;                     break;  }; |

### **Looping Statements :**

Looping statements are used to execute a set of statements repeatedly until supplied condition becomes FALSE.

There are four types of loop statements in Java.

1. for loop
2. while loop
3. do-while loop
4. for-each loop

**1) for Loop**

for statement or for loop provides way to specify initialization, termination condition and increment/decrement in a single statement. It is often used to iterate over an array or a collection.

|  |  |
| --- | --- |
| 1  2  3  4 | for (initialization; termination; increment/decrement)  {         //Set Of Statements  } |

**2) while loop**

In while loop, a set of statements are executed repeatedly until condition is TRUE. If the condition becomes FALSE, control goes out of while loop. Use this loop when you don’t know exact number of iterations.

This loop is entry controlled loop as condition is checked while entering the loop.

|  |  |
| --- | --- |
| 1  2  3  4 | while (condition)  {      //Set of statements : These are executed until condition is TRUE  } |

**3) do-while loop**

It is same as while loop, but here condition is checked while exiting the loop. Hence, this loop is exit controlled loop.

|  |  |
| --- | --- |
| 1  2  3  4  5 | do  {      //Set of statements : These are executed until condition is TRUE  }  while (condition); |

**4) for-each loop or advanced for loop**

It is used to iterate all the elements of an array or a collection at a time. You can’t iterate only few elements of a collection/array using this loop. This is the main difference between basic for loop and advanced for loop.

|  |  |
| --- | --- |
| 1  2  3  4 | for (data\_type variable\_Name : array/collection)  {      //Set of statements  } |

### **Branching Statements :**

Branching statements or jump statements are used to transfer the control of execution to some other part of the program.

There are three branching statements in Java.

1. break statement
2. continue statement
3. return statement

**1) break statement**

break statement is used to break the current loop and transfer the control of execution to next statement after the loop. It is used in loop and switch statement.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | while (condition)  {      //Some statements        if (condition to break)      {          break;      }        //Some statements  } |

**2) continue Statement**

continue statement is used to skip the current iteration of a loop and jump to next iteration.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | while (condition)  {      //Some statements        if (condition to continue)      {          continue;      }        //Some statements  } |

**3) return Statement**

return statement is used to exit the current method and return the control of execution to where the method is called.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | return\_Type anymethod(Pass\_arguments\_Here)  {      //Some statements        return anyThing or Nothing;  } |

# 5. for-each Loop In Java (Enhanced for Loop):

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  2  3  4 | for(Data\_Type variable : array or collection)  {    } |

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  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class ForEachLoop  {      public static void main(String[] args)      {          //An array of strings          String[] str = {"First", "Second", "Third", "Fourth", "Fifth"};            //iterating every element of str using for-each loop          for (String s : str)          {              System.out.println(s);          }      }  } |

**Output :**  
First  
Second  
Third  
Fourth  
Fifth

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ForEachLoop  {      public static void main(String[] args)      {          //An ArrayList of strings          ArrayList<String> list = new ArrayList<String>();            //Adding elements to ArrayList          list.add("First");          list.add("Second");          list.add("Third");          list.add("Fourth");            //iterating every element of list using for-each loop          for (String s : list)          {              System.out.println(s);          }      }  } |

**Output :**  
First  
Second  
Third  
Fourth

## **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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class ForEachLoop  {      public static void main(String[] args)      {          //Two Dimensional Array          int[][] twoDArray = { {1, 2, 3, 4}, {5, 6, 7, 8} };            //iterating every element of twoDArray using for-each loop          for (int[] oneDArray : twoDArray)          {              System.out.print("[");                //iterating every element of oneDArray using for-each loop              for (int i : oneDArray)              {                  System.out.print(i+"\t");              }              System.out.println("]");          }      }  } |

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

## **Advantages of for-each loop :**

* You need not to specify the initialization, condition and increment or decrement as you specify in the normal for loop.
* It increases the readability of the code.

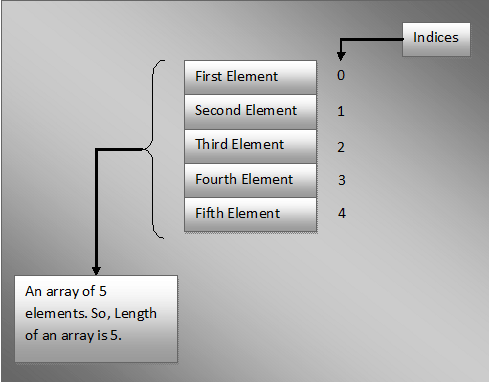
## **Disadvantages Of for-each loop :**

* You can’t iterate only few elements of an array or collection using for-each loop.

# 6. Arrays:

# Arrays In Java:

Array is a set of values where each value is identified by an index. You can make an array of int’s, double’s, boolean’s or any other types but all the values of array must be of same type. The index of an array starts from 0. The following diagram shows how the array elements are stored in an array object.



Today we will discuss about declaring an array variable, creating an array object, initializing and accessing the arrays elements in java.

## **Declaring Arrays In Java :**

In java, you can declare arrays in two ways. Those two ways of declaring arrays in java are,

**Data\_Type[] Variable\_Name;**

AND

**Data\_Type Variable\_Name[];**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] arrayOfInts;    //Declaring an array of ints            double arrayOfDoubles[];   //Declaring an array of doubles            char[] arrayOfChars;     //Declaring an array of characters            boolean arrayOfBooleans[];   //Declaring an array of booleans      }  } |

Note : As both styles of declaring arrays in java are valid but the style **Data\_Type[] Variable\_Name** is preferred. The style **Data\_Type Variable\_Name[]** comes from C/C++ and it is included in java to accommodate C/C++ programmers.

## **Instantiating An Array Object :**

You can instantiate or create an array object using new operator. The syntax for instantiating arrays in java is,

**Variable\_Name = new Data\_Type[ArraySize];**

For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] arrayOfInts;      //Declaring an array of int            arrayOfInts = new int[10];   //Instiantiating an array of int using new operator       }  } |

**Note :** When you create an array object of specified type and length with new operator, by default all array elements in the array object are initialized with default values.

## **Initializing Array Elements :**

The syntax for initializing array elements is,

**Variable\_Name[index] = Value;**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] arrayOfInts;      //Declaring an array of int            arrayOfInts = new int[10];   //Instantiating an array of int using new operator            arrayOfInts[2] = 12;         //Initializing 3rd element            arrayOfInts[5] = 56;         //Initializing 6th element      }  } |

Note : Uninitialized array elements will get default values.

## **Combining declaration, instantiate and initialization into one statement :**

You can combine declaration, instantiate and initialization in one statement like below,

**Data\_Type[] Variable\_Name = new Data\_Type[] {Value0, Value1, Value2, Value3 ….. };**

You should not mention array size here. If you mention array size, you will get compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ArraysInJava  {      public static void main(String[] args)      {          //Declaring, instantiating and Initializing an array in one statement            double[] arrayOfDoubles = new double[] {12.56, 45.87, 14.85};            //This is also ok.            int[] arrayOfInts = {12, 21, 0, 5, 7};      }  } |

## **Accessing Array Elements :**

Array elements are accessed through index of that element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ArraysInJava  {      public static void main(String[] args)      {          //Declaring and Initializing an array in one statement            int[] arrayOfInts = {12, 21, 0, 5, 7};            System.out.println(arrayOfInts[0]);   //accessing 1st element            System.out.println(arrayOfInts[3]);   //accessing 4th element            System.out.println(arrayOfInts[2]);   //accessing 3rd element      }  } |

# How Are the Arrays stored in the memory?

Internally, Arrays are like objects. That’s why they are created using new operator. Array objects can hold two types of data.

1) primitive type of data

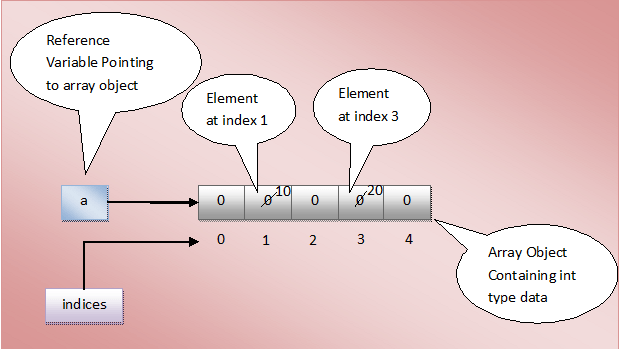
2) References to derived type of data.

Let’s discuss how the memory is allocated to an array containing primitive data or references to other objects.

## **Array Containing Primitive Type Of Data :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a;           //Declaring int type array reference variable which will be referring to int type array object            a = new int[5];    //Creating an int type Array Object containing 5 elements of int type            a[1] = 10;         //Changing Value of element at index 1            a[3] = 20;         //Changing Value of element at index 3      }  } |

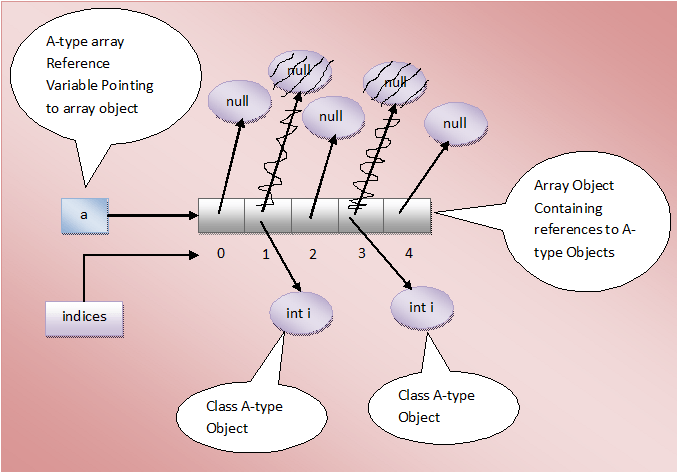
In the above example, First statement of main method (Line 5) declares an array reference variable of int type. In the next statement, an array object containing 5 elements of int type is created and assigned to already declared array reference variable. Initially, all elements of array are initialized with default values (i.e 0 in this example). In the next statement (Line 9), Value of element at index 1 is changed to 10. In the next statement, value of element at index 3 is changed to 20. This can be well explained with the diagram like below.



## **Array Containing References To Derived Type Of Data :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class A  {      int i;  }    public class ArraysInJava  {      public static void main(String[] args)      {          A[] a;           //Declaring array reference variable of A-type which will be referring to an array object containing references to A-type objects            a = new A[5];    //Creating an Array Object containing 5 references to A-type objects            a[1] = new A();         //Creating an object of A-type and assigning it to element at index 1            a[3] = new A();         //Creating an object of A-type and assigning it to element at index 3      }  } |

In the above example, first statement of main method (Line 10) declares an array reference variable of class A-type. This array reference variable must refer to an array object holding references to class A-type objects. In the next statement, one array object which can hold 5 references to class A-type objects is created. Please remember that, in this statement only an array object is created not 5 objects of class A-type are created . Initially, all elements of array will be pointing to null. In the next two statements (Line 14 and 16), two objects of class A-type are created and assigned to elements at index 1 and 3 respectively. It can be represented using diagram like below.

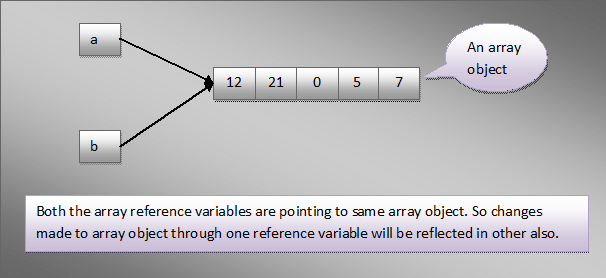


# Copying An Array In Java:

Consider the following program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a = {12, 21, 0, 5, 7};   //Declaring and initializing an array of ints            int[] b = a;             //copying array 'a' to array 'b'            //Printing elements of array 'b'            for (int i = 0; i < b.length; i++)          {              System.out.println(b[i]);          }            a[2] = 56;     //Changing value of 3rd element of array 'a'            System.out.println(b[2]);    //value of 3rd element of array 'b' also changes            b[4] = 100;     //Changing value of 5th element of array 'b'            System.out.println(a[4]);     //value of 5th element of array 'a' also changes      }  } |

In the above example, we are defining an array of ints. We are copying that array to another array of ints. You can notice that changing the value of one array will be reflected in another array also. That means, both the array variables(a and b) are pointing to same object in the memory. This can be illustrated using diagram as below,



If you copy an array using above method, changes made in one array will be reflected in other. Normally, you don’t want such behaviour in real time. You may need two different array objects with same set of elements. In such scenarios, Instead of using the above method, use any one of the following methods.

## **1) Copying An Array Using for Loop :**

The following example shows how to copy an array using for-loop.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a = {12, 21, 0, 5, 7};   //Declaring and initializing an array of ints            int[] b = new int[a.length];             //Declaring and instantiating another array of ints with same length            for (int i = 0; i < a.length; i++)          {              b[i] = a[i];          }            //Now changing values of one array will not reflect in another array            a[2] = 56;       //Changing value of 3rd element in array 'a'            System.out.println(b[2]);    //value of 3rd element in array 'b' will not change            b[4] = 100;     //Changing value of 5th element in array 'b'            System.out.println(a[4]);    //value of 5th element in array 'a' will not change      }  } |

## **2) Copying An Array Using copyOf() Method of java.util.Array Class :**

|  |  |
| --- | --- |
| 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 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a = {12, 21, 0, 5, 7};   //Declaring and initializing an array of ints            //creating a copy of array 'a' using copyOf() method of java.util.Arrays class            int[] b = Arrays.copyOf(a, a.length);            //Printing elements of array 'b'            for (int i = 0; i < b.length; i++)          {              System.out.println(b[i]);          }            //Now changing values of one array will not reflect in other array            a[2] = 56;       //Changing value of 3rd element in array 'a'            System.out.println(b[2]);    //value of 3rd element in array 'b' will not change            b[4] = 100;     //Changing value of 5th element in array 'b'            System.out.println(a[4]);    //value of 5th element in array 'a' will not change      }  } |

## **3) Copying An Array Using clone() Method :**

All arrays will have clone() method inherited from java.lang.Object class. Using this method, you can copy an array.

|  |  |
| --- | --- |
| 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 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a = {12, 21, 0, 5, 7};   //Declaring and initializing an array of ints            //creating a copy of array 'a' using clone() method            int[] b = a.clone();            //Printing elements of array 'b'            for (int i = 0; i < b.length; i++)          {              System.out.println(b[i]);          }            //Now changing values of one array will not reflect in other array            a[2] = 56;       //Changing value of 3rd element in array 'a'            System.out.println(b[2]);    //value of 3rd element in array 'b' will not change            b[4] = 100;     //Changing value of 5th element in array 'b'            System.out.println(a[4]);    //value of 5th element in array 'a' will not change      }  } |

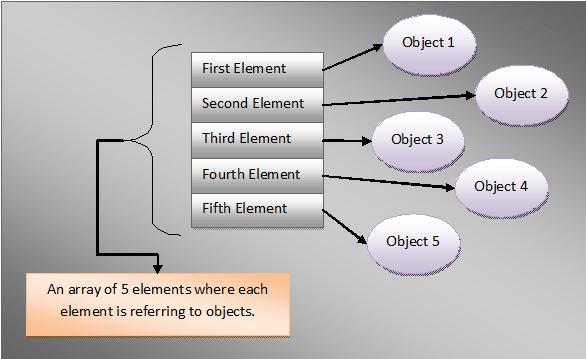
## **4) Copying An Array Using arraycopy() Method Of System Class :**

|  |  |
| --- | --- |
| 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 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a = {12, 21, 0, 5, 7};   //Declaring and initializing an array of ints            //Creating an array object of same length as array 'a'            int[] b = new int[a.length];            //creating a copy of array 'a' using arraycopy() method of System class            System.arraycopy(a, 0, b, 0, a.length);            //Printing elements of array 'b'            for (int i = 0; i < b.length; i++)          {              System.out.println(b[i]);          }            //Now changing values of one array will not reflect in other array            a[2] = 56;       //Changing value of 3rd element in array 'a'            System.out.println(b[2]);    //value of 3rd element in array 'b' will not change            b[4] = 100;     //Changing value of 5th element in array 'b'            System.out.println(a[4]);    //value of 5th element in array 'a' will not change      }  } |

**Note :** Using arraycopy() method of System class, you can copy a part of an array into another array.

# Array Of Objects In Java:

Array can hold the references to any type of objects. It is important to note that array can contain only references to the objects, not the objects itself. This can be well explained with the diagram.



For example, In the below example ‘**arrayOfUserDefinedTypeObjects**‘ is an array containing only reference variables of type ‘**UserDefinedType**‘ type but not objects of ‘UserDefinedType’.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class UserDefinedType  {      int i;        void methodOne()      {          System.out.println("From User Defined Type Object");      }  }    public class ArraysInJava  {      public static void main(String[] args)      {          UserDefinedType[] arrayOfUserDefinedTypeObjects = new UserDefinedType[10];            arrayOfUserDefinedTypeObjects[1].methodOne();    //This statement throws NullPointerException            //because array elements are not initialized      }  } |

The statement **UserDefinedType[] arrayOfUserDefinedTypeObjects = new UserDefinedType[10];** creates an array which can hold references to 10 objects of ‘UserDefinedType’. This statement does not create 10 objects of ‘UserDefinedType’. You have to explicitly create the objects and assign to each element of the array. Otherwise, It gives NullPointerException at run time like in the above example.

To avoid the exception, create 10 objects and assign them to each element of array.

|  |  |
| --- | --- |
| 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 | class UserDefinedType  {      int i;        void methodOne()      {          System.out.println("From User Defined Type Object");      }  }    public class ArraysInJava  {      public static void main(String[] args)      {          UserDefinedType[] arrayOfUserDefinedTypeObjects = new UserDefinedType[10];            for (int i = 0; i < arrayOfUserDefinedTypeObjects.length; i++)          {              //Creating an object of 'UserDefinedType' and              //assigning it to an element of array                arrayOfUserDefinedTypeObjects[i] = new UserDefinedType();          }            arrayOfUserDefinedTypeObjects[1].methodOne();   //accessing member of 'UserDefinedType'            //Now, above statement does not throw NullPointerException      }  } |

An array containing super class reference variables can point to sub class objects. For example,

|  |  |
| --- | --- |
| 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 | class A  {      int i;        void methodOne()      {          System.out.println("From Class A");      }  }    class B extends A  {      @Override      void methodOne()      {          System.out.println("From Class B");      }  }    public class ArraysInJava  {      public static void main(String[] args)      {          A[] arrayOfSuperClassReferences = new A[5];   //Defining an array of super class reference variables            for (int i = 0; i < arrayOfSuperClassReferences.length; i++)          {              arrayOfSuperClassReferences[i] = new B();     //array element pointing to sub class object                arrayOfSuperClassReferences[i].methodOne();   //accessing member of sub class object          }      }  } |

# Arrays As Parameters And Return Types Of A Method:

Arrays can be passed to method as arguments and methods can return an array. Arrays are Passed-By-Reference. That means, When an array is passed to a method, reference of an array object is passed not the copy of the object. So, Any changes made to object in the method will be reflected in the actual object.

Let’s see some of examples where arrays are parameters and return types of a method.

## **Passing Array Of Primitive Type As A Parameter to The Method :**

|  |  |
| --- | --- |
| 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 | public class ArraysInJava  {      public static void main(String[] args)      {          //declaring and creating An array object of double type containing 8 elements            double[] d = new double[8];            d[2] = 10.25;   //Changing the value of element at index 2            d[5] = 15.35;   //Changing the value of element at index 5            methodOne(d);   //Passing array to methodOne()            //Changes made to array object in the method are reflected in the actual object            System.out.println(d[2]);     //Output : 25.32            System.out.println(d[5]);     //Output : 45.89      }        static void methodOne(double[] d)      {          d[2] = 25.32;   //Changing the value of element at index 2            d[5] = 45.89;   //Changing the value of element at index 5      }  } |

## **Passing Array Of Derived Type As A Parameter to The Method :**

|  |  |
| --- | --- |
| 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 | class A  {      int i;  }    public class ArraysInJava  {      public static void main(String[] args)      {          //declaring and creating An array object of A-type containing 8 references to A-type objects            A[] a = new A[8];            a[2] = new A();   //creating A-type object and assigning it to element at index 2            a[5] = new A();   //creating A-type object and assigning it to element at index 5            a[2].i = 10;     //Changing value of field of A-type object referred by element at index 2            a[5].i = 20;     //Changing value of field of A-type object referred by element at index 5            methodOne(a);   //Passing array to methodOne()            //Changes made to array object in the method are reflected here            System.out.println(a[2].i);     //Output : 30            System.out.println(a[5].i);     //Output : 40      }        static void methodOne(A[] a)      {          a[2].i = 30;   //Changing value of field of A-type object referred by element at index 2            a[5].i = 40;   //Changing value of field of A-type object referred by element at index 5      }  } |

## **Method Returning An Array Of Primitive Type :**

|  |  |
| --- | --- |
| 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 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a = methodOne();            System.out.println(a[2]);     //Output : 10            System.out.println(a[5]);     //Output : 20      }        //Method returns an array of int type        static int[] methodOne()      {          //declaring and creating An array object of int type containing 8 elements            int[] a = new int[8];            a[2] = 10;   //Changing value of element at index 2            a[5] = 20;   //Changing value of element at index 5            return a;      }  } |

## **Method Returrning An Array Of Derived Type :**

|  |  |
| --- | --- |
| 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 | class A  {      int i;  }    public class ArraysInJava  {      public static void main(String[] args)      {          A[] a = methodOne();            System.out.println(a[2].i);     //Output : 10            System.out.println(a[5].i);     //Output : 20      }        //Method returns an array of A-type        static A[] methodOne()      {          //declaring and creating An array object of A-type containing 8 references to A-type objects            A[] a = new A[8];            a[2] = new A();   //creating A-type object and assigning it to element at index 2            a[5] = new A();   //creating A-type object and assigning it to element at index 5            a[2].i = 10;   //Changing value of field of A-type object referred by element at index 2            a[5].i = 20;   //Changing value of field of A-type object referred by element at index 5            return a;      }  } |

# Multidimensional Arrays In Java:

We have seen that an array can contain references to other derived types. Array is also derived type. What if an array contains references to other array objects?. These arrays are called multidimensional arrays. Multidimensional arrays in java can be seen as **arrays of arrays**. i.e an array containing references of other array objects. Multidimensional arrays in java can be two, three, four or more dimensional.

## **Two-Dimensional Array :**

Two dimensional array is an array containing references of one dimensional arrays.

|  |  |
| --- | --- |
| 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 | public class MultiDimensionalArraysInJava  {      public static void main(String[] args)      {          int[] fisrtElement = {1, 2, 3};      //One Dimensional Array            int[] secondElement = {4, 5, 6};    //One Dimensional Array            int[] thirdElement = {7, 8, 9};    //One Dimensional Array            //Two Dimensional Array          int[][] twoDimensionalArray = {fisrtElement, secondElement, thirdElement};            //Printing elements of two dimensional array          for (int i = 0; i < twoDimensionalArray.length; i++)          {              System.out.print("{");              for (int j = 0; j < twoDimensionalArray[i].length; j++)              {                  System.out.print(twoDimensionalArray[i][j] +"\t");              }              System.out.print("}");              System.out.println();          }      }  } |

Output :  
{1 2 3 }  
{4 5 6 }  
{7 8 9 }

## **Three-Dimensional Array :**

Three dimensional array is an array containing references of two-dimensional arrays.

|  |  |
| --- | --- |
| 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  47  48  49  50  51 | public class MultiDimensionalArraysInJava  {      public static void main(String[] args)      {          //One Dimensional Arrays          int[] fisrtArray = {1, 2, 3};            int[] secondArray = {4, 5, 6};            int[] thirdArray = {7, 8, 9};            int[] fourthArray = {10, 11, 12};            int[] fifthArray = {13, 14, 15};            int[] sixthArray = {16, 17, 18};            int[] seventhArray = {19, 20, 21};            int[] eighthArray = {22, 23, 24};            int[] ninthArray = {25, 26, 27};            //Two Dimensional Arrays          int[][] twoDimensionalArray1 = {fisrtArray, secondArray, thirdArray};            int[][] twoDimensionalArray2 = {fourthArray, fifthArray, sixthArray};            int[][] twoDimensionalArray3 = {seventhArray, eighthArray, ninthArray};            //Three Dimensional Array          int[][][] threeDimensionalArray = {twoDimensionalArray1, twoDimensionalArray2, twoDimensionalArray3};            //Printing elements of three dimensional array          for (int i = 0; i < threeDimensionalArray.length; i++)          {              System.out.print("{");              for (int j = 0; j < threeDimensionalArray[i].length; j++)              {                  System.out.print("{");                  for (int k = 0; k < threeDimensionalArray[i][j].length; k++)                  {                      System.out.print(threeDimensionalArray[i][j][k] +"\t");                  }                  System.out.print("} ");              }              System.out.print("}");              System.out.println();          }      }  } |

Output :  
{{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 } }

## **Four-Dimensional Array :**

Four dimensional array is an array containing references of three dimensional arrays.

|  |  |
| --- | --- |
| 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  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111 | public class MultiDimensionalArraysInJava  {      public static void main(String[] args)      {          //One Dimensional Arrays          int[] fisrtArray = {1, 2, 3};            int[] secondArray = {4, 5, 6};            int[] thirdArray = {7, 8, 9};            int[] fourthArray = {10, 11, 12};            int[] fifthArray = {13, 14, 15};            int[] sixthArray = {16, 17, 18};            int[] seventhArray = {19, 20, 21};            int[] eighthArray = {22, 23, 24};            int[] ninthArray = {25, 26, 27};            int[] tenthArray = {28, 29, 30};            int[] eleventhArray = {31, 32, 33};            int[] twelvethArray = {34, 35, 36};            int[] thirteenthArray = {37, 38, 39};            int[] fourteenththArray = {40, 41, 42};            int[] fifteenthArray = {43, 44, 45};            int[] sixteenthArray = {46, 47, 48};            int[] seventeenthArray = {49, 50, 51};            int[] eighteenthArray = {52, 53, 54};            int[] ninteenthArray = {55, 56, 57};            int[] twentiethArray = {58, 59, 60};            int[] twentyFirstArray = {61, 62, 63};            int[] twentySecondArray = {64, 65, 66};            int[] twentyThirdArray = {67, 68, 69};            int[] twentyFourthArray = {70, 71, 72};            int[] twentyFifthArray = {73, 74, 75};            int[] twentySixthArray = {76, 77, 78};            int[] twentySeventhArray = {79, 80, 81};            //Two Dimensional Arrays          int[][] twoDimensionalArray1 = {fisrtArray, secondArray, thirdArray};            int[][] twoDimensionalArray2 = {fourthArray, fifthArray, sixthArray};            int[][] twoDimensionalArray3 = {seventhArray, eighthArray, ninthArray};            int[][] twodimentionalArray4 = {tenthArray, eleventhArray, twelvethArray};            int[][] twodimentionalArray5 = {thirteenthArray, fourteenththArray, fifteenthArray};            int[][] twoDimentionalArray6 = {sixteenthArray, seventeenthArray, eighteenthArray};            int[][] twoDimentionalArray7 = {ninteenthArray, twentiethArray, twentyFirstArray};            int[][] twoDimentionalArray8 = {twentySecondArray, twentyThirdArray, twentyFourthArray};            int[][] twoDimentionalArray9 = {twentyFifthArray, twentySixthArray, twentySeventhArray};            //Three Dimensional Array          int[][][] threeDimensionalArray1 = {twoDimensionalArray1, twoDimensionalArray2, twoDimensionalArray3};            int[][][] threeDimentionalArray2 = {twodimentionalArray4, twodimentionalArray5, twoDimentionalArray6};            int[][][] threeDimensionalArray3 = {twoDimentionalArray7, twoDimentionalArray8, twoDimentionalArray9};            //Four Dimensional Array          int[][][][] fourthDimentionalArray = {threeDimensionalArray1, threeDimentionalArray2, threeDimensionalArray3};            //Printing elements of four dimensional array          for (int i = 0; i < fourthDimentionalArray.length; i++)          {              System.out.print("{");              for (int j = 0; j < fourthDimentionalArray[i].length; j++)              {                  System.out.print("{");                  for (int k = 0; k < fourthDimentionalArray[i][j].length; k++)                  {                      System.out.print("{");                      for (int m = 0; m < fourthDimentionalArray[i][j][k].length; m++)                      {                          System.out.print(fourthDimentionalArray[i][j][k][m] +"\t");                      }                      System.out.print("} ");                  }                  System.out.print("} ");              }              System.out.print("}");              System.out.println();          }      }  } |

Output :  
{{{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 47 48 } {49 50 51 } {52 53 54 } } }  
{{{55 56 57 } {58 59 60 } {61 62 63 } } {{64 65 66 } {67 68 69 } {70 71 72 } } {{73 74 75 } {76 77 78 } {79 80 81 } } }

Only one dimensional and two dimensional arrays are commonly used. Three dimensional and onward are rarely used.

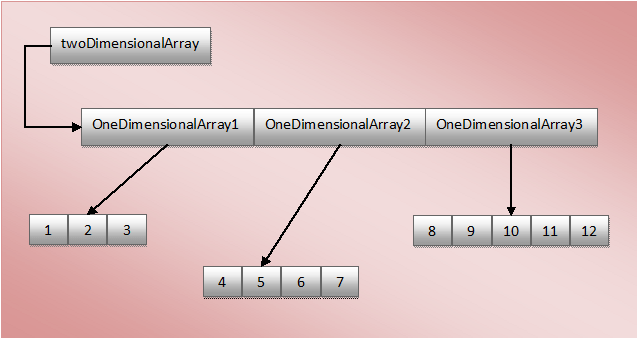
# Jagged Arrays In Java:

**Jagged arrays** in java are arrays containing arrays of different length. Jagged arrays are also multidimensional arrays. Jagged arrays in java sometimes are also called as **ragged arrays.**

The following example shows two dimensional jagged array containing 3 one dimensional arrays of length 3, 4 and 5 respectively.

|  |  |
| --- | --- |
| 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 | public class JaggedArraysInJava  {      public static void main(String[] args)      {          //One Dimensional Array of lenghth 3          int[] OneDimensionalArray1 = {1, 2, 3};            //One Dimensional Array of lenghth 4          int[] oneDimensionalArray2 = {4, 5, 6, 7};            //One Dimensional Array of lenghth 5          int[] oneDimensionalArray3 = {8, 9, 10, 11, 12};            //Jagged Two Dimensional Array          int[][] twoDimensionalArray = {OneDimensionalArray1, oneDimensionalArray2, oneDimensionalArray3};            //Printing elements of Two Dimensional Array          for (int i = 0; i < twoDimensionalArray.length; i++)          {              for (int j = 0; j < twoDimensionalArray[i].length; j++)              {                  System.out.print(twoDimensionalArray[i][j]+"\t");              }              System.out.println();          }      }  } |

The jagged array in the above program can be represented as,



## **Instantiating Jagged Array :**

While instantiating a jagged array using new operator, you need not to mention the size of it’s contained arrays. It indicates that array contains the arrays of varied length. For example,

|  |  |
| --- | --- |
| 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 | public class JaggedArraysInJava  {      public static void main(String[] args)      {          //Jagged Two Dimensional Array, No need to mention the size of it's contained arrays          int[][] jaggedTwoDimensionalArray = new int[5][];            //One Dimensional Array of lenghth 3          int[] OneDimensionalArray1 = {1, 2, 3};            //One Dimensional Array of lenghth 4          int[] oneDimensionalArray2 = {4, 5, 6, 7};            //One Dimensional Array of lenghth 5          int[] oneDimensionalArray3 = {8, 9, 10, 11, 12};            //Initializing elements of Jagged Array            jaggedTwoDimensionalArray[0] = OneDimensionalArray1;            jaggedTwoDimensionalArray[1] = oneDimensionalArray2;            jaggedTwoDimensionalArray[2] = oneDimensionalArray3;      }  } |

# java.util.Arrays Class In Java:

java.util.Arrays class in java is used to perform some operations like copying, sorting and searching on the arrays. You can check the documentation of Arrays class [here](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html).

Let’s discuss some of the useful methods of Arrays Class in java.

## **toString() method :**

toString() method of Arrays class is used to display all elements of an array. This method returns string representation of all elements of an array. String representation consists of all elements enclosed within “[ ]”. All elements are seperated by “, “(Comma and Space). The following example shows usage of toString() method.

|  |  |
| --- | --- |
| 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  47 | import java.util.Arrays;    public class ArraysClassInJava  {      public static void main(String[] args)      {          //An array of byte          byte[] b = {10, 20, 30};            System.out.println(Arrays.toString(b));        //Output : [10, 20, 30]            //An array of short          short[] s = {40, 50, 60, 70};            System.out.println(Arrays.toString(s));       //Output : [40, 50, 60, 70]            //An array of int          int[] i = {12, 21, 42, 68};            System.out.println(Arrays.toString(i));      //Output : [12, 21, 42, 68]            //An array of long          long[] l = {100, 110, 120, 130, 140, 150};            System.out.println(Arrays.toString(l));      //Output : [100, 110, 120, 130, 140, 150]            //An array of double          double[] d = {12.5, 14.9, 87.4, 55.8};            System.out.println(Arrays.toString(d));      //Output : [12.5, 14.9, 87.4, 55.8]            //An array of char          char[] c = {'A', 'B', 'C', 'D', 'E'};            System.out.println(Arrays.toString(c));     //Output : [A, B, C, D, E]            //An array of boolean          boolean[] bln = {true, false, false, true};            System.out.println(Arrays.toString(bln));     //Output : [true, false, false, true]            //An array of String          String[] str = {"java", "concepts", "Arrays", "methods"};            System.out.println(Arrays.toString(str));     //Output : [java, concepts, Arrays, methods]      }  } |

## **sort() method :**

This method sorts elements of an array in ascending order. This method internally uses quiksort algorithm to sort the elements. The following example shows usage of sort() method.

**Note : There is no sort() method in Arrays class which sorts boolean type of array.**

|  |  |
| --- | --- |
| 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  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | import java.util.Arrays;    public class ArraysClassInJava  {      public static void main(String[] args)      {          //An array of byte          byte[] b = {51, 22, 8, 37};            Arrays.sort(b);     //sorts elements of the specified array in ascending order            System.out.println(Arrays.toString(b));        //Output : [8, 22, 37, 51]            //An array of short          short[] s = {24, 5, 21, 12, 19};            Arrays.sort(s);            System.out.println(Arrays.toString(s));       //Output : [5, 12, 19, 21, 24]            //An array of int          int[] i = {42, 12, 68, 21};            Arrays.sort(i);            System.out.println(Arrays.toString(i));      //Output : [12, 21, 42, 68]            //An array of long          long[] l = {879, 412, 258, 985, 856};            Arrays.sort(l);            System.out.println(Arrays.toString(l));      //Output : [258, 412, 856, 879, 985]            //An array of double          double[] d = {12.5, 87.4, 41.24, 14.9, 55.8};            Arrays.sort(d);            System.out.println(Arrays.toString(d));      //Output : [12.5, 14.9, 41.24, 55.8, 87.4]            //An array of char          char[] c = {'Z', 'B', 'X', 'L', 'b', 'A'};            Arrays.sort(c);            System.out.println(Arrays.toString(c));     //Output : [A, B, L, X, Z, b]            //An array of String          String[] str = {"java", "concepts", "Arrays", "methods", "core"};            Arrays.sort(str);            System.out.println(Arrays.toString(str));     //Output : [Arrays, concepts, core, java, methods]            //An array of boolean          boolean[] bln = {true, false, false, true};            Arrays.sort(bln);   //Compile time error            //Because, there is no sort method in Arrays class which sorts boolean type array      }  } |

## **binarySearch() method :**

This method Searches the specified array for the specified value using the binary search algorithm. The array must be sorted before calling this method. If it is not sorted, the results will be undefined.

If the value is found in the array, it returns index of that value in the array. If not found, it returns (-n-1). Where, n is called **insertion point**. The insertion point is the point at which the specified value would be inserted into the sorted array. For example, it will be the index of the first element greater than the specified value or it will be the length of the array if all elements in the array are less than the specified value.

If the array contains multiple elements same as specified value, there is no guarantee which one will be found.

**Note : There is no binarySearch() method in Arrays class which searches for a boolean value in the boolean type of array.**

The following example shows usage of binarySearch() method.

|  |  |
| --- | --- |
| 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  47  48  49  50  51  52  53  54  55 | import java.util.Arrays;    public class ArraysClassInJava  {      public static void main(String[] args)      {          //An array of int          int[] i = {24, 13, 45, 37, 84, 13, 28};            //Sorting the int array          Arrays.sort(i);            //Printing sorted array          System.out.println(Arrays.toString(i));       //Output : [13, 13, 24, 28, 37, 45, 84]            //Searching the value          System.out.println(Arrays.binarySearch(i, 37));  //Output : 4            System.out.println(Arrays.binarySearch(i, 13));  //Output : 1            System.out.println(Arrays.binarySearch(i, 55));     //Output : -7            //An array of char          char[] c = {'X', 'n', 'F', 's', 'D', 'J', 'j', 'F'};            //Sorting the char array          Arrays.sort(c);            //Printing Sorted array          System.out.println(Arrays.toString(c));      //Output : [D, F, F, J, X, j, n, s]            //Searching the character          System.out.println(Arrays.binarySearch(c, 'F'));     //Output : 1            System.out.println(Arrays.binarySearch(c, 'J'));     //Output : 3            System.out.println(Arrays.binarySearch(c, 'Z'));     //Output : -6            //An array of String          String[] str = {"First", "second", "Third", "second", "Four", "fifth"};            //Sorting the String array          Arrays.sort(str);            //Printing Sorted array          System.out.println(Arrays.toString(str));       //Output : [First, Four, Third, fifth, second, second]            //Searching the string in the array          System.out.println(Arrays.binarySearch(str, "Third"));    //Output : 2            System.out.println(Arrays.binarySearch(str, "second"));   //Output : 4            System.out.println(Arrays.binarySearch(str, "sixth"));    //Output : -7      }  } |

# java.util.Arrays Class In Java (Contd…):

 we will discuss about some more useful methods of java.util.Arrays Class in java.

## **fill() Method :**

This method assigns specified value to each element of an array. This method is useful in initializing all elements of an array with one value. The following example shows usage of fill() method.

|  |  |
| --- | --- |
| 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 | import java.util.Arrays;    public class ArraysClassInJava  {      public static void main(String[] args)      {          //An array of int          int[] i = new int[5];            Arrays.fill(i, 10);    //Assigns 10 to each element of the array            System.out.println(Arrays.toString(i));   //Output : [10, 10, 10, 10, 10]            //An array of double          double[] d = {12.5, 14.8, 45.9, 23.5};            Arrays.fill(d, 53.6);    //Assigns 53.6 to each element of the array            System.out.println(Arrays.toString(d));   //Output : [53.6, 53.6, 53.6, 53.6]            //An array of boolean          boolean[] bln = new boolean[5];            Arrays.fill(bln, true);     //Assigns true to each element of the array            System.out.println(Arrays.toString(bln));     //Output : [true, true, true, true, true]            //An array of char          char[] c = new char[10];            Arrays.fill(c, 'P');    //Assigns P to each element of the array            System.out.println(Arrays.toString(c));     //Output : [P, P, P, P, P, P, P, P, P, P]            //An array of String          String[] str = {"Java", "Concepts", "basic java", "Arrays Class"};            Arrays.fill(str, "value");    //Assigns value to each element of the array            System.out.println(Arrays.toString(str));     //Output : [value, value, value, value]      }  } |

## **copyOf() Method :**

This method copies the specified array into new array of same type. While copying, the new array may be truncated or padded with default values so that it has the specified length.

|  |  |
| --- | --- |
| 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  47 | import java.util.Arrays;    public class ArraysClassInJava  {      public static void main(String[] args)      {          //An array of int          int[] i = {1, 21, 15, 48, 79};            //Copying array i into array i1          int[] i1 = Arrays.copyOf(i, 10);            System.out.println(Arrays.toString(i1));   //Output : [1, 21, 15, 48, 79, 0, 0, 0, 0, 0]            //An array of double          double[] d = {12.5, 45.8, 56.2, 47.9, 23.6, 89.5};            //Copying array d into array d1          double[] d1 = Arrays.copyOf(d, 4);            System.out.println(Arrays.toString(d1));    //Output : [12.5, 45.8, 56.2, 47.9]            //An array of boolean          boolean[] bln = {true, false, true, true, false};            //Copying array bln into array bln1          boolean[] bln1 = Arrays.copyOf(bln, 10);            System.out.println(Arrays.toString(bln1));    //Output : [true, false, true, true, false, false, false, false, false, false]            //An array of char          char[] c = {'X', 'B', 'Z', 'H', 'I', 'J'};            //Copying array c into array c1          char[] c1 = Arrays.copyOf(c, 5);            System.out.println(Arrays.toString(c1));    //Output : [X, B, Z, H, I]            //An array of String          String[] str = {"java", "j2ee", "struts", "hibernate"};            //Copying array str into array str1          String[] str1 = Arrays.copyOf(str, 7);            System.out.println(Arrays.toString(str1));      }  } |

## **copyOfRange() Method :**

This method is used to copy some part of an array into another array of same type. While copying the new array may be truncated or padded with default values to obtain the required length.

|  |  |
| --- | --- |
| 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  47 | import java.util.Arrays;    public class ArraysClassInJava  {      public static void main(String[] args)      {          //An array of int          int[] i = {1, 21, 15, 48, 79};            //Copying some part of array i into array i1          int[] i1 = Arrays.copyOfRange(i, 2, 7);            System.out.println(Arrays.toString(i1));   //Output : [15, 48, 79, 0, 0]            //An array of double          double[] d = {12.5, 45.8, 56.2, 47.9, 23.6, 89.5};            //Copying some part of array d into array d1          double[] d1 = Arrays.copyOfRange(d, 2, 5);            System.out.println(Arrays.toString(d1));    //Output : [56.2, 47.9, 23.6]            //An array of boolean          boolean[] bln = {true, false, true, true, false};            //Copying some part of array bln into array bln1          boolean[] bln1 = Arrays.copyOfRange(bln, 1, 8);            System.out.println(Arrays.toString(bln1));    //Output : [false, true, true, false, false, false, false]            //An array of char          char[] c = {'X', 'B', 'Z', 'H', 'I', 'J'};            //Copying some part of array c into array c1          char[] c1 = Arrays.copyOfRange(c, 2, 4);            System.out.println(Arrays.toString(c1));    //Output : [Z, H]            //An array of String          String[] str = {"java", "j2ee", "struts", "hibernate"};            //Copying some part of array str into array str1          String[] str1 = Arrays.copyOfRange(str, 4, 8);            System.out.println(Arrays.toString(str1));   //Output : [null, null, null, null]      }  } |

# 10 Interesting Observations about arrays in java:

The followings are 10 interesting observations about arrays in java. You may be asked in the interviews or Java certification exams about these observations.

***1)*** The size of an array can not be negative. If you give size of an array as negative, you don’t get any errors while compliing. But, you will get NegativeArraySizeException at run time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] i = new int[-5];   //No Compile Time Error            //You will get java.lang.NegativeArraySizeException at run time      }  } |

***2)*** The size of an array must be an integer or an expression which results an integer. Auto-widening is also allowed.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] i = new int[10+5];    //Array size can be an expression resulting an integer            int[] i1 = new int[(byte)10];    //byte is auto-widened to int            int i3 = new int[10.25];     //Compile Time Error : size can not be double      }  } |

***3)*** Declaration and instantiating of an array strictly must be of same type. No auto-widening, auto-boxing and auto-unboxing is allowed. But only auto-upcasting is allowed.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ArraysInJava  {      public static void main(String[] args)      {          Integer[] I = new int[5];   //Compile Time Error : Auto-Boxing not allowed            int[] i = new Integer[10];   //Compile Time Error : Auto-UnBoxing not allowed            long[] l = new byte[10];    //Compile Time Error : Auto-widening not allowed            Object[] o = new String[10];    //No Compile Time Error : Auto-Upcasting is allowed, String[] is upcasted to Object[]      }  } |

***4)*** The type of elements of an array must be compatible with type of the array object. If you try to store non-compatible element in an array object, you will get ArrayStoreException at run time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ArraysInJava  {      public static void main(String[] args)      {          Object[] o = new String[10];    //No Compile Time Error : String[] is auto-upcasted to Object[]            //i.e array object of strings can be referred by array reference variable of Object type            o[2] = "java";            o[5] = 20;   //No Compile time error,            //but you will get java.lang.ArrayStoreException at run time.      }  } |

***5)*** If you are supplying the contents to the array without new operator, then it should be at the time of declaration only. Not at any other places.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] i = {1, 2, 3, 4};   //This is the correct way            i = {1, 2, 3 , 4};     //Compile time error            i = new int[]{1, 2, 3, 4};  //This is also correct way      }  } |

***6)*** Another way of declaring multi dimensional arrays.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ArraysInJava  {      public static void main(String[] args)      {          int[][] twoDArray;    //Normal way of declaring two-dimensional array            int[] TwoDArray [];   //Another way of declaring two-dimensional array            int[][][] threeDArray;  //Normal way of declaring three-dimensional array            int[] ThreeDArray [][];    //This is also legal      }  } |

***7)*** While creating multi dimensional arrays, you can not specify an array dimension after an empty dimension.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ArraysInJava  {      public static void main(String[] args)      {          int[][][] threeDArray = new int[10][][10];    //Compile Time Error            int[][][] threeDArray1 = new int[][10][];     //Compile Time Error            int[][][] threeDArray2 = new int[][][10];     //Compile Time Error      }  } |

***8)*** You can create an anonymous array i.e an array without reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ArraysInJava  {      public static void main(String[] args)      {          //Creating anonymous array            System.out.println(new int[]{1, 2, 3}.length);    //Output : 3            System.out.println(new int[]{47, 21, 58, 98}[1]);   //Output : 21      }  } |

***9)*** While assigning one array reference variable to another, compiler checks only type of the array not the size.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ArraysInJava  {      public static void main(String[] args)      {          int[] a = new int[10];            int[] b = new int[100];            double[] c = new double[20];            a = b;            b = c;     //Compile Time Error : can not convert from double[] to int[]      }  } |

***10)*** The size of an array can not be changed once you define it. You can not insert or delete array elements after creating an array. Only you can change is the value of the elements. This is the main drawback of arrays.

# How To Check The Equality Of Two Arrays In Java?

Two arrays are said to be equal if the arrays have equal number of elements and all corresponding pairs of elements in two arrays are equal. In the other words, two arrays are said to be equal if both the arrays have same elements in the same order. You are very often need to compare two arrays for equality while developing the applications. This is also one of the important  java interview question for 1 or 2 years experienced java professionals.

In this post, I have tried to point out different methods to check the equality of two arrays.

### **1) Iterative Method :**

In this method, first we check length of two given arrays. If the length of both arrays is same, then we compare corresponding pairs of elements of both the arrays. If all corresponding pairs of elements are equal, then given arrays will be considered as equal. This method will be time consuming if the arrays have lots of elements. This method is not recommended to check the equality of two arrays if the arrays are big in size. But in the interview, interviewer may ask you to compare two arrays without using in-built functions. In that time, this method may help you.

|  |  |
| --- | --- |
| 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 | public class EqualityOfTwoArrays  {      public static void main(String[] args)      {          int[] arrayOne = {2, 5, 1, 7, 4};            int[] arrayTwo = {2, 5, 1, 7, 4};            boolean equalOrNot = true;            if(arrayOne.length == arrayTwo.length)          {              for (int i = 0; i < arrayOne.length; i++)              {                  if(arrayOne[i] != arrayTwo[i])                  {                      equalOrNot = false;                  }              }          }          else          {              equalOrNot = false;          }            if (equalOrNot)          {              System.out.println("Two Arrays Are Equal");          }          else          {              System.out.println("Two Arrays Are Not equal");          }      }  } |

### **2) Using Arrays.equals() Method :**

In this method, we use in-built equals() method of Arrays class to check the equality of two arrays. This method takes two arrays as parameters and returns true if both the arrays have same number of elements and corresponding pairs of elements of both arrays are equal.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class EqualityOfTwoArrays  {      public static void main(String[] args)      {          String[] s1 = {"java", "j2ee", "struts", "hibernate"};            String[] s2 = {"jsp", "spring", "jdbc", "hibernate"};            String[] s3 = {"java", "j2ee", "struts", "hibernate"};            System.out.println(Arrays.equals(s1, s2));        //Output : false            System.out.println(Arrays.equals(s1, s3));      //Output : true      }  } |

If you want to compare two arrays which have same number of elements and same set of elements but in different positions, then first sort both arrays using Arrays.sort() method and then compare using Arrays.equals() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class EqualityOfTwoArrays  {      public static void main(String[] args)      {          String[] s1 = {"java", "swings", "j2ee", "struts", "jsp", "hibernate"};            String[] s2 = {"java", "struts", "j2ee", "hibernate", "swings", "jsp"};            Arrays.sort(s1);            Arrays.sort(s2);            System.out.println(Arrays.equals(s1, s2));       //Output : true      }  } |

### **3) Using Arrays.deepEquals() Method :**

If you are checking multidimensional arrays for equality, then use deepEquals() method of Arrays class instead of equals() method. Because, deepEquals() performs deep comparison of both the arrays.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class EqualityOfTwoArrays  {      public static void main(String[] args)      {          String[][] s1 = { {"java", "swings", "j2ee" }, { "struts", "jsp", "hibernate"} };            String[][] s2 = { {"java", "swings", "j2ee" }, { "struts", "jsp", "hibernate"} };            System.out.println(Arrays.deepEquals(s1, s2));     //Output : true            //Calling equals() method on same arrays will return false            System.out.println(Arrays.equals(s1, s2));        //Output : false            //That's why use deepEquals() method to compare multidimensional arrays      }  } |

# Arrays.deepToString() Method Example

**java.util.Arrays** class has many useful methods to perform the operations on the arrays. **deepToString() method** is one such method. **Arrays.deepToString()** method is used to get the string representation of multidimensional arrays. This method returns the deep contents of the specified array. If the specified array contains other arrays as it’s elements then it returns the contents of those arrays also.

Below example shows how to use **deepToString()** method to print the contents of the multidimensional arrays.

|  |  |
| --- | --- |
| 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  47  48  49  50  51  52  53 | public class MainClass  {      public static void main(String[] args)      {          //One Dimensional Array            String[] oneDArray = new String[] {"ONE", "TWO", "THREE", "FOUR", "FIVE"};            System.out.println("One Dimensional Array : ");            //Printing one dimensional array contents using deepToString() method            System.out.println(Arrays.deepToString(oneDArray));            //Two Dimensional Array            String[][] twoDArray = new String[][] {                                                      {"ONE", "TWO", "THREE", "FOUR"},                                                      {"FIVE", "SIX", "SEVEN"},                                                      {"EIGHT", "NINE", "TEN", "ELEVEN", "TWELVE"}                                                  };            System.out.println("Two Dimensional Array : ");            //Printing two dimensional array contents using deepToString() method            System.out.println(Arrays.deepToString(twoDArray));            //Three Dimensional Array            String[][][] threeDArray = new String[][][] {                                                          {                                                              {"ONE", "TWO", "THREE"},                                                              {"FOUR", "FIVE", "SIX", "SEVEN"}                                                          },                                                          {                                                              {"EIGHT", "NINE", "TEN", "ELEVEN"},                                                              {"TWELVE", "THIRTEEN", "FOURTEEN"}                                                          },                                                          {                                                              {"FIFTEEN", "SIXTEEN"} ,                                                              {"SEVENTEEN", "EIGHTEEN", "NINETEEN"},                                                              {"TWENTY", "TWENTY ONE"}                                                          }                                                      };            System.out.println("Three Dimensional Array : ");            //Printing three dimensional array contents using deepToString() method            System.out.println(Arrays.deepToString(threeDArray));      }  } |

**Output :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | One Dimensional Array :  [ONE, TWO, THREE, FOUR, FIVE]  Two Dimensional Array :  [[ONE, TWO, THREE, FOUR], [FIVE, SIX, SEVEN], [EIGHT, NINE, TEN, ELEVEN, TWELVE]]  Three Dimensional Array :  [[[ONE, TWO, THREE], [FOUR, FIVE, SIX, SEVEN]], [[EIGHT, NINE, TEN, ELEVEN], [TWELVE, THIRTEEN, FOURTEEN]], [[FIFTEEN, SIXTEEN], [SEVENTEEN, EIGHTEEN, NINETEEN], [TWENTY, TWENTY ONE]]] |

If you want to print the contents of one dimensional arrays, then use **Arrays.toString()** method or **normal for loop** or **enhanced for loop**. You can also use **Arrays.deepToString()** method to print the contents of one dimensional arrays. But, If you want to print the contents of multidimensional arrays, instead of nesting multiple for loops, use **Arrays.deepToString()** method. It is the easiest method to print the contents of multidimensional arrays

# 7. Static Import In Java:

Before discussing about **static import** in java, first let’s know what normal **import** does.

Consider this java file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package pack1;    public class A  {      //Member Of Pack1  } |

If you want to use the member of pack1 (Class A) in another package, then you have to refer it through package name like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | package pack2;    //Member Of pack2  public class B  {      public static void main(String[] args)      {          pack1.A a = new pack1.A();    //Accessing member of pack1      }  } |

It will be very annoying if pack1 member has to be used hundreds of times. Each time, member has to be referred through it’s package name. You can avoid this using **import statement** just after the package declaration statement. If you import the required package in the current file, then you can access the members of that package without using package name just like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | package pack2;    import pack1.\*;      //Importing all members of pack1    //Member Of pack2  public class B  {      public static void main(String[] args)      {          A a = new A();     //Accessing member of pack1 without using package name      }  } |

Now come to **static import in java**. Using static import, you can access static members of a class without referring them through their class name. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package pack1;    public class A  {      public static int i;    //Static Member        public static void methodOne()      {          //Static Member      }  } |

The above static members of class A can be directly used in another package if you import them as static.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package pack2;    import static pack1.A.\*;      //Importing all static members of Class A    public class B  {      public static void main(String[] args)      {          System.out.println(i);      //accessing static member of Class A without using class name.            methodOne();     //accessing static member of Class A without using class name.      }  } |

**Notes :**

* Using import statement, you can import only members of a package like classes, interfaces etc into the current file.
* Using static import, you can import only static members of a class like fields, methods etc into the current file.
* You can import only individual members of a package or a class like **import pack1.A;** or **import static pack1.A.i**

# 8. instanceof Operator In Java:

Consider the following class hierarchy.

|  |  |
| --- | --- |
| 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 | class Subject  {      void methodOfSubject()      {          System.out.println("From Subject");      }  }    class Physics extends Subject  {      @Override      void methodOfSubject()      {          System.out.println("From Physics");      }  }    class Biology extends Subject  {      @Override      void methodOfSubject()      {          System.out.println("From Biology");      }  }    class Botany extends Biology  {      @Override      void methodOfSubject()      {          System.out.println("From Botany");      }  }    class Zoology extends Biology  {      @Override      void methodOfSubject()      {          System.out.println("From Zoology");      }  } |

Now, let’s assume that we have a requirement in which some block of code has to be executed if the object is a type of Biology. Let’s try to meet this requirement.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class InstanceOfOperator  {      public static void main(String[] args)      {          //An array of Subjects          Subject[] subjects = {new Biology(), new Botany(), new Zoology(), new Subject(), new Physics()};            for (int i = 0; i < subjects.length; i++)          {              Biology biology = (Biology) subjects[i];                biology.methodOfSubject();          }      }  } |

The above code runs successfully if an object is a type of Biology or it’s sub types(Botany and Zoology). But. it gives ClassCastException at run time if an object is a type of other than Biology (Subject and Physics). ClassCastException is a dangerous exception which will be known only at run time and hence causing the program termination. Therefore, It has to be handled properly. Of course, you can handle it by using try-catch blocks. But, there is one more easy method to handle it. That is using **instanceof operator.**

**instanceof operator in java** is a binary operator which checks whether an object is of specified type. The syntax for using instanceof operator is as follows.

**objref instanceof Type**

Where, **objref** is a reference to the object and **type** is a class type. instanceof operator evaluates to true If objref is of the specified type or can be cast into the specified type, otherwise it returns false. Now, let’s try to implement above requirement using instanceof operator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class InstanceOfOperator  {      public static void main(String[] args)      {          //An array of Subjects          Subject[] subjects = {new Biology(), new Botany(), new Zoology(), new Subject(), new Physics()};            for (int i = 0; i < subjects.length; i++)          {              //Checking whether the object is a type of Biology using instanceof operator              if(subjects[i] instanceof Biology)              {                  subjects[i].methodOfSubject();              }          }      }  } |

Now, run this program. It does not throw ClassCastException at run time and program will run smoothly. That is the use of instanceof operator.

# 9. isNaN() And isInfinite() Methods In Java:

**isNaN()** and **isInfinite()** methods are the static members of java.lang.Double and java.lang.Float classes. Both these two wrapper classes have their own version of these two methods.

Before discussing about these two methods, let’s get to know about **NaN**, **POSITIVE\_INFINITY and NEGATIVE\_INFINITY.**

NaN, POSITIVE\_INFINITY and NEGATIVE\_INFINITY are static and final constants of java.lang.Double and java.lang.Float wrapper classes.

**NaN** : This constant holds Not-a-Number value of specified type. It represents mathematically undefined or unrepresentable number such as number obtained by dividing zero by zero or square root of a negative number.

**POSITIVE\_INFINITY** : This constant holds positive infinity value.

**NEGATIVE\_INFINITY** : This constant holds negative infinity value.

Following example shows usage of these three constants.

|  |  |
| --- | --- |
| 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 | public class IsNanAndIsInfinite  {      public static void main(String[] args)      {          //Assigning Not-A-Number value of type double to d1          double d1 = Double.NaN;            System.out.println(d1);            //Output : NaN            //Assigning positive infinity value of type double to d2          double d2 = Double.POSITIVE\_INFINITY;            System.out.println(d2);           //Output : Infinity            //Assigning negative infinity value of type double to d3          double d3 = Double.NEGATIVE\_INFINITY;            System.out.println(d3);           //Output : -Infinity            //Assigning Not-A-Number value of type float to f1          float f1 = Float.NaN;            System.out.println(f1);         //Output : NaN            //Assigning positive infinity value of type float to f2          float f2 = Float.POSITIVE\_INFINITY;            System.out.println(f2);        //Output : Infinity            //Assigning negative infinity value of type float to f3          float f3 = Float.NEGATIVE\_INFINITY;            System.out.println(f3);       //Output : -Infinity      }  } |

## **isNaN() :**

This method returns true, if the specified number is Not-a-Number i.e mathematically undefined or unrepresentable number. Otherwise it returns false. Both, Double and Float wrapper classes have this method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class IsNanAndIsInfinite  {      public static void main(String[] args)      {          double d = 0.0/0.0;            System.out.println(Double.isNaN(d));    //Output : true            d = Math.sqrt(-1.2);            System.out.println(Double.isNaN(d));    //Output : true            float f = 0.0f/0.0f;            System.out.println(Float.isNaN(f));            //Output : true            f = 0 \* Float.POSITIVE\_INFINITY;            System.out.println(Float.isNaN(f));          //Output : true      }  } |

## **isInfinite() :**

This method returns true if the specified number is positively or negatively infinite. Both Double and Float wrapper classes have this method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class IsNanAndIsInfinite  {      public static void main(String[] args)      {          double d = Double.POSITIVE\_INFINITY / 0.0;            System.out.println(Double.isInfinite(d));    //Output : true            d = Double.NEGATIVE\_INFINITY / 0.0;            System.out.println(Double.isInfinite(d));    //Output : true            float f = Float.POSITIVE\_INFINITY \* 2.2f;            System.out.println(Float.isInfinite(f));            //Output : true            f = Float.NEGATIVE\_INFINITY \* 4.12f;            System.out.println(Float.isInfinite(f));          //Output : true      }  } |

# 10. Unreachable Code And Dead Code In Java:

## **Unreachable Code :**

Every statement in any java program must be reachable i.e every statement must be executable at least once in any one of the possible flows. If any code can not be executable in any of the possible flows, then it is called unreachable code. Unreachable code in java is a compile time error.

Look at the following example. Try to compile this example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class UnreachableCodeInJava  {      void UnreachableCode\_method(boolean b)      {          if(b)          {              return;          }          else          {              return;          }            System.out.println("Unreachable Statement");     //Compile Time Error : Unreachable Code      }  } |

In this example, Line 14 gives compile time error : Unreachable Code. Because this statement can not be reached in any of the flows. If boolean “b” is true, then if block will be executed and control will be returned. If boolean “b” is false, then else block will be executed and control will be returned. So, in both the flows control is not coming to Line 14. It remains unreachable. Hence, it gives compile time error.

Have a look at this example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class UnreachableCodeInJava  {      void UnreachableCode\_method(boolean b)      {          System.out.println("Reachable Statement");            return;            System.out.println("Unreachable Statement");     //Compile Time Error      }  } |

In this example, UnreachableCode\_method() has only one flow. In this flow only, all the statements has to be executed at least once. But, control is returning at second statement only. So, Third statement becomes unreachable.

## **Dead Code :**

Dead code is also unreachable code, but dead code doesn’t give compile time error. It just gives you a warning.  
For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class UnreachableCodeInJava  {      void UnreachableCode\_method(boolean b)      {          System.out.println("Reachable Statement");            if(true)          {              return;          }            System.out.println("Unreachable Statement");     //Dead Code      }  } |

In the above example, if block will be always executed as we are passing constant value “true” to it. So, Line 12 becomes unreachable. Although compiler will not complain about it. It just gives you a warning saying that it is a dead code.

Now, make little modification to above example.Replace the if block with while block.Compile it and see what happens.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class UnreachableCodeInJava  {      static void UnreachableCode\_method(boolean b)      {          System.out.println("Reachable Statement");            while(true)          {              return;          }            System.out.println("Unreachable Statement");     //Compile Time Error      }  } |

This example gives compile time error : Unreachable code. But why? Why compiler changes it’s mind when it sees while block? Why it does not complain about unreachable code after if block.?

This is because,The rules of unreachable code for different blocks like constructors, methods, try, catch, if, for, while etc are defined individually in the Java Language Specification. According to Java Language Specification, The conditional expression of If block will no be evaluated at compile time to decide the unreachable code. That’s why you will not get compile time error in case of if block. You can check the rules of unreachable code [here](http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.21).