JAVA DOCUMENTATION

1.CLASS & OBJECT

Class:

Class has many definitions.

Def:1:- A class is a top level block that is used for grouping variables and methods for developing logic.

Def:2:- A class is specification or blue print or template of an object that defines what goes to make up a particular sort of object.

A class defines the structure, state and Behavior (data & code) that will be shared by a set of objects. Hence each object of a given class contains the structure, state, and behavior defined by that class.

Syntax of a class:

```
<Modifiers> class <ClassName>{

    //body of the class.

    <Constructor>

    <static members>

    <Non-static members>

    <nested inner classes>
}
```

Object:

Object is the physical reality of a class.

- Technically object can be defined as “it is an encapsulated form of all non-static variables and non-static methods of a particular class”|.
- An instance of a class is the other technical term of an object.
- The process of creating an object out of a class is called “instantiation”.

When we create a class we specify the data and the code that constitute that class. Collectively these elements are called members of that class. Specifically the data defined by that class are referred to as member variables or instance variables or attributes. The code that operates on that data is referred to as member member methods or methods.

Syntax:

<ClassName> <object reference> = new <ClassName()>;

2.Static and Non-static data members:

Static members:

A member that is declared with static keyword inside a class and outside of all methods is called static member.

The use of static member is to provide one copy memory for storing values and executing logic common to all objects and all methods of a class.

Java supports 5 different static members. They are:

1.static variable

2.static block

3.static method

4.main method

5.Static inner class.

Example of using static members:

```
import static java.lang.System;                          static import statement.
   class Test{
        static int a;                                    static variable

        static {                                         static block
                ---------
                ----------
        }

        static void m1() {                               static method
                ---------
                ---------
        }

        public static void main(String[] args){          main method
                -------------
                -------------
        }

        static class A{                                  static inner class.
                ---------
                ----------
        }
  }
```

The above example defines the static members

1.Static import statement: A import statement that has static keyword is called static import.

- It is used for accessing a static variable and static method of one class from other packaged class without using class name. It is java 5 new feature.

2.Static Variable: A variable that is declared inside a class outside of methods with the keyword static is called static variable.

- It is used for providing one copy memory common to all objects and all members of a class.
- Every static variable , both primitive & referenced, are provided memory at the time of class loaded into JVM by class loader

3.Static Block: A name less block that is created inside a class outside of all methods by having static keyword in its prototype is called static block.

- A Static block is used for executing some logic at the time of class loaded in to JVM only once. This logic can be either SV's initialization logic or some other logic which is required to execute only once in the life time of a class.

4.Static Method: A method that is declared inside a class with static keyword is called static method.

- It is used for developing either initializing logic, modification, printing logic or some business logic by using the static variables , parameters and local variable common to all objects of this class.
- If we created a method as static(), we can call it and execute without creating object.

5.Main Method: Main method is a user defined method with predefined prototype syntax.

- It is used for starting class logic execution as a stand alone application.
- Main method is a mediator method between JVM and java programmer.

6.Static Inner Class: A class level inner class that has static keyword in its declaration us called Static Inner Class.

- It is used for creating an inner object common to all instances of an outer object.

Non-Static Members:

A member that is created or declared inside a class and outside of all methods without using static keyword is called non-static members.

The use of non-static member is to provide separate copy of memory for each object separately for storing specific values of this object and performing operations specific to one object by using this object values.,

Java supports 5 different non-static members.

1.non-static variable/ instance variable

2.non-static block/ instance initializer block

3.constructor.

4.non-static method/instance method

5.non-static inner class/instance inner class.

Example of using Non-static members:

```
class Example{
    int a = 10;                                  Non-static variable

    {                                            Non-static block
        ---------
        ----------
    }

    Example{                                     Constructor
        -------------
        -------------

    }
```

```
        void m1() {                                    Non-static method
                ---------
                ---------
        }
        class A{                                       Non-static inner class.
                ---------
                ---------
         }
}
```

1.Non-Static Variable: A variable that is created inside a class and outside a method (or) a constructor

(or) a block without static keyword is called "*Non-Static Variable".*

- A non-static variable is used for allocating memory separately for every object in its instance to share this object specific value.

2.Non-Static Block: A name less block that is created inside a class outside of all method is called *"Non-static block".*

3.Constructor: A constructor is kind of method whose name is same as class name and doesn't have any return type.

- Both non-static block & constructor are used for initializing an object (non-static variables inside an instance) with this objects specific values given by user.
- Non-static block & constructor are also used for executing some logic only once at the time of every new object creation. It means NSB & constructor are executed for every object creation.

4.Non-Static Method: A method that is created inside a class without using static keyword is called a *"Non-Static Method"*.

- Non-static method is used for developing an initialization logic (or) a business logic common to all objects, but to executing this logic specific to one object with that object data available in its instance.

5.Non-Static Inner Class: A class that is created inside another class without using static keyword is called a *"Non-Static Inner Class"*.

- A non-static inner class is also called as member inner class.
- A non-static inner class is used for representing an inner object of an outer object and for creating its instance with the association of outer class instance.

5.CONSTRUCTORS & TYPES OF CONSTRUCTORS

Constructor: A constructor is kind of method whose name is same as class name and doesn't have any return type.

- In java, a constructor is used only for initializing object which is created by "new" keyword.

Rules in Creating a Constructor:

1. Its name must be same as the class name
2. It must not have any return type, even void also not allowed.
3. It doesn't allow all the 8 execution level modifiers.
4. It allows all the 4 Accessibility modifiers
5. In logic section we can't use the return value.

6. We must call it by using new keyword. We cannot call it directly by its name as method, compiler will throw error, because constructor is meant for initializing object, so it must be called with new keyword.

Test Cases:

```
class Example{
    //Ex(){}                    CE:invalid method declaration,return type required.
    Void Example(){}            //no CE. But it is consider as a method.
    //Static Example(){}        CE. Modifier static is not allowed here.
    //Final Example(){}         CE. Modifier static is not allowed here
    //abstract Example(){}      CE. Modifier static is not allowed here
    //native Example(){}        CE. Modifier static is not allowed here
    //volatile Example(){}      CE. Modifier static is not allowed here
    //synchronized Example(){}  CE. Modifier static is not allowed here
    //strictfp Example(){}      CE. Modifier static is not allowed here
```

Types Of Constructors:

Java supports 3 types of constructors

1. Default Constructor(DC)

    -It is compiler generated constructor at the time of compilation in .class file.

2.Non-Parametrized Constructor(NPC)/no param/no-arg/0-param/nullary constructor

3.Parametrized Constructor(PC)

-NPC AND PC are programmer defined constructor

Syntax:

1.Default Constructor class:

```
class Example{
                                        DC
}
```

2.Non-Parametrized Constructor:

```
class Example{

    Example(){              NPC

    }
}
```

3.Parametrized Constructor:

```
    class Example{

        Example(int a, int b){          PC

        }
    }
```

6.MEHTOD OVERLOADING

***Def*:**

Defining multiple methods(either static or non-static) with same name but with different parameters type or list or order is called *"method overloading"*

Syntax:

```
void m1() { }
void m1(int i) { }
void m1(float f1) { }
void (String s1) { }
void m1(String s1,String s2) { }
void m1(int i1,float f1) { }
void m1(float f1,int i1) { }
```

7.TYPES OF CLASSES

(A) Top level class:

A top level class is a top level block that is used for grouping variables and methods and inner classes for developing logic.

Syntax of a class:

```
<Modifiers> class <ClassName>{

    //body of the class.
```

<Constructor>

<static members>

<Non-static members>

<nested inner classes>

}

B)Class With In Class(Inner classes):

The class defined inside another class or interface is called inner class. We can also create an interface in another class or interface.

Eg:

```
class Example{

    class Sample{

    }
}
```

Need of Inner Classes:

Inner classes are used to create an object logically inside another object with clear separation of properties region. So then we know that the object belongs to which object.

Eg:

```
class Student{
```

```
        int sno;
        String sname:

        class Address{
            int hno;
            String city;
        }
    }
```

Types of Inner classes:

1.Member class
   I. Static inner class
   II. Non-Static inner class
2.Method local class (local class)
3.Anonymous class (argument inner class)

Syntax to create all above types of inner classes:

```
//Example.java                          //javac Example.java

class Example{                          Example.class

    static class A { }                  Example$A.class
        class B { }                     Example$B.class
```

```
        void m1() {

            class C { }                    Example$1C.class

            m2(new Thread() {});           Example$1.class
    }
```

Compiler generates .class file separately for every inner class as shown above . So that the inner class logic is completely separated from outer class and stored in another .class file.

1.For Static and Non-Static inner class:

outerClassName$innerClassName.class

- *Example$A.class*
- *Example$B.class*

2.For Method Local inner class:

outerClassName$<n>innerClassName.class

- *Example$1C.class*

3.For Anonymous class:

outerClassName$<n>.class

- *Example$1C.class*

where 'n' is an index number starts with 1, incremented by 1 only if class is repeated in another method.

Let us understand about these inner classes:

1.Static inner class:

The inner class defined at class level with static keyword is called *"static inner class"*.

Syntax:

```
class Example{
        static class A {}
}
```

Allowed Modifiers:

- private
- protected
- public
- final
- abstract
- strictfp

Types of members allowed:

1) static variable
2) static block
3) static method
4) main method
5) non-static variable
6) non-static block
7) non-static method
8) constructor

2.Non-Static inner class:

The inner class defined at class level without static keyword is called *"non-static inner class"*.

We must develop non-static inner class to create a separate object in outer class object. So that when an outer class is initiated this inner class members are provided memory as part of every outer class instance

Syntax:

```
class Example{

        class B {}
}
```

Allowed Modifiers:

- private
- protected
- public
- final
- abstract
- strictfp

Types of members allowed:

1) non-static variable
2) non-static block
3) non-static method
4) constructor

3.Method local inner class:

The inner class defined inside a method of outer class is called *"method local class"*.

Syntax:

```
class Example{

    void m1() {

        class B {}
    }
}
```

Allowed Modifiers:

- final
- abstract
- strictfp

Types of members allowed:

1) non-static variable
2) non-static block
3) non-static method
4) constructor

4.Anonymous inner class:

Anonymous inner class is one type of inner class. It is a nameless subclass of a class/interface. Like other inner classes it is not individual class, it is a subclass of some other existed class or interface.

Using anonymous inner class we can do 3 things at a time.

1. Inner class creation as a subclass of outer class.
2. Overriding outer class method.
3. Creating and sending its object as argument or return type to another method.

Syntax:

```
new outer ClassName{

        //overriding outer class methods
}
```

Allowed Modifiers:

No modifiers are allowed

Types of members allowed:

1) non-static variable
2) non-static block
3) non-static method

8.ASSOCIATION, COMPOSITION , AGGREGATION

Association:

Association refers to the relationship between multiple objects. It refers to how objects are related to each other and how they are using each others functionality.

Composition and aggregation are two types of association.

Example for Association: [Has-a relationship]

```
//Employee.class

class Employee {

    String name;
    Address address;

}
```

```
//Address.class

class Address {

    String address;

}
```

Composition:

The composition is the strong type of association. An association is said to composition, if an object owns another object and another object cannot exists without the owner object.

Eg: Consider the case of Human having heart. Here Human object contains the heart & heart can't exist without Human.

Example for Composition:

```
//Car.class
//car must have engine.
```

```
//Engine.class
//Engine object
```

```
public class Car {
                                                    class Engine { }
//engine is a mandatory part of car
        private final Engine engine;

        public car() {
                engine = new engine();
        }

}
```

Aggregation:

Aggregation is a week association. An association is said to aggregation, if both objects can exist independently.

Eg: A team object & a player object. The team contains multiple players but a player can exist without a team

Example for Aggregation:

```
//Team.class                                    //Player.class
                                                //player object
public class Team {
                                                class Player { }
//players can be 0 or more.

        private List players;
```

```
        public Team() {
                players = new ArrayList();
        }

}
```

9.PACKAGES:

Package: A folder that is linked with java class is called a *"package"*.

Need of a Package: It is used to group related classes, interfaces and enums. Also it is used to separate new classes from existed classes. So by using packages we can create multiple classes with the same name, also we can create user defined classes with predefined class names.

Creation of a Package: To create a package we have a keyword called "package".

Syntax:

```
package <package_name>;
```

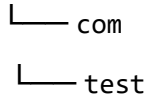
for eg: *package p1;*

Rule on creating a package statement:

package statement should be the first statement in a java file and also The package name must be unique.

Java uses file system directories to store packages. Let's create a Java file inside another directory.

For example:



Now, edit file, and at the beginning of the file, write the package statement as:

package com.test;

Here, any class that is declared within the test directory belongs to the com.test package.

Eg:

```
package com.test;
class Test {
        public static void main(String[] args){
                System.out.println("Hello World!");
        }
}
```

Importing classes from a package:

Java has an import statement that allows you to import an entire package or use only certain classes and interfaces defined in the package.

The general form of import statement is:

```
import package.name.ClassName;   // To import a certain class only
import package.name.*   // To import the whole package
```

For example,

```
import java.util.Date;          // imports only Date class
import java.io.*;               // imports everything inside java.io package
```

The import statement is optional in Java.

If you want to use class/interface from a certain package, you can also use its fully qualified name, which includes its full package hierarchy.

Example to import a package using the import statement.

```
import java.util.Date;

class MyClass implements Date {
        // body
}
```

Creating Jar files:

jar stands for java Achieve. It is a java based compressed file. It is used for grouping java library files(i.e., class files) for distributing library files as a part of software.

To create a .jar file, we can use jar cf command in the following way:

jar cf jarfilename inputfiles

Here, cf represents create the file. For example , assuming our package pack is available in C:\directory, to convert it into a jar file into the pack.jar , we can give the command as:

C:\> jar cf pack.jar pack

Now, pack.jar file is created

Viewing a JAR file: To view the contents of .jar files, we can use the command as:

jar tf jarfilename

Here , tf represents table view of file contents. For example, to view the contents of our pack.jar file , we can give the command:

C:/> jar tf pack.jar

Now, the contents of pack.jar are displayed as:

META-INF/

META-INF/MANIFEST.MF

pack/

pack/class1.class

pack/class2.class

..

..

where class1, class2 etc are the classes in the package pack. The first two entries represent that there is a manifest file created and added to pack.jar. The third entry represents the sub-directory with the name pack and the last two represent the files name in the directory pack.

When we create. .jar files , it automatically receives the default manifest file. There can be only one manifest file in an archive , and it always has the path name.

META-INF/MANIFEST.MF

This manifest file is useful to specify the information about other files which are packaged.

10.WRAPPER CLASS

Def: The classes which are used to represent primitive values as objects are called *"wrapper classes"*.

In java.lang package we have 8 wrapper classes one per each primitive type to represent them as an object.

Among them 6 wrapper classes represent number type values these wrapper classes are called as number wrapper classes. These 6 number wrapper classes are subclasses of a class Number which is an abstract class.

Primitive Types	**Wrapper classes**	**Number**
byte	Byte	public byte byteValue() {}
short	Short	public short shortValue() {}
int	Integer	public abstract int intValue();
long	Long	public abstract long longValue();
float	Float	public abstract float floatValue();
double	Double	public abstract double doubleValue();
char	Char	public char charValue() {}
boolean	Boolean	public boolean booleanValue() {}
void	Void	

Example:

```
public class Main {
        public static void main(String [] args) {
                Integer myInt = 5;
                Double myDouble = 5.99;
                Character myChar = 'A';
                System.out.println(myInt);
                System.out.println(myDouble);
                System.out.println(myChar);
                }
        }
```

MUTABLE AND IMMUTABLE OBJECTS

Java is an object-oriented programming language. As it is an object-oriented programming language, it's all methods and mechanism revolves around the objects. One object-based concept is mutable and immutable in Java. Objects in Java are either mutable or immutable; it depends on how the object can be iterated.

1.Mutable Objects:

The mutable objects are objects whose value can be changed after initialization. We can change the object's values, such as field and states, after the object is created. For example, Java.util.Date, StringBuilder, StringBuffer, etc.

When we made a change in existing mutable objects, no new object will be created; instead, it will alter the value of the existing object. These object's classes provide methods to make changes in it.

The Getters and Setters (get() and set() methods) are available in mutable objects. The Mutable object may or may not be thread-safe.

2.Immutable Objects:

The immutable objects are objects whose value can not be changed after initialization. We can not change anything once the object is created. For example, primitive objects such as int, long, float, double, all legacy classes, Wrapper class, String class, etc.

In a nutshell, immutable means unmodified or unchangeable. Once the immutable objects are created, its object values and state can not be changed.

Only Getters (get() method) are available not Setters (set() method) for immutable objects.

Creating classes for mutable and immutable objects:

Creating mutable objects:

The following two things are essential for creating a mutable class:

•Methods for modifying the field values

•Getters and Setters of the objects

Eg:

```
public class Example {
        private String s;
        Example(String s) {
                this.s = s;
        }
        public String getName() {
                return s;
        }
        public void setName(String coursename){
                this.s = coursename;
        }
        public static void main(String[] args) {
                Example obj = new Example("Training Session");
                System.out.println(obj.getName());
                // Here, we can update the name using the setName method.
                obj.setName("Java Training");
                System.out.println(obj.getName());
        }
}
```

output: *Training Session*

Java Training

Creating an Immutable Class:

The following things are essential for creating an immutable class:

1.Final class, which is declared as final so that it can't be extended.

2.All fields should be private so that direct access to the fields is blocked.

3.No Setters

4.All mutable fields should be as final so that they can not be iterated once initialized.

Eg:

```
public class Example1 {
        private final String s;
        Example1(final String s) {
                this.s = s;
        }
        public final String getName() {
        return s;
        }
        public static void main(String[] args) {
                Example obj = newExample("Core Java Training");
                System.out.println(obj.getName());
        }
}
```

output: *Core Java Training*

Differences between Mutable & Immutable Objects:

Mutable	Immutable
We can change the value of mutable objects after initialization.	Once an immutable object is initiated; We can not change its values.
The state can be changed.	The state can not be changed.
In mutable objects, no new objects are formed.	In immutable objects, a new object is formed when the value of the object is altered.
It provides methods to change the object.	It does not provide any method to change the object value.
It supports get() and set() methods to deal with the object.	It only supports get() method to pass the value of the object.
Mutable classes are may or may not be thread-safe.	Immutable classes are thread-safe.
The essentials for creating a mutable class are methods for modifying fields, getters and setters.	The essentials for creating an immutable class are final class, private fields, final mutable objects.

11.STRING, STRINGBUFFER, STRINGBUILDER, STRINGTOKENZER

(1) String: String is a sequence of characters placed in double quote(“ “).

- performing different operations on string data is called *“String Handling”*.
- In general String can be defined as it is a sequence of characters. Strings are constant; their values cannot be changed in the same memory after they are created.
- So String is defines as it is an *“Immutable sequence of characters”*.

Creating a String Object:

String object can be created in two ways:

1. By assigning string literal directly *EG: => String s = "abc";*
2. By using any one of the string class constructors.

String class has totally 15 constructors among them below are the 8 important constructors.

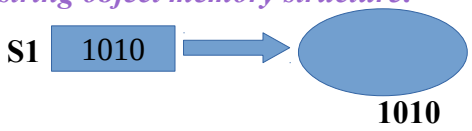
1. **string()**

   - Creates empty string object, not null string object.

For eg:

```
string s1 = new String();
System.out.println(s1);  //no output
```

string object memory structure:



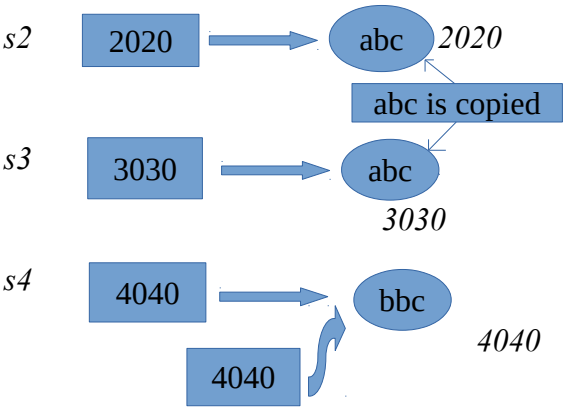
2. **String(String value)**

   - Creates String object with given String object characters. It performs String copy operation.

For eg:

```
String s2 = "abc";
//string copy
String s3 = new String(s2);
//creating string with direct string literal
String s4 = new String("bbc");
//String assignment
String s5 = s4;

System.out.println("s2: "+s2); //abc
System.out.println("s3: "+s3); //abc
System.out.println("s4: "+s4); //bbc
System.out.println(s2 == s3); //false
```



System.out.println(s4 == s5); //true

Note:

- ***In String copy*** two different String objects are created with same data, it is like a file copy operations in OS
- ***In String assignment*** current object reference is copied to destination variable, new object is not created

3. **String(StringBuffer sb)**

   - Creates new String object with the given StringBuffer object data. Performs string copy operation from stringBuffer object to string object.

4. **String(StringBuilder sb)**

   - Creates new String object with the given StringBuilder object data. Performs string copy operation from stringBuilder object to string object.

5. **String(char[] ch)**

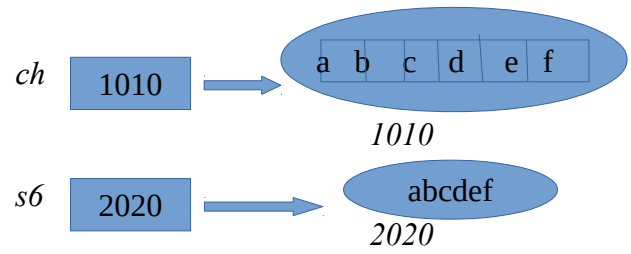
   - Creates String object with the given char array values. Performs string copy operation from char[] object to string object.

For eg:

char[] ch = {'a', 'b', 'c', 'd', 'e', 'f'};

String s6 = new String(ch);

System.out.println("s6: "+s6); //abcdef



6. **String(char[] ch, int offset, int count)**

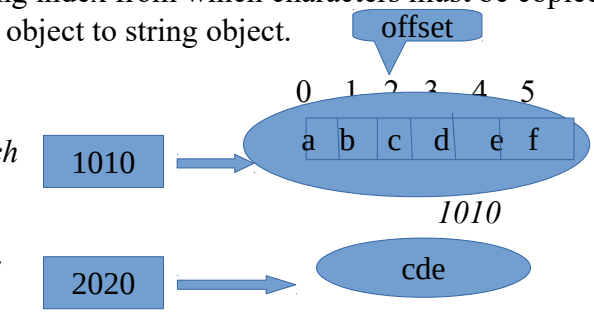
   - Creates new String object with the given count number of characters from the given offset in the char[] object. Here offset is the starting index from which characters must be copied. Performs string copy operation from char[] object to string object.

For eg:

char[] ch = {'a', 'b', 'c', 'd', 'e', 'f'};

String s7 = new String(ch,2,3);

System.out.println("s7: "+s7); //cde



2020

7. String(byte[] b)

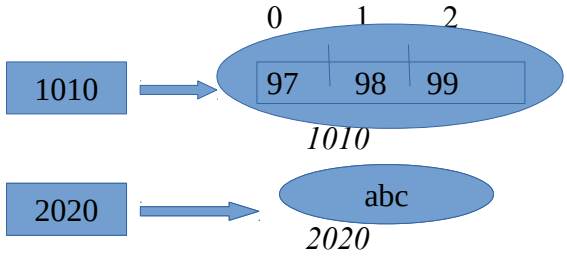
- Creates new String object by copying the given byte[] numbers by converting them into their ASCII characters

For eg:

byte[] b = {97,98,99};

String s8 = new String(b);

System.out.println(b); //abc



8. String(byte[] b, int offset, int count)

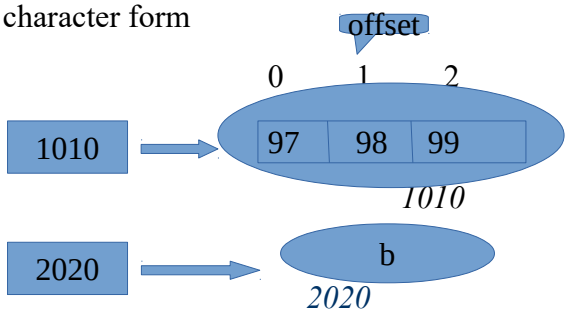
- Creates new String object with the given count number of bytes from the given off in the byte[]. All the bytes are stored in their ASCII character form

For eg:

byte[] b = {97,98,99};

String s9 = new String(b,1,1);

System.out.println("s9: "+s9); //b



Why are Strings in Java Immutable?

String in Java is a very special class, as it is used almost in every Java program. That's why it is Immutable to enhance performance and security. Let's understand it in detail:

In Java, strings use the concept of literals. Suppose we have an object having many reference variables. In such a scenario, if we will change the value of a reference variable, it will affect the entire object and all of its values.

Apart from the above reasons, the following reasons are also responsible for making the String immutable:

Immutable objects are very simple; there is no need for synchronization and are inherently thread-safe. But, Immutable objects make good building blocks for other objects, so we have to provide

them special care.

Most Developers make the String the final object so that it can not be altered later.

We can also say that because of String pooling concept.

(2) String Buffer:

It is a thread safe, mutable sequence of characters. A String buffer is like a string, but can be modified in the same memory location.

StringBuffer object van be created only in one way i.e., "by using its available constructor".

In StringBuffer class we have below 4 constructors:

1. **public StringBuffer()**

   - Creates empty StingBuffer object with default capacity I 16 buffers, it means it has 16 empty locations.

For eg:

```
StringBuffer sb1 = new StringBuffer();
System.out.println("sb1: "+sb1); //sb
System.out.println("sb1 capacity:" sb1.capacity()); //16
```

2. **public StringBuffer(int capacity)**

   - Creates empty StingBuffer object with given capacity.

For eg:

```
StringBuffer sb2 = new StringBuffer(3);
System.out.println("sb2: "+sb2); //sb2
System.out.println("sb2 capacity:" sb2.capacity()); //3
```

Rule: capacity value should be>=0, if we pass -ve number, JVM Throws *"Java.lang.NegativeArraySizeException"*

```
StringBuffer sb = new StringBuffer(-5); //RE: NASE
```

3. **public StringBuffer(String S)**

   - Creates StingBuffer object with given String object data. It performs String copy from String to StringBuffer object. The default capacity is [16+s.length()]

For eg:

```
StringBuffer sb3 = new StringBuffer("abc);
System.out.println("sb3: "+sb3); //sb3: abc
System.out.println("sb3 capacity:" sb3.capacity());  //19
```

4. **public StringBuffer(CharSequence cs)**

   - Creates new StingBuffer object with given cs object characters. It performs String copy from cs object to StringBuffer object. Its capacity is [16+cs.length()]

For eg:

```
StringBuffer sb4 = new StringBuffer("abc);
StringBuffer sb5 = new StringBuffer(sb4);
System.out.println("sb4: "+sb4); //sb4: abc
System.out.println("sb5: "+sb5); //sb5: abc
System.out.println("sb4 capacity:" sb4.capacity()); //19
System.out.println("sb5 capacity:" sb5.capacity()); //19
System.out.println(sb4 == sb5);  //false
```

Rule:

we cannot create String Buffer object by passing null.It leads to RE: NPE.

In StringBuffer case CE: ambiguous error is not raised for null argument because its constructors are overloaded with super and subclasses CharSequence and String.

For null argument String parameter constructor is called.

```
StringBuffer sb  = new StringBuffer(null); //RE:NPE
```

Methods used in stringBuffer: Some of the most used methods are:

1. **length() and capacity():** The length of a StringBuffer can be found by the length() method, while the total allocated capacity can be found by the capacity() method.

   Eg:

```
import java.io.*;
class GFG {
        public static void main(String[] args) {
                StringBuffer s = new StringBuffer("Training");
                int p = s.length();
                int q = s.capacity();
                System.out.println("Length of string Training=" + p);
                System.out.println("Capacity of string Training=" + q);
        }
}
```

Output:

Length of string Training=8

Capacity of string Training=24

2. **append( ):** It is used to add text at the end of the existence text. Here are a few of its forms:

   StringBuffer append(String str)

   StringBuffer append(int num)

   Eg:

```
import java.io.*;
class Example {
        public static void main(String[] args)
{
        StringBuffer s = new StringBuffer("Training");
        s.append("Session");
        System.out.println(s); // returns TrainingSession
```

```
        s.append(1);
        System.out.println(s); // returns TrainingSession
    }
}
```

Output:

TrainingSession

TrainingSession

3. **insert():** It is used to insert text at the specified index position. These are a few of its forms:

    StringBuffer insert(int index, String str)

    StringBuffer insert(int index, char ch)

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

Eg:

```
import java.io.*;
class Example {
    public static void main(String[] args){
        StringBuffer s = new StringBuffer("TrainingStudents");
        s.insert(8, "for");
        System.out.println(s); // returns TrainingforStudents

        s.insert(0, 5);
        System.out.println(s); // returns 5TrainingforStudents
```

```
                s.insert(3, true);
                System.out.println(s); // returns 5TrtrueainingforStudents
                s.insert(5, 41.35d);
                System.out.println(s); // returns 5Trai41.35ningforStudents

                s.insert(8, 41.35f);
                System.out.println(s); // returns 5Trai41.41.3535ningforStudents

                char geeks_arr[] = { 'p', 'a', 'w', 'a', 'n' };

                // insert character array at offset 9
                s.insert(2, geeks_arr);
                System.out.println(s); // returns 5Tpawanrai41.41.3535ningforStudents
        }
}
```

OUTPUT:

TrainingforStudents
5TrainingforStudents
5TrtrueainingforStudents
5Trai41.35ningforStudents
5Trai41.41.3535ningforStudents
5Tpawanrai41.41.3535ningforStudents

4. **reverse():** It can reverse the characters within a StringBuffer object using reverse().This method returns the reversed object on which it was called.

   Eg:

```
import java.io.*;
class Example {
        public static void main(String[] args){
```

```
                StringBuffer s = new StringBuffer("Training");
                s.reverse();
                System.out.println(s); // returns gniniarT
        }
    }
```

OUTPUT:

gniniarT

5. **delete() and deleteCharAt( ):** It can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).

   The delete( ) method deletes a sequence of characters from the invoking object. Here, start Index specifies the index of the first character to remove, and end Index specifies an index one past the last character to remove.

   Thus, the substring deleted runs from start Index to endIndex–1. The resulting StringBuffer object is returned. The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

   These methods are shown here:

   StringBuffer delete(int startIndex, int endIndex)

   StringBuffer deleteCharAt(int loc)

   Eg:

```
import java.io.*;
class Example {
    public static void main(String[] args){
        StringBuffer s = new StringBuffer("TrainingSession");
        s.delete(0, 5);
        System.out.println(s); // returns ingSession
        s.deleteCharAt(9);
        System.out.println(s); // returns ingSessio
    }
```

```
}
```

OUTPUT:

ingSession

ingSessio

6. **replace( ):** It can replace one set of characters with another set inside a StringBuffer object by calling replace( ). The substring being replaced is specified by the indexes start Index and endIndex. Thus, the substring at start Index through endIndex–1 is replaced. The replacement string is passed in str.The resulting StringBuffer object is returned.

   Its signature is shown here:

   StringBuffer replace(int startIndex, int endIndex, String str)

   Eg:

```
import java.io.*;
class Example {
    public static void main(String[] args){
        StringBuffer s = new StringBuffer("TrainingforStudents");
        s.replace(8,11, "are");
        System.out.println(s); // returns  TrainingareStudents
    }
}
```

OUTPUT:

TrainingareStudents

7. **ensureCapacity( ):** It is used to increase the capacity of a StringBuffer object. The new capacity will be set to either the value we specify or twice the current capacity plus two (i.e. capacity+2), whichever is larger. Here, capacity specifies the size of the buffer.

   Syntax:

```
void ensureCapacity(int capacity)
```

   Besides that all the methods that are used in String class can also be used.

8. **StringBuffer appendCodePoint(int codePoint):** This method appends the string representation of the codePoint argument to this sequence.

   Syntax:

   public StringBuffer appendCodePoint(int codePoint)

9. **char charAt(int index):** This method returns the char value in this sequence at the specified index.

   Syntax:

   public char charAt(int index)

10. **IntStream chars():** This method returns a stream of int zero-extending the char values from this sequence.

    Syntax:

    public IntStream chars()

11. **int codePointAt(int index):** This method returns the character (Unicode code point) at the specified index.

    Syntax:

    public int codePointAt(int index)

11. **int codePointBefore(int index):** This method returns the character (Unicode code point) before the specified index.

    Syntax:

    public int codePointBefore(int index)

12. **int codePointCount(int beginIndex, int endIndex):** This method returns the number of Unicode code points in the specified text range of this sequence.

Syntax:

public int codePointCount(int beginIndex, int endIndex)

13. **IntStream codePoints():** This method returns a stream of code point values from this sequence.

Syntax:

public IntStream codePoints()

14. **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin):** In this method, the characters are copied from this sequence into the destination character array dst.

Syntax:

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

15. **int indexOf(String str):** This method returns the index within this string of the first occurrence of the specified substring.

Syntax:

public int indexOf(String str)

public int indexOf(String str, int fromIndex)

16. **int lastIndexOf(String str):** This method returns the index within this string of the last occurrence of the specified substring.

Syntax:

public int lastIndexOf(String str)

public int lastIndexOf(String str, int fromIndex)

17. **int offsetByCodePoints(int index, int codePointOffset):** This method returns the index within this sequence that is offset from the given index by codePointOffset code points.

Syntax:

public int offsetByCodePoints(int index, int codePointOffset)

18. **void setCharAt(int index, char ch)**: In this method, the character at the specified index is set to ch.

Syntax:

public void setCharAt(int index, char ch)

19. **void setLength(int newLength):** This method sets the length of the character sequence.

Syntax:

public void setLength(int newLength)

20. **CharSequence subSequence(int start, int end):** This method returns a new character sequence that is a subsequence of this sequence.

Syntax:

public CharSequence subSequence(int start, int end)

21. **String substring(int start)**: This method returns a new String that contains a subsequence of characters currently contained in this character sequence.

Syntax:

public String substring(int start)

public String substring(int start, int end)

22. **String toString():** This method returns a string representing the data in this sequence.

Syntax:

public String toString()

23. void trimToSize(): This method attempts to reduce storage used for the character sequence.

Syntax:

public void trimToSize()

(3) StringBuilder:

The StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.

The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters.

However the StringBuilder class differs from the StringBuffer class on the basis of synchronization. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does.

Therefore this class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.

Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used.

Class Hierarchy:

java.lang.Object

↳*java.lang*

↳*Class StringBuilder*

Syntax:

public final class StringBuilder

extends Object

implements Serializable, CharSequence

Constructors in Java StringBuilder:

- **StringBuilder():**Constructs a string builder with no characters in it and an initial capacity of 16 characters.

- **StringBuilder(int capacity):**Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.

- **StringBuilder(CharSequence seq):**Constructs a string builder that contains the same characters as the specified CharSequence.

- **StringBuilder(String str):**Constructs a string builder initialized to the contents of the specified string.

Eg:

```
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;
public class GFG1 {
        public static void main(String[] argv) throws Exception {
                // create a StringBuilder object using StringBuilder() constructor
                StringBuilder str = new StringBuilder();
                str.append("GFG");
                // print string
                System.out.println("String = "+ str.toString());
                // create a StringBuilder object using StringBuilder(CharSequence) constructor
                StringBuilder str1 = new StringBuilder("AAAABBBCCCC");
                // print string
                System.out.println("String1 = "+ str1.toString());
                // create a StringBuilder object using StringBuilder(capacity) constructor
```

```
                StringBuilder str2 = new StringBuilder(10);
                // print string
                System.out.println("String2 capacity = "+ str2.capacity());
                // create a StringBuilder object using StringBuilder(String) constructor
                StringBuilder str3 = new StringBuilder(str1.toString());
                // print string
                System.out.println("String3 = "+ str3.toString());
        }
    }
```

Output:

String = GFG

String1 = AAAABBBCCCC

String2 capacity = 10

String3 = AAAABBBCCCC

Methods in Java StringBuilder:

1.StringBuilder append(X x): This method appends the string representation of the X type argument to the sequence

2.StringBuilder appendCodePoint(int codePoint): This method appends the string representation of the codePoint argument to this sequence.

3.int capacity(): This method returns the current capacity.

4.char charAt(int index): This method returns the char value in this sequence at the specified index.

5.IntStream chars(): This method returns a stream of int zero-extending the char values from this sequence.

6.int codePointAt(int index): This method returns the character (Unicode code point) at the specified index.

7.int codePointBefore(int index): This method returns the character (Unicode code point) before the specified index.

8.int codePointCount(int beginIndex, int endIndex): This method returns the number of Unicode code points in the specified text range of this sequence

9.IntStream codePoints(): This method returns a stream of code point values from this sequence.

10.StringBuilder delete(int start, int end): This method removes the characters in a substring of this sequence.

11.StringBuilder deleteCharAt(int index): This method removes the char at the specified position in this sequence.

12.void ensureCapacity(int minimumCapacity): This method ensures that the capacity is at least equal to the specified minimum.

13.void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin): This method characters are copied from this sequence into the destination character array dst.

14.int indexOf(): This method returns the index within this string of the first occurrence of the specified substring.

15.StringBuilder insert(int offset, boolean b): This method inserts the string representation of the booalternatelean argument into this sequence.

16.StringBuilder insert(): This method inserts the string representation of the char argument into this sequence.

17.int lastIndexOf(): This method returns the index within this string of the last occurrence of the specified substring

18.int length(): This method returns the length (character count).

19.int offsetByCodePoints(int index, int codePointOffset): This method returns the index within this sequence that is offset from the given index by codePointOffset code points.

20.StringBuilder replace(int start, int end, String str): This method replaces the characters in a substring of this sequence with characters in the specified String.

21.StringBuilder reverse(): This method causes this character sequence to be replaced by the reverse of the sequence.

22.void setCharAt(int index, char ch): In this method, the character at the specified index is set to ch.

23.void setLength(int newLength): This method sets the length of the character sequence.

24.CharSequence subSequence(int start, int end): This method returns a new character sequence

that is a subsequence of this sequence.

25.String substring(): This method returns a new String that contains a subsequence of

26.String toString(): This method returns a string representing the data in this sequence.

27.void trimToSize(): This method attempts to reduce storage used for the character sequence.

Eg using SB methods:

```
// methods of StringBuilder
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;
public class GFG1 {
    public static void main(String[] args)throws Exception
    {
    // create a StringBuilder object with a String pass as parameter
    StringBuilder str = new StringBuilder("AAAABBBCCCC");
    // print string
    System.out.println("String = "+ str.toString());
    // reverse the string
    StringBuilder reverseStr = str.reverse();
    // print string
    System.out.println("Reverse String = "+ reverseStr.toString());
    // Append ', '(44) to the String
    str.appendCodePoint(44);
    // Print the modified String
    System.out.println("Modified StringBuilder = "+ str);
    // get capacity
    int capacity = str.capacity();
    // print the result
```

```
        System.out.println("StringBuilder = " + str);
        System.out.println("Capacity of StringBuilder = "+ capacity);
    }
}
```

Output:

String = AAAABBBCCCC

Reverse String = CCCCBBBAAAA

Modified StringBuilder = CCCCBBBAAAA,

StringBuilder = CCCCBBBAAAA,

Capacity of StringBuilder = 27

(4) StringTokenzer:

The java.util.StringTokenizer class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

Constructors of StringTokenizer class:

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimeter.
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Eg:

```
import java.util.StringTokenizer;
public class Example{
    public static void main(String args[]){
```

```
                StringTokenizer st = new StringTokenizer("my name is Brahmini"," ");
                while (st.hasMoreTokens()){
                        System.out.println(st.nextToken());
                }
        }
}
```

OUTPUT:

my

name

is

Brahmini

Methods of StringTokenizer class:

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

Eg using one of the method:

```
import java.util.*;
public class Test {
        public static void main(String[] args) {
        StringTokenizer st = new StringTokenizer("my,name,is,Brahmini");
```

```
        //printing next token
        System.out.println("Next token is : " + st.nextToken(","));
        }
    }
```

OUTPUT:

Next token is : my

12.ARRAYS, SINGLE DIM, TWO DIM, MULTIPLE DIM ARRAYS

(1) Arrays:

An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

- In Java all arrays are dynamically allocated.(discussed below)
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The size of an array must be specified by an int or short value and not long.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Array can contain primitives (int, char, etc.) as well as object (or non-primitive) references of a class depending on the definition of the array. In case of primitive data types, the actual values are stored in contiguous memory locations. In case of objects of a class, the actual objects are stored in heap segment.

33	56	10	64	72	5
0	1	2	3	4	5

=>Array indices

Array length = 6

first index = 0

last index = 5

Creating, Initializing, and Accessing an Array:

(2) One-Dimensional Arrays :

The general form of a one-dimensional array declaration is

Syntax:

type var-name[];

OR

type[] var-name;

An array declaration has two components: the type and the name.

Type declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc. or user-defined data types (objects of a class). Thus, the element type for the array determines what type of data the array will hold.

Example:

```
// both are valid declarations
int intArray[];
or int[] intArray;

byte byteArray[];
short shortsArray[];
boolean booleanArray[];
long longArray[];
float floatArray[];
double doubleArray[];
char charArray[];
```

```
// an array of references to objects of the class MyClass (a class created by user)
MyClass myClassArray[];
Object[]  ao,      // array of Object
Collection[] ca;  // array of Collection of unknown type
```

Although the first declaration above establishes the fact that intArray is an array variable, no actual array exists. It merely tells the compiler that this variable (intArray) will hold an array of the integer type. To link intArray with an actual, physical array of integers, you must allocate one using new and assign it to intArray.

Instantiating an Array in Java:

When an array is declared, only a reference of array is created. To actually create or give memory to array, you create an array like this:The general form of new as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and var-name is the name of array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate.

Example:

```
int intArray[];    //declaring array
intArray = new int[20];  // allocating memory to array
                OR
int[] intArray = new int[20]; // combining both statements in one
```

Note:

1. The elements in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference types). Refer Default array values in Java
2. Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

Array Literal:

In a situation, where the size of the array and variables of array are already known, array literals

can be used.

Syntax:

```
int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
// Declaring array literal
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java

Accessing Java Array Elements using for Loop:

Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1.

All the elements of array can be accessed using Java for Loop.

Syntax:

```
// accessing the elements of the specified array
for (int i = 0; i < arr.length; i++){
        System.out.println("Element at index " + i +   " : "+ arr[i]);
}
```

Eg:

```
class Example {
    public static void main (String[] args) {
        // declares an Array of integers.
        int[] arr;

        // allocating memory for 5 integers.
        arr = new int[5];

        // initialize the first elements of the array
        arr[0] = 10;
```

```
                // initialize the second elements of the array
                arr[1] = 20;
                //so on...
                arr[2] = 30;
                arr[3] = 40;
                arr[4] = 50;

                // accessing the elements of the specified array
                for (int i = 0; i < arr.length; i++)
                System.out.println("Element at index " + i +" : "+ arr[i]);
        }
}
```

Arrays of Objects:

An array of objects is created just like an array of primitive type data items in the following way.

Syntax:

```
Student[] arr = new Student[7]; //student is a user-defined class
```

The student Array contains seven memory spaces each of size of student class in which the address of seven Student objects can be stored. The Student objects have to be instantiated using the constructor of the Student class and their references should be assigned to the array elements in the following way.

```
        Student[] arr = new Student[5];
```

Eg:

```
class Student {
        public int roll_no;
        public String name;
```

```
        Student(int roll_no, String name) {
                this.roll_no = roll_no;
                this.name = name;
        }
}

// Elements of the array are objects of a class Student.
public class College {
        public static void main (String[] args){
                // declares an Array of integers.
                Student[] arr;

                // allocating memory for 5 objects of type Student.
                arr = new Student[5];

                // initialize the first elements of the array
                arr[0] = new Student(1,"Brahmini");

                // initialize the second elements of the array
                arr[1] = new Student(2,"Sandeep");

                // so on...
                arr[2] = new Student(3,"Kavya");
                arr[3] = new Student(4,"Nandini");
                arr[4] = new Student(5,"Bhanu");

                // accessing the elements of the specified array
```

```
                for (int i = 0; i < arr.length; i++) {
                        System.out.println("Element at " + i + " : " +arr[i].roll_no
                                                                    +"arr[i].name);
                }
        }
}
```

Output:

Element at 0 : 1 Brahmini

Element at 1 : 2 Sandeep

Element at 2 : 3 Kavya

Element at 3 : 4 Nandini

Element at 4 : 5 Bhanu

What happens if we try to access element outside the array size?

JVM throws ArrayIndexOutOfBoundsException to indicate that array has been accessed with an illegal index. The index is either negative or greater than or equal to size of array.

Eg:

```
class Example {
        public static void main (String[] args){
                int[] arr = new int[2];
                arr[0] = 10;
                arr[1] = 20;
                for (int i = 0; i <= arr.length; i++){
                        System.out.println(arr[i]);
                }
```

```
        }
    }
```

Runtime error

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2 at Example.main(File.java:12)

(3) Two Dimensional & Multidimensional Arrays:

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Examples:

```
int[][] intArray = new int[10][20]; //a 2D array or matrix
int[][][] intArray = new int[10][20][10]; //a 3D array
```

Eg code:

```
class multiDimensional {
    public static void main(String args[]) {
    // declaring and initializing 2D array
    int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

    // printing 2D array
    for (int i=0; i< 3 ; i++) {
        for (int j=0; j < 3 ; j++) {
            System.out.print(arr[i][j] + " ");
```

```
                }
                System.out.println();
            }
        }
}
```

Output:

2 7 9

3 6 1

7 4 2

Passing Arrays to Methods:

Like variables, we can also pass arrays to methods. For example, below program pass array to method sum for calculating sum of array's values.

Eg:

```
// Java program to demonstrate passing of array to method
class Test{
    // Driver method
    public static void main(String args[]) {
        int arr[] = {3, 1, 2, 5, 4};
        // passing array to method m1
```

```
                sum(arr);
        }

        public static void sum(int[] arr) {
                // getting sum of array values
                int sum = 0;

                for (int i = 0; i < arr.length; i++) {
                        sum+=arr[i];
                }
                System.out.println("sum of array values : " + sum);
        }
}
```

Output:

1 2 3

Class Objects for Arrays:

Every array has an associated Class object, shared with all other arrays with the same component type.

Eg:

```
// Java program to demonstrate Class Objects for Arrays
class Test{
        public static void main(String args[]){
```

```
                int intArray[] = new int[3];
                byte byteArray[] = new byte[3];
                short shortsArray[] = new short[3];

                // array of Strings
                String[] strArray = new String[3];

                System.out.println(intArray.getClass());
                System.out.println(intArray.getClass().getSuperclass());
                System.out.println(byteArray.getClass());
                System.out.println(shortsArray.getClass());
                System.out.println(strArray.getClass());
        }
}
```

Output:

```
class [I
class java.lang.Object
class [B
class [S
class [Ljava.lang.String;
```

Explanation :

1.The string "[I" is the run-time type signature for the class object "array with component type int".

2.The only direct superclass of any array type is java.lang.Object.

3.The string "[B" is the run-time type signature for the class object "array with component type byte".

4.The string "[S" is the run-time type signature for the class object "array with component type short".

5.The string "[L" is the run-time type signature for the class object "array with component type of a Class". The Class name is then followed.

Array Members:

Now as you know that arrays are object of a class and direct superclass of arrays is class Object. The members of an array type are all of the following:

•The public final field length, which contains the number of components of the array. Length may be positive or zero.

•All the members inherited from class Object; the only method of Object that is not inherited is its clone method.

•The public method clone(), which overrides clone method in class Object and throws no checked exceptions.

Cloning of arrays:

• When you clone a single dimensional array, such as Object[], a "deep copy" is performed with the new array containing copies of the original array's elements as opposed to references.

Eg:

```
// Java program to demonstrate cloning of one-dimensional arrays
class Test{
    public static void main(String args[]){
        int intArray[] = {1,2,3};
        int cloneArray[] = intArray.clone();
    // will print false as deep copy is created for one-dimensional array
    System.out.println(intArray == cloneArray);

    for (int i = 0; i < cloneArray.length; i++) {
```

```
                System.out.print(cloneArray[i]+" ");
            }
        }
}
```

Output:

false

1 2 3

- A clone of a multi-dimensional array (like Object[][]) is a “shallow copy” however, which is to say that it creates only a single new array with each element array a reference to an original element array, but subarrays are shared.

Eg:

```
// Java program to demonstrate cloning of multi-dimensional arrays
class Test{
    public static void main(String args[]){
        int intArray[][] = {{1,2,3},{4,5}};
        int cloneArray[][] = intArray.clone();

        // will print false
        System.out.println(intArray == cloneArray);

        // will print true as shallow copy is created i.e. sub-arrays are shared
        System.out.println(intArray[0] == cloneArray[0]);
        System.out.println(intArray[1] == cloneArray[1]);
    }
}
```

Output:

false

true

true

13.FOR EACH LOOP & NEW CHANGES WITH SWITH STATEMENTS

For each loop:

For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5.

•It starts with the keyword for like a normal for-loop.

•Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.

•In the loop body, you can use the loop variable you created rather than using an indexed array element.

•It's commonly used to iterate over an array or a Collections class (eg, ArrayList)

Syntax:

```
for (type var : array)
{
        statements using var;
}
```

is equivalent to:

```
for (int i=0; i<arr.length; i++)
{
         type var = arr[i];
        statements using var;
}
```

Eg:

```
// Java program to illustrate for-each loop
class For_Each {
	public static void main(String[] args) {
		int[] marks = { 125, 132, 95, 116, 110 };
		int highest_marks = maximum(marks);
		System.out.println("The highest score is " + highest_marks);
	}

	public static int maximum(int[] numbers){
		int maxSoFar = numbers[0];

		// for each loop
		for (int num : numbers)
		{
			if (num > maxSoFar)
			{
				maxSoFar = num;
			}
		}
	return maxSoFar;
	}
}
```

Output:

The highest score is 132

Limitations of for-each loop:

1.For-each loops are not appropriate when you want to modify the array:

```
for (int num : marks)
{
    // only changes num, not the array element
    num = num*2;
}
```

2.For-each loops do not keep track of index. So we can not obtain array index using For-Each loop

```
for (int num : numbers)
{
     if (num == target)
    {
            return ???;   // do not know the index of num
     }
}
```

3.For-each only iterates forward over the array in single steps

```
// cannot be converted to a for-each loop
for (int i=numbers.length-1; i>0; i--)
{
     System.out.println(numbers[i]);
}
```

4.For-each cannot process two decision making statements at once

```
// cannot be easily converted to a for-each loop
for (int i=0; i<numbers.length; i++)
{
    if (numbers[i] == arr[i])
     { .... }
}
```

New changes with switch statements:

Java SE 13 introduces the yield statement. It takes one argument, which is the value that the case label produces in a switch expression.

The yield statement makes it easier for you to differentiate between switch statements and switch expressions. A switch statement, but not a switch expression, can be the target of a break statement. Conversely, a switch expression, but not a switch statement, can be the target of a yield statement.

Using "break" statement:

Eg:

```
public enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
                  SATURDAY; }
int numLetters = 0;
Day day = Day.WEDNESDAY;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
    numLetters = 6;
    break;
    case TUESDAY:
    numLetters = 7;
    break;
    case THURSDAY:
    case SATURDAY:
    numLetters = 8;
    break;
    case WEDNESDAY:
     numLetters = 9;
    break;
    default:
        throw new IllegalStateException("Invalid day: " + day);
    }
```

```
System.out.println(numLetters);
```

Using "yield" statement:

Eg:

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> {
        System.out.println(6);
        yield 6;
    }
    case TUESDAY -> {
        System.out.println(7);
        yield 7;
    }
    case THURSDAY, SATURDAY -> {
        System.out.println(8);
        yield 8;
    }
    case WEDNESDAY -> {
        System.out.println(9);
        yield 9;
    }
    default -> {
        throw new IllegalStateException("Invalid day: " + day);
    }
};
```

14. VAR ARG

In JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called varargs and it is short-form for variable-length arguments. A method that takes a variable number of arguments is a varargs method.

Prior to JDK 5, variable-length arguments could be handled two ways. One using overloaded

method(one for each) and another put the arguments into an array, and then pass this array to the method. Both of them are potentially error-prone and require more code. The varargs feature offers a simpler, better option.

Syntax of varargs :

A variable-length argument is specified by three periods(…).

For Example,

```
public static void fun(int ... a)
{
 // method body
}
```

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here a is implicitly declared as an array of type int[].

Eg code:

```
// Java program to demonstrate varargs
class Test{
    // A method that takes variable number of integer arguments.
    static void fun(int ...a) {
        System.out.println("Number of arguments: " + a.length);
        // using for each loop to display contents of a
        for (int i: a) {
            System.out.print(i + " ");
            System.out.println();
        }
    }

    // Driver code
    public static void main(String args[])
    {
        // Calling the varargs method with different number of parameters
        fun(100);           // one parameter
        fun(1, 2, 3, 4); // four parameters
```

```
            fun();          // no parameter
        }
    }
```

Output:

Number of arguments: 1

100

Number of arguments: 4

1 2 3 4

Number of arguments: 0

Explanation of above program:

- The … syntax tells the compiler that varargs has been used and these arguments should be stored in the array referred to by a.
- The variable a is operated on as an array. In this case, we have defined the data type of a as int. So it can take only integer values. The number of arguments can be found out using a.length, the way we find the length of an array in Java.

Note: A method can have variable length parameters with other parameters too, but one should ensure that there exists only one varargs parameter that should be written last in the parameter list of the method declaration.

int nums(int a, float b, double … c)

In this case, the first two arguments are matched with the first two parameters and the remaining arguments belong to c.

Eg

```
// Java program to demonstrate varargs with normal arguments
class Test2 {
    // Takes string as a argument followed by varargs
    static void fun2(String str, int ...a) {
        System.out.println("String: " + str);
        System.out.println("Number of arguments is: "+ a.length);

        // using for each loop to display contents of a
```

```
                for (int i: a) {
                        System.out.print(i + " ");
                }
                System.out.println();
        }

        public static void main(String args[]) {
                // Calling fun2() with different parameter
                fun2("Training", 100, 200);
                fun2("CSPortal", 1, 2, 3, 4, 5);
                fun2("forStrudents");
        }
}
```

Output:

String: Training

Number of arguments is: 2

100 200

String: CSportal

Number of arguments is: 5

1 2 3 4 5

String: forStudents

Number of arguments is: 0

Important points:

- Vararg Methods can also be overloaded but overloading may lead to ambiguity.
- Prior to JDK 5, variable length arguments could be handled into two ways : One was using overloading, other was using array argument.
- There can be only one variable argument in a method.
- Variable argument (varargs) must be the last argument.