Java API Packages and imp. Classes

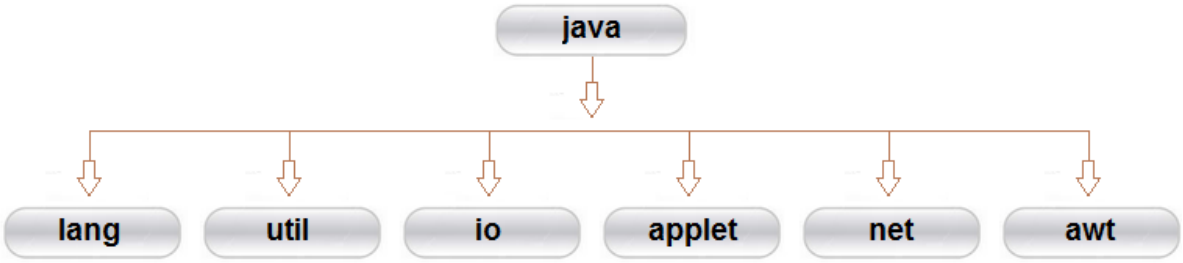
- A java **package is a group of similar types of classes, interfaces and sub-packages**. Think of it as a folder in a file directory. We can say that Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces.

Advantage of Java Package

- Java package is used **to categorize the classes and interfaces** so that they can be easily maintained.
- Java package **provides access protection.** protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Java package **removes naming collision**. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Packages can be considered as **data encapsulation (or data-hiding).**
- Packages are divided into two categories:
  1. Built-in Packages (packages from the Java API)
  2. User-defined Packages (create your own packages)

a. Built-in Packages (packages from the Java API)

- Java **API(Application Program Interface)** provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the the Java API. Following figure shows the system packages that are frequently used in the programs.



Java System Packages and Their Classes

java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.

java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for creating and implementing applets.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

- The Java **API is a library of prewritten classes, that are free to use, included in the Java Development Environment**.
- The library contains components for managing input, database programming, and much more.
- The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
- To use a class or a package from the library, you need to use the import keyword:

Syntax

```
import packagename.Class;   // Import a single class
import packagename.*;   // Import the whole package
```

1. java.lang package

- Provides classes that are **fundamental to the design of the Java programming language.**
- The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.
- The following are the classes under the java.lang package.

Class	Description
Integer	The Integer class wraps the int primitive data type into an object. This class includes helpful methods in converting value from string to Integer.
Math	The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

String	Java String class is represented by character strings.
String Buffer	The StringBuffer is a thread safe, mutable sequence of characters. Its much the same as String however StringBuffer can be modified.
Double	The Double class wraps a value of the primitive type double in an object. An object of type Double contains a single field whose type is double.
Number	The numeric wrapper classes are subclasses of Number Instance methods for converting between numeric types

A. java.lang.Number class

- The numeric wrapper classes are subclasses of Number Instance methods for converting between numeric types.
- The java.lang.Number class is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.
- The Subclasses of Number must provide methods to convert the represented numeric value to byte, double, float, int, long, and short.

Class Declaration

Syntax:

public abstract class Number extends Object implements Serializable

Methods	
Modifier and Type	**Method and Description**
byte	**byteValue**() Returns the value of the specified number as a byte.
abstract double	**doubleValue**() Returns the value of the specified number as a double.
abstract float	**floatValue**() Returns the value of the specified number as a float.
abstract int	**intValue**() Returns the value of the specified number as an int.
abstract long	**longValue**()

	Returns the value of the specified number as a long.
short	**shortValue**() Returns the value of the specified number as a short.

Constructor Detail

public Number()

Example:

```
public class NumberDemo {
  public static void main(String[] args) {
    // get a number as float
    Float x = new Float(123456f);
    // get a number as double
    Double y = new Double(9876);
    // print their value as int
    System.out.println("x as float:" + x + ", x as Integer:" + x.intValue());
    System.out.println("y as double:" + y + ", y as Integer:" + y.intValue());
  }
}
```

B. Java.lang.Math Class

- The java.lang.Math class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Class Declaration

Syntax:

public final class Math extends Object

Field:

Following are the fields for java.lang.Math class −

- **static double E** − This is the double value that is closer than any other to e, the base of the natural logarithms.
- **static double PI** − This is the double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

Basic Math Functions

- **Math.abs()**

- The Math.abs() function returns the absolute value of the parameter passed to it.
- The absolute value is the positive value of the parameter.
- If the parameter value is negative, the negative sign is removed and the positive value corresponding to the negative value without sign is returned.
- **Examples:**
  int abs1 = Math.abs(10); // abs1 = 10
  int abs2 = Math.abs(-20); // abs2 = 20
- The Math.abs() method is overloaded in 4 versions:
  - Math.abs(int)
  - Math.abs(long)
  - Math.abs(float)
  - Math.abs(double)

- **Math.ceil()**

- The Math.ceil() function rounds a floating point value up to the nearest integer value. The rounded value is returned as a double.
- **Example:**
  double ceil = Math.ceil(7.343); // ceil = 8.0

- **Math.floor()**

- The Math.floor() function rounds a floating point value down to the nearest integer value.
- The rounded value is returned as a double.
- **Example:**
  double floor = Math.floor(7.343); // floor = 7.0

- **Math.min()**

- The Math.min() method returns the smallest of two values passed to it as parameter
- **Example:**
  int min = Math.min(10, 20);

- **Math.max()**

- The Math.max() method returns the largest of two values passed to it as parameter.
- **Example:**
  int max = Math.max(10, 20);

- **Math.round()**

- The Math.round() method rounds a float or double to the nearest integer using normal math round rules (either up or down).
- **Example:**
  double roundedDown = Math.round(23.445);
  double roundedUp = Math.round(23.545);

- After executing these two Java statements the roundedDown variable will contain the value 23.0 , and the roundedUp variable will contain the value 24.0.

➢ **Math.random()**

- The Math.random() method returns a random floating point number between 0 and 1. Of course the number is not fully random, but the result of some calculation which is supposed to make it as unpredictable as possible.
- **Example:**
  double random = Math.random();

Exponential and Logarithmic Math Functions

➢ **Math.exp()**

- The Math.exp() function returns *e* (Euler's number) raised to the power of the value provided as parameter.
- **Example:**
  double exp1 = Math.exp(1);
  System.out.println("exp1 = " + exp1);
  double exp2 = Math.exp(2);
  System.out.println("exp2 = " + exp2);
  Output:
  exp1 = 2.718281828459045
  exp2 = 7.38905609893065

➢ **Math.log()**

- The Math.log() method provides the logarithm of the given parameter. The base for the logarithm is *i* (Euler's number). Thus, Math.log() provides the reverse function of Math.exp().
- **Example:**
  double log1 = Math.log(1);
  System.out.println("log1 = " + log1);
  double log10 = Math.log(10);
  System.out.println("log10 = " + log10);
  output:
  log1 = 0.0
  log10 = 2.302585092994046

➢ **Math.log10()**

- The Math.log10 method works like the Math.log() method except is uses 10 as is base for calculating the logarithm instead of *e* (Euler's Number).
- **Example:**
  double log10_1 = Math.log10(1);
  System.out.println("log10_1 = " + log10_1);
  double log10_100 = Math.log10(100);

System.out.println("log10_100 = " + log10_100);
Output:
log10_1 = 0.0
log10_100 = 2.0

- **Math.pow()**

- The Math.pow() function takes two parameters. The method returns the value of the first parameter raised to the power of the second parameter.
- **Example:**
  double pow2 = Math.pow(2,2);
  System.out.println("pow2 = " + pow2);
  double pow8 = Math.pow(2,8);
  System.out.println("pow8 = " + pow8);
  Output:
  pow2 = 4.0
  pow8 = 256.0

- **Math.sqrt()**

- The Math.sqrt() method calculates the square root of the parameter given to it.
- **Example:**
  double sqrt4 = Math.sqrt(4);
  System.out.println("sqrt4 = " + sqrt4);
  double sqrt9 = Math.sqrt(9);
  System.out.println("sqrt9 = " + sqrt9);
  Output:
  sqrt4 = 2.0
  sqrt9 = 3.0

Trigonometric Math Functions

- **Math.PI**

- The Math.PI constant is a double with a value that is very close to the value of PI - the mathematical definition of PI.

- **Math.sin()**

- The Math.sin() method calculates the sine value of some angle value in radians.
- **Example:**
  double sin = Math.sin(Math.PI);
  System.out.println("sin = " + sin);
  Output:
  Sin=1.224646.....

- **Math.cos()**

- The Math.cos() method calculates the cosine value of some angle value in radians.
- **Example:**

```
double cos = Math.cos(Math.PI);
System.out.println("cos = " + cos);
```

Output:

cos=1.224646.....

- **Math.tan()**

- The Math.tan() method calculates the tangens value of some angle value in radians.
- **Example:**

```
double tan = Math.tan(Math.PI);
System.out.println("tan = " + tan);
```

Output:

tan=-1.224646.....

- **Math.asin()**

- The Math.asin() method calculates the arc sine value of a value between 1 and 1.
- **Example:**

```
double asin = Math.asin(1.0);
System.out.println("asin = " + asin);
```

Output:

asin=1.570796.....

- **Math.acos()**

- The Math.acos() method calculates the arc cosine value of a value between 1 and -1.
- **Example:**

```
double acos = Math.acos(1.0);
System.out.println("acos = " + acos);
```

Output:

acos=0.0

- **Math.atan()**

- The Math.atan() method calculates the arc tangens value of a value between 1 and -1.
- **Example:**

```
double atan = Math.atan(1.0);
System.out.println("atan = " + atan);
```

Output:

atan=0.78539816....

- **Math.toRadians()**

- The Math.toRadians() method converts an angle in degrees to radians.
- **Example:**

```
double radians = Math.toRadians(180);
System.out.println("radians = " + radians);
```

Output:

radians=3.141....

➢ **Math.toDegrees()**

- The Math.toDegrees() method converts an angle in radians to degrees.
- **Example:**
  double degrees = Math.toDegrees(Math.PI);
  System.out.println("degrees = " + degrees);
  Output:
  degrees=180

C. Java.lang.String Class

- The java.lang.String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
- Strings are constant, their values cannot be changed after they are created. That is called immutable.

Class Declaration
Syntax:
public final class String extends Object implements Serializable, Comparable<String>, CharSequence

- By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

What is String in java

- Generally, String is a sequence of characters. But in Java, **string is an object that represents a sequence of characters.** The java.lang.String class is used to create a string object.

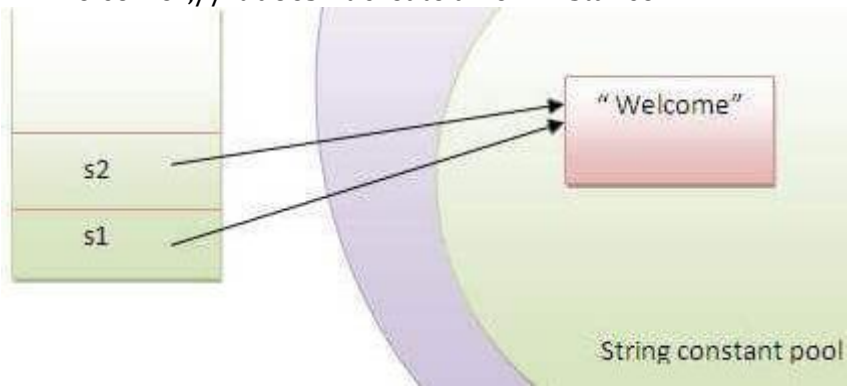
How to create a string object?
There are two ways to create String object:

1. By string literal
2. By new keyword

1. String Literal

- Java String literal is created by using double quotes.
- **Example:**
  String s="welcome";
- Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.
- **Example:**
  String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance



- String objects are stored in a special memory area known as the "string constant pool"

Why Java uses the concept of String literal?

- To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2. By new keyword

- **Example:**

  String s=**new** String("Welcome");//creates two objects and one reference variable
- In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Example:

```
public class StringMethodsDemo {
        public static void main(String[] args) {
                String targetString = "Java is fun to learn";
                String s1= "JAVA";
                String s2= "Java";
                String s3 = "  Hello Java  ";
                System.out.println("Char at index 2(third position): " +
targetString.charAt(2));
                System.out.println("After Concat: "+ targetString.concat("-Enjoy-"));
                System.out.println("Checking equals ignoring case: "
+s2.equalsIgnoreCase(s1));
                System.out.println("Checking equals with case: " +s2.equals(s1));
                System.out.println("Checking Length: "+ targetString.length());
                System.out.println("Replace function: "+ targetString.replace("fun", "easy"));
```

```
                System.out.println("SubString of targetString: "+ targetString.substring(8));
                System.out.println("SubString of targetString: "+ targetString.substring(8,
12));
                System.out.println("Converting to lower case: "+
targetString.toLowerCase());
                System.out.println("Converting to upper case: "+
targetString.toUpperCase());
                System.out.println("Triming string: " + s3.trim());
                System.out.println("searching s1 in targetString: " +
targetString.contains(s1));
                System.out.println("searching s2 in targetString: " +
targetString.contains(s2));
                char [] charArray = s2.toCharArray();
                System.out.println("Size of char array: " + charArray.length);
                System.out.println("Printing last element of array: " + charArray[3]);
        }}
```

D. Java.lang.Wrapper class

- The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.
- autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

char	Character

Use of Wrapper classes in Java

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
    - **Serialization** in Java is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only

Autoboxing

- The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing.
- For example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short. Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Primitive to Wrapper Example:

```
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}}
```

Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper to Primitive Example:

```
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally
System.out.println(a+" "+i+" "+j);
}}
```

E. Java.lang.StringBuffer class

- Java **StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.**
- Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Constructors:

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Methods:

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex, int	is used to return the substring from the specified beginIndex. and endIndex.

	endIndex)	

Example:

```
public class StringBufferExample {
        public static void main(String[] args) {
                StringBuffer sb = new StringBuffer("Hello");
                int sbLength = sb.length();
                int sbCapacity = sb.capacity();
                System.out.println("String Length of " + sb + " is " + sbLength);
                System.out.println("Capacity of " + sb + " is " + sbCapacity);
                sb.append("World ");
                System.out.println(sb);
                sb.insert(5, " ");
                sb.insert(sb.length(), 2017);
                System.out.println(sb);
                System.out.println(sb.reverse());
                System.out.println(sb.delete(5,11));
                StringBuffer sb2 = new StringBuffer("Hello World!");
                System.out.println(sb2.deleteCharAt(11));
                System.out.println(sb.replace(6,11,"Earth"));
        }
}
```

F. Java.lang.StringBuilder Class

- Java StringBuilder class is used to create mutable (modifiable) string.
- The Java **StringBuilder class is same as StringBuffer class except that it is non-synchronized. which means it is not thread safe. Its because StringBuilder methods are not synchronised.**

Constructors:

Constructor	Description
StringBuilder()	creates an empty string Builder with the initial capacity of 16.
StringBuilder(String str)	creates a string Builder with the specified string.
StringBuilder(int	creates an empty string Builder with the specified capacity as

length)	length.

Difference between String and StringBuffer

- There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Difference between StringBuffer and StringBuilder

- Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder.
- The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable.
- There are many differences between StringBuffer and StringBuilder.

No.	StringBuffer	StringBuilder
1)	StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than StringBuffer.

G. Java.lang.Package Class

- The **java.lang.Package class contain version information about the implementation and specification of a Java package.It provides methods such as getName(), getImplementationTitle(),getImplementationVendor(), getImplementationVersion() etc.**
- **Example:**

```
class PackageInfo{
public static void main(String args[]){
        Package p=Package.getPackage("java.lang");
        System.out.println("package name: "+p.getName());
        System.out.println("Specification Title: "+p.getSpecificationTitle());
        System.out.println("Specification Vendor: "+p.getSpecificationVendor());
        System.out.println("Specification Version: "+p.getSpecificationVersion());
        System.out.println("Implementaion Title: "+p.getImplementationTitle());
        System.out.println("Implementation Vendor: "+p.getImplementationVendor(
        ));
        System.out.println("Implementation Version: "+p.getImplementationVersion
        ());
        System.out.println("Is sealed: "+p.isSealed());
         }
}
```

2.Java util package

- Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

A. Java.util.random class

- class is part of java.util package.
- An instance of java Random class is **used to generate random numbers.**
- This class provides several methods to generate random numbers of type integer, double, long, float etc.

Constructors

- **Random():** creates new random generator
- **Random(long seed):** creates new random generator using specified seed

Java Random Class Methods

- **nextBoolean():** This method returns next pseudorandom which is a boolean value

from random number generator sequence.

- **nextDouble():** This method returns next pseudorandom which is double value between 0.0 and 1.0.
- **nextFloat():** This method returns next pseudorandom which is float value between 0.0 and 1.0.
- **nextInt():** This method returns next int value from random number generator sequence.
- **nextInt(int n):** This method return a pseudorandom which is int value between 0 and specified value from random number generator sequence.\

Example:

```
import java.util.Random;
public class RandomNumberExample {
        public static void main(String[] args) {
                Random random = new Random ();
                System.out.println(random.nextBoolean());
                System.out.println(random.nextDouble());
                System.out.println(random.nextFloat());
                System.out.println(random.nextInt());
                System.out.println(random.nextInt(20));
        }
}
```

B. Java.util. Vector Class

- Java Vector class comes under the java.util package. The vector class **implements a growable array of objects.** Like an array, it contains the component that can be accessed using an integer index.
- Vector is very useful if we don't know the size of an array in advance or we need one that **can change the size of array over the lifetime of a program**.
- Vector **implements a dynamic array that means it can grow or shrink as required. It is similar to the ArrayList, but with two differences-**
  - **Vector is synchronized for thread safety.**
  - **The vector contains many legacy methods that are not the part of a collections framework**
- **Class declaration**
- **Syntax:**

  public class Vector<E>extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

- Here <E> represents an Element, which could be any class.
- **Example:**

  ArrayList<Integer> list = new ArrayList<Integer>();
- **Arraylist is an ordered collection (by index), but not sorted**.
- To use the ArrayList class, you must use the following import statement:

  import java.util.ArrayList;
- Then, to declare an ArrayList, you can use the default constructor, as in the following example:

  ArrayList names = new ArrayList();
- The default constructor **creates an ArrayList with a capacity of 10 items.** The capacity of an ArrayList is the number of items it can hold without having to increase its size.

constructors

ArrayList names = new ArrayList(int size);
ArrayList names = new ArrayList(Collection c);

- You can also specify a capacity/size of initial ArrayList as well as create ArrayList from other collection types.

Advantages Array List has over arrays are

- It can grow dynamically.
- It provides more powerful insertion and search mechanisms than arrays.

Vector Class Constructors

- Vector class supports four types of constructors. These are:

SN	Constructor	Description
1)	vector()	It constructs an empty vector with the default size as 10.
2)	vector(int initialCapacity)	It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
3)	vector(int initialCapacity, int capacityIncrement)	It constructs an empty vector with the specified initial capacity and capacity increment.
4)	Vector(Collection<?extends E>	It constructs a vector that contains the elements of a

	c)	collection c.

Commonly used methods of Vector Class:

1. **void addElement(Object element):** It inserts the element at the end of the Vector.
2. **int capacity():** This method returns the current capacity of the vector.
3. **int size():** It returns the current size of the vector.
4. **void setSize(int size):** It changes the existing size with the specified size.
5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement():** It is used for getting the first element of the vector.
9. **Object lastElement():** Returns the last element of the array.
10. **Object get(int index):** Returns the element at the specified index.
11. **boolean isEmpty():** This method returns true if Vector doesn't have any element.
12. **boolean removeElement(Object element):** Removes the specifed element from vector.
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.

Example1:

```
import java.util.*;
public class VectorExample1 {
    public static void main(String args[]) {
      //Create an empty vector with initial capacity 4
      Vector<String> vec = new Vector<String>(4);
      //Adding elements to a vector
      vec.add("Tiger");
      vec.add("Lion");
      vec.add("Dog");
      vec.add("Elephant");
      //Check size and capacity
      System.out.println("Size is: "+vec.size());
      System.out.println("Default capacity is: "+vec.capacity());
```

```
        //Display Vector elements
        System.out.println("Vector element is: "+vec);
        vec.addElement("Rat");
        vec.addElement("Cat");
        vec.addElement("Deer");
        //Again check size and capacity after two insertions
        System.out.println("Size after addition: "+vec.size());
        System.out.println("Capacity after addition is: "+vec.capacity());
        //Display Vector elements again
        System.out.println("Elements are: "+vec);
        //Checking if Tiger is present or not in this vector
         if(vec.contains("Tiger"))
         {
           System.out.println("Tiger is present at the index " +vec.indexOf("Tiger"));
         }
         else
         {
           System.out.println("Tiger is not present in the list.");
         }
         //Get the first element
        System.out.println("The first animal of the vector is = "+vec.firstElement());
        //Get the last element
        System.out.println("The last animal of the vector is = "+vec.lastElement());
    }
}
```

Example2:

```
import java.util.*;
public class VectorExample2 {
    public static void main(String args[]) {
     //Create an empty Vector
     Vector < Integer > in = new Vector < > ();
      //Add elements in the vector
     in.add(100);
     in.add(200);
     in.add(300);
     in.add(200);
```

```
        in.add(400);
        in.add(500);
        in.add(600);
        in.add(700);
        //Display the vector elements
        System.out.println("Values in vector: " +in);
        //use remove() method to delete the first occurence of an element
        System.out.println("Remove first occourence of element 200: "+in.remove((Integer)20
0));
        //Display the vector elements afre remove() method
        System.out.println("Values in vector: " +in);
        //Remove the element at index 4
        System.out.println("Remove element at index 4: " +in.remove(4));
        System.out.println("New Value list in vector: " +in);
        //Remove an element
        in.removeElementAt(5);
        //Checking vector and displays the element
        System.out.println("Vector element after removal: " +in);
        //Get the hashcode for this vector
        System.out.println("Hash code of this vector = "+in.hashCode());
        //Get the element at specified index
        System.out.println("Element at index 1 is = "+in.get(1));
         }
}
```

C. Java.util.SimpleTimeZone Class

- The java.util.SimpleTimeZone class is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.Following are the important points about SimpleTimeZone –
  - The class holds an offset from GMT(Greenwich Mean Time), called raw offset.
  - This class also holds start and end rules for a daylight saving time schedule.

Class declaration

Syntax:

public class SimpleTimeZone extends TimeZone

Field:

- **static int STANDARD_TIME** – This is the constant for a mode of start or end time specified as standard time.

- **static int UTC_TIME** – This is the constant for a mode of start or end time specified as UTC(Coordinated Universal Time).
- **static int WALL_TIME** – This is the constant for a mode of start or end time specified as wall clock time.

Methods of Java TimeZone

Method	Description
static String[] getAvailableIDs()	It is used to get all the available IDs supported.
static TimeZone getDefault()	It is used to get the default TimeZone for this host.
String getDisplayName()	It is used to return a name of this time zone suitable for presentation to the user in the default locale.
String getID()	It is used to get the ID of this time zone
int getOffset(long date)	It is used to return the offset of this time zone from UTC at the specified date.
void setID(String ID)	It is used to set the time zone ID

Example1:

```
import java.util.*;
public class TimeZoneExample1 {
  public static void main( String args[] ){
  String[] id = TimeZone.getAvailableIDs();
  System.out.println("In TimeZone class available Ids are: ");
  for (int i=0; i<id.length; i++){
  System.out.println(id[i]);
  }
TimeZone zone = TimeZone.getTimeZone("Asia/Kolkata");
System.out.println("The Offset value of TimeZone: "+zone.getOffset(Calendar.ZONE_OFFSET));
 TimeZone timezone1 = TimeZone.getTimeZone("Asia/Kolkata");
```

```
  System.out.println("Value of ID is: " + timezone1.getID());
TimeZone zone2 = TimeZone.getDefault();
  String name = zone2.getDisplayName();
  System.out.println("Display name for default time zone: "+ name);
  }
}
```

D. Java.util.Date Class

- The java.util.Date class represents a **specific instant in time, with millisecond precision.** It is inherited by java.sql.Date, java.sql.Time and java.sql.Timestamp interfaces.

Class declaration

Syntax:

public class Date extends Object implements Serializable, Cloneable, Comparable<Date>

constructors

Sr.No.	Constructor & Description
1	**Date()** This constructor allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.
2	**Date(long date)** This constructor allocates a Date object and initializes it to represent the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.

methods

Method	Syntax	Description
toString()	public String toString()	Displays the Current date and time.
setTime()	public void setTime(long time)	Sets this Date object to represent a point in time that is time milliseconds after January 1, 1970 00:00:00 GMT
hashCode()	public int hashCode()	Returns a hash code value for the Date object.
after()	public boolean after(Date d)	tests if current date is after the given date.
clone()	public Object clone()	returns the duplicate of passed Date object
before()	public boolean after(Date d)	if current date is before the given date
compareTo()	public int compareTo(Date argDate)	compares two dates and results in -1, 0 or 1 based on the comparison
equals.()	public boolean equals(Object argDate)	checks whether two dates are equal or not based on their millisecond difference
getTime()	public long getTime()	ethod results in count of milliseconds of the argumented date, referencing January 1, 1970, 00:00:00 GMT

to print date in java using java.util.Date class.

1st way:

```
java.util.Date date=new java.util.Date();
System.out.println(date);
```

2nd way:

```
long millis=System.currentTimeMillis();
java.util.Date date=new java.util.Date(millis);
System.out.println(date);
```

Example:

```
import java.util.*;
public class DateDemo
{
        public static void main(String[] args)
        {
                Date mydate = new Date();
```

```
// Displaying the current date and time
System.out.println("System date : "+ mydate.toString() );
// Is used to set time by milliseconds. Adds 15680
// milliseconds to January 1, 1970 to get new time.
mydate.setTime(15680);
System.out.println("Time after setting: " + mydate.toString());
int d = mydate.hashCode();
System.out.println("Amount (in ms) by which time" + " is shifted : " + d);
Date date1 = new Date(2016, 11, 18);
 Date date2 = new Date(1997, 10, 27);
       // Use of after() to check date2 is after date1
      boolean a = date2.after(date1);
      System.out.println("Is date2 is after date1 : " + a);
       // Use of after() to check date2 is after date1
      a = date1.after(date2);
      System.out.println("Is date1 is after date2 : " + a);
      System.out.println("");
       // Use of clone() method
      Object date3 = date1.clone();
      System.out.println("Cloned date3 :" + date3.toString());
      System.out.println("");
       // Use of before() to check date2 is after date1
      boolean b = date2.before(date1);
      System.out.println("Is date2 is before date1 : " + a);
      int comparison = date1.compareTo(date2);    // d1 > d2
            int comparison2 = date2.compareTo(date1);   // d2 > d1
            int comparison3 = date1.compareTo(date1);   // d1 = d1
      System.out.println("date1 > date2 : " + comparison);
      System.out.println("date1 < date2 : " + comparison2);
      System.out.println("date1 = date1 : " + comparison3);
      System.out.println("");
     // Use of equal() method
    boolean r1 = date1.equals(date2);
    System.out.println("Result of equal() r1 : " + r1);
     boolean r2 = date1.equals(date1);
    System.out.println("Result of equal() r2 : " + r2);
    System.out.println("");
       // Use of getTime() method
```

```
        long count1 = date1.getTime();
        long count2 = date1.getTime();
        System.out.println("Milliseconds of d1 : " + count1);
        System.out.println("Milliseconds of d2 : " + count2);
    }
}
```

E. Java.util.Hashtable Class

- The java.util.Hashtable class implements a hashtable**, which maps keys to values**.
- Any non-null object can be used as a key or as a value. It inherits Dictionary class and implements the Map interface.
- To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.
- It is **similar to HashMap, but is synchronised.**

Constructors:

- **Hashtable():** This is the default constructor.
- **Hashtable(int size):** This creates a hash table that has initial size specified by size.
- **Hashtable(int size, float fillRatio):** This version creates a hash table that has initial size specified by size and fill ratio specified by fillRatio.
- **fill ratio:** Basically it determines how full hash table can be before it is resized upward.and its Value lie between 0.0 to 1.0
- **Hashtable(Map m):** This creates a hash table that is initialised with the elements in m.

class declaration

Syntax:.

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

Field:

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.\

Methods:

void clear() :

method clears the hashtable so that it contains no keys

Object clone() :

used to create a shallow copy of this hashtable.

computeIfAbsent(Key, Function):

The computeIfAbsent(Key, Function) method of Hashtable class which allows you to compute value of a mapping for specified key if key is not already associated with a value (or is mapped to null).

contains (Object value):
The java.util.Hashtable.contains(Object value) method in Java is used to check whether a particular value is being mapped by any keys present in the Hashtable.
boolean contains Key(Object key) :
tests if some key maps into the specified value in this hashtable.
boolean contains Value(Object value) :
returns true if this hash table maps one or more keys to this value.
Enumeration elements() :
Returns an enumeration of the values obtained in hash table
entrySet() :
used to get a set view of the entries contained in this hash table.
boolean equals(Object o) :
used to compare specified object with this Map for equality.
Object get(Object key) :
used to get the object that contains the value associated with key.
int hashCode() :
returns the hash code value for this Map as per the definition in the Map interface.
boolean isEmpty() :
used to test if this hashtable maps no keys to values
Enumeration keys() :
used to get enumeration of the keys contained in the hash table.
Object put (Object key, Object value) :
maps the specified key to the specified value in this hashtable.
putIfAbsent(Key, Function):
The putIfAbsent(Key, value) method of Hashtable class which allows to map a value to a given key if given key is not associated with a value or mapped to null. A null value is returned if such key-value set is already present in the HashMap
KeySet() :
used to get a Set view of the keys contained in this hash table
void putAll(Map t) :
copies all of the mappings from the specified map to this hashtable.
protected void rehash() :
Increase the size of the hash table and rehashes all its keys.
Object remove(Object key) :
Removes key and its value.
int size() :
returns the number of entries in hash table

String toString() :
returns the string equivalent of a hash table
values() :
used to get a Collection view of the values contained in this Hashtable

Example1:

```
import java.util.*;
class HashTableDemo1 {
   public static void main(String[] arg)
   {
      Hashtable<Integer, String> h = new Hashtable<>();
        Hashtable<Integer, String> h1 = new Hashtable<>();
        h.put(3, "DSTC");
      h.put(2, "MCA");
      h.put(1, "5");
        // create a clone or shallow copy of hash table h
      h1 =(Hashtable<Integer, String>) h.clone();
        // checking clone h1
      System.out.println("values in clone: " + h1);
        // clear hash table h
      h.clear();
        // checking hash table h
      System.out.println("after clearing: " + h);
Hashtable<String, Integer> table = new Hashtable<>();
      table.put("Pen", 10);
      table.put("Book", 500);
     table.put("Clothes", 400);
      table.put("Mobile", 5000);
System.out.println("hashTable: " + table.toString());
        // provide value for new key which is absent
      // using computeIfAbsent method
      table.computeIfAbsent("newPen", k -> 600);
      table.computeIfAbsent("newBook", k -> 800);
        // print new mapping
      System.out.println("new hashTable: " + table);
System.out.println("Initial Table is: " + table);
      // Checking for the Value 'pen'
```

```
        System.out.println("Is the value 'Pen' present? " +  table.contains("Pen"));
          // Checking for the Value 'World'
        System.out.println("Is the value 'World' present? " +  table.contains("World"));
    }
}
```

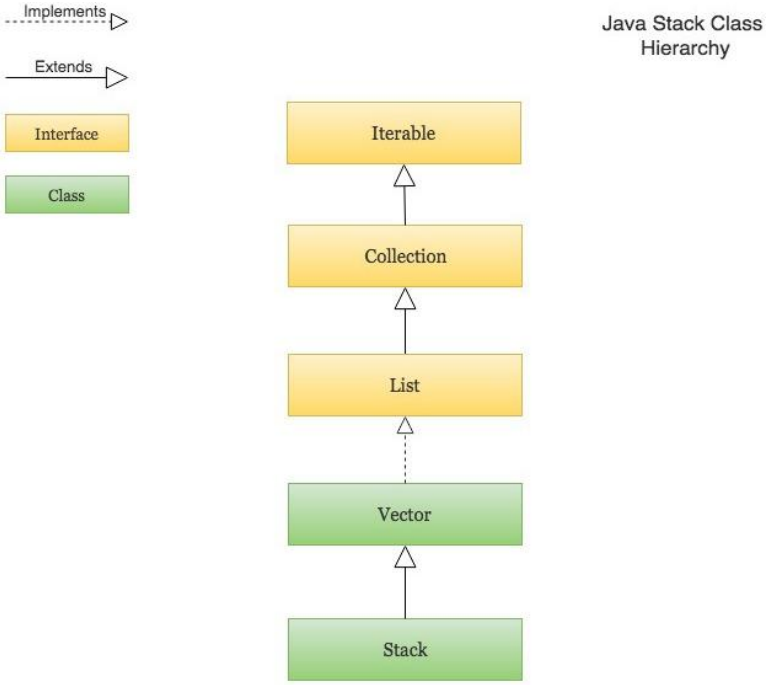
Example2:

```
import java.util.*;
 class hashTabledemo2 {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<String, Integer> marks = new Hashtable<String, Integer>();
          // enter name/marks pair
        marks.put("tweener", new Integer(345));
        marks.put("krantz", new Integer(100));
        marks.put("burrows", new Integer(790));
        marks.put("tancredi", new Integer(800));
        marks.put("bellick", new Integer(435));
          // check whether a value exists or not
        if (marks.containsKey("burrows"))
           System.out.println("Key found in table");
            if (marks.containsValue(345))
           System.out.println("value found in table");
              Enumeration e = h.elements();
         System.out.println("display values:");
         while (e.hasMoreElements()) {
           System.out.println(e.nextElement());
        }
// creating set view for hash table
        Set s = h.entrySet();
          // printing set entries
        System.out.println("set entries: " + s);
        Hashtable<String, Integer> marks1 = new Hashtable<String, Integer>();
          // enter name/marks pair
        marks1.put("tweener", new Integer(345));
        marks1.put("krantz", new Integer(100));
              if (h.equals(h1))
          System.out.println("both are equal");
            // get the value mapped with key krantz
         System.out.println(marks.get("krantz"));
                    System.out.println("hash code is: " + h.hashCode());
```

```
            h1.clear();
      // checking whether hash table h is empty or not
    if (h1.isEmpty())
       System.out.println("yes hash table is empty");
}
}
```

F. Java.util.Stack Class

- The java.util.Stack class represents a last-in-first-out (LIFO) stack of objects.
- When a stack is first created, it contains no items.
- In this class, the last element inserted is accessed first.
- It extends class Java Vector with five more operations that allow a vector to be treated as a stack.
- The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top. When a stack is first created, it contains no items.



Constructor:

Stack():Creates an empty Stack.

Stack Methods:

Method	Description
boolean empty()	Tests if this stack is empty.
Object peek()	Looks at the object at the top of this stack without removing it from the stack.
Object pop()	Removes the object at the top of this stack and returns that object as the value of this function.
Object push(Object element)	Pushes an item onto the top of this stack.
int search(Object element)	Returns the 1-based position where an object is on this stack

Example:

```
import java.util.Stack;
public class StackDemo {
   public static void main(String a[]){
      Stack<Integer> stack = new Stack<>();
      System.out.println("Empty stack : "  + stack);
      System.out.println("Empty stack : "  + stack.isEmpty());
      // Exception in thread "main" java.util.EmptyStackException
      // System.out.println("Empty stack : Pop Operation : "  + stack.pop());
      stack.push(1001);
      stack.push(1002);
      stack.push(1003);
      stack.push(1004);
      System.out.println("Non-Empty stack : "  + stack);
      System.out.println("Non-Empty stack: Pop Operation : "  + stack.pop());
      System.out.println("Non-Empty stack : After Pop Operation : "  + stack);
System.out.println("Top element is :"+ stack.peek());
      System.out.println("Non-Empty stack : search() Operation : "  + stack.search(1002));
      System.out.println("Non-Empty stack : "  + stack.isEmpty());
   }
}
```

Java Array to Stack Example

```
import java.util.Stack;
public class ArrayToStackDemo {
   public static void main(String a[]){
      Integer[] intArr = { 1001,1002,1003,1004};
      Stack<Integer> stack = new Stack<>();
      for(Integer i : intArr){
         stack.push(i);
      }
      System.out.println("Non-Empty stack : "  + stack);
   }
}
```

G. Java.util.GregorianCalendar Class

- The java.util.GregorianCalendar class is a concrete **subclass of Calendar and provides the standard calendar system** used by most of the world.Following are the important points about GregorianCalendar –
- It is a hybrid calendar that supports both the Julian and Gregorian calendar systems with the support of a single discontinuity, which corresponds by default to the Gregorian date when the Gregorian calendar was instituted.
- The Julian calendar specifies leap years every four years, whereas the Gregorian calendar omits century years which are not divisible by 400.

Class declaration

Syntax:

public class GregorianCalendar extends Calendar

Fields defined :

GregorianCalendar Class defines two fields:

AD : referring to the common era(anno Domini)

BC : referring to before common era(**Before Christ**)

Constructors:

Constructor	Description
GregorianCalendar()	Constructs a default GregorianCalendar using the current time in the default time zone with the default locale.
GregorianCalendar(int year,int month,int	Constructs a GregorianCalendar with the given

dayOfMonth)	date set in the default time zone with the default locale.
GregorianCalendar(int year,int month,int dayOfMonth,int hourOfDay,int minute)	Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.
GregorianCalendar(int year,int month,int dayOfMonth,int hourOfDay,int minute,int second)	Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.
GregorianCalendar(Locale aLocale)	Constructs a GregorianCalendar based on the current time in the default time zone with the given locale.
GregorianCalendar(TimeZone zone)	Constructs a GregorianCalendar based on the current time in the given time zone with the default locale.
GregorianCalendar(TimeZone zone,Locale aLocale)	Constructs a GregorianCalendar based on the current time in the given time zone with the given locale.

Methods:

Method	Summary
void	**add**(int field,int amount) Overrides Calendar Date Arithmetic function.
protected void	**computeFields**() Overrides Calendar Converts UTC as milliseconds to time field values.
protected void	**computeTime**() Overrides Calendar Converts time field values to UTC as milliseconds.
int	**getActualMaximum**(int field) Return the maximum value that this field could have, given the current date.
int	**getActualMinimum**(int field) Return the minimum value that this field could have, given the current

	date.
int	**getGreatestMinimum**(int field) Returns highest minimum value for the given field if varies.
Date	**getGregorianChange**() Gets the Gregorian Calendar change date.
int	**getLeastMaximum**(int field) Returns lowest maximum value for the given field if varies.
int	**getMaximum**(int field) Returns maximum value for the given field. e.g. for Gregorian DAY_OF_MONTH, 31 Please see Calendar.getMaximum for descriptions on parameters and the return value.
int	**getMinimum**(int field) Returns minimum value for the given field. e.g. for Gregorian DAY_OF_MONTH, 1 Please see Calendar.getMinimum for descriptions on parameters and the return value.
int	**hashCode**() Override hashCode.
boolean	**isLeapYear**(int year) Determines if the given year is a leap year.
void	**roll**(int field,boolean up) Overrides Calendar Time Field Rolling function.
void	**roll**(int field,int amount) Roll a field by a signed amount.
void	**setGregorianChange**(Date date) Sets the GregorianCalendar change date.

Example1:

```
import java.util.Calendar;
import java.util.GregorianCalendar;
 class CalendarGFG {
   public static void main(String[] args)
   {
      // Creating an object of Calendar Class
```

```
        Calendar cal = Calendar.getInstance();
          /* Creating an object of
             GregorianCalendar Class */
        GregorianCalendar gcal = new GregorianCalendar();
          /* Displaying Current Date using
             Calendar Class */
        System.out.println("Calendar date: " + cal.getTime());
          /* Displaying Current Date using  GregorianCalendar Class */
        System.out.print("Gregorian date: " + gcal.getTime());
    } // end of main function
} // end of class
```

H. Java.util.StringTokenizer Class

- The **java.util.StringTokenizer class allows you to break a string into tokens. It is simple way to break string. Each split string part is called *Token*.**
- It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class.
- This class is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.

Class declaration

Syntax:

public class StringTokenizer extends Object implements Enumeration<Object>

Constructors:

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.

Methods:

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.

Example:

```
import java.util.StringTokenizer;
public class StringTokenizerDemo{
 public static void main(String args[]){
  StringTokenizer st = new StringTokenizer("Java is secure language"," ");
   while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
   }
  String mystr = "JAVA : Code  /String : Tokenizer : /Dataflair";
        StringTokenizer st1 = new StringTokenizer(mystr,"/");
    // checking next token
     System.out.println("Next token is : " + st1.nextToken(":"));
while (st1.hasMoreTokens()) {
      System.out.println(st1.nextToken());
   }
  }
}
```

I. Java.util.Scanner Class

- The **java.util.Scanner class is a simple text scanner which can parse primitive types and strings using regular expressions**.Following are the important points

about Scanner –

- A Scanner **breaks its input into tokens using a delimiter pattern, which by default matches whitespace.**
- A scanning operation may block waiting for input.
- A Scanner is not safe for multithreaded use without external synchronization.

Class declaration

Syntax:

public final class Scanner extends Object implements Iterator<String>

How to get Java Scanner

- To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class.
- For Example:

  Scanner in = **new** Scanner(System.in);
- To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class.
- For Example:

  Scanner in = **new** Scanner("Hello Java");

Constructors

SN	Constructor	Description
1)	Scanner(File source)	It constructs a new Scanner that produces values scanned from the specified file.
2)	Scanner(File source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.
3)	Scanner(InputStream source)	It constructs a new Scanner that produces values scanned from the specified input stream.
4)	Scanner(InputStream source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified input stream.
5)	Scanner(Readable source)	It constructs a new Scanner that produces

		values scanned from the specified source.
6)	Scanner(String source)	It constructs a new Scanner that produces values scanned from the specified string.
7)	Scanner(ReadableByteChannel source)	It constructs a new Scanner that produces values scanned from the specified channel.
8)	Scanner(ReadableByteChannel source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified channel.
9)	Scanner(Path source)	It constructs a new Scanner that produces values scanned from the specified file.
10)	Scanner(Path source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.

Methods

Modifier & Type	Method	Description
void	close()	It is used to close this scanner.
pattern	delimiter()	It is used to get the Pattern which the Scanner class is currently using to match delimiters.
Stream<MatchResult>	findAll()	It is used to find a stream of match results that match the provided pattern string.
String	findInLine()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.

string	findWithinHorizon()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
boolean	hasNext()	It returns true if this scanner has another token in its input.
IOException	ioException()	It is used to get the IOException last thrown by this Scanner's readable.
boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
byte	nextByte()	It scans the next token of the input as a byte.
double	nextDouble()	It scans the next token of the input as a double.
float	nextFloat()	It scans the next token of the input as a float.
int	nextInt()	It scans the next token of the input as an Int.
String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
long	nextLong()	It scans the next token of the input as a long.

Example:

```
import java.util.*;
public class ScannerDemo {
public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
          String s = "Hello,  Java.";
```

```
        //Create scanner Object and pass string in it
        Scanner scan = new Scanner(s);
        //Check if the scanner has a token
        System.out.println("Boolean Result: " + scan.hasNext());
        //Print the string
        System.out.println("String: " +scan.nextLine());
        System.out.print("Enter your age: ");
        int i = in.nextInt();
        System.out.println("Age: " + i);
        System.out.print("Enter your salary: ");
        double d = in.nextDouble();
        System.out.println("Salary: " + d);
           String st = "Hello/This is Java/we use scanner class.";
           Scanner sc = new Scanner(st);
           sc.useDelimiter("/");
    //Printing the tokenized Strings
 System.out.println("---Tokenizes String---");
 while(sc.hasNext()){
          System.out.println(sc.next());
      }
        //Display the new delimiter
        System.out.println("Delimiter used: " +sc.delimiter());
        }
}
```

3. java.net package

- Java programs are built specially to be run on a network. For the practice of these network applications, a set of classes is provided under this package.
- The java.net package can be roughly divided in two sections:
  - **A Low Level API**, which deals with the following abstractions:
    - **Addresses,** which are networking identifiers, like IP addresses.
    - **Sockets,** which are basic bidirectional data communication mechanisms.
    - **Interfaces**, which describe network interfaces.
  - **A High Level API**, which deals with the following abstractions:
    - **URIs**, which represent Universal Resource Identifiers.
    - **URLs**, which represent Universal Resource Locators.

- **Connections,** which represents connections to the resource pointed to by URLs.

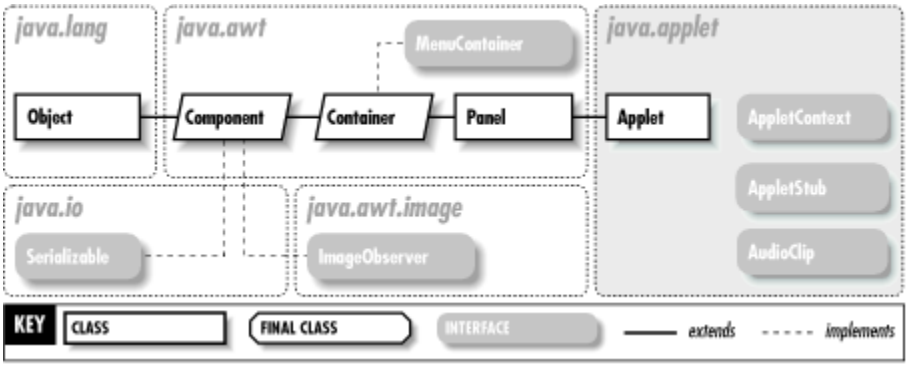
Example:

```
URI uri = new URI("http://java.sun.com/");
URL url = uri.toURL();
InputStream in = url.openStream();
```

Class	Summary
Authenticator	The class Authenticator represents an object that knows how to obtain authentication for a network connection.
CacheRequest	Represents channels for storing resources in the ResponseCache.
CacheResponse	Represent channels for retrieving resources from the ResponseCache.
ContentHandler	The abstract class ContentHandler is the superclass of all classes that read an Object from a URLConnection.
CookieHandler	A CookieHandler object provides a callback mechanism to hook up a HTTP state management policy implementation into the HTTP protocol handler.
HttpURLConnection	A URLConnection with support for HTTP-specific features.
InetAddress	This class represents an Internet Protocol (IP) address.
NetPermission	This class is for various network permissions.
PasswordAuthentication	The class PasswordAuthentication is a data holder that is used by Authenticator.
URI	Represents a Uniform Resource Identifier (URI) reference.
URL	Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.
URLClassLoader	This class loader is used to load classes and resources from a search path of URLs referring to both JAR files and directories.
URLConnection	The abstract class URLConnection is the superclass of all classes that represent a communications link between the application and a URL.

4. java.applet Package

- The java.applet package is a small but important package that defines the Applet class--the superclass of all applets. It also defines the AppletContext and AppletStub interfaces, which are implemented by web browsers and other applet viewers.



- Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
- The applet framework involves two entities: the *applet* and the *applet context*. **An applet is an embeddable window with a few extra methods that the applet context can use to initialize, start, and stop the applet.**
- The **applet context is an application that is responsible for loading and running applets.** For example, the applet context could be a Web browser or an applet development environment.

Methods:

1) **getAppletInfo()** should return text suitable for display in an About dialog posted by the web browser or applet viewer.
2) **getParameterInfo()** should return an arbitrary-length array of three-element arrays of strings
3) **paint()** method to draw the applet on the screen.
4) **showStatus()** displays text in the web browser or applet viewer's status line. getImage()
5) **getAudioClip()** read image (GIF and JPEG formats) and audio files (AU format) over the network and return corresponding Java objects.
6) **getParameter()** looks up the value of a parameter specified with a<param> tag within <applet></applet>pair.
7) **getCodeBase()** returns the base URL from which the applet's code was loaded,
8) **getDocumentBase()** returns the base URL from which the HTML document containing the applet was loaded.
9) **getAppletContext()** returns an AppletContext object.

5. java.awt Package

- The java.awt package is the **Abstract Windowing Toolkit**. The classes of this package may be roughly divided into three categories
  - **Graphics:** These classes define colors, fonts, images, polygons, and so forth.
  - **Components:** These classes are GUI (graphical user interface) components such as buttons, menus, lists, and dialog boxes.
  - **Layout Managers:** These classes control the layout of components within their container objects.
- Java **AWT** (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.
- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.
- The java.awt package provides classes for AWT api such as TextField, Label, TextArea .RadioButton, CheckBox, Choice, List etc.

6. javax.swing package

- **Provides a set of "lightweight" (all-Java language) components** that, to the maximum degree possible, work the same on all platforms. Java Swing is a part of **Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.**
- Unlike AWT, Java Swing provides platform-independent and lightweight components.
- The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

7. java.awt.event Package

- The java.awt.event package defines classes and interfaces used for event handling in the AWT and Swing.
- The members of this package fall into three categories:
- **Events:** The classes with names ending in "Event" represent specific types of events, generated by the AWT or by one of the AWT or Swing components.
- **Listeners:** The interfaces in this package are all event listeners; their names end with "Listener". These interfaces define the methods that must be implemented by any object that wants to be notified when a particular event occurs. Note that there is a Listener interface for each Event class.
- **Adapters:** Each of the classes with a name ending in "Adapter" provides a no-op implementation for an event listener interface that defines more than one method. When you are interested in only a single method of an event listener interface, it is

easier to subclass an Adapter class than to implement all of the methods of the corresponding Listener interface.

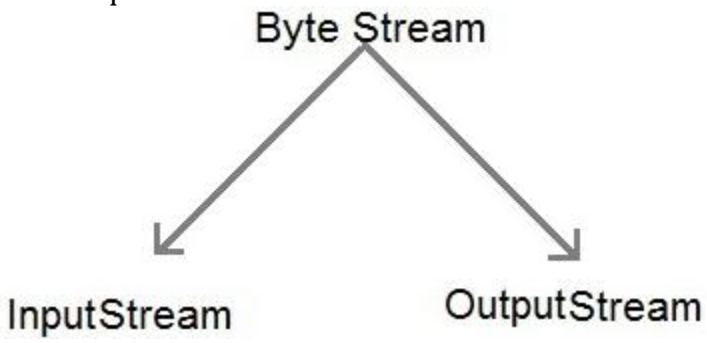
- Events are fired by event sources. An event listener registers with an event source to receive notifications about the events of a particular type. This package defines events and event listeners, as well as event listener adapters, which are convenience classes to make easier the process of writing event listeners.

8. java.io package

- **Provides for system input and output through data streams, serialization and the file system.**
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform file handling in Java by Java I/O API.
- Java encapsulates Stream under java.io package. Java defines two types of streams. They are,
- **Byte Stream :** It provides a convenient means for handling input and output of byte.
- **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Java Byte Stream Classes

- Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



- These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

Byte stream classes.

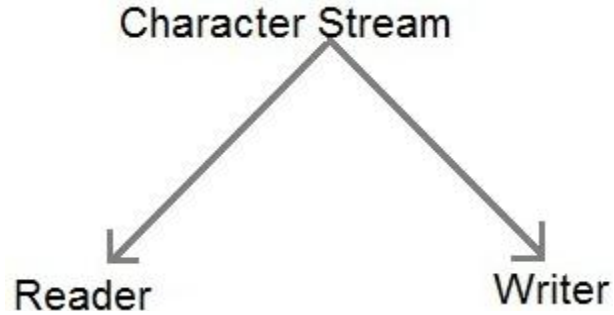
Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.

BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

- These classes define several key methods. Two most important are
  - read() : reads byte of data.
  - write() : Writes byte of data.

Java Character Stream Classes

- Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



- These two abstract classes have several concrete classes that handle unicode character.

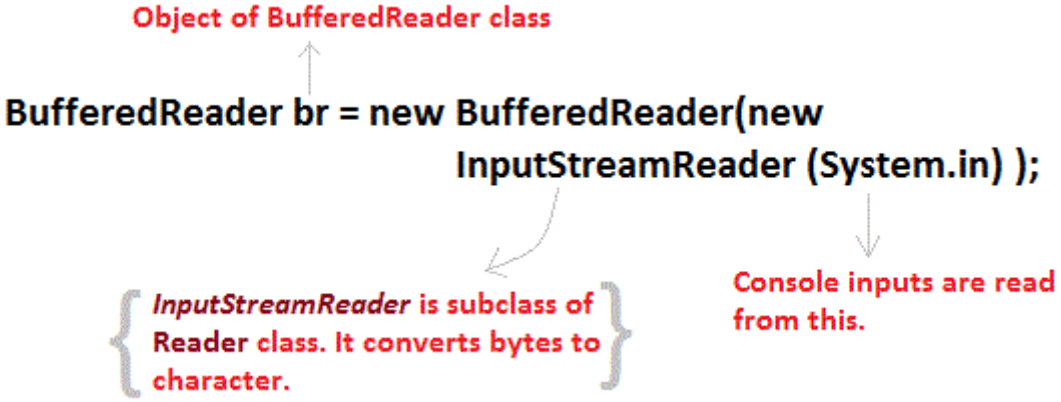
Character stream classes

Stream class	Description
BufferedReader	Handles buffered input stream.

BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

Reading Console Input

We use the object of BufferedReader class to take inputs from the keyboard.



Reading Characters

- read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into char type.

Example:

```
class StreamDemo
{
        public static void main(String[] args)
        {
```

```
        try
        {
                File fl = new File("d:/sbj/bca/java prog/myfile.txt");
                BufferedReader br = new BufferedReader(new FileReader(fl)) ;
                String str;
                while ((str=br.readLine())!=null)
                {
                        System.out.println(str);
                }
         }
        catch(IOException  e) {
                e.printStackTrace();
        }
       }
}
```

b. User defined package:

- Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc

➢ **Creating a Package**

- Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
    └── mypack
        └── MyPackageClass.java
```

➢ **To create a package, use the package keyword:**

- The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.
- If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.
- To compile the Java programs with package statements, you have to use -d option as shown below.

Syntax:

```
javac -d Destination_folder file_name.java
```

- Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example:

Simple.java

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

- Save the file in current folder and directory as Simple.java, and compile it:
  d:\bca\java>javac –d . Simple.java
- if you want to create package in another directory or folder then compile as:
  d:\bca\java>javac –d e:/jprog Simple.java
- To run the Simple.java file, write the following:
  d:\bca\java>java mypack.Simple

➢ **How to access package from another package?**

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

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

1. Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example:

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

```
  obj.msg();
 }
}
```

2. Using packagename.classname

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

Example

```
//save by A.java
package pack;
public class cl1{
 public void msg(){System.out.println("Hello");}}
//save by B.java
package mypack;
import pack.cl1;
 class cl2{
 public static void main(String args[]){
  cl1 obj = new cl1();
  obj.msg();
 }
}
```

3. Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example

```
//save by A.java
package pack;
public class cl3{
 public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class cl4{
 public static void main(String args[]){
```

```
  pack.cl3 obj = new pack.cl3();//using fully qualified name
  obj.msg();
 }
}
```

Subpackage in java

- Package inside the package is called the subpackage. It should be created **to categorize the package further**.
- If you import a package, subpackages will not be imported.
- If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

➢ **Creating subpackage:**

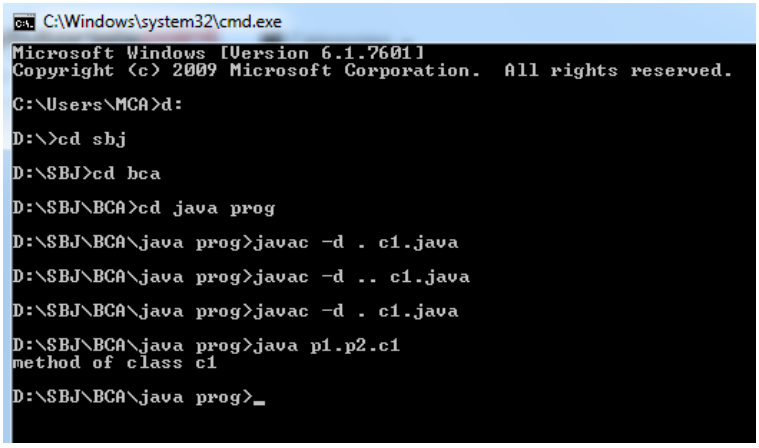
In package p1:

```
package p1;
class c1
{
    void m1()
    {
        System.out.println("method of class c1");
    }
    public static void main(String args[])
    {
        c1 ct=new c1();
        ct.m1();
    }
}
```

Create a subpackage p2

```
package p1.p2;
class c1
{
    void m1()
    {
        System.out.println("method of class c1");
    }
    public static void main(String args[])
    {
        c1 ct=new c1();
        ct.m1();
    }
```

}



- **Create a subpackage and import it in another package:**

Create subpackage subpack1 in pack package

```
package pack.subpack1;
public class cl5 {
        public void product(int a, int b){
                System.out.println(a*b);
        }
}
```

Import subpack1 in mypack package

```
package mypack;
import pack.subpack1.cl5;
 class cl6{
 public static void main(String args[]){
cl5 obj = new  cl5();
  obj.product(4,5);
 }
}
```

Output:

```
D:\SBJ\BCA\java prog>javac -d . cl5.java

D:\SBJ\BCA\java prog>javac -d . cl6.java

D:\SBJ\BCA\java prog>java mypack.cl6
20
```