

Classes and Objects in Java

Basics of Classes in Java

Contents

- Introduce to classes and objects in Java.
- Understand how some of the OO concepts learnt so far are supported in Java.
- Understand important features in Java classes.

Learning Outcomes

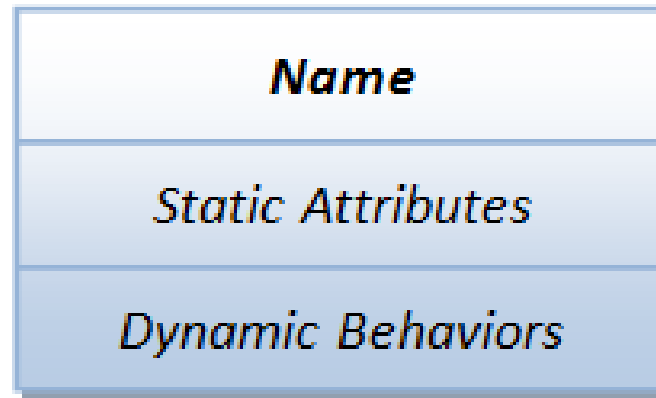
- Defining classes, creating objects and methods, Passing and Returning object form Method, Method overloading.
- String class, StringBuffer class, Operations on string, String Joiner class, Wrapper Class.
- Access control, modifiers, this keyword, static keyword.
- Constructors: Default constructors, Parameterized constructors, Copy constructors, Private constructor, and Constructor Overloading.

Introduction

- Java is a **true OO language** and therefore the underlying structure of all Java programs is classes.
- Anything we wish to represent in Java must be encapsulated in a class that defines the “**state**” and “**behaviour**” of the basic program components known as objects.
- Classes create objects and objects use methods to communicate between them. **They provide a convenient method for packaging a group of logically related data items and functions that work on them.**
- A class essentially serves as **a template for an object** and behaves like a basic data type “int”.
- It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OO concepts such as **encapsulation, inheritance, and polymorphism.**

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- A Class is a 3-Compartment Box Encapsulating Data and Operations

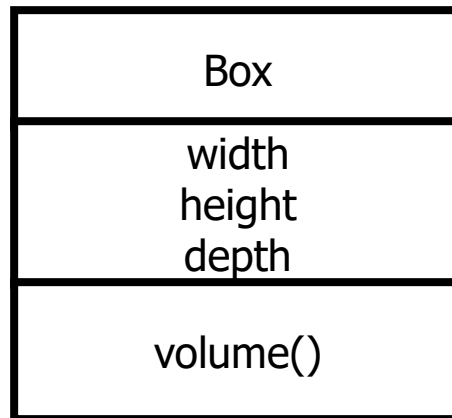


A class is a 3-compartment box

- *Name* (or identity): identifies the class.
- *Variables* (or attribute, state, field): contains the *static attributes* of the class.
- *Methods* (or behaviors, function, operation): contains the *dynamic behaviors* of the class.
- a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

Classes

- Defines a new data type.
- *Template for an object, and an object is an instance of a class.*



Name (Identifier)	Student	Circle	SoccerPlayer	Car
Variables (Static attributes)	name gpa	radius color	name number xLocation yLocation	plateNumber xLocation yLocation speed
Methods (Dynamic behaviors)	getName() setGpa()	getRadius() getArea()	run() jump() kickBall()	move() park() accelerate()

Examples of classes

Name	<u>paul:Student</u>	<u>peter:Student</u>
Variables	name="Paul Lee" gpa=3.5	name="Peter Tan" gpa=3.9
Methods	getName() setGpa()	getName() setGpa()

Two instances - paul and peter - of the class Student

Class Definition in Java

- The general form for a class definition:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```


A Simple Class

- Class with no methods

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

A Simple Class

```
public class Circle { // class name
    double radius; // variables
    String color;
    double getRadius() { ..... } // methods
    double getArea() { ..... }
}
```

```
public class SoccerPlayer { // class name
    int number; // variables
    String name;
    int x, y;
    void run() { ..... } // methods
    void kickBall() { ..... }
}
```

Data Abstraction

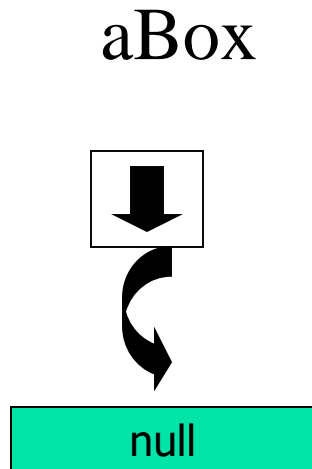
- Declare the Box class, have created a new data type – Data Abstraction
- Can define variables (objects) of that type:

```
Box aBox;
```

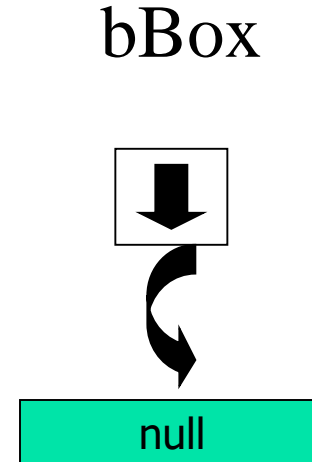
```
Box bBox;
```

Class of Box cont.

- aBox, bBox simply refers to a Box object, not an object itself.



Points to nothing (Null Reference)



Points to nothing (Null Reference)

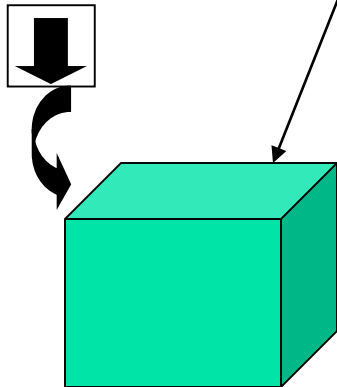
Creating objects/Instance of a class

- To create *an instance/object of a class*, you have to:
 - **Declare** an instance identifier (instance name) of a particular class.
 - **Construct** the instance (i.e., allocate storage for the instance and initialize the instance) using the "**new**" operator.
- Objects/Instances are created dynamically using the new keyword.

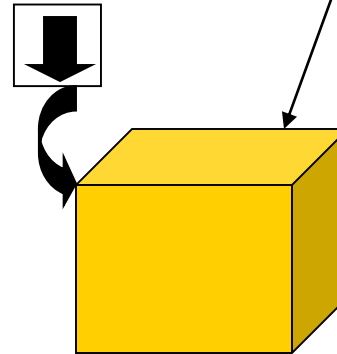
Creating objects/Instance of a class

- aBox and bBox refer to Box objects

aBox = new Box() ;



bBox = new Box() ;

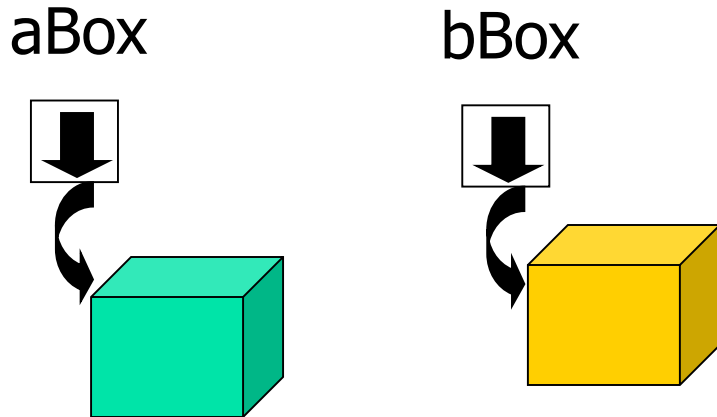


Creating objects/Instance of a class

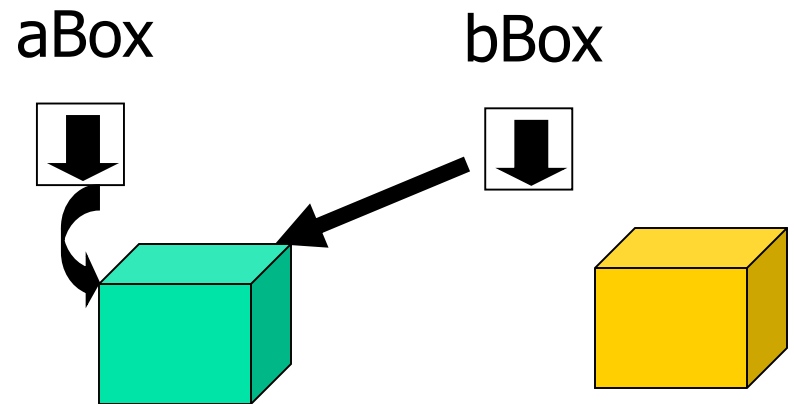
```
aBox = new Box();  
bBox = new Box();
```

```
bBox = aBox;
```


Before Assignment



After Assignment



Automatic garbage collection

- The object  does not have a reference and cannot be used in future.
- The object becomes a candidate for automatic garbage collection.
- Java automatically collects garbage periodically and releases the memory used to be used in the future.

Creating objects/Instance of a class

```
// Declare 3 instances of the class Circle, c1, c2,  
and c3  
    Circle c1, c2, c3; // They hold a special value  
called null //  
    Construct the instances via new operator  
c1 = new Circle();  
c2 = new Circle(2.0);  
c3 = new Circle(3.0, "red");  
// You can Declare and Construct in the same  
statement  
    Circle c4 = new Circle();
```

Dot (.) Operator

- The *variables* and *methods* of a class are formally called *member variables* and *member methods*.
- To reference a member variable or method, you must:
 - First identify the *instance* you are interested in, and then,
 - Use the *dot operator* (.) to reference the desired member variable or method.
- For example,
 - In a class Circle, with two member variables (radius and color) and two member methods (getRadius() and getArea()).
 - We have created three instances of the class Circle, namely, c1, c2 and c3.
 - To invoke the method getArea(), you must first identify the instance of interest, says c2, then use the *dot operator*, in the form of *c2.getArea()*.

Dot (.) Operator

```
// Suppose that the class Circle has variables radius
// and color,
// and methods getArea() and getRadius().
// Declare and construct instances c1 and c2 of the
// class Circle
Circle c1 = new Circle ();
Circle c2 = new Circle ();
// Invoke member methods for the instance c1 via dot
// operator
System.out.println(c1.getArea());
System.out.println(c1.getRadius());

// Reference member variables for instance c2 via dot
// operator
c2.radius = 5.0;
c2.color = "blue";
```

Member Variables

- *A member variable has a name (or identifier) and a type; and holds a value of that particular type.*
 - **Variable Naming Convention:** A variable name shall be a **noun or a noun phrase** made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case),
e.g., `fontSize`, `roomNumber`, `xMax`, `yMin` and `xTopLeft`.
- *The formal syntax for variable definition in Java is:*
 - *[AccessControlModifier] type variableName [= initialValue];*
[AccessControlModifier] type variableName-1 [= initialValue-1] [, type variableName-2 [= initialValue-2]] ... ;
 - For example,
private double radius;
public int length = 1, width = 1;

Member Methods

- A method (as described in the earlier chapter):
 1. receives arguments from the caller,
 2. performs the operations defined in the method body, and
 3. returns a piece of result (or void) to the caller.
- The syntax for method declaration in Java is as follows:
 - *[AccessControlModifier] returnType methodName ([parameterList]) { // method body or implementation }*
- For examples:
 - *// Return the area of this Circle instance*
*public double getArea() { return radius * radius * Math.PI; }*

Member Methods

■ Method Naming Convention:

- A method name **shall be a verb, or a verb phrase** made up of several words.
- The first word is in lowercase and the rest of the words are initial-capitalized (camel-case).

- For example,

getArea(), setRadius(), getParameterValues(), hasNext().

- A variable name is a noun, denoting an attribute;
- A method name is a verb, denoting an action.
- A class name is a noun beginning with uppercase.

Example

```
/* A program that uses the Box class.

    Call this file BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}

// This class declares an object of type Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;

        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Volume is " + vol);
    }
}
```

Adding a Method to the Box Class

// This program includes a method inside the box class.

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // display volume of a box  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}
```


Adding a Method to the Box Class

```
class BoxDemo3 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        // assign values to mybox1's instance variables  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;  
        /* assign different values to mybox2's instance variables */  
        mybox2.width = 3;  
        mybox2.height = 6;  
        mybox2.depth = 9;  
        // display volume of first box  
        mybox1.volume();  
        // display volume of second box  
        mybox2.volume();  
    }  
}
```

Method That Takes Parameters

```
// sets dimensions of box
```

```
void setDim(double w, double h, double d) {
```

```
    width = w;
```

```
    height = h;
```

```
    depth = d;
```

```
}
```

```
/* in main program method can be called to set box
```

```
Box mybox1 = new Box();
```

```
Box mybox2 = new Box();
```

```
double vol;
```

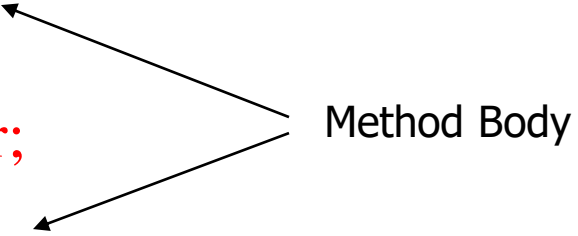
```
// initialize each box
```

```
mybox1.setDim(10, 20, 15);
```

```
mybox2.setDim(3, 6, 9);
```

Circle Class

```
public class Circle {  
  
    public double x, y; // centre of the circle  
    public double r;    // radius of circle  
  
    //Methods to return circumference and area  
    public double circumference() {  
        return 2*3.14*r;  
    }  
    public double area() {  
        return 3.14 * r * r;  
    }  
}
```



Method Body

Circle Program

```
// Circle.java:  Contains both Circle class and its user class
//Add Circle class code here
class MyMain{
    public static void main(String args[])    {
        Circle aCircle;  // creating reference
        aCircle = new Circle(); // creating object
        aCircle.x = 10;  // assigning value to data field
        aCircle.y = 20;
        aCircle.r = 5;
        double area = aCircle.area(); // invoking method
        double circumf = aCircle.circumference();
        System.out.println("Radius="+aCircle.r+" Area="+area);
        System.out.println("Radius="+aCircle.r+" Circumference
        ="+circumf);
    }
}
```

Better way of Initialising or Access Data Members x, y, r

- When there too many items to update/access and also to develop a readable code, generally it is done by defining specific method for each purpose.
- To initialise/Update a value:
 - `aCircle.setX(10)`
- To access a value:
 - `aCircle.getX()`
- These methods are informally called as Accessors or Setters/Getters Methods.

Accessors – “Getters/Setters”

```
public class Circle {  
    private double x,y,r;  
  
    //Methods to return circumference and area  
    public double getX() { return x;}  
    public double getY() { return y;}  
    public double getR() { return r;}  
    public double setX(double x_in) { x = x_in;}  
    public double serY(double y_in) { y = y_in;}  
    public double setR(double r_in) { r = r_in;}  
  
}
```

How does this code looks ? More readable ?

```
// Circle.java:  Contains both Circle class and its user
class
//Add Circle class code here
class MyMain{
    public static void main(String args[]){
        Circle aCircle;  // creating reference
        aCircle = new Circle(); // creating object
        aCircle.setX(10);
        aCircle.setY(20);
        aCircle.setR(5);
        double area = aCircle.area(); // invoking method
        double circumf = aCircle.circumference();
        System.out.println("Radius="+aCircle.getR()+"
Area="+area);
        System.out.println("Radius="+aCircle.getR()+"
Circumference =" +circumf);
    }
}
```

Object Initialisation

- When objects are created, the initial value of data fields is unknown unless its users explicitly do so. For example,
 - `ObjectName.DataField1 = 0; // OR`
 - `ObjectName.SetDataField1(0);`
- In many cases, it makes sense if this initialisation can be carried out by default without the users explicitly initializing them.
 - For example, if you create an object of the class called “Counter”, it is natural to assume that the counter record-keeping field is initialized to zero unless otherwise specified differently.

```
class Counter
{
    int CounterIndex;
    ...
}
Counter counter1 = new Counter();
```

- What is the value of “counter1.CounterIndex” ?
- In Java, this can be achieved through a mechanism called constructors.

Visibility Controls/Access Modifiers of JAVA

- Java provides a number of **access modifiers** to set access levels for **classes, variables, methods** and **constructors**.
- The four access levels are:
- **Package/friendly (default)** -Visible to the **package**.
No modifiers are needed.
- **Private** - Visible to the **class** only.
- **Public**- Visible to the **class** as well as **outside the class**.
- **Protected**- Visible to the **package** and all **sub Classes**.

Visibility Controls/Access Modifiers of JAVA

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Visibility Controls of JAVA

Default Access Modifier – No Keyword

- **Default** access modifier means no need to declare an access modifier for a **class, field, method** etc.
- A variable or method declared without any access control modifier is **available to any other class in the same package**.
- The **default** modifier cannot be used for methods in an **interface** because the methods in an interface are by default **public**.

Visibility Controls of JAVA

Default Access Modifier – Example

```
class BaseClass
{
    void display()        //no access modifier indicates default modifier
    {
        System.out.println("BaseClass::Display with 'dafault' scope");
    }
}
class Main
{
    public static void main(String args[])
    {
        //access class with default scope
        BaseClass obj = new BaseClass();

        obj.display();    //access class method with default scope
    }
}
```

BaseClass::Display with
'dafault' scope

Visibility Controls of JAVA

Public Access Modifier - public

- The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members.
- When a class member is preceded by **public**, then that member may be accessed by code **outside the class**.
- A class, method, constructor, interface etc declared public can be accessed from any other class.
- Therefore, methods or blocks declared inside a public class can be accessed from any class belonging to the Java world.
- However, if the public class we are trying to access is in a different package, and then the public class still need to be **imported**. Because of class inheritance, all public methods and variables of a class are inherited by its **subClasses**.

Visibility Controls of JAVA

Public Access Modifier - public

■ Example:

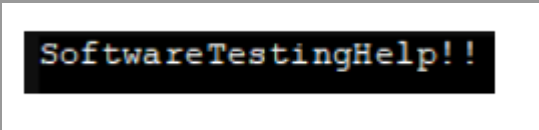
```
public static void main(String[] args)
{
// ...
}
```

- The **main()** method of an application needs to be **public**. Otherwise, it could not be called by a **Java interpreter** (such as java) to run the class.

Visibility Controls of JAVA

Public Access Modifier - Example

```
class A
{
    public void display()
    {
        System.out.println("SoftwareTestingHelp!!");
    }
}
class Main
{
    public static void main(String args[])
    {
        A obj = new A ();
        obj.display();
    }
}
```



Visibility Controls of JAVA

Protected Access Modifier – Protected

- Variables, methods and constructors which are declared **protected** in a super class can be accessed only by the **subClasses** in other package or any class within the package of the protected members' class.
- The protected access modifier cannot be applied to **class** and **interfaces**.
- **Methods** can be declared **protected**, however **methods** in a **interface** cannot be declared **protected**.
- Protected access gives chance to the subClass to use the helper method or variable, while prevents a non-related class from trying to use it.

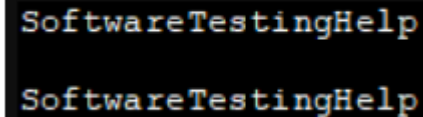
Visibility Controls of JAVA

Protected Access Modifier – Protected

```
class A
{
    protected void display()
    {
        System.out.println("SoftwareTestingHelp");
    }
}

class B extends A {}
class C extends B {}

class Main{
    public static void main(String args[])
    {
        B obj = new B();           //create object of class B
        obj.display();              //access class A protected method using obj
        C cObj = new C();           //create object of class C
        cObj.display ();            //access class A protected method using cObj
    }
}
```



SoftwareTestingHelp
SoftwareTestingHelp

Visibility Controls of JAVA

Protected Access Modifier – Protected

Example:

```
package p1; class A
{ float f1; protected int i1; }
// note that class B belongs to the same package as class A
package p1; class B
{
public void getData()
{
// create an instance of class A
A a1 = new A(); a1.f1 = 19;
a1.i1 = 12;
}}
```

Visibility Controls of JAVA

Private Access Modifier – private

- Methods, Variables and Constructors that are declared **private** can only be accessed within the **declared class itself**.
- Private access modifier is the most **restrictive access level**. **Class** and **interfaces** cannot be **private**.
- Variables that are declared private can be accessed outside the class if **public getter** methods are present in the **class**.
- Using the private modifier, an object **encapsulates** itself and **hides** data from the outside world.

Visibility Controls of JAVA

Private Access Modifier – private

- Private access modifier cannot be used for classes and interfaces.
- The scope of private entities (methods and variables) is limited to the class in which they are declared.
- A class with a private constructor cannot create an object of the class from any other place like the main method.

Visibility Controls of JAVA

Private Access Modifier – Example

```
class TestClass{
    //private variable and method
    private int num=100;
    private void printMessage(){System.out.println("Hello java");}
}

public class Main{
    public static void main(String args[]){
        TestClass obj=new TestClass();
        System.out.println(obj.num);//try to access private data member - Compile Time Error
        obj.printMessage();//Accessing private method - Compile Time Error
    }
}
```

Visibility Controls of JAVA

Private Access Modifier – private

Example:

```
class A  
{private String s1 = "Hello";  
public String getName()  
{return this.s1;}}
```

- Here, s1 variable of A class is private, so there's no way for other classes to retrieve.
- So, to make this variable available to the outside world, we defined public methods: getName(), which returns the value of s1.

Visibility Controls of JAVA

Protected Access Modifier – Protected

- In above example, class A and B are in same package p1. Class A has i1 variable which is declared as protected.
- So, it can be accessed through entire package and all its subclasses.
- Thus, in getData() method of class B we can access variable f1 as well as i1.

this Keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the **this** keyword. Keyword **this** can be used inside any method or constructor of class to refer to the current object.
- It means, **this** is always a reference to the object on which the method was invoked.
- **this** keyword can be very useful in case of Variable Hiding.
- You can use **this** anywhere a reference to an object of the current class' type is permitted.
- We cannot create two **Instance/Local** variables with same name. But it is legal to create one instance variable & one local variable or method parameter with same name.
- **Local Variable** will hide the instance variable which is called Variable Hiding.

this Keyword

Usage of Java this Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

01

this can be used to refer current class instance variable.

04

this can be passed as an argument in the method call.

02

this can be used to invoke current class method (implicitly)

05

this can be passed as argument in the constructor call.

03

this() can be used to invoke current class Constructor.

06

this can be used to return the current class instance from the method

this Keyword

- **this: to refer current class instance variable**
- The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
- **this: to invoke current class method**
- You may invoke the method of the current class by using the this keyword.
- **this: to reuse constructor call**
- The this() constructor call should be used to reuse the constructor from the constructor.
- It maintains the chain between the constructors i.e. it is used for constructor chaining.

this: to refer current class instance variable

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class test{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }}

```

111 ankit 5000.0
112 sumit 6000.0

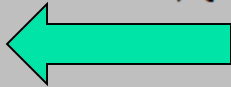
this: to invoke current class method

```
class A{  
void m(){System.out.println("hello m");}  
void n(){  
System.out.println("hello n");  
//m(); //same as this.m()  
this.m();           hello n  
                        hello m  
}  
}  
  
class TestThis4{  
public static void main(String args[]) {  
A a=new A();  
a.n();  
}}
```

this: to reuse constructor call

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

111 ankit java 0.0
112 sumit java 6000.0



Static Keyword

- The static keyword in Java is used for memory management mainly.
- We can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.
- The static can be:
 - **Variable (also known as a class variable)**
 - **Method (also known as a class method)**
 - **Block**
 - **Nested class**

Static Keyword

Static Variable:-

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable:-

- It makes your program memory efficient (i.e., it saves memory).

Static Keyword

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

- Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created.
- All students have its unique rollno and name, so instance data member is good in such case.
- Here, "college" refers to the common property of all objects.
- If we make it static, this field will get the memory only once.
- **Java static property is shared to all objects.**

Static Keyword

```
class Student{
    int rollno;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r,String n){
        rollno = r;
        name = n;
    }
    //method to display the values
    void display ()
        {System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of static variable
public class TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        s1.display();
        s2.display();
    }
}
```

111 Karan ITS
222 Aryan ITS

//we can change the college of all objects by the single line of code
//Student.college="SSGP"

Static Keyword

■ Program of the counter without static variable:-

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.
class Counter{
    int count=0;//will get memory each time when the instance is created

    Counter(){
        count++; //incrementing value
        System.out.println(count);
    }

    public static void main(String args[]){
        //Creating objects
        Counter c1=new Counter();
        Counter c2=new Counter();
        Counter c3=new Counter();
    }
}
```

1
1
1

Static Keyword

■ Program of counter by static variable

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
class Counter2{
    static int count=0;//will get memory only once and retain its value

    Counter2(){
        count++; //incrementing the value of static variable
        System.out.println(count);
    }

    public static void main(String args[]){
        //creating objects
        Counter2 c1=new Counter2();
        Counter2 c2=new Counter2();
        Counter2 c3=new Counter2();
    }
}
```

1
2
3

Static Keyword

Static Variable	Non- Static Variable
Static variables can be accessed using class name. Syntax class_name.variable_name	Non static variables can be accessed using instance of a class. Syntax obj_ref.variable_name
Static variables can be accessed by static and non static methods	Non static variables cannot be accessed inside a static method.
Static variables reduce the amount of memory used by a program.	Non static variables do not reduce the amount of memory used by a program
Static variables are shared among all instances of a class.	Non static variables are specific to that instance of a class.
Static variable is like a global variable and is available to all methods.	Non static variable is like a local variable and they can be accessed through only instance of a class.
Memory is allocated at the time of loading of class so that these are also known as class variable.	Non-static variable also known as instance variable while because memory is allocated whenever instance is created.

Static Keyword

Java static method:-

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Static Keyword

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

    static void change(){
        college = "SSGPIT";
    }

    Student(int r, String n){
        rollno = r;
        name = n;
    }

    void display(){System.out.println(rollno+" "+name+" "+college);}
}

public class TestStaticMethod{
    public static void main(String args[]){
        Student.change();//calling change method
        //creating objects
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        Student s3 = new Student(333,"Sonoo");
        //calling display method
        s1.display();
        s2.display();
        s3.display();
    }
}
```

111 Karan SSGPIT

222 Aryan SSGPIT

333 Sonoo SSGPIT

Static Keyword

//Java Program to get the cube of a given number using the static method

```
class Calculate{  
    static int cube(int x){  
        return x*x*x;  
    }  
  
    public static void main(String args[]){  
        int result=Calculate.cube(5);  
        System.out.println(result);  
    }  
}
```

Static Keyword

Restrictions for the static method :-

- There are two main restrictions for the static method.
- The static method can not use non static data member or call non-static method directly.
- this and super cannot be used in static context.

Why is the Java main method static?

- It is because the object is not required to call a static method.
- If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

Static Keyword

Java program to call a non-static method

```
class Test{
    // static method
    public int sum(int a, int b)
    {
        return a + b;
    }
}

class Main {
    public static void main(String[] args)
    {
        int n = 3, m = 6;
        Test g = new Test();
        int s = g.sum(n, m);
        // call the non-static method
        System.out.print("sum is = " + s);
    }
}
```

Output:

sum is = 9|

Static Keyword

Static Method	Non- Static Method
<p>These method always preceded by static keyword.</p> <p>Syntax static void fun2() {....}</p>	<p>These method never be preceded by static keyword</p> <p>Syntax void fun2() {....}</p>
<p>Memory is allocated only once at the time of class loading.</p>	<p>Memory is allocated multiple time whenever method is calling.</p>
<p>These are common to every object so that it is also known as member method or class method.</p>	<p>It is specific to an object so that these are also known as instance method.</p>
<p>These property always access with class reference</p> <p>Syntax: className.methodname();</p>	<p>These methods always access with object reference</p> <p>Syntax: Objref.methodname();</p>
<p>If any method wants to be execute only once in the program that can be declare as static .</p>	<p>If any method wants to be execute multiple time that can be declare as non static.</p>

Static Keyword

Java static block :-

- It is used to initialize the static data member.
- It is executed before the main method at the time of class loading.

```
class A2{  
    static{System.out.println("static block is invoked");}  
    public static void main(String args[]){  
        System.out.println("Hello main");  
    }  
}
```

Output:static block is invoked
Hello main

Static Keyword

Static block executes before the main method:-

- A Class has to be loaded in main memory before we start using it.
- Static block is executed during class loading.
- This is the reason why static block executes before the main method.

Static Keyword

```
class Main {  
    // static variables  
    static int a = 23;  
    static int b;  
    static int max;  
  
    // static blocks  
    static {  
        System.out.println("First Static block.");  
        b = a * 4;  
    }  
    static {  
        System.out.println("Second Static block.");  
        max = 30;  
    }  
  
    // static method  
    static void display() {  
  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("max = " + max);  
    }  
  
    public static void main(String args[]) {  
        // calling the static method  
        display();  
    }  
}
```

First Static block.
Second Static block.

a = 23
b = 92
max = 30

Method Overloading

- If class have multiple methods with same name but different parameters is known as Method Overloading.
- Method overloading is also known as compile time (static) polymorphism.
- The same method name will be used with different number of parameters and parameters of different type.
- Overloading of methods with different return types is not allowed.
- Compiler identifies which method should be called among all the methods have same name using the type and number of arguments.
- However, the two functions with the same name must differ in at least one of the following,
 - The number of parameters
 - The data type of parameters
 - The order of parameter

Method Overloading

```
class overloadingDemo
{
void sum(int a,int b)
{
System.out.println("Sum of (a+b) is:: "+(a+b));
}
void sum(int a,int b,int c)
{
System.out.println("Sum of (a+b+c) is:: "+(a+b+c));
}
void sum(double a,double b)
{
System.out.println("Sum of double (a+b) is:: "+(a+b));
}
public static void main(String args[])
{
    overloadingDemo o1 = new overloadingDemo();
    o1.sum(10,10);           // call method1
    o1.sum(10,10,10);        // call method2
    o1.sum(10.5,10.5);       // call method3
}
}
```

Sum of (a+b) is:: 20
Sum of (a+b+c) is:: 30
Sum of double (a+b) is:: 21.0

Constructors

- Constructor is a special method that gets **invoked** “**automatically**” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the **same name as the class name**.
- Constructor **cannot return values**.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

Constructors

- A constructor is different from an ordinary method in the following aspects:
 - The name of the constructor method is the same as the class name.
 - Constructor has no return type. It implicitly returns void. No return statement is allowed inside the constructor's body.
 - Constructor can only be invoked via the "new" operator. It can only be used *once* to initialize the instance constructed. Once an instance is constructed, you cannot call the constructor anymore.
 - Constructors are not inherited

Constructors

- Types of Constructors:
- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Private Constructor

Constructors

■ Default Constructor:

- A constructor with **no parameter** is called the *default constructor*.
- It initializes the member variables to their default value.
- For example,
 - Circle() constructor initialize member variables radius and color to their default value.

Defining a Constructor

- Like any other method

```
public class ClassName {  
  
    // Data Fields..  
  
    // Constructor  
    public ClassName()  
    {  
        // Method Body Statements initialising Data  
        Fields  
    }  
  
    //Methods to manipulate data fields  
}
```

- Invoking:

- There is NO explicit invocation statement needed:
When the object creation statement is executed, the constructor method will be executed automatically.

Defining a Constructor: Example

```
class Bike1{  
    //creating a default constructor  
    Bike1()  
    {  
        System.out.println("Bike is created");  
    }  
    //main method  
    public static void main(String args[])  
    {  
        //calling a default constructor  
        Bike1 b=new Bike1();  
    } }
```

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

```
class BoxDemo6 {  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box objects  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
  
        double vol;  
  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

When this program is run, it generates the following results:

```
Constructing Box  
Constructing Box  
Volume is 1000.0  
Volume is 1000.0
```

Defining a Constructor: Example

```
public class Counter {  
    int CounterIndex;  
  
    // Constructor  
    public Counter()  
    {  
        CounterIndex = 0;  
    }  
    //Methods to update or access counter  
    public void increase()  
    {  
        CounterIndex = CounterIndex + 1;  
    }  
    public void decrease()  
    {  
        CounterIndex = CounterIndex - 1;  
    }  
    int getCounterIndex()  
    {  
        return CounterIndex;  
    }  
}
```


Trace counter value at each statement and What is the output ?

```
class MyClass {  
    public static void main(String args[]){  
        Counter counter1 = new Counter();  
        counter1.increase();  
        int a = counter1.getCounterIndex();  
        counter1.increase();  
        int b = counter1.getCounterIndex();  
        if ( a > b )  
            counter1.increase();  
        else  
            counter1.decrease();  
        System.out.println(counter1.getCounterIndex());  
    }  
}
```

A Counter with User Supplied Initial Value ?

- This can be done by adding another constructor method to the class.

```
public class Counter {  
    int CounterIndex;  
  
    // Constructor 1  
    public Counter()  
    {  
        CounterIndex = 0;  
    }  
    public Counter(int InitValue )  
    {  
        CounterIndex = InitValue;  
    }  
}  
  
// A New User Class: Utilising both constructors  
Counter counter1 = new Counter();  
Counter counter2 = new Counter (10);
```

Adding a Multiple-Parameters Constructor to our Circle Class

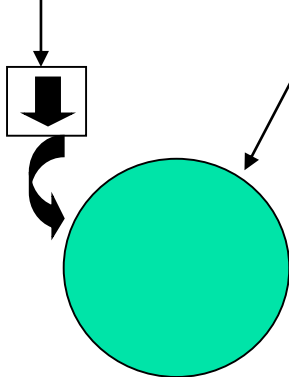
```
public class Circle {  
    public double x,y,r;  
    // Constructor  
    public Circle(double centreX, double centreY,  
                  double radius)  
    {  
        x = centreX;  
        y = centreY;  
        r = radius;  
    }  
    //Methods to return circumference and area  
    public double circumference() { return 2*3.14*r; }  
    public double area() { return 3.14 * r * r; }  
}
```

Constructors initialise Objects

- Recall the following OLD Code Segment:

```
Circle aCircle = new Circle();  
aCircle.x = 10.0; // initialize center and radius  
aCircle.y = 20.0  
aCircle.r = 5.0;
```

aCircle = new Circle() ;



At creation time the center and radius are not defined.

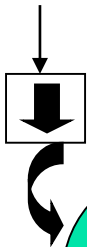
These values are explicitly set later.

Constructors initialise Objects

- With defined constructor

```
Circle aCircle = new Circle(10.0, 20.0, 5.0);
```

```
aCircle = new Circle(10.0, 20.0, 5.0) ;
```



aCircle is created with center (10, 20)
and radius 5

Parameterized Constructors

- A constructor that has parameters is known as parameterized constructor.
- It is used to provide different values to the distinct objects.
- It is required to pass parameters on creation of objects.
- If we define only parameterized constructors, then we cannot create an object with default constructor. This is because compiler will not create default constructor. You need to create default constructor explicitly.

Parameterized Constructors

```
//Java Program to demonstrate the use of the parameterized constructor.
class A{
    int id;
    String name;
    //creating a parameterized constructor
    A(int i,String n){
        id = i;
        name = n;
    }
    //method to display the values
    void display()
    {System.out.println(id+" "+name);}

    public static void main(String args[]){
        //creating objects and passing values
        A a = new A(111,"Karan");
        A a1 = new A(222,"Aryan");
        //calling method to display the values of object
        a.display();
        a1.display();
    }
}
```

111 Karan
222 Aryan

Parameterized Constructors

```
class salary
{
    int basic;
    double da,hra;
    salary()
    {
        System.out.println("Default constructor called");
    }
    salary(int b)
    {
        System.out.println("parametreised constructor called");
        basic=b;
        da=basic*0.2;
        hra=basic*0.1;
    }
    double computesalary()
    { return basic+da+hra; }
}
class basic
{
    public static void main(String args[])
    {
        double sal;
        salary e1=new salary();
        salary e2=new salary(5000);
        sal=e2.computesalary();
        System.out.println("salary is :"+sal);
    }
}
```

Default constructor called

parametreised constructor called

salary is :6500.0

Parameterized Constructors

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

Parameterized Constructors

```
class BoxDemo7 {  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box objects  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(3, 6, 9);  
  
        double vol;  
  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

The output from this program is shown here:

```
Volume is 3000.0  
Volume is 162.0
```

Copy Constructors

- A copy constructor is a constructor that takes only one parameter which is the same type as the class in which the copy constructor is defined.
- A copy constructor is used to create another object that is a copy of the object that it takes as a parameter. But, the newly created copy is totally independent of the original object.
- It is independent in the sense that the copy is located at different address in memory than the original.

Copy Constructors

```
class student
{
    int id;
    String name;
    student(int i,String n)
    {
        id=i;
        name=n;
    }
    student(student s)
    {
        id=s.id;
        name=s.name;
    }
    void display()
    {
        System.out.println(id+" "+name);
    }
    public static void main(String args[])
    {
        student s1=new student(10,"xyz");
        student s2=new student(20,"abc");
        student s3=new student(s1);
        s1.display();
        s2.display();
        s3.display();
    }
}
```

10xyz

20abc

10xyz

Copy Constructors

Advantages of copy constructor:

- The Copy constructor is easier to use when our class contains a complex object with several parameters.
- Whenever we want to add any field to our class, then we can do so just by changing the input to the constructor.
- One of the most crucial importance of copy constructors is that there is no need for any typecasting.
- Copy Constructors allow us to change the fields declared as final.

Private Constructors

- Java allows us to declare a constructor as private.
- We can declare a constructor private by using the private access specifier.
- Note that if a constructor is declared private, we are not able to create an object of the class.
- we can use this private constructor in **Singleton Design Pattern**.
- i.e. a class must ensure that only single instance should be created and single object can be used by all other classes.

Rules for Private Constructors

- It does not allow a class to be sub-classed.
- It does not allow to create an object outside the class.
- If a class has a private constructor and when we try to extend the class, a compile-time error occurs.
- We cannot access a private constructor from any other class.
- If all the constant methods are there in our class, we can use a private constructor.
- If all the methods are static then we can use a private constructor.
- We can use a public function to call the private constructor if an object is not initialized.
- We can return only the instance of that object if an object is already initialized.

Private Constructors

```
class Scaler{  
    private Scaler () {  
        System.out.println("Hello World!");  
    }  
    public static void main(String args[]){  
        Scaler obj = new Scaler();  
    }  
}
```

Hello World!

```
class A{  
    private A() {  
        System.out.println("Hello World!");  
    }  
}  
class Scaler{  
    public static void main(String args[]){  
        A obj = new A();  
    }  
}
```

```
Scaler.java:8: error: A() has private access in A  
    A obj = new A();  
                ^
```

1 error

Private Constructors

```
class A{  
    private A() {  
        System.out.println("Hello World!");  
    }  
    static A getInstanceOfA(){  
        return new A();  
    }  
}  
class Scaler{  
    public static void main(String args[]){  
        A obj = A.getInstanceOfA();  
    }  
}
```

Hello World!

Private Constructors

```
class SingletonObject {  
    private SingletonObject() {  
        System.out.println("In a private constructor");  
    }  
    public static SingletonObject getObject() {  
        // we can call this constructor  
        if (ref == null)  
            ref = new SingletonObject();  
        return ref;  
    }  
    private static SingletonObject ref;  
}  
public class PrivateConstructorDemo {  
    public static void main(String args[]) {  
        SingletonObject sObj = SingletonObject.getObject();  
    }  
}
```

In a private constructor

Constructor Overloading

- Constructor overloading in java allows to more than one constructor inside one Class.
- It is not much different than method overloading. In Constructor overloading you have multiple constructors with different signature with only difference that constructor doesn't have return type.
- These types of constructor known as overloaded constructor.

Constructor Overloading

```
/* programe defines three constructors to initialize the dimensions of a
box various ways using constructor overloading. */
class Box {
    double width;
    double height;
    double depth;
    // constructor used when all dimensions specified
    Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
    Box() {                // constructor used when no dimensions specified
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }
    Box(double len) {      // constructor used when cube is created
        width = height = depth = len;
    }
    double volume() {      // compute and return volume
        return width * height * depth;
    }
}
```

Constructor Overloading

```
class OverloadCons {  
    public static void main(String args[]) {  
        // create boxes using the various constructors  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box();  
        Box mycube = new Box(7);  
        double vol;  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume of mybox2 is " + vol);  
        // get volume of cube  
        vol = mycube.volume();  
        System.out.println("Volume of mycube is " + vol);  
    }  
}
```

The output produced by this program is shown here:

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of mycube is 343.0

Constructor Overloading

```
class salary
{
    int basic;
    double da,hra;
    salary()
    {
        System.out.println("Default constructor called");
        basic=0;
        da=0.0;
        hra=0.0;
    }
    salary(int b, double d, double h)
    {
        System.out.println("parametreised constructor called");
        basic=b;
        da=basic*d;
        hra=basic*h;
    }
    salary(int b)
    {
        basic=b;
        da=hra=0.0;
    }
    double computesalary()
    { return basic+da+hra;}
}
```

Constructor Overloading

```
class basic
{
    public static void main(String args[])
    {
        double sal;
        salary e1=new salary();
        sal=e1.computesalary();
        System.out.println("salary is :"+sal);
        salary e2=new salary(5000,0.2,0.1);
        sal=e2.computesalary();
        System.out.println("salary is :"+sal);
        salary e3=new salary(10000);
        sal=e3.computesalary();
        System.out.println("salary is :"+sal);
    }
}
```

Default constructor called salary is :0.0
parametreised constructor called salary is :6500.0
salary is :10000.0

String Class:-

- Strings are widely used in JAVA Programming, are not only a sequence of characters but it defines object.
- String Class is defined in java.lang package.
- The String type is used to declare string variables. Also we can declare array of strings.
- A variable of type String can be assign to another variable of type String.
- It is **Final** class
- Due to Final, String class cannot be inherited.
- It is immutable.

String Class:-

- **Creating a String:-**
- **String literal :-**
- `String s = "Ghandhy College";`
- To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).
- **Using new keyword :-**
- `String s = new String ("Ghandhy College");`
- **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).

Methods/Operations of String Class

Method	Description
<u>length()</u>	Returns the length of this string.
<u>toUpperCase()</u>	Converts all of the characters in this String to upper case.
<u>toLowerCase()</u>	Converts all of the characters in this String to lower case.
substring():	extracts a substring from the string and returns it.
<u>charAt</u>(int index)	Returns the char value at the specified index.
<u>compareTo</u>(String anotherString)	Compares two strings lexicographically.
<u>equals</u>(Object anObject)	Compares this string to the specified object.
contains():	method checks whether the specified string (sequence of characters) is present in the string or not
<u>split</u>(String regex)	Splits this string around matches of the given regular expression.

String Class:-

- **Length():**To get the length of the string

```
public class StringExample{  
    public static void main(String args[]){  
  
        char ch[]={'s','t','r','i','n','g','s'};  
        String s1=new String(ch);  
        String s2="java";  
        String s3=new String("example");  
  
        System.out.println("length of string:"+s1.length());  
        System.out.println("length of string:"+s2.length());  
        System.out.println("length of string:"+s3.length());  
    }}
```

```
length of string:7  
length of string:4  
length of string:7
```

String Class:-

- toUpperCase(): Converts all of the characters in this String to upper case.
- toLowerCase(): Converts all of the characters in this String to lower case.

```
public class StringLowerExample{  
    public static void main(String args[]) {  
        String s1="Information Technology Department";  
        System.out.println(s1);  
        System.out.println(s1.toLowerCase());  
        System.out.println(s1.toUpperCase());  
    }  
}
```

```
Information Technology Department  
information technology department  
INFORMATION TECHNOLOGY DEPARTMENT
```

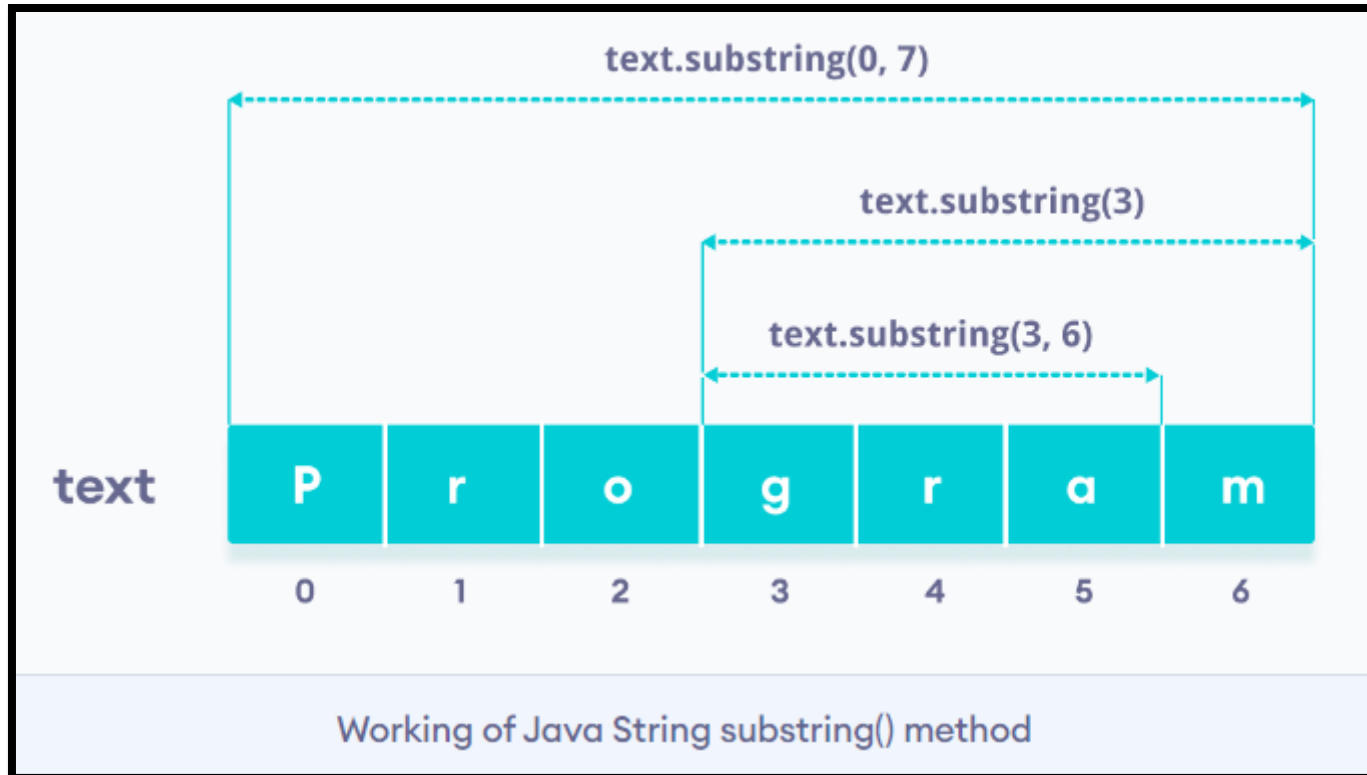
String Class:-

- **substring()**: extracts a substring from the string and returns it.

Syntax :

- **string.substring(int startIndex, int endIndex)**
- Here, string is an object of the String class.
- startIndex - the beginning index
- endIndex (optional) - the ending index
- The substring begins with the character at the startIndex and extends to the character at index endIndex - 1.
- If the endIndex is not passed, the substring begins with the character at the specified index and extends to the end of the string.

String Class:-



String Class:-

```
class Main {  
    public static void main(String[] args) {  
        String str1 = "program";  
  
        // from the first character to the end  
        System.out.println(str1.substring(0)); // program  
  
        // from the 4th character to the end  
        System.out.println(str1.substring(3)); // gram  
        // from 1st to the 7th character  
        System.out.println(str1.substring(0, 7)); // program  
  
        // from 1st to the 5th character  
        System.out.println(str1.substring(0, 5)); // progr  
  
        // from 4th to the 5th character  
        System.out.println(str1.substring(3, 5)); // gr  
    }  
}
```

String Class:-

- `charAt()`:method returns the character at the specified index.

Syntax:

- **`string.charAt(int index)`**
- Here, string is an object of the String class.
- `charAt()` Parameters:
- index - the index of the character (an int value)
- `charAt()` Return Value:
- returns the character at the specified index

String Class:-

```
class Main {  
    public static void main(String[] args) {  
        String str1 = "Learn Java";  
        String str2 = "Learn\nJava";  
  
        // first character  
        System.out.println(str1.charAt(0)); // 'L'  
  
        // seventh character  
        System.out.println(str1.charAt(6)); // 'J'  
  
        // sixth character  
        System.out.println(str2.charAt(5)); // '\n'  
    }  
}
```

String Class:-

- **Java String compare:-**
- There are three ways to compare String in Java:
- By Using equals() Method
- By Using == Operator
- By compareTo() Method

String Class:-

- equals():The equals() method returns true if two strings are equal. If not, it returns false.
- Syntax:
- **string.equals(String str)**
- Here, string is an object of the String class.
- equals() Parameters:-
- The equals() method takes a single parameter.
- str - the string to be compared
- equals() Return Value:-
- returns true if the strings are equal
- returns false if the strings are not equal
- returns false if the str argument is null

String Class:-

```
class Main {  
    public static void main(String[] args) {  
        String str1 = "Learn Java";  
        String str2 = "Learn Java";  
        String str3 = "Learn Kolin";  
        boolean result;  
  
        // comparing str1 with str2  
        result = str1.equals(str2);  
        System.out.println(result); // true  
  
        // comparing str1 with str3  
        result = str1.equals(str3);  
  
        System.out.println(result); // false  
  
        // comparing str3 with str1  
        result = str3.equals(str1);  
        System.out.println(result); // false  
    }  
}
```

String Class:-

- By Using == operator
- The == operator compares references not values.

```
class Teststringcomparison3{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3=new String("Sachin");  
        System.out.println(s1==s2);//true (because both refer to same instance)  
        System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)  
    }  
}
```

String Class:-

- By Using compareTo() method:-
- The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.
- Syntax:
- **string.compareTo(String str)**
- Suppose s1 and s2 are two String objects. If:
- $s1 == s2$: The method returns 0.
- $s1 > s2$: The method returns a positive value.
- $s1 < s2$: The method returns a negative value.

String Class:-

```
class Teststringcomparison4{  
    public static void main(String args[]){  
  
        String s1="Rohan";  
        String s2="Rohan";  
        String s3="Sohil";  
  
        System.out.println(s1.compareTo(s2));//0  
        System.out.println(s1.compareTo(s3));//1(because s1>s3)  
        System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )  
    }  
}
```

String Class:-

- `contains()`: method checks whether the specified string (sequence of characters) is present in the string or not.
- **Syntax:**
- **`string.contains(CharSequence ch)`**
- Here, `string` is an object of the `String` class.
- `contains()` Parameters:
- The `contains()` method takes a single parameter.
- `ch (charSequence)` - a sequence of characters
- Note: A `charSequence` is a sequence of characters such as: `String`, `CharBuffer`, `StringBuffer` etc.
- returns `true` if the string contains the specified character
- returns `false` if the string doesn't contain the specified character

String Class:-

```
class Main {  
    public static void main(String[] args) {  
        String str1 = "Learn Java";  
        Boolean result;  
  
        // check if str1 contains "Java"  
        result = str1.contains("Java");  
        System.out.println(result); // true  
  
        // check if str1 contains "Python"  
        result = str1.contains("Python");  
        System.out.println(result); // false  
  
        // check if str1 contains ""  
        result = str1.contains("");  
  
        System.out.println(result); // true  
    }  
}
```

String Class:-

- `split()`:method divides the string at the specified regex and returns an array of substrings.
- **Syntax:-**
- **`string.split(String regex, int limit)`**
- The string `split()` method can take two parameters:
- `regex` - the string is divided at this regex (can be strings)
- `limit` (optional) - controls the number of resulting substrings
- If the `limit` parameter is not passed, `split()` returns all possible substrings.
- **`split()` Return Value:-**
- returns an array of substrings

String Class:-

```
public class SplitExample{  
    public static void main(String args[]) {  
        String s1="java string split method of string ";  
        String[] words=s1.split(" "); //splits the string based on string  
        //using java foreach loop to print elements of string array  
        for(String w:words){  
            System.out.println(w);  
        }  
    }  
}
```

java
string
split
method
of
string

StringBuffer Class:-

- Java StringBuffer class is used to create mutable (modifiable) String objects.
- The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.
- Java StringBuffer class is a thread-safe, mutable sequence of characters.
- Every string buffer has a capacity.
- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
- StringBuffer defines 4 constructors.

StringBuffer Class:-

Constructor	Description
StringBuffer()	This constructs a string buffer with no characters in it and an initial capacity of 16 characters.
StringBuffer(CharSequence seq)	This constructs a string buffer that contains the same characters as the specified CharSequence .
StringBuffer(int capacity)	This constructs a string buffer with no characters in it and the specified initial capacity .
StringBuffer(String str)	This constructs a string buffer initialized to the contents of the specified string .

StringBuffer Class:-

```
class Test {  
    public static void main(String args[])  
    {  
        String str = "study";  
        str.concat("tonight");  
        System.out.println(str);           // Output: study  
  
        StringBuffer strB = new StringBuffer("study");  
        strB.append("tonight");  
        System.out.println(strB);         // Output: studytonight  
    }  
}
```

StringBuffer Class:-

■ Methods of StringBuffer Class:

Method	Description
capacity()	Returns the current capacity of the String buffer.
charAt(int index)	This method returns the char value in this sequence at the specified index.
toString()	This method returns a string representing the data in this sequence.
insert(int offset, char c)	Inserts the string representation of the char argument into this character sequence.
<u>append</u> (<u>String</u> str)	Appends the string to this character sequence.
reverse()	The character sequence contained in this string buffer is replaced by the reverse of the sequence.

StringBuffer Class:-

- `append()`:-This method will concatenate the string representation of any type of data to the end of the StringBuffer object.
- `append()` method has several overloaded forms.
- `StringBuffer append(String str)`
- `StringBuffer append(int n)`
- `StringBuffer append(Object obj)`

```
public class Demo {  
    public static void main(String[] args)  
    {  
        StringBuffer str = new StringBuffer("test");  
        str.append(123);  
        System.out.println(str);  
    }  
}
```

OUTPUT:TEST123

StringBuffer Class:-

- insert():method inserts one string into another. Here are few forms of insert() method.
- StringBuffer insert(int index, String str)
- StringBuffer insert(int index, int num)
- StringBuffer insert(int index, Object obj)
- Here the first parameter gives the index at which position the string will be inserted and string representation of second parameter is inserted into StringBuffer object.

```
public class Demo {  
    public static void main(String[] args) {  
        StringBuffer str = new StringBuffer("test");  
        str.insert(2, 123);  
        System.out.println(str);  
    }  
}
```

OUTPUT:TEST123

StringBuffer Class:-

- `reverse()`: method reverses the characters within a `StringBuffer` object.

```
public class Demo {  
    public static void main(String[] args) {  
        StringBuffer str = new StringBuffer("Hello");  
        str.reverse();  
        System.out.println(str);  
    }  
}
```

OUTPUT:OLLEH

StringBuffer Class:-

- `replace()`: method replaces the string from specified start index to the end index.

```
public class Demo {  
    public static void main(String[] args) {  
        StringBuffer str = new StringBuffer("Hello World");  
        str.replace( 6, 11, "java");  
        System.out.println(str);  
    }  
}
```

OUTPUT:HELLO JAVA

StringBuffer Class:-

- `capacity()`:- method returns the current capacity of StringBuffer object.

```
public class Demo {  
    public static void main(String[] args) {  
        StringBuffer str = new StringBuffer();  
        System.out.println( str.capacity() );  
    }  
}
```

OUTPUT:16

String v/s StringBuffer Class:-

String	StringBuffer
It is immutable means you cannot modify.	It is mutable means you can modify.
String class is slower than the StringBuffer.	StringBuffer class is faster than the String.
String is not safe for use by multiple threads .	String buffers are safe for use by multiple threads .
String Class not provides insert() Operation.	StringBuffer Class provides insert() Operation.
String Class provides split() Operation.	StringBuffer Class not provides split() Operation.
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.

StringJoiner Class:-

- Java StringJoiner class is included in the Java 8 version that allows us to construct a sequence of characters separated by a delimiter.
- It is located in java.util package and used to provide utility to create a string by a user-defined delimiter.
- the delimiter can be anything like comma (,), colon(:) etc. We can also pass suffix and prefix to the string joiner object.
- **Declaration:-**
- **public final class StringJoiner extends Object**

StringJoiner Class:-

Method	Description
Public StringJoiner add(CharSequence newElement)	It adds a copy of the given CharSequence value as the next element of the StringJoiner value. If newElement is null,"null" is added.
Public StringJoiner merge(StringJoiner other)	It adds the contents of the given StringJoiner without prefix and suffix as the next element if it is non-empty. If the given StringJoiner is empty, the call has no effect.
Public int length()	It returns the length of the String representation of this StringJoiner.
Public StringJoiner setEmptyValue(CharSequence emptyValue)	It sets the sequence of characters to be used when determining the string representation of this StringJoiner and no elements have been added yet, that is, when it is empty.
public String toString()	It returns the current value, consisting of the prefix, and the suffix or the empty value characters are returned.

StringJoiner Class:-

```
import java.util.StringJoiner;
public class STDemo {
    public static void main(String[] args) {
        StringJoiner sj = new StringJoiner(",");
        sj.add("India");
        sj.add("China");
        sj.add("US");
        sj.add("UK");
        System.out.println(sj);
    }
}
```

OUTPUT:

India,China,US,UK

StringJoiner Class:-

```
import java.util.StringJoiner;
public class STDemo {
    public static void main(String[] args) {
        StringJoiner sj = new StringJoiner(":", "(");
        sj.add("India");
        sj.add("China");
        sj.add("US");
        sj.add("UK");
        System.out.println(sj);
    }
}
```

OUTPUT:

(India:China:US:UK)

Wrapper Class:-

- Wrapper class wraps (encloses) around a data type and gives it an object appearance.
- Wrapper classes are used to convert any data type into an object.
- The primitive data types are not objects and they do not belong to any class.
- So, sometimes it is required to convert data types into objects in java.
- Wrapper classes include methods to unwrap the object and give back the data type.
- Example:
 - `int k = 100;`
 - `Integer it1 = new Integer(k);`
 - The int data type k is converted into an object, it1 using Integer class.

Wrapper Class:-

- The it1 object can be used wherever k is required an object.
- To unwrap (getting back int from Integer object) the object it1.
Example:
 - `int m = it1.intValue();`
 - `System.out.println(m*m); // prints 10000`
 - `intValue()` is a method of Integer class that returns an int data type.

Wrapper Class:-

- Eight wrapper classes exist in java.lang package that represent 8 data types:

Primitive Data Type	Wrapper Class	Unwrap Methods
byte	Byte	byteValue()
short	Short	shortValue()
int	Integer	intValue()
long	Long	longValue()
float	Float	floatValue()
double	Double	doubleValue()
char	Character	charValue()
boolean	Boolean	booleanValue()

Wrapper Class:-

- There are mainly two uses with wrapper classes.
- 1) To convert simple data types into objects.
- 2) To convert strings into data types (known as parsing operations), here methods of type
- `parseX()` are used. (Ex. `parseInt()`)
- The wrapper classes also provide methods which can be used to convert a `String` to any of the
- primitive data types, except character.
- •These methods have the format `parseX()` where x refers to any of the primitive data types except `char`.
- •To convert any of the primitive data type value to a `String`, we use the `valueOf()` methods of the `String` class.

Wrapper Class:-

- `int x = Integer.parseInt("34"); // x=34`
- `double y = Double.parseDouble("34.7"); // y =34.7`
- `String s1= String.valueOf('a'); // s1="a"`
- `String s2=String.valueOf(true); // s2="true"`