Class Fundamentals

- A ***class*** is a template that defines the form of an object.
- It specifies both data and the code that will operate on that data.
- Java uses a class specification to construct **objects.**
- **Objects** are instances of a class.
- **A class** is essentially a set of plans that specify how to build an objects
- **General form of class:**

```
class classname {
  // declare instance variables
  type var1;
  type varN;
     // declare methods
     type method1(parameters) {
     //body of method
     }
} //end class
```

- Example:

```
class Vehicle {
        int passengers; // number of passengers
        int fuelCap; // fuel capacity in litres
        int mpg; // fuel consumption in miles per litres
       }
```

- A class definition creates a new data type. In this case, the **new data type** is called **Vehicle.**

How objects are Created

- The following line is used to create object:
  - *It declares variable called v of the class type Employee.*
  - *The declaration creates a physical copy of the object and assigns to v a reference to that object. This is done by using new operator.*
  - *the new keyword dynamically allocates memory for an object and returns a reference to it.*
  - Vehicle v; // declaring reference
  - v = new Vehicle // allocating a vehicle object
  - The **dot operator(.)** links the name of an object with the name of a member.
  - The general form of the **dot operator** is : **object.member**
  - In general the dot(.)operator is used to access **both instance variables and methods.**

Example: Vehicle v =new Vehicle( );

```
class Vehicle {
    int passengers;
    int fuelCap;
    int mpg;
}
class VehicleDemo {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
            int range;
              v.passengers = 7;
              v.fuelCap = 16;
              v.mpg = 21;
              range = v.fuelCap * v.mpg;
    System.out.println("Minivan can carry " + v.passengers + " with a range of
    " + range); }
}
```

Reference Variables and Assignments

- Assigning one object reference variable to another.

```
Rectangle r1 =  new Rectangle( );
Rectangle r2 = r1; // assigning r1
```

- r1 is reference variable which contain the address of Actual Rectangle Object
- r2 is another reference variable
- r2 is initialized with r1 means – "**r1 and r2**" both are referring same object , thus it does not create duplicate object , nor does it allocate extra memory.

```
class Rectangle {
  double length;
  double breadth;
}

class RectangleDemo {

  public static void main(String args[]) {

  Rectangle r1 = new Rectangle();
  Rectangle r2 = r1;

  r1.length = 10;
  r2.length = 20;

  System.out.println("Value of R1's Length : " + r1.length); //
  System.out.println("Value of R2's Length : " + r2.length); //

  }
}
```

Methods

- A method contains the statement that define its action.
  - Each method has a name
  - It is the same name used to call the method
  - General form

    re-type name(parameter-list)
    {
    //body of method
    }
  - ***Here, return_type*** is nothing but the value to be returned to an calling method.
  - ***method_name*** is an name of method that we are going to call through any method.
  - ***arg1,arg2,arg3*** are the different parameters that we are going to pass to a method
- **Return type of method**
  1. Method can return any type of value.
  2. Method can return any Primitive data type.
  3. Method can return Object of Class Type.
  4. Method sometimes may not return value.
- ***Method Name***
  1. Method name must be valid identifier.
  2. All Variable naming rules are applicable for writing Method Name.
- ***Parameter List***

1. 1. Method can accept ***any*** number of parameters.
2. Method can accept any ***data type*** as parameter.
3. Method can accept ***Object*** as Parameter

4. Method can accept ***no*** Parameter.
5. Parameters are separated by ***Comma***.
6. Parameter must have ***Data Type***

Returning from a method

```
class Rectangle {
  double length;
  void setLength(int len)
          {
                  length = len;
           }
}
class RectangleDemo {
  public static void main(String args[]) {
  Rectangle r1 = new Rectangle();
  r1.setLength(20);
  System.out.println("After Function Length : " + r1.length);
  }
}
```

Returning value

- Return values are used for a variety of purposes in programming.
- In some cases, such as **sqrt(),** the return value contains the outcome of some calculation.
- In other cases, the return value simply indicate success or failure.
- In still others, it may contain a status code.
- Methods return a value to the calling routine using this form of **return:**
    - return value;

Using Parameters

- It is possible to pass one or more values to a method when the method is called.
- A value passed to a method is called an ***argument.***
- Inside the method, the variable that receives the argument is called a ***parameter.***
- Parameters are declared inside the parentheses that follow the method's name.
- The parameter declaration syntax is the same as that used for variables

```
class Rectangle {

        double lenght,  breadth;

        void setLength(double len){
                length= len;
        }
        int getLength()
        {
                return lenght*length;
        }
class RectDemo {
        public static void main(String [] args)

        {       Rectangle r1 = new Rectangle();
                r1. setlength(10);
                int sum = r1.getLenght();
                System.out.println("length"+sum);
        }

}
```

Constructors

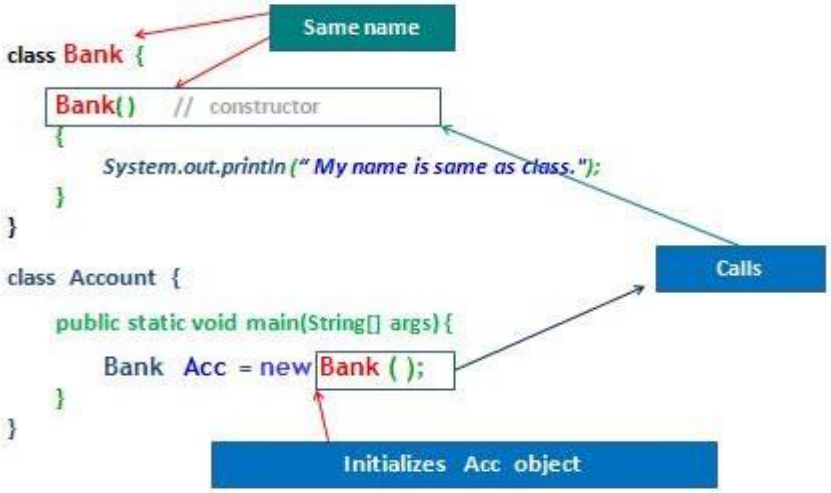
- a member function which initializes a class
- A constructor has:
  - (i) the same name as the class itself
  - (ii) no return type

Types of Constructors

1. Default constructor
2. Parameterized constructor

Default Constructor

- A constructor that have no parameter is known as default constructor.



Parameterized Constructors

- A constructor that have parameters is known as parameterized constructor.
- Parameterized constructor are used to provide different values to distinct object.

```
class Student4{
   int id;
   String name;

   Student4(int i,String n){
   id = i;
   name = n;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```

Garbage Collection and Finalizers

- A destructor is a special method typically used to perform cleanup after an object is no longer needed.
- No destructors in java.
- Alternative in JAVA : garbage collection
- Garbage Collection is carried by a *daemon thread* called *Garbage Collector*.
- Destroying object by garbage collector
  - Before Destroy invokes finalize () method.
- User can't force Garbage collection; JVM triggers it if needed

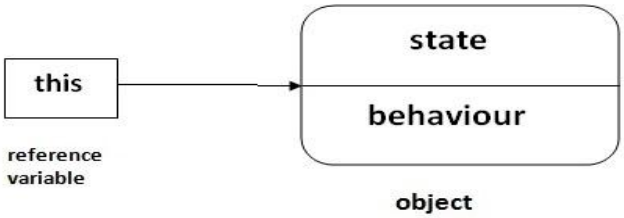
- Requesting garbage collection
  - System.gc() and Runtime.gc()
  - send request to JVM but Garbage collection is *not guaranteed*

```
protected void finalize ( )
{
    //finalize code
}
```

- *A finalizer* in Java is the opposite of a constructor.
- a finalizer method performs finalization for the object.
- Garbage collector can't free resources, such as open files and network connections.
- you need to write a finalize() method for any object that needs to perform such tasks.

The this Keyword

- *this* is a reference variable that refers to the *current object.*



- **5 usage of this keyword**
  - this.variable : access instance variable
  - this.method() : calls method of current class
  - this() : calls constructor of current class
  - method(this) :passes current object as an argument
  - return this : returns instance of current class

- **this.variable:**

```
class Student {
    int id;
    String name;
    Student(int id, String name){
        this.id=id;
        this.name=name;

        System.out.println("In constructor"+id+""+name);
    }
    public static void main(String[] args){
        Student s =new Student(10,"John");
    }
}
```

- **this.method( )**

```
class Student {

    void one(){
        System.out.println("one" );
        this.two();
    }

    void two(){

        System.out.println("Wellcome to Wase" );
    }

    public static void main(String args[]){
        Student s = new Student();
        s.one();
    }
}
```

- **this( )**
  - The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

```
class Demo{
    Demo()
    {
        System.out.println("In first constructor");

    }
    Demo(String name){
        this(); //should be first line
        System.out.println("In Second constructor "+name);
    }

    public static void main(String args[]){

        Demo d= new Demo("John");
    }
}
```

- **method(this)**
  - The this keyword can also be passed as an argument in the method. It is mainly used in the event handling.

```
class Student {

    void two( Student stud){
        System.out.println("two" );
        stud.three();
    }
    void three()
    {
        System.out.println("three");
    }
    void one()
    {
        System.out.println("one" );
        two(this);
    }

    public static void main(String args[]){
        Student s = new Student();
        s.one();
    }
}
```

- **return this**
  - to return the current class instance

```
class Rectangle{
 int length, breadth;

 Rectangle(int l, int b){
  length = l;
  breadth = b;
 }

 Rectangle getObj(){
  return this;
 }
}
class This5{
 public static void main (String[] args){
  Rectangle r1 = new Rectangle(15,20);
     int a;
  Rectangle r2;
  r2 = r1.getObj();
  System.out.println("length: " + r1.length + " breadth: " + r1.breadth);
  System.out.println("length: " + r2.length + " breadth: " + r2.breadth);
 }
}
```