Class: A **class** is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. **Java class** objects exhibit the properties and behaviors **defined** by its **class**. A **class** can contain fields and methods to describe the behavior of an object. For example Automobile,chair, Book are some examples of a class .

Object:Object refers to a particular instance of a class where the **object** can be a combination of variables, functions, and data structures. For example for automobile class, Lexus can be an instance/object.

Syntax for defining a class in Java:

```
public class <class name>
{
    // data items or fields
   //  Constructors
   //  methods

}
```

Example:

```
public class Bicycle {

    // the Bicycle class has
    // three fields
    public int cadence;                    // public is a modifier
    public int gear;
    public int speed;

    // the Bicycle class has
    // one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has
    // four methods
    public void setCadence(int newValue) {
        cadence = newValue;
```

```
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

You can compile a Java class before you have a program in which to use it. The compiled bytecode for the class will be stored in a file of the same name, but ending in .class rather than .java. So compiling the file Bicycle.java will create a file called Bicycle.class. Later, you can compile a program file that uses the class Bicycle, and you will not need to recompile the class definition for Bicycle.

Complete example of a class, and then using it in a program:

```
public class Dog
{
    public String name;
    public String breed;
    public int age;
    public void writeOutput()
    {
        System.out.println("Name: " + name);
        System.out.println("Breed: " + breed);
        System.out.println("Age in calendar years: " +
                                    age);
        System.out.println("Age in human years: " +
                            getAgeInHumanYears());
        System.out.println();
    }
    public int getAgeInHumanYears()
    {
        int humanAge = 0;
        if (age <= 2)
        {
            humanAge = age * 11;
        }
        else
        {
            humanAge = 22 + ((age-2) * 5);
        }
            return humanAge;
    }
}
```

Later in this chapter we will see that the modifier `public` *for instance variables should be replaced with* `private`.

Class that creates an instance of Dog Class:

```
public class DogDemo
{
    public static void main(String[] args)
    {
        Dog balto = new Dog();
        balto.name = "Balto";
```

```
        balto.age = 8;
        balto.breed = "Siberian Husky";
        balto.writeOutput();
        Dog scooby = new Dog();
        scooby.name = "Scooby";
        scooby.age = 42;
        scooby.breed = "Great Dane";
        System.out.println(scooby.name + " is a " +
                           scooby.breed + ".");
        System.out.print("He is " + scooby.age +
                         " years old, or ");
        int humanYears = scooby.getAgeInHumanYears();
        System.out.println(humanYears + " in human years.");
    }
}
```

The following three lines from the start of the class definition define three instance variables:
public String name;
public String breed;
public int age;

The word public simply means that there are no restrictions on how these instance variables are used. Each of these lines declares one instance variable. Notice that each instance variable has a data type. For example, the instance variable name is of String String.
Each object, or instance, of the class has its own copy of these three instance variables, which is why they are called instance variables.

Use of This :
Within a method definition, you can use the keyword **this** as a name for the object receiving the method call.

Example:

```
public void writeOutput()
{ System.out.println("Name = " + this.name);
  System.out.println("Population = " + this.population);
  System.out.println("Growth rate = " + this.growthRate + "%");
 }
```

Constructor: A constructor is a method in the class which is used to initialize the data members in the class. A class contains constructors that are invoked to create objects from

the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type.
For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;
}
```

Creating objects of class: Objects of a class can be created after class definition using the following syntax:
<class name> <object name> = new Constructor(argument 1, argument 2,...);

Example:
To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

This statements has three parts :

1. Declaration: Associate a variable name with an object type.
2. Instantiation: The new keyword is a Java operator that creates the object.
3. Initialization: The new operator is followed by a call to a constructor, which initializes the new object.

The **new operator** creates space in memory for the object and initializes the data members or fields. **For example:**
new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

A class can have multiple constructors but they should have a unique argument list. Depending upon how many arguments are passed at object creation, respective constructor is called.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
   gear = 1;
   cadence = 10;
   speed = 0;
}
```

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.
You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass.

// Example for Nested Class and Class with four constructors

```
public class Point
{
   public int x = 0;
   public int y = 0;
   //constructor
   public Point(int a, int b)
  {
      x = a;
      y = b;
   }
}


public class Rectangle
 {
   public int width = 0;
   public int height = 0;
   public Point origin;

   //  constructors
   public Rectangle()
  {
      origin = new Point(0, 0);
   }
   public Rectangle(Point p)
  {
      origin = p;
   }
   public Rectangle(int w, int h)
  {
      origin = new Point(0, 0);
      width = w;
```

```
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y)
    {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing the area of the rectangle
    public int getArea()
    {
        return width * height;
    }
}
```

Referencing an Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

Fields may be accessed with their names within its own class. For example, we can add a statement within the Bicycle class that prints the gear and speed:

```
System.out.println("Speed and Gear are: " + speed + ", " + gear);
```

In this case, speed and gear are simple names.

Usage of Dot Operator outside the class

Fields and methods of a class can be accessed outside the class using an object reference followed by the dot (.) operator, followed by a simple field/method name using the following syntax:

```
objectReference.fieldName
```

For example:

```
System.out.println("Width of rectOne: "  + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
```

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. Recall that the new operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new Rectangle object and immediately gets its height. In essence, the statement calculates the default height of a Rectangle. Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
objectReference.methodName(argumentList);
```

or:

```
objectReference.methodName();
```

For Example:

```
System.out.println("Area of rectOne: " + rectOne.getArea());

int areaOfRectangle = new Rectangle(100, 50).getArea();
```

In this case, the object that getArea() is invoked on is the rectangle returned by the constructor.