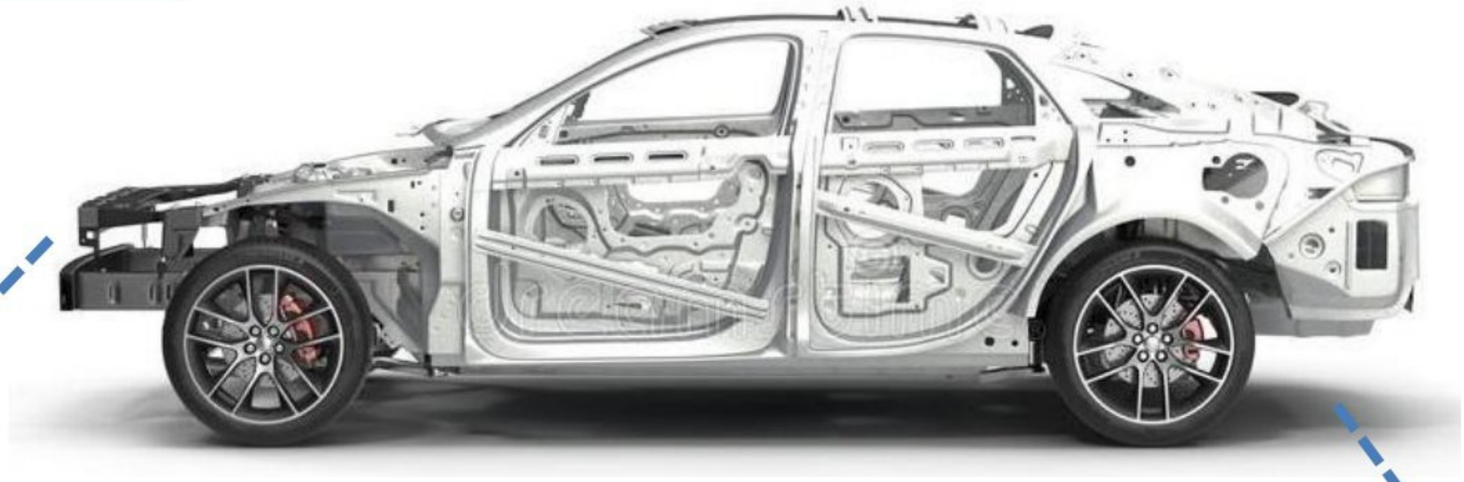
Class

- Class is **derived datatype**, it combines members of different datatypes into one.
- Defines new datatype (primitive ones are not enough).
  For Example : **Car**
- This new datatype can be used to create objects.
- A class is a template for an object .

Example :

```
class Car{
        String company;
        String model;
        double price;
        double milage;
        ……
}
```

Class: Car



Properties (Describe)
Company
Model
Color
Mfg. Year
Price
Fuel Type
Mileage
Gear Type
Power Steering
Anti-Lock braking system

Methods (Functions)
Start
Drive
Park
On_break
On_lock
On_turn

Object

- An object is an **instance** of a **class**.
- An object has a **state** and **behavior**.

  Example: A dog has

  states - color, name, breed as well as

  behaviors – wagging the tail, barking, eating.

- The **state** of an object is stored in **fields** (variables), while **methods** (functions) display the object's **behavior**.

Objects of a class Car

Honda City

Hyundai i20

Sumo Grand

Mercedes E class

Swift Dzire

Creating Object & Accessing members

- **new** keyword creates new object
- Syntax:

  ClassName objName = **new** ClassName();

  Example :

  SmartPhone iPhone = **new** SmartPhone();

- Object variables and methods can be accessed using the **dot (.)** operator
- Example:

  iPhone.storage = 8000;

Introducing methods

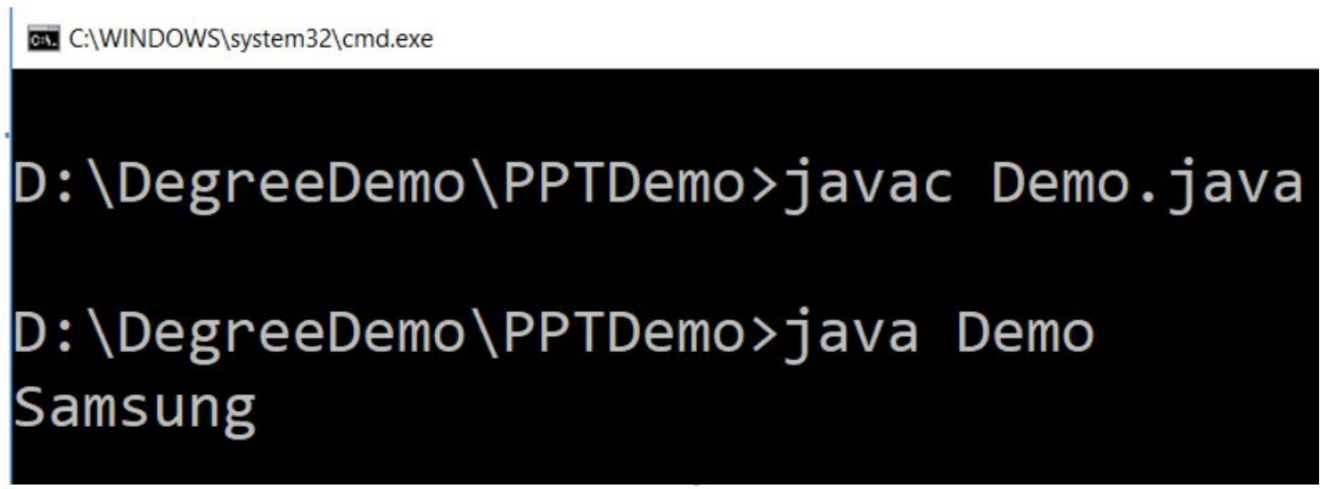
- Methods represents the **behavior** of a class.
- Remember : a Method is always invoked relative to an object. (except ….)
- Syntax:

```
access_specifier return_type method_name(argument_list)
{
    // code
}
```

Example (method)

```
public class SmartPhone {
    String manufacturer;
    String model;
    double storage;
    double screenSize;
    public String getManufacturer(){
        return manufacturer;
    }
    public void setManufacturer(String a){
        manufacturer = a;
    }
}
```

```
public class Demo {
    public static void main(String args[]) {
        SmartPhone sp = new SmartPhone();
        sp.setManufacturer("Samsung");
        String name = sp.getManufacturer();
        System.out.println(name);
    }
}
```



this keyword

- **this** is a reference variable that refers to the **current object**.
- Usage of java **this** keyword
  - this can be used to refer **current class instance variable**.
  - this can be used to invoke **current class method** (implicitly)
  - this() can be used to invoke **current class constructor**.
  - this can be **passed** as an **argument** in the **method call**.
  - this can be **passed** as **argument** in the **constructor call**.
  - this can be used to **return** the current class instance from the method.

Example (method) using this operator

```
public class SmartPhone {

    String manufacturer;
    String model;
    double storage;
    double screenSize;

    public String getManufacturer()
    {
        return manufacturer;
    }
    public void setManufacturer(String manufacturer)
    {
        this.manufacturer = manufacturer;
    }
}
```

Method Overloading

- Overloading allows **different methods** to have **same name**, but **different signatures**.
- Signature can differ by **number of input** parameters or **type of input** parameters or **both**.
- Overloading is related to **compile time** (or **static**) polymorphism.

Example (Method Overloading)

```
public class OverloadingMethods {
    public static void main(String[] ar) {
        int ans1 = sum(5,2); // will return 7
        int ans2 = sum(5,2,6); // will return 13
        double ans3 = sum(5.8,6.4); // will return 12.2
        // print ans1,ans2,ans3 in order to see the result
    }
    // Overloaded sum(). This sum takes two int parameters
    public static int sum(int x, int y) {
        return (x + y);
    }
    // Overloaded sum(). This sum takes three int parameters
    public static int sum(int x, int y, int z) {
        return (x + y + z);
    }
    // Overloaded sum(). This sum takes two double parameters
    public static double sum(double x, double y) {
        return (x + y);
    }
}
```

Constructor

- A **constructor** is a method that is called automatically when an instance of an object is created.
- Here are the key differences between a constructor and a method:
  - A constructor **doesn't** have a **return** type.
  - The **name** of the constructor **must** be **same** as the name of the **class**.
  - Unlike methods, **constructors** are not considered **members of a class**.
  - A constructor is called **automatically** when a **new instance** of an object is **created**.
- Syntax :

```
public ClassName (parameter-list)
{
   statements...
}
```

Constructor Overloading

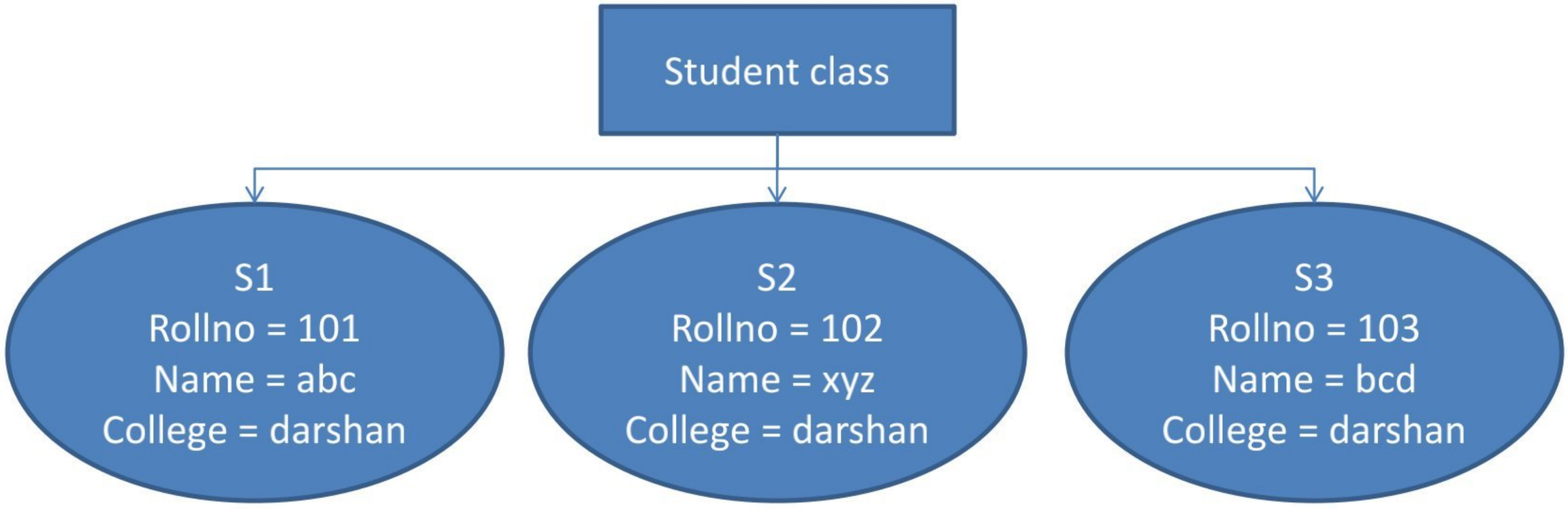
```
public class Human {

    double noOfBreath;
    double luck;
    String name;

    public Human()
    {
        noOfBreath = 1000;
        luck = 0.5;
    }
    public Human(String name)
    {
        this();
        this.name = name;
    }
}
```

static keyword

- The **static** keyword is mainly used for memory management.
- The static keyword **belongs** to the **class** rather than **instance** of the class.
- The static can be:
  - **variable** (also known as class variable)
  - **method** (also known as class method)
  - **block**
  - **nested class**
- The static variable can be used to refer the **common property** of all objects (that is not unique for each object).
- The static variable gets **memory** only once in class area at the time of **class loading**.
- **Advantage** : It makes your program memory efficient (i.e it saves memory).

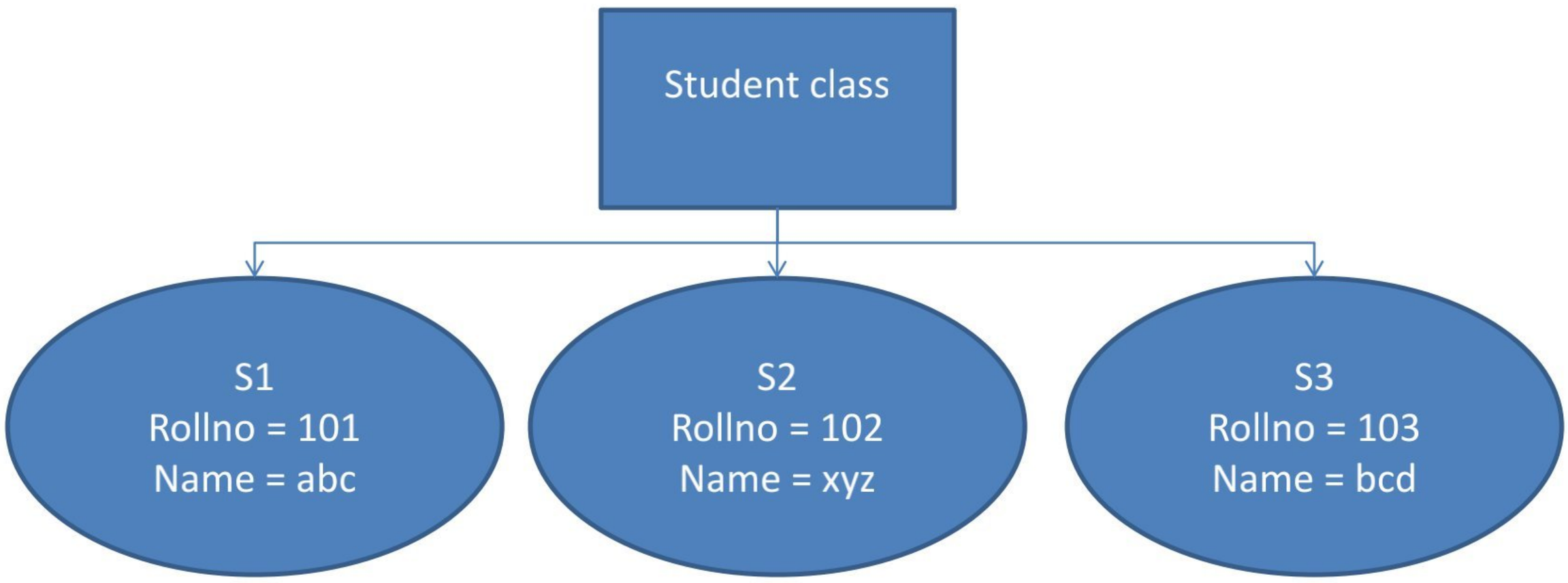
Example without static variable

```
class Student{
        int rollNo;
        String name;
        String college="darshan";
}
```



Example with static variable

```
class Student{
        int rollNo;
        String name;
        static String college="darshan";
}
```



static methods

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

static method (Example)

```java
public class StaticMethod {
    static String college = "abc";
    int rollNo ;
    public void displayCollege(){
        System.out.println("College Name = "+this.college);
    }
    public static void setCollege(String col){
         StaticMethod.college = col;
         rollNo = 10;
    }
    public static void main(String[] ar){
        StaticMethod s1 = new StaticMethod();
        StaticMethod s2 = new StaticMethod();
        s1.displayCollege();
        StaticMethod.setCollege("Darshan");
        s2.displayCollege();
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo>javac StaticMethod.java

D:\DegreeDemo>java StaticMethod
College Name = abc
College Name = Darshan
```

static block

- Java supports a special block, called **static block** (also called static clause) which can be used for **static initializations** of a class.
- This code inside static block is **executed only once** when the first time you make an object of that class **or,** the first time you access a static member of that class

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

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo>javac MyStaticBlock.java

D:\DegreeDemo>java MyStaticBlock
static block is invoked
Hello main
```

finalize() method

- The **finalize()** is belong to **java.lang.Object** class.
- finalize() method is called by the **garbage collector** on an object when garbage collector determines that there are no more references to the object.
- finalize() might be used to make sure that some **system resource** not managed by the JRE is **properly released**.
- A subclass **overrides** the finalize method to **dispose** of **system resources** or to perform other **cleanup**.

```
protected void finalize() throws Exception
{
        // code to dispose resources here
}
```

Example – finalize() method

- We can also manually call finalize method by activating garbage collector using **System.gc()** method.

```
public class DemoFinalize
{
    public static void main(String[] args)
    {
        DemoFinalize obj = new DemoFinalize();
        obj=null; // to remove the reference of the object
        System.gc();
    }
    public void finalize()
    {
         System.out.println("Garbage Collected");
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo>javac DemoFinalize.java

D:\DegreeDemo>java DemoFinalize
Garbage Collected
```

Nested Class / Inner Class

- Java inner class or nested class is a class which is **declared inside** the **class or interface**.
- Inner classes are used to **logically group** classes and interfaces in one place so that it can be more **readable** and **maintainable**.
- Additionally, it **can access** all the **members of outer class** including private data members and methods.

- Advantages of Inner Class
  - It can **access** all the **members** of Outer Class
  - It is **more readable** and **maintainable**
  - It requires **less code** (Code Optimization)

Example (Inner Class)

File Name
DemoInnerClass.java

```
public class DemoInnerClass{

    private int a = 10;
    class InnerOne
    {
        public void printMe()
        {
                System.out.println("Hello "+ a);
        }
    }

    public static void main(String[] ar)
    {
        DemoInnerClass d = new DemoInnerClass();
        InnerOne i = d.new InnerOne();
        i.printMe();
    }
}
```

Abstraction

- As per dictionary, **abstraction** is the quality of dealing with **ideas** rather than **events**.
- For example,
  - when you drive a car, you need not to worry about the complex mechanism of how car is working.
  - You just need to know how to drive, not how it works.

Abstraction in Java

- Likewise in Object-oriented programming, **abstraction** is a **process of hiding** the **implementation details** from the **user**, only the functionality will be provided to the user.
- In other words, the user will have the information on what the object does instead of how it does it.
- **Abstraction** is achieved using **Abstract classes** and **interfaces**.

Abstract Class

- A class which contains the abstract keyword in its declaration is known as abstract class.
  - Abstract classes **may or may not** contain **abstract methods**, i.e., methods without body (public void get();)
  - But, if a class has **at least one** abstract method, then the class must be declared **abstract**.
  - If a class is declared abstract, it **cannot** be instantiated.
  - To use an abstract class, you have to inherit it to another class and provide **implementations** of the abstract methods in it.

Abstract Class (Example)

```
abstract class Car {
        public abstract double getAverage();
}
class Swift extends Car{
        public double getAverage(){
                return 22.5;
        }
}
class Baleno extends Car{
        public double getAverage(){
                return 23.2;
        }
}

public class MyAbstractDemo{
        public static void main(String ar[]){
            Swift b1 = new Swift();
            Baleno b2 = new Baleno();
            System.out.println(b1.getAverage());
            System.out.println(b2.getAverage());
        }
}
```

File Name
MyAbstractDemo.java

```
C:\WINDOWS\system32\cmd.exe
D:\DegreeDemo\PPTDemo>javac MyAbstractDemo.java

D:\DegreeDemo\PPTDemo>java MyAbstractDemo
22.5
23.2
```