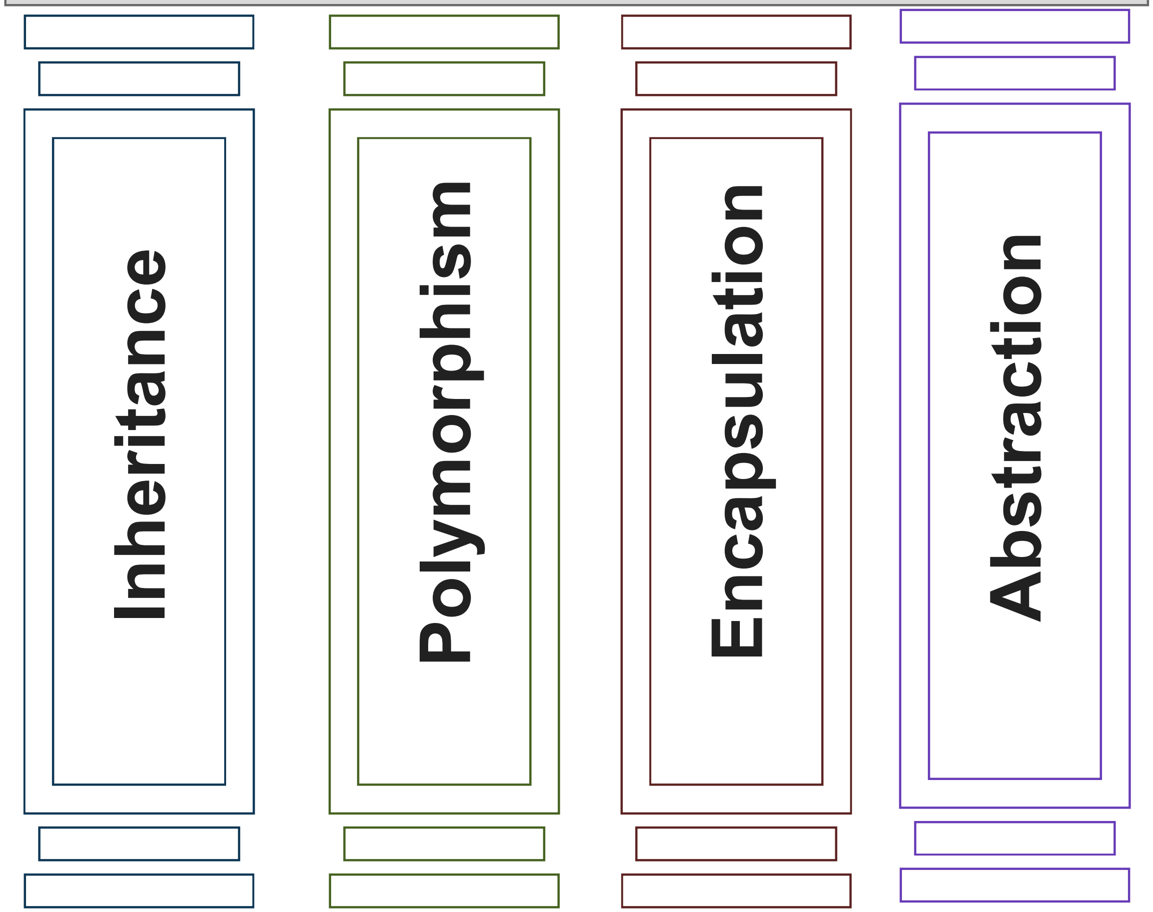
Pillars of OOP



Inheritance

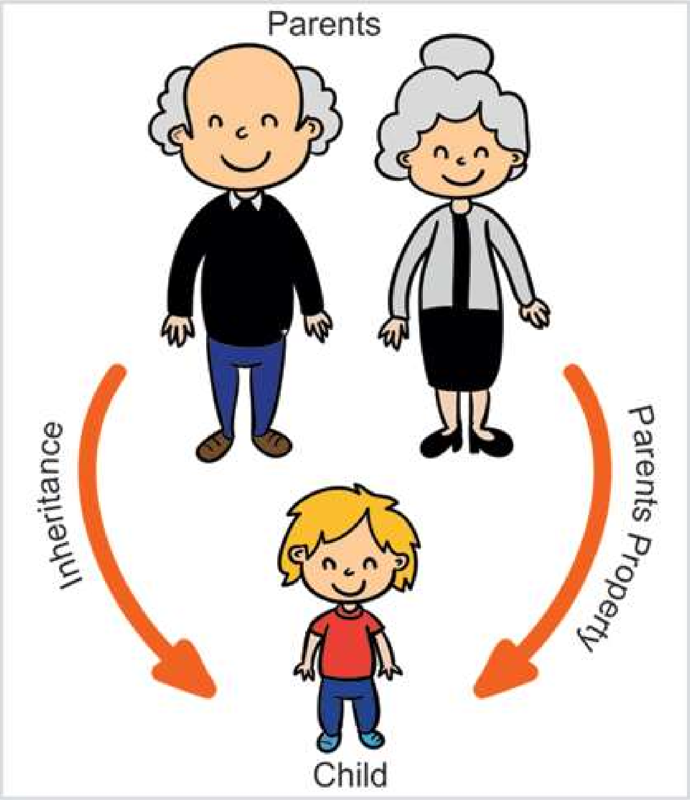
Polymorphism

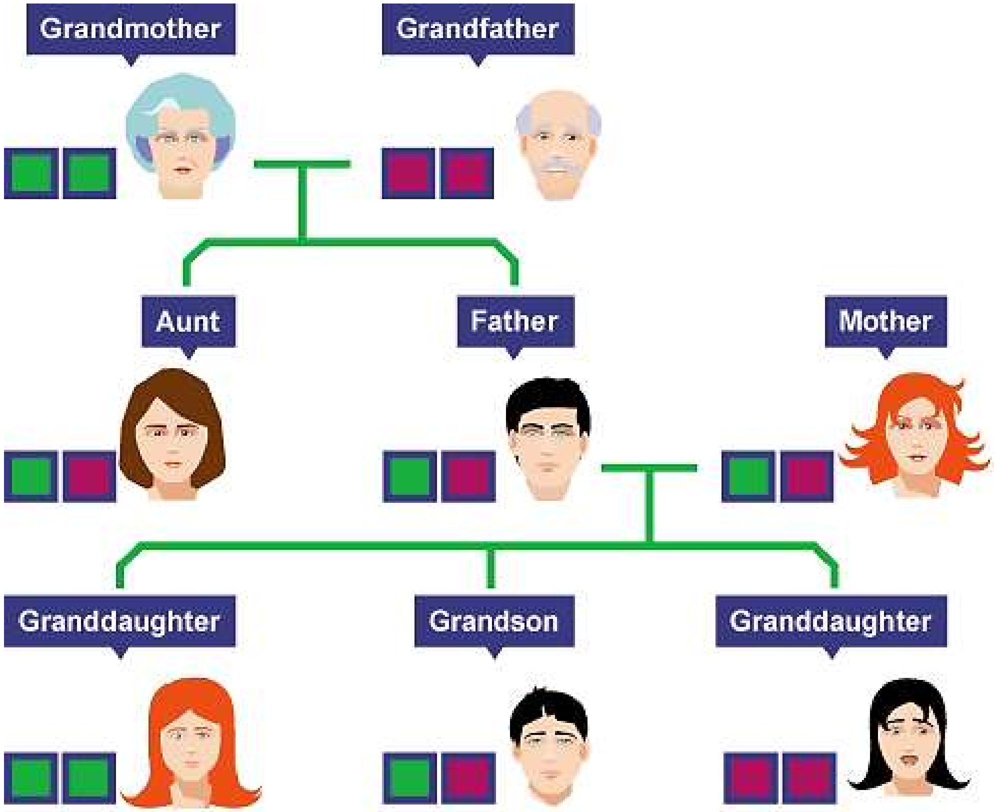
Encapsulation

Abstraction

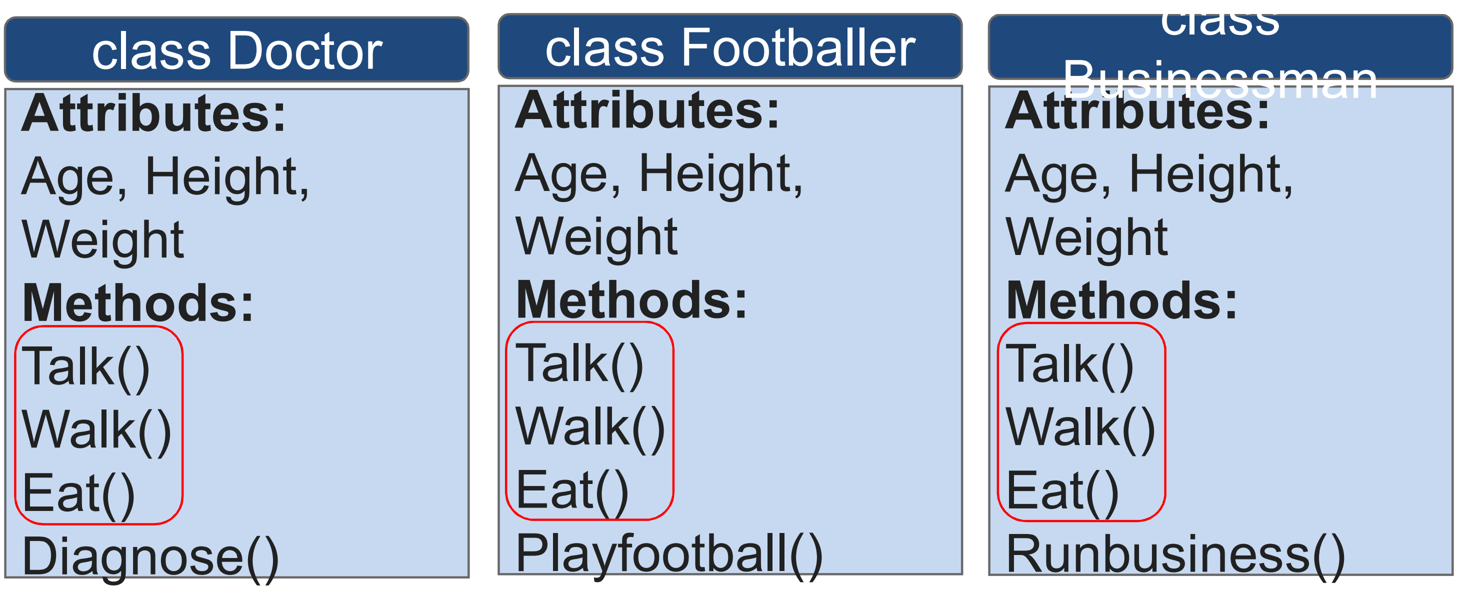
Inheritance

Inheritance





Inheritance



class Doctor

Attributes:

Age, Height, Weight

Methods:

Talk()
Walk()
Eat()
Diagnose()

class Footballer

Attributes:

Age, Height, Weight

Methods:

Talk()
Walk()
Eat()
Playfootball()

class Businessman

Attributes:

Age, Height, Weight

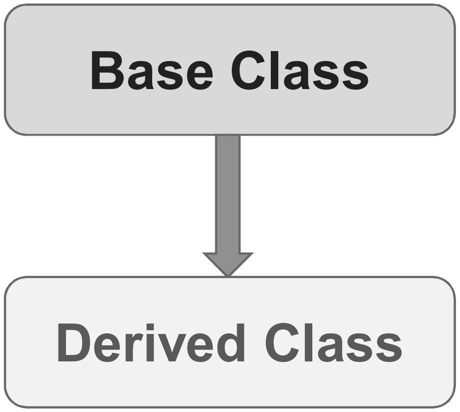
Methods:

Talk()
Walk()
Eat()
Runbusiness()

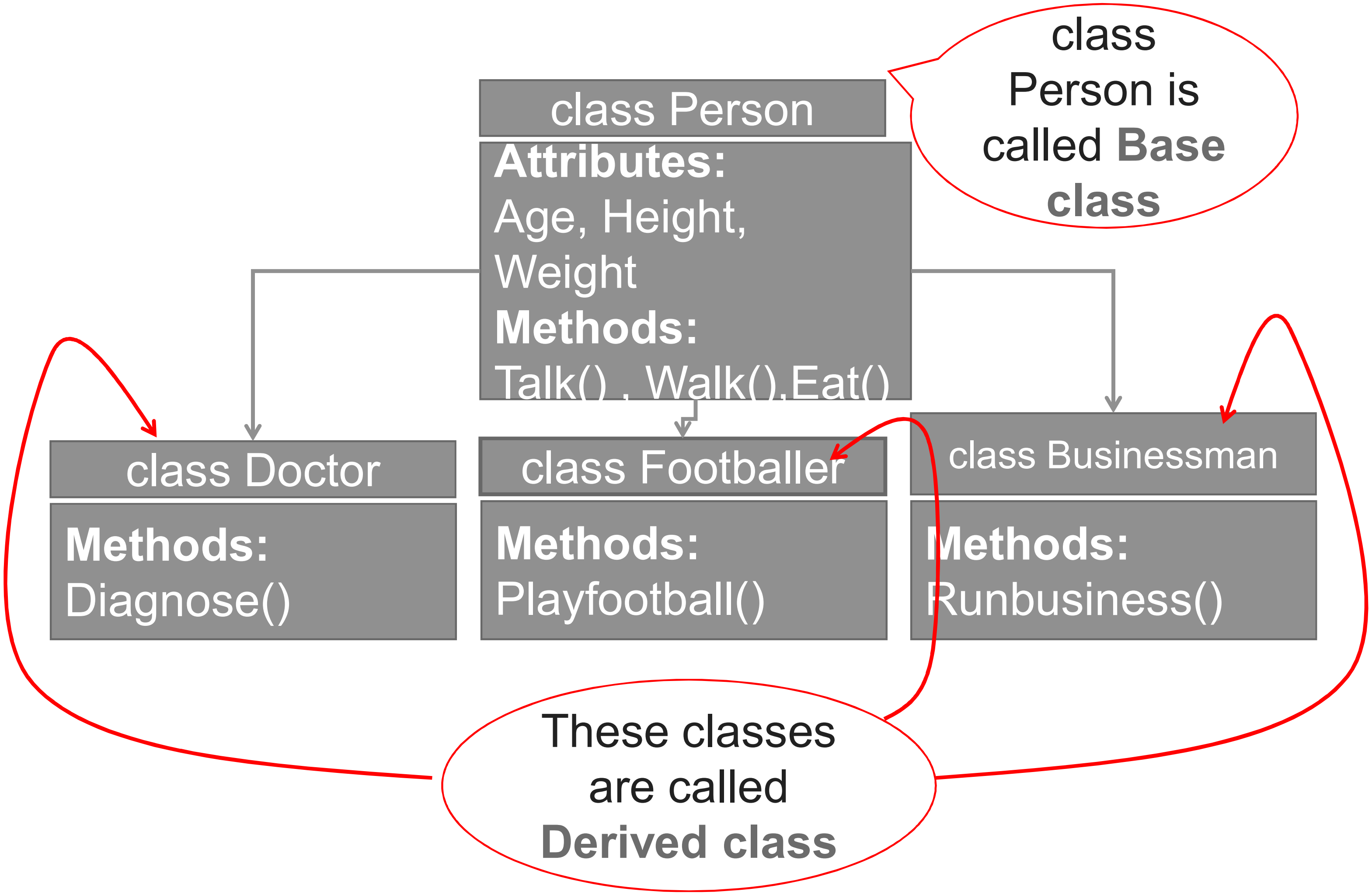
- All of the classes have common attributes (Age, Height, Weight) and methods (Walk, Talk, Eat).
- However, they have some special skills like Diagnose, Playfootball and Runbusiness.
- In each of the classes, you would be copying the same code for Walk, Talk and Eat for each character.

Inheritance

- The mechanism of a class to derive properties and characteristics from another class is called **Inheritance**.
- It is the most important feature of Object Oriented Programming.
- Inheritance is the process, by which class can acquire(reuse) the properties and methods of another class.
- **Base Class:** The class whose properties are inherited by sub class is called Base Class/Super class/Parent class.
- **Derived Class:** The class that inherits properties from another class is called Sub class/Derived Class/Child class.
- Inheritance is implemented using super class and sub class relationship in object-oriented languages.



Inheritance



class Person

Attributes:
Age, Height,
Weight
Methods:
Talk() , Walk(),Eat()

class Person is called **Base class**

class Doctor

Methods:
Diagnose()

class Footballer

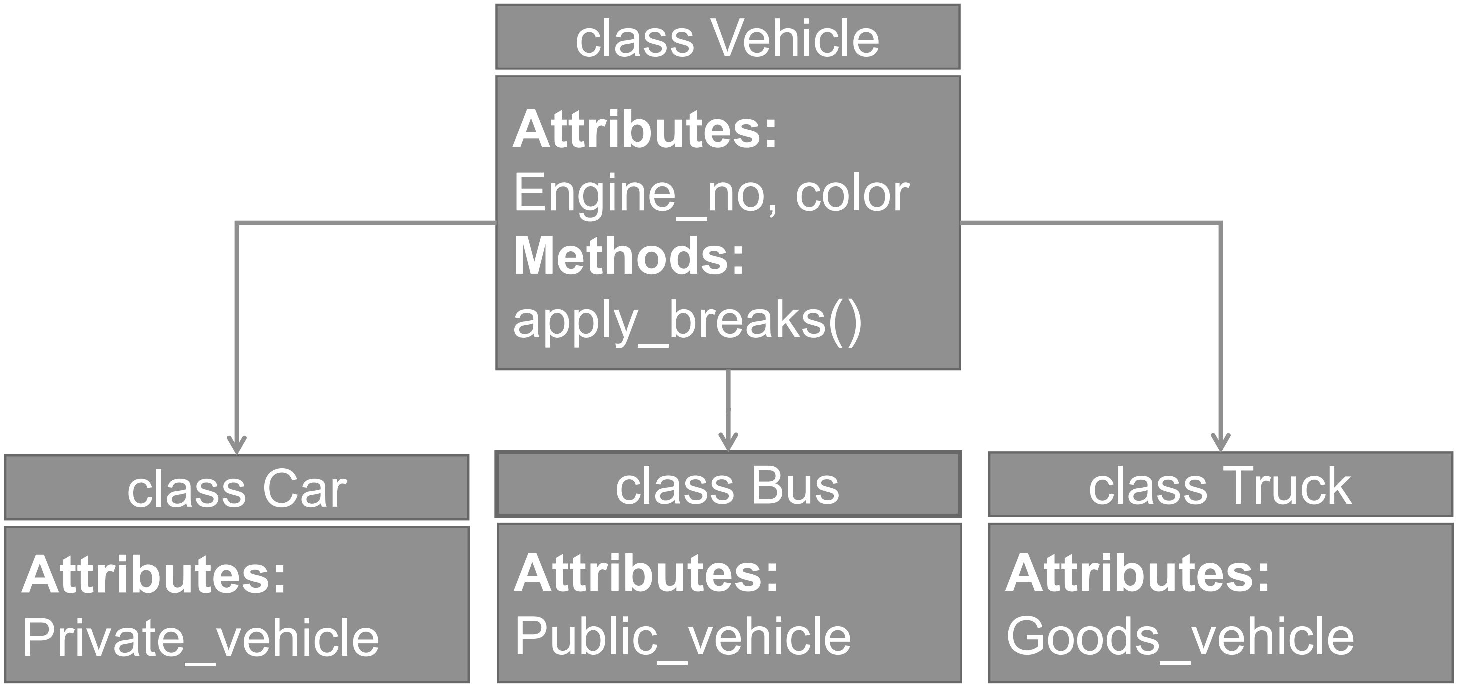
Methods:
Playfootball()

class Businessman

Methods:
Runbusiness()

These classes are called **Derived class**

Inheritance



class Vehicle

Attributes:
Engine_no, color
Methods:
apply_breaks()

class Car

Attributes:
Private_vehicle

class Bus

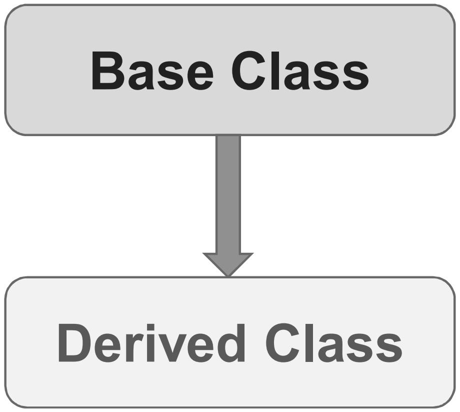
Attributes:
Public_vehicle

class Truck

Attributes:
Goods_vehicle

Inheritance: Advantages

- Promotes reusability
    - When an existing code is reused, it leads to less development and maintenance costs.
- It is used to generate more dominant objects.
- Avoids duplicity and data redundancy.
- Inheritance makes the sub classes follow a standard interface.



Base Class

Derived Class

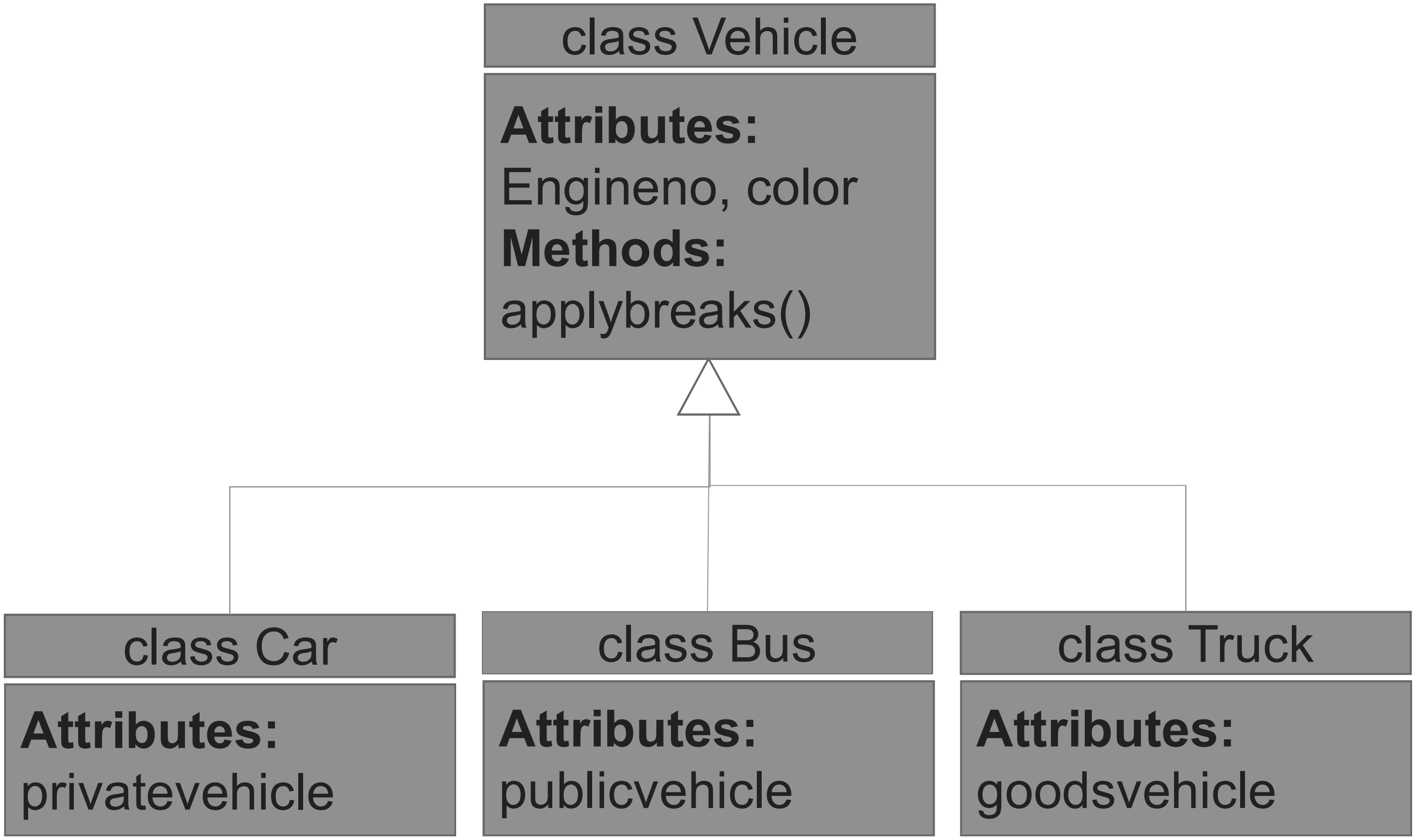
Implementing Inheritance

Subclass-Superclass Relationship

Introduction

- **Inheritance** is the process, by which a class can acquire(reuse) the properties and methods of another class.
- The mechanism of deriving a new class from an old class is called **inheritance**.
- The new class is called **derived class** and old class is called **base class**.
- The derived class may have all the features of the base class and the programmer can add new features to the derived class.
- Inheritance is also known as "IS-A relationship" between parent and child classes.
- For Example :
  - Car **IS A** Vehicle
  - Bike **IS A** Vehicle
  - EngineeringCollege **IS A** College
  - MedicalCollege **IS A** College
  - MCACollege **IS A** College

Inheritance Example



class Vehicle

Attributes:
Engineno, color
Methods:
applybreaks()

class Car

Attributes:
privatevehicle

class Bus

Attributes:
publicvehicle

class Truck

Attributes:
goodsvehicle

How to implement Inheritance in java

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

Syntax:

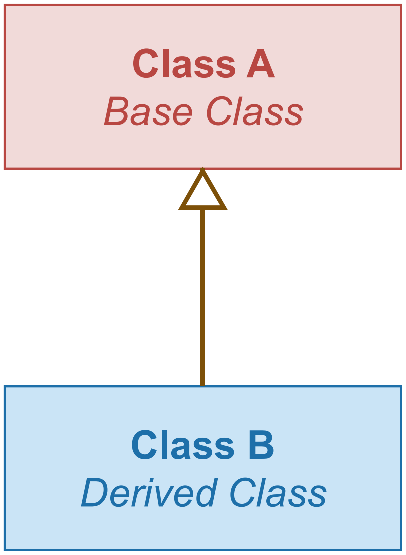
```
class subclass-name extends superclass-name
{
     // body of class…
}
```

Implementing Inheritance in java

```
class A
{
  //SuperClass or ParentClass or BaseClass
}
```

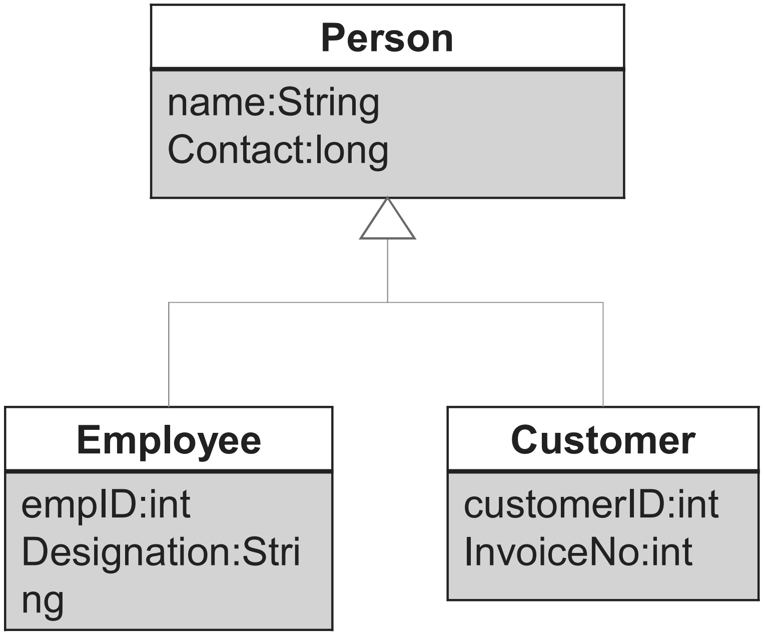
the keyword **“extends”** is used to create a subclass of **A**

```
class B extends A
{
  //SubClass or ChildClass or DerivedClass
}
```



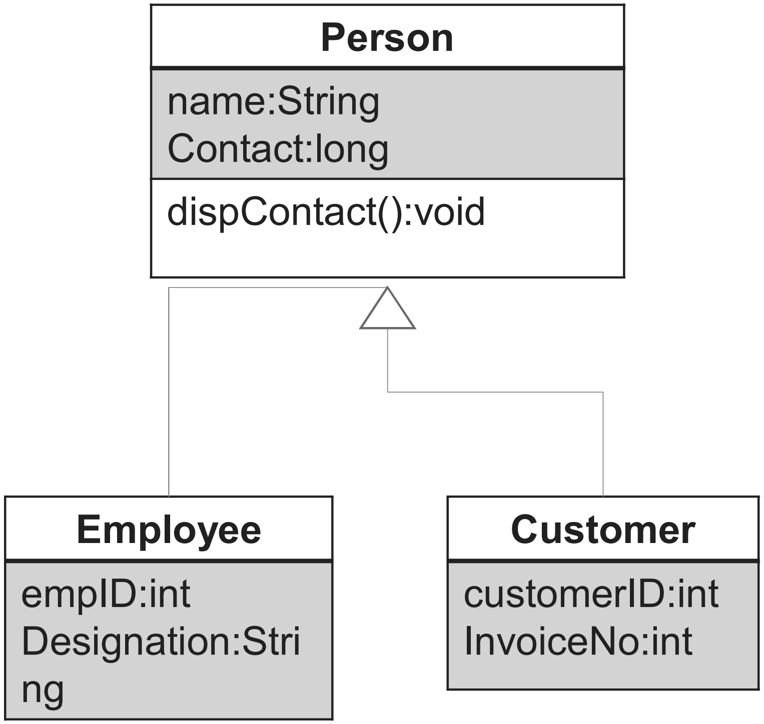
Implementing Inheritance in java

```
class Person
{
    String name;
    long contact;
}

class Employee extends Person
{
    int empID;
    String designation;
}

Class Customer extends Person
{
    int customerID;
    int invoiceNo;
}
```



Implementing Inheritance in java

```
class Person
{
    String name;
    long contact;
    public void  dispContact()
    { System.out.println(“num=”+contact);
    }
}
class Employee extends Person
{
    int empID;
    String designation;
}
Class Customer extends Person
{
    int customerID;
    int invoiceNo;
}
```



Person

name:String
Contact:long

dispContact():void

Employee

empID:int
Designation:String

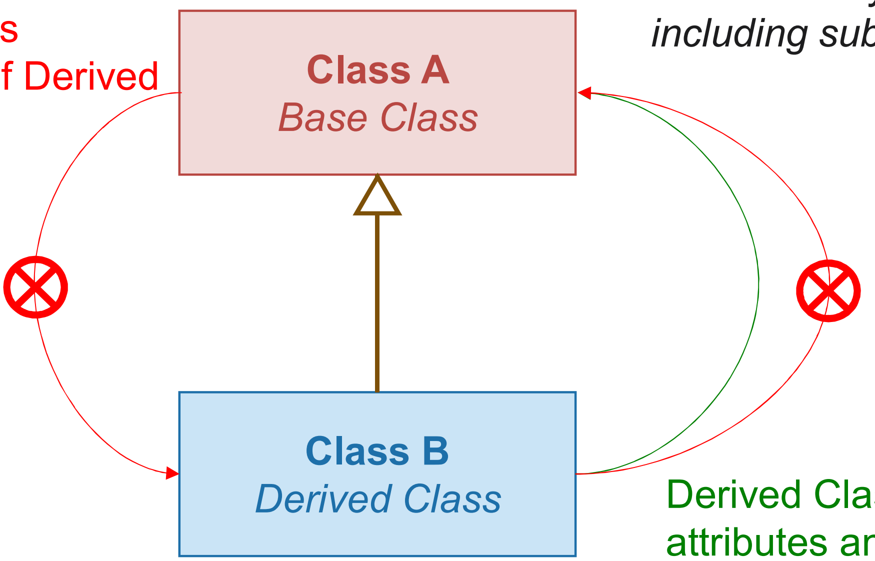
Customer

customerID:int
InvoiceNo:int

Property of Inheritance

Property of Inheritance

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.



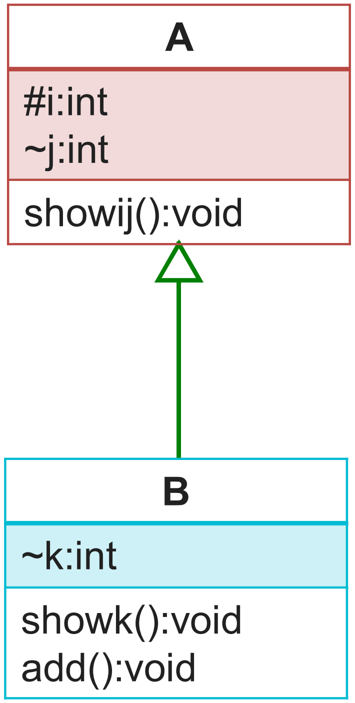
Base class A can't access attributes and methods of Derived Class B

Derived Class B can't access **private members** of Base class A

Derived Class B can access attributes and methods of Base class A

Inheritance by Example

Example1: InheritanceDemo1



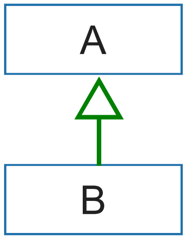
Example1: InheritanceDemo.java

```
class A{
    protected int i;
    int j;
    void showij(){
    System.out.println("i="+i+" j="+j);
    }
}

class B extends A{ //inheritance
    int k;
    void showk(){
            System.out.println("k="+k);
    }
    void add(){
    System.out.println("i+j+k="+(i+j+k));
    }
}
```

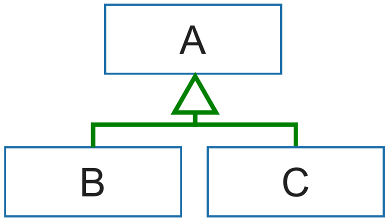
```
class InheritanceDemo{
public static void main(String[]
                                  args)
  {
            A superObjA= new A();
            superObjA.i=10;
            superObjA.j=20;

            B subObjB= new B();
            subObjB.k=30;

            superObjA.showij();
            subObjB.showk();
            subObjB.add();
   }
}
```

Types of Inheritance in Java

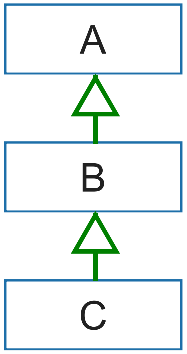
1. Single Inheritance



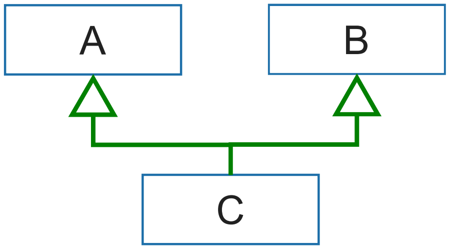
2. Hierarchical Inheritance



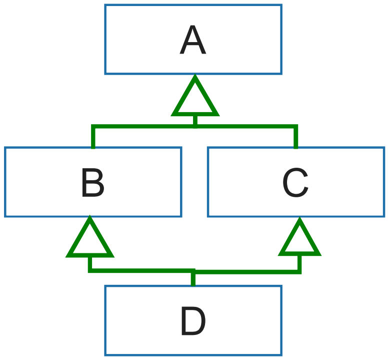
3. Multilevel Inheritance



~~4. Multiple Inheritance~~

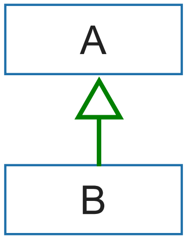


~~5. Hybrid Inheritance~~



Note: Multiple and Hybrid Inheritance is **not supported** in **Java** with the Class Inheritance, we can still use those Inheritance with Interface which we will learn in later part of the Unit

Single Inheritance

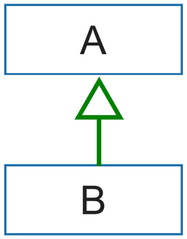


Single Inheritance: InheritanceDemo.java

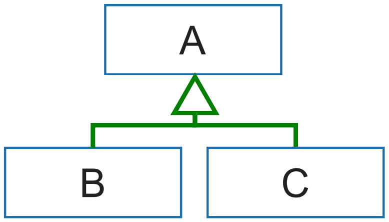
```java
class A{
    protected int i;
    int j;
    void showij(){
    System.out.println("i="+i+" j="+j);
    }
}
```

```java
class B extends A{ //inheritance
    int k;
    void showk(){
        System.out.println("k="+k);
    }
    void add(){
    System.out.println("i+j+k="+(i+j+k));
    }
}
```

```java
class InheritanceDemo{
public static void main(String[] args)
  {
         A superObjA= new A();
         superObjA.i=10;
         superObjA.j=20;

         B subObjB= new B();
         subObjB.i=100
         subObjB.j=100;
         subObjB.k=30;

         superObjA.showij();
         subObjB.showk();
         subObjB.add();
   }
}
```



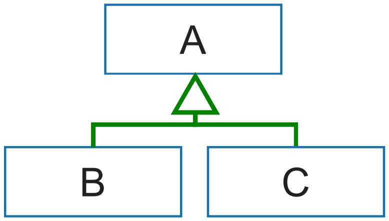
Hierarchical Inheritance



```
class A{
    protected int i;
    int j;
    void showij(){
    System.out.println("inside
                class A:i="+i+" j="+j);
    }
}
```

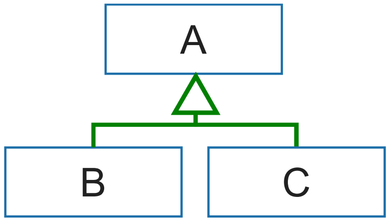
```
class B extends A{
    int k;
    void showk(){
    System.out.println("inside
                        class B:k="+k);
    }
    void add_ijk(){
    System.out.println(i+"+"+j+"+"+
                        k+"="+(i+j+k));
    }}
```

```
class C extends A{
    int m;
    void showm(){
    System.out.println("inside
              class C:k="+m);
    }
    void add_ijm(){
    System.out.println(i+"+"+j+
             "+"+m+"="+(i+j+m));
    }
}
```

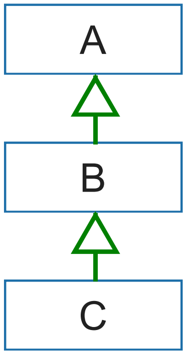


InheritanceLevel.ja

```java
class InheritanceLevel{
    public static void main(String[] args) {
        A superObjA= new A();
        superObjA.i=10;
        superObjA.j=20;
        superObjA.showij();

        B subObjB= new B();
        subObjB.i=100;
        subObjB.j=200;
        subObjB.k=300;
        subObjB.showk();
        subObjB.add_ijk();

        C subObjC= new C();
        subObjC.i=1000;
        subObjC.j=2000;;
        subObjC.m=3000;
        subObjC.showm();
        subObjC.add_ijm();
    }
}
```



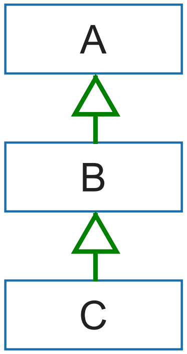
Multilevel Inheritance



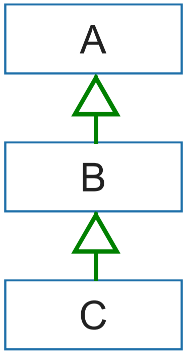
```
class A{
    protected int i;
    int j;
    void showij(){
    System.out.println("inside
                class A:i="+i+" j="+j);
    }
}
```

```
class B extends A{
    int k;
    void showk(){
    System.out.println("inside
                        class B:k="+k);
    }
    void add_ijk(){
    System.out.println(i+"+"+j+"+"+
                        k+"="+(i+j+k));
    }}
```

```
class C extends B{
    int m;
    void showm(){
    System.out.println("inside
                class C:k="+m);
    }
    void add_ijkm(){
    System.out.println(i+"+"+j+
    "+"+k+"+"+m+"="+(i+j+k+m));
    }
}
```

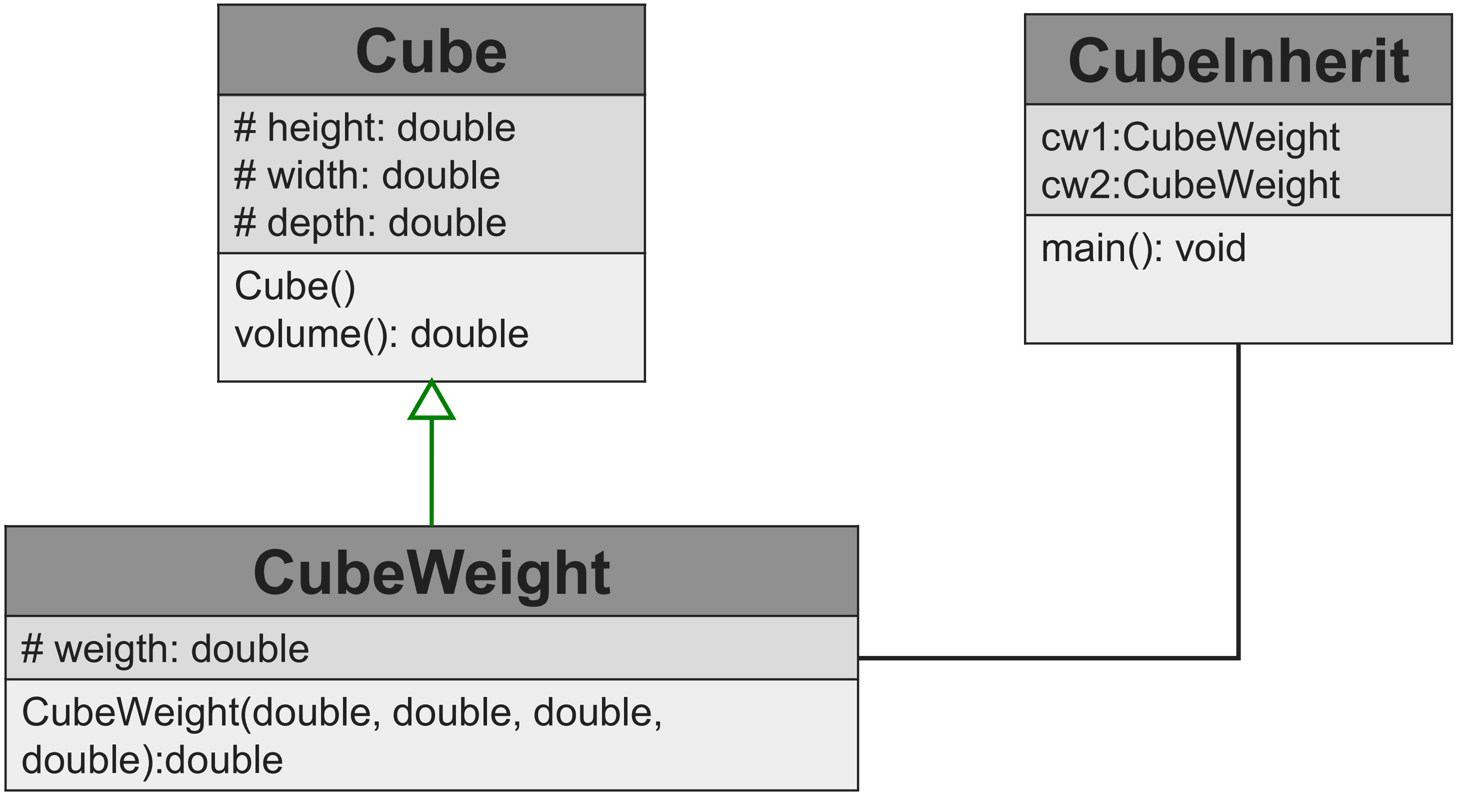


```
class InheritanceMultilevel{
    public static void main(String[] args) {
        A superObjA= new A();
        superObjA.i=10;
        superObjA.j=20;
        superObjA.showij();

        B subObjB= new B();
        subObjB.i=100;
        subObjB.j=200;
        subObjB.k=300;
        subObjB.showk();
        subObjB.add_ijk();

        C subObjC= new C();
        subObjC.i=1000;
        subObjC.j=2000;
        subObjC.k=3000;
        subObjC.m=4000;
        subObjC.showm();
        subObjC.add_ijkm();
    }
}
```



Derived Class with Constructor

Derived Class with Constructor



Cube

# height: double
# width: double
# depth: double

Cube()
volume(): double

CubeInherit

cw1:CubeWeight
cw2:CubeWeight

main(): void

CubeWeight

# weigth: double

CubeWeight(double, double, double, double):double

```
class Cube{
    protected double
                 height,width,depth;
    Cube(){
       System.out.println("inside
       default Constructor:CUBE");
    }
    double volume(){
      return height*width*depth;
    }
}
```

```
class CubeWeight extends Cube{
double weigth;
CubeWeight(double h,double w,double d, double m)
{
  System.out.println("inside Constructor:
                        CUBEWEIGTH");
          height=h;
          width=w;
          depth=d;
          weigth=m;
   }}
```

```
class CubeInherit{
   public static void main(String[] args) {
CubeWeight cw1= new
             CubeWeight(10,10,10,20.5);
CubeWeight cw2= new
             CubeWeight(100,100,100,200.5);
System.out.println("cw1.volume()="
                                +cw1.volume());
System.out.println("cw2.volume()="
                                +cw2.volume());
  }}
```

CubeInherit.jav

Outpu
t

```
inside default Constructor:CUBE
inside Constructor:CUBEWEIGTH
inside default Constructor:CUBE
inside Constructor:CUBEWEIGTH
cw1.volume()=1000.0
cw2.volume()=1000000.0
```

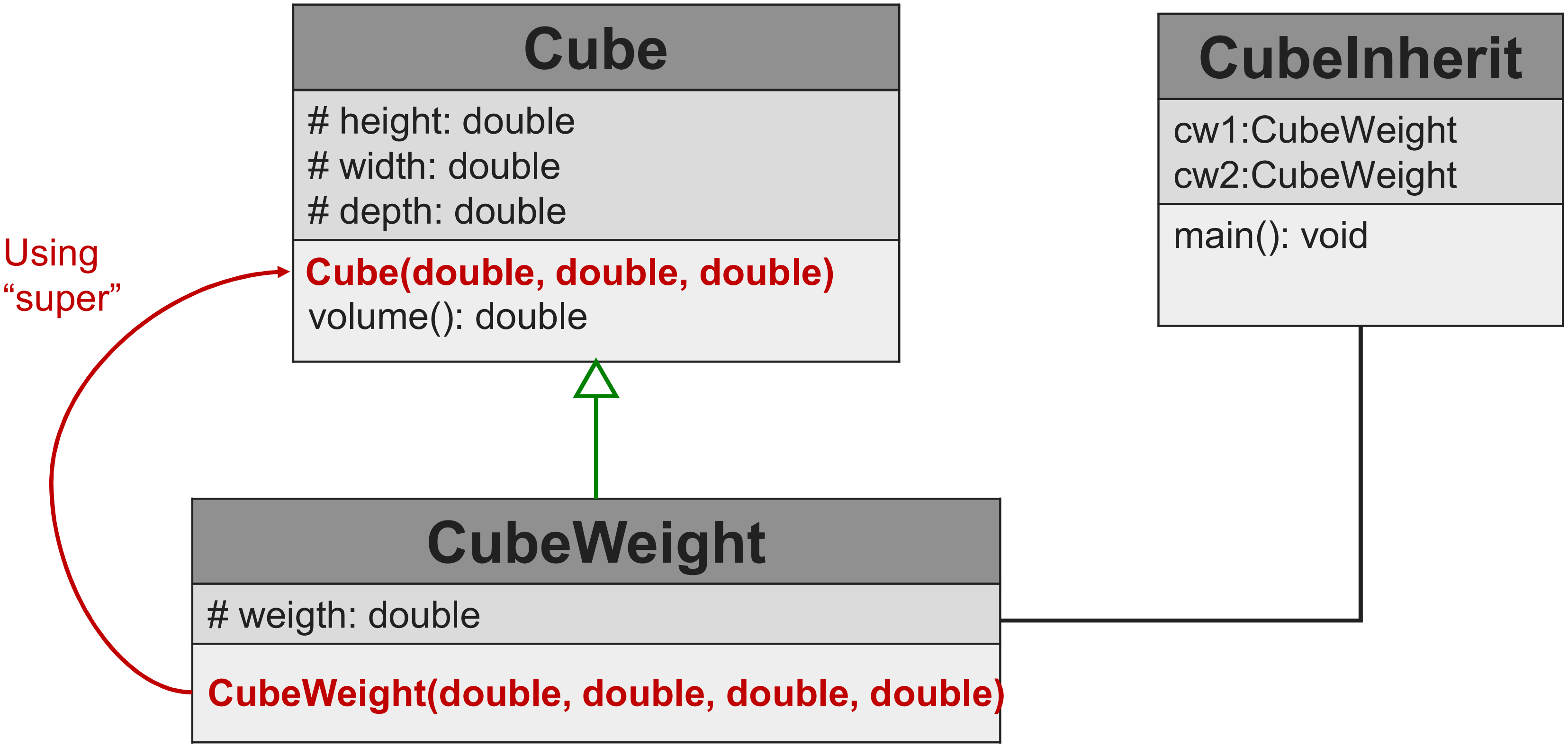
Super Keyword

Super Keyword

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. Super has two general forms:
  1. Calls the superclass constructor.
  2. Used to access a members(i.e. *instance variable or method*) of the superclass.

Using super to Call Superclass Constructors

- Call to super must be first statement in constructor



```
class Cube{
    protected double
                  height,width,depth;
Cube(double h,double w,double d){
System.out.println("Constructor:
                                  CUBE");
            height=h;
            width=w;
            depth=d;
    }
    double volume(){
    return height*width*depth;
    }
}
class CubeWeight extends Cube{
    double weigth;
    CubeWeight(double h,double w,double d, double m){
      super(h,w,d); //call superclassConstructor
      System.out.println("Constructor:CUBEWEIGTH");
      weigth=m;
    }
}
```

```
class CubeInheritSuper{
    public static void main(String[] args) {
            CubeWeight cw1= new
                    CubeWeight(10,10,10,20.5);
            CubeWeight cw2= new
                    CubeWeight(100,100,100,200.5);
System.out.println("cw1.volume()="+cw1.volume());
System.out.println("cw1.weigth="+cw1.weigth);
System.out.println("cw2.volume()="+cw2.volume());
System.out.println("cw2.weigth="+cw2.weigth);
  }
}
```

CubeInheritSuper.ja

Outpu
t

```
Constructor:CUBE
Constructor:CUBEWEIGTH
Constructor:CUBE
Constructor:CUBEWEIGTH
cw1.volume()=1000.0
cw1.weigth=20.5
cw2.volume()=1000000.0
cw2.weigth=200.5
```

Using super to Call Superclass Constructors

Using super to access members

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.
- Syntax:

```
super.member
```

member can be either a **method** or an **instance variable**.

- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Using super to access members: SuperMemberDemo.java

```
class A{
     int i;
}

class B extends A{
    int i,k;
    B(int a,int b){
        super.i=a;
        this.i=b;
    }
    void show(){
    System.out.println("super.i="+super.i);
    System.out.println("this.i="+this.i);
    }
}
class SuperMemberDemo{
   public static void main(String[] args)
  {
        B b= new B(12,56);
        b.show();
   }
}
```

Output

```
super.i=12
this.i=56
```

Using super to access members:

```java
class A{
    int i=33;
    void show(){
    System.out.println("inside A:i="+i);
    }
}

class B extends A{
    int i,k;
    B(int a,int b){
            super.show();
            super.i=a;
            this.i=b;
    }
    void show(){
    System.out.println("super.i="+super.i);
    System.out.println("this.i="+this.i);
    }
}
```

```java
class SuperMemberDemo{
    public static void main(String[] args)
  {
            B b= new B(12,56);
            b.show();
    }
}
```

Output

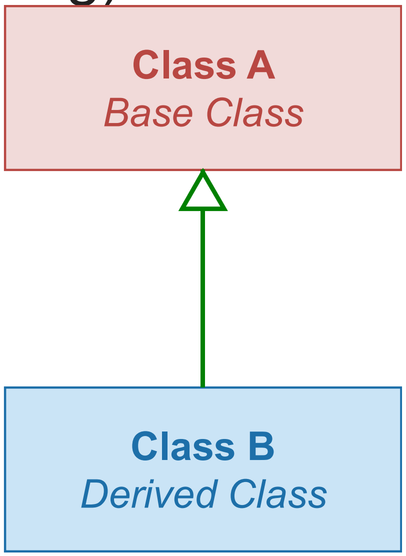
```
inside A:i=33
super.i=12
B.i=56
```

Points to remember for super

- When a subclass calls **super( )**, it is calling the constructor of its immediate superclass.
- This is true even in a multileveled hierarchy.
- **super( )** must always be the **first statement** executed inside a subclass constructor.
- If a constructor does not explicitly call a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass.
- The most common application of super keyword is to eliminate the ambiguity between members of superclass and sub class.

Why Inheritance

1. Reusability of code
2. To implement polymorphism at run time (method overriding).



Assignment

- Emp(id, name, salary)private
- Use parameterized constructor (3 Arg)
- Display emp
- Manager inherited from emp(bonus)
- Use parameterized constructor for manager(4 Arg)
- Display (call base class display)
- Write main method

Access Control

Access Control

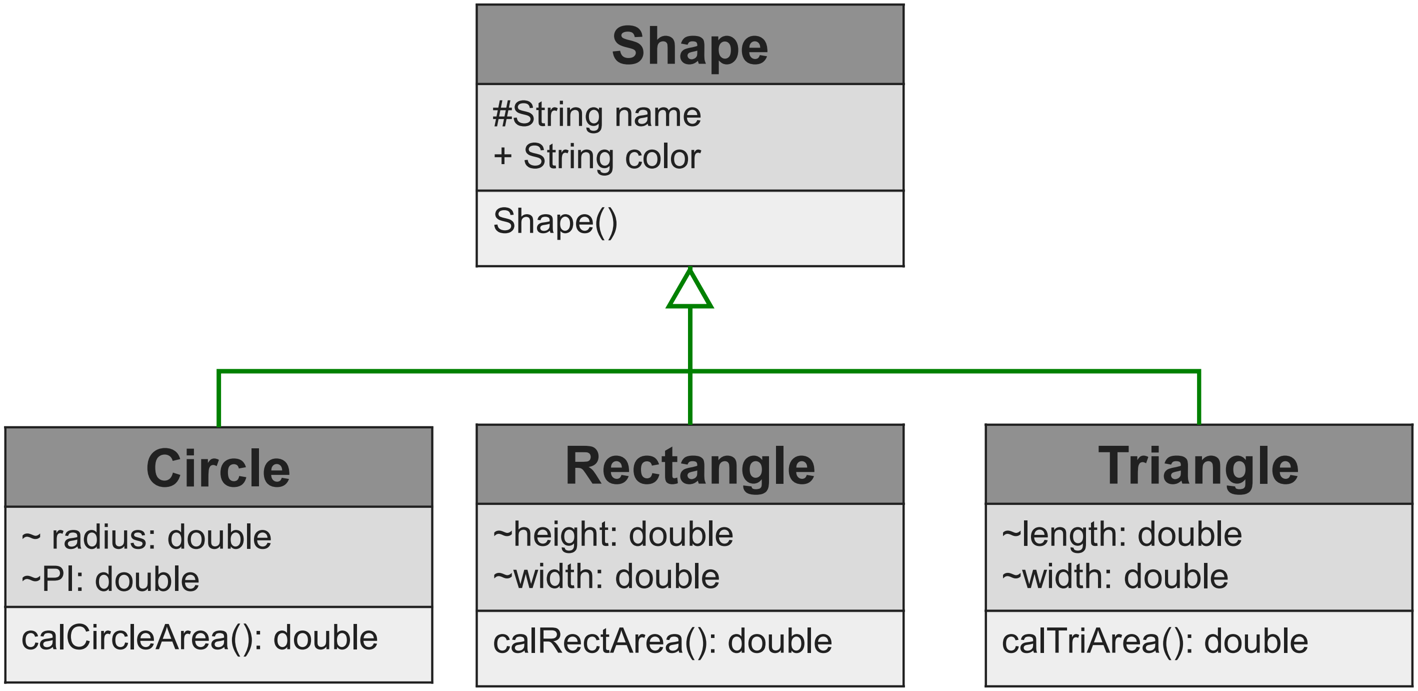
Access Modifier	Description
Private(-)	The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
Default(~)	The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
Protected(#)	The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
Public(+)	The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Control

Access Modifier	Same Class	Same Package	Sub Class	Universal
Private	✓			
Default	✓	✓		
Protected	✓	✓	✓	
Public	✓	✓	✓	✓

Exercise

1. Why multiple and Hybrid inheritance is not supported in java.
2. Implement inheritance in java using following diagram.



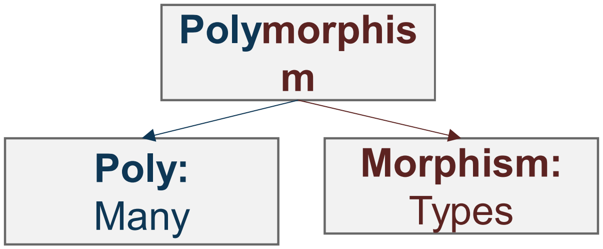
Interview Questions

1. Which class in Java is superclass of every other class?
2. Can a class extend itself?
3. Can we assign superclass to subclass?
4. Can a class extend more than one class?

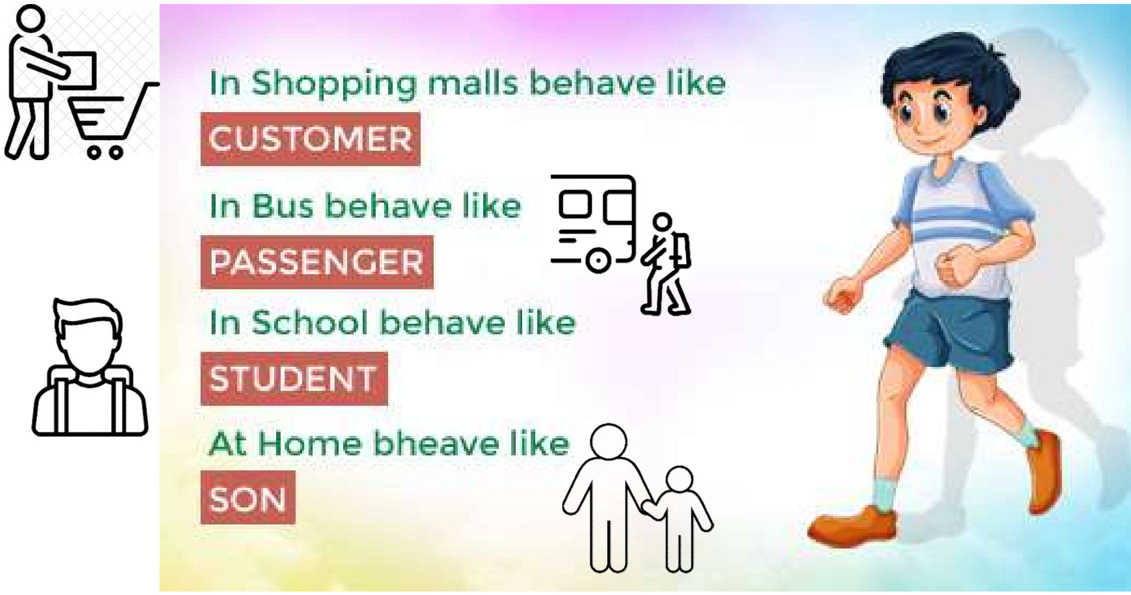
Polymorphism

Polymorphism

Polymorphism: It is a Greek term means, ***“One name many Forms”***

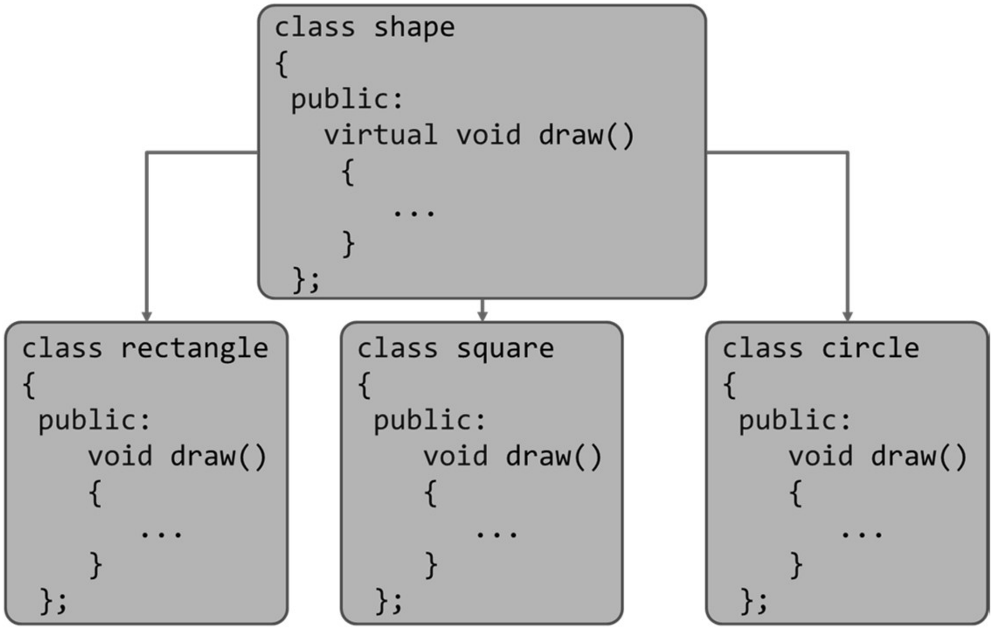


- Most important concept of object oriented programming
- In OOP, Polymorphism is the ability of an object to take on many forms.



Polymorphism

- Polymorphism is the method in an object-oriented programming language that does different things depending on the class of the object which calls it.
- Polymorphism can be implemented using the concept of overloading and overriding.



```
class shape
{
 public:
   virtual void draw()
    {
        ...
    }
 };
```

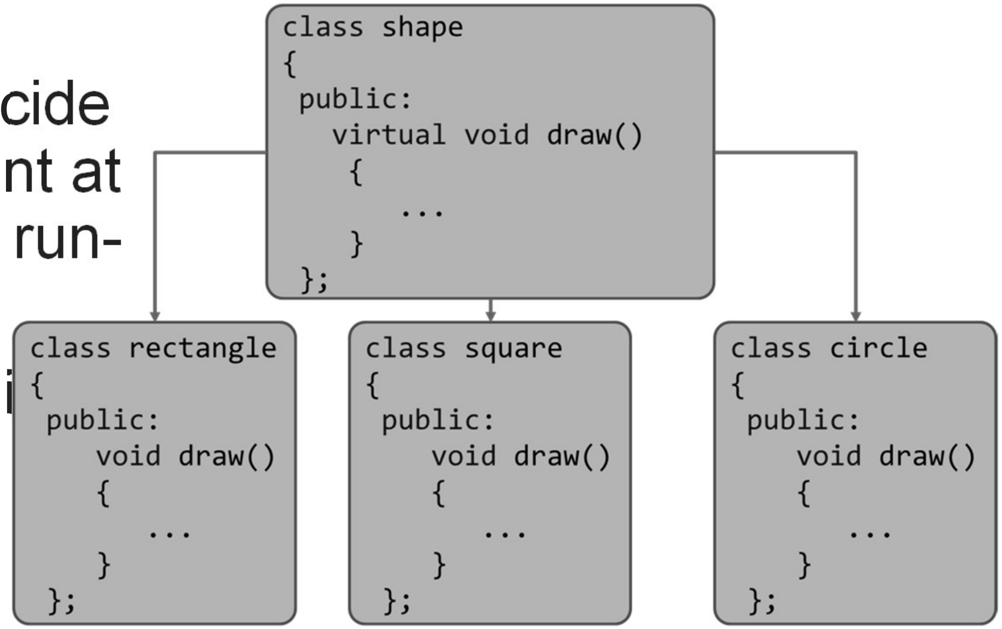
```
class rectangle
{
 public:
    void draw()
    {
        ...
    }
 };
```

```
class square
{
 public:
    void draw()
    {
        ...
    }
 };
```

```
class circle
{
 public:
    void draw()
    {
        ...
    }
 };
```

Polymorphism: Advantages

- Single variable can be used to store multiple data types.
- Easy to debug the codes.
- It allows to perform a single act in different ways.
- Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
- Reduces coupling, increases reusabili and makes code easier to read.



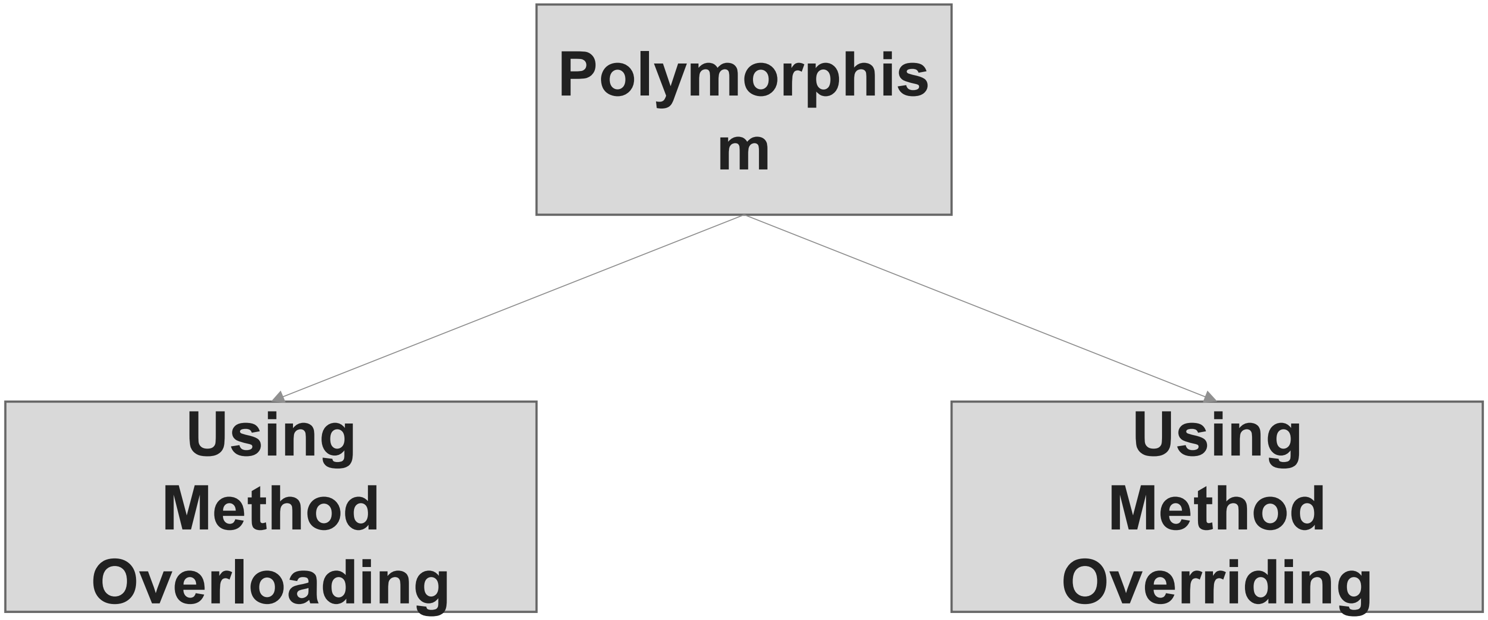
```
class shape
{
 public:
   virtual void draw()
    {
        ...
    }
 };
```

```
class rectangle
{
 public:
    void draw()
    {
        ...
    }
 };
```

```
class square
{
 public:
    void draw()
    {
        ...
    }
 };
```

```
class circle
{
 public:
    void draw()
    {
        ...
    }
 };
```

Implementing Polymorphism



Polymorphism

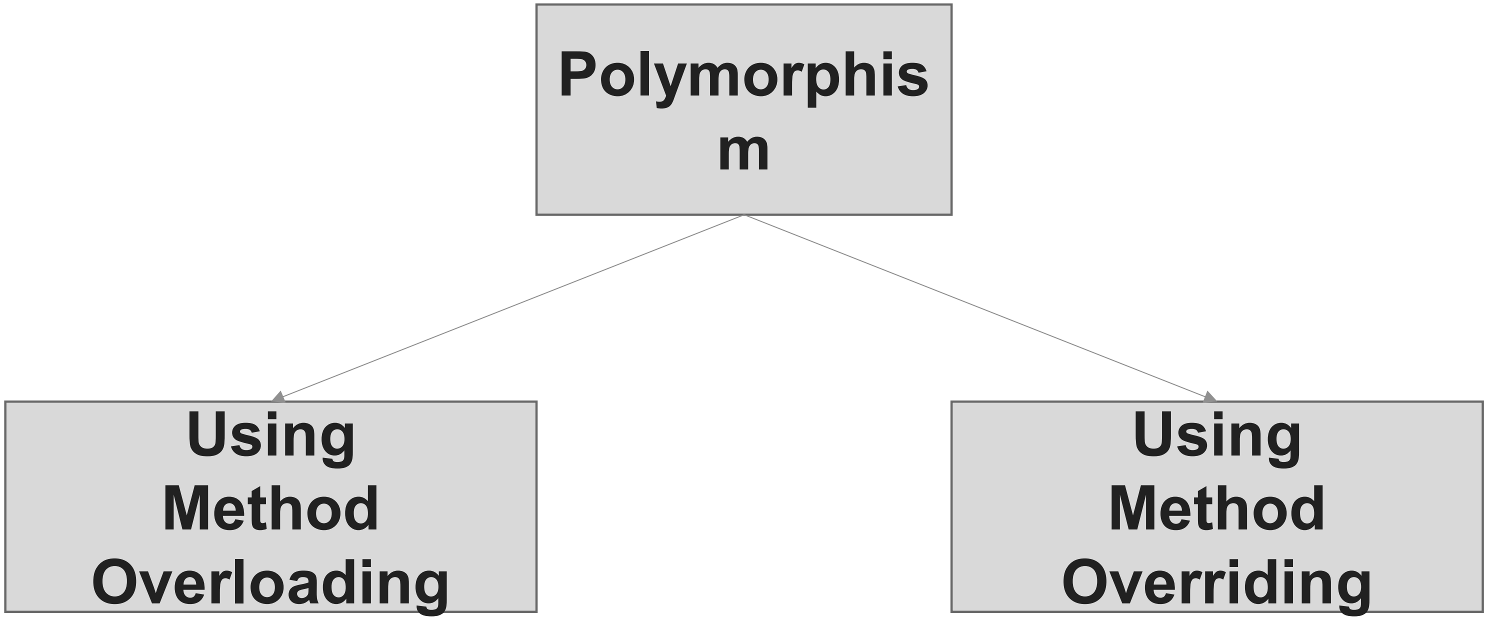
Using Method Overloading

Using Method Overriding

Implementing Polymorphism

Overloading & Overrding

Implementing Polymorphism



Polymorphism

Using Method Overloading

Using Method Overriding

Method Overloading

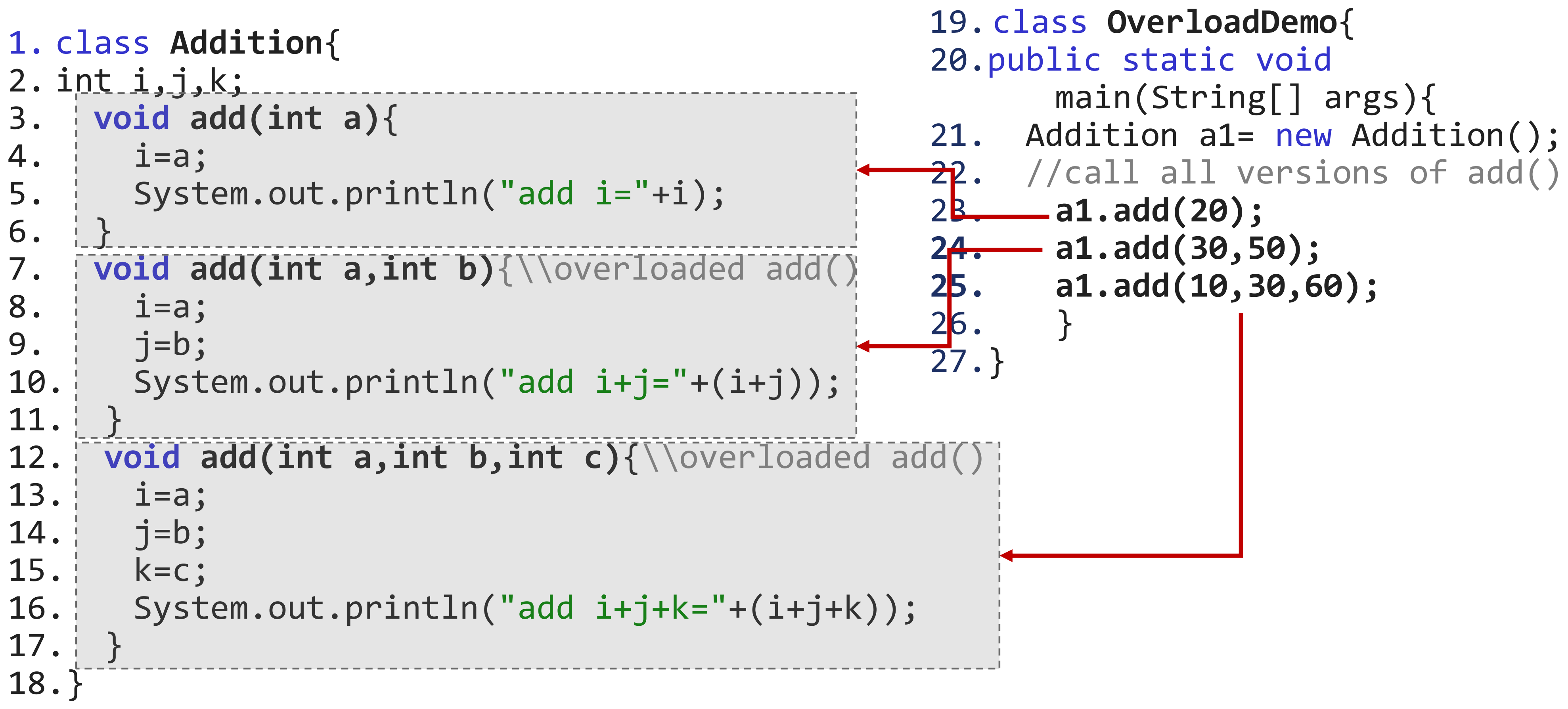
Method Overloading: Compile-time Polymorphism

- **Definition**: When two or more methods are implemented that share same name but different parameter(s), the methods are said to be ***overloaded***, and the process is referred to as ***method overloading***
- Method overloading is one of the ways that Java implements polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
  - E.g.

```
public void draw()
public void draw(int height, int width)
public void draw(int radius)
```

- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While in overloaded methods with different return types and same name & parameter are not allowed ,as the return type alone is insufficient for the compiler to distinguish two versions of a method.

Method Overloading: Compile-time Polymorphism



```java
class Addition{
int i,j,k;
  void add(int a){
    i=a;
    System.out.println("add i="+i);
  }
  void add(int a,int b){\\overloaded add()
    i=a;
    j=b;
    System.out.println("add i+j="+(i+j));
  }
  void add(int a,int b,int c){\\overloaded add()
    i=a;
    j=b;
    k=c;
    System.out.println("add i+j+k="+(i+j+k));
  }
}
```

```java
class OverloadDemo{
public static void
      main(String[] args){
  Addition a1= new Addition();
  //call all versions of add()
    a1.add(20);
    a1.add(30,50);
    a1.add(10,30,60);
    }
}
```

Method Overriding

Method Overriding: Run-time Polymorphism

- In a class hierarchy, when a method in a **subclass** has the same name and type signature as a method in its **superclass**, then the method in the subclass is said to *override* the method in the superclass.
- **Definition**: If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

Method Overriding: OverrideDemo.java

```java
class Shape{
   void draw(){
    System.out.println("Draw Shape");
   }
}
class Circle extends Shape{
   void draw(){
    System.out.println("Draw Circle");
   }
}
class Square extends Shape{
   void draw(){
    System.out.println("Draw Square");
   }
}
```

Output

```
Draw Circle
Draw Square
Draw Shape
```

```java
class OverrideDemo{
    public static void
          main(String[] args) {
    Circle c= new Circle();
    c.draw(); //child class meth()
    Square sq= new Square();
    sq.draw();//child class meth()
    Shape sh= new Shape();
    sh.draw();//parentClass meth()
    }
}
```

*When an overridden method is called from within a **subclass**, it will always refer to the version of that method defined by the **subclass**. The version of the method defined by the **superclass** will be **hidden**.*

Method Overriding: OverrideDemo.java

```java
class Shape{
   void draw(){
    System.out.println("Draw Shape");
   }
}
class Circle extends Shape{
   void draw(){
   super.draw();
    System.out.println("Draw Circle");
   }
}
class Square extends Shape{
   void draw(){
    System.out.println("Draw Square");
   }
}
```

```java
class OverrideDemo{
    public static void main(String[] args) {
    Circle c= new Circle();
    c.draw();
    Square sq= new Square();
    sq.draw();
    }
}
```

*Here, **super.draw( )** calls the superclass version of **draw( )**.*

Output

```
Draw Shape
Draw Circle
Draw Square
```

Overridden methods in Java are similar to virtual functions in C++ and C#.

Why Overriding?

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for **runtime polymorphism.**
- By combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.

Method Overriding: Points to remember

- Method overriding occurs *only* when the names and the type signatures of the two methods are **identical**. If they are not, then the two methods are simply overloaded.
- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an **IS-A relationship (inheritance).**

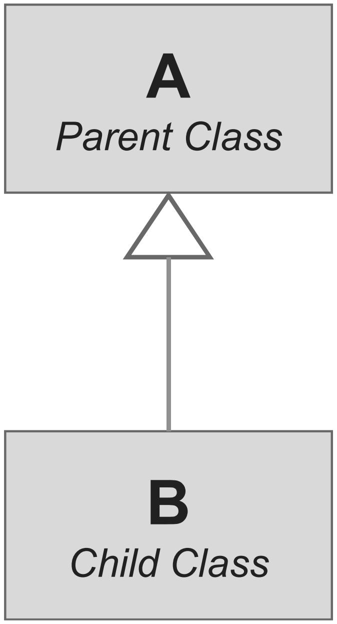
Overloading vs Overriding

Overloading vs Overriding: Java Methods

Method Overloading	Method Overriding
Overloading: Method with same name different signature	**Overriding:**Method with same name same signature
Known as Compile-time Polymorphism	Known as Run-time Polymorphism
It is performed *within class*.	It occurs *in two classes* with IS-A (inheritance) relationship.
Inheritance and method hiding is not involved here.	Here subclass method hides the super class method.

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.



A
Parent Class

B
Child Class

```
A a = new A(); //object of parent class
B b = new B(); //object of child class
```

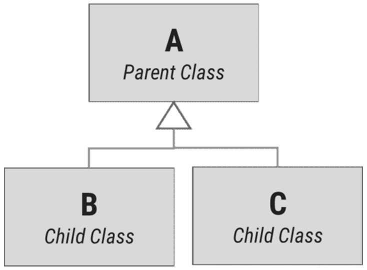
```
A a = new B();
//Up casting(Dynamic Method Dispatch)
```

```
B b= new A();
//Error! Not Allowed
```

Dynamic Method Dispatch: Example

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

class B extends A{
void display(){
    System.out.println("inside class B");
 }
}

class C extends A{
void display(){
    System.out.println("inside class C");
 }
}
```

```
class DispatchDemo{
   public static void
     main(String[] args) {
          A a = new A();
          B b = new B();
          C c = new C();
          A r; //obtain a reference
                                of type A

          r=a;
          r.display();

          r=b;
          r.display();

          r=c;
          r.display();
   }
}
```



Dynamic binding occurs during run-time known as Run-time Polymorphism.

Dynamic Method Dispatch: Example

```
class Game {
  public void type() {
    System.out.println("Indoor & outdoor");
  }
}
class Cricket extends Game {
  public void type() {
    System.out.println("outdoor game");
  }
}
class Badminton extends Game {
  public void type() {
    System.out.println("indoor game");
  }
}
class Tennis extends Game {
  public void type() {
    System.out.println("Mix game");
  }
}
```

```
public class MyProg {
  public static void main(String[] args) {
    Game g = new Game();
    Cricket c = new Cricket();
    Badminton b = new Badminton();
    Tennis t = new Tennis();
    Scanner s = new Scanner(System.in);
    System.out.print("Please Enter name of
                           the game = ");
    String op = s.nextLine();
    if (op.equals("cricket")) {
          g = c;
    } else if (op.equals("badminton")) {
          g = b;
    } else if (op.equals("tennis")) {
    g = t;
    }
    g.type();
  }
}
```

“final” keyword

- The final keyword is used for **restriction**.
- final keyword can be used in many context
- Final can be:
  1. Variable

     If you make any variable as final, you **cannot change the value** of final variable(It will be constant).
  2. Method

     If you make any method as final, you **cannot override** it.
  3. Class

     If you make any class as final, you **cannot extend** it.

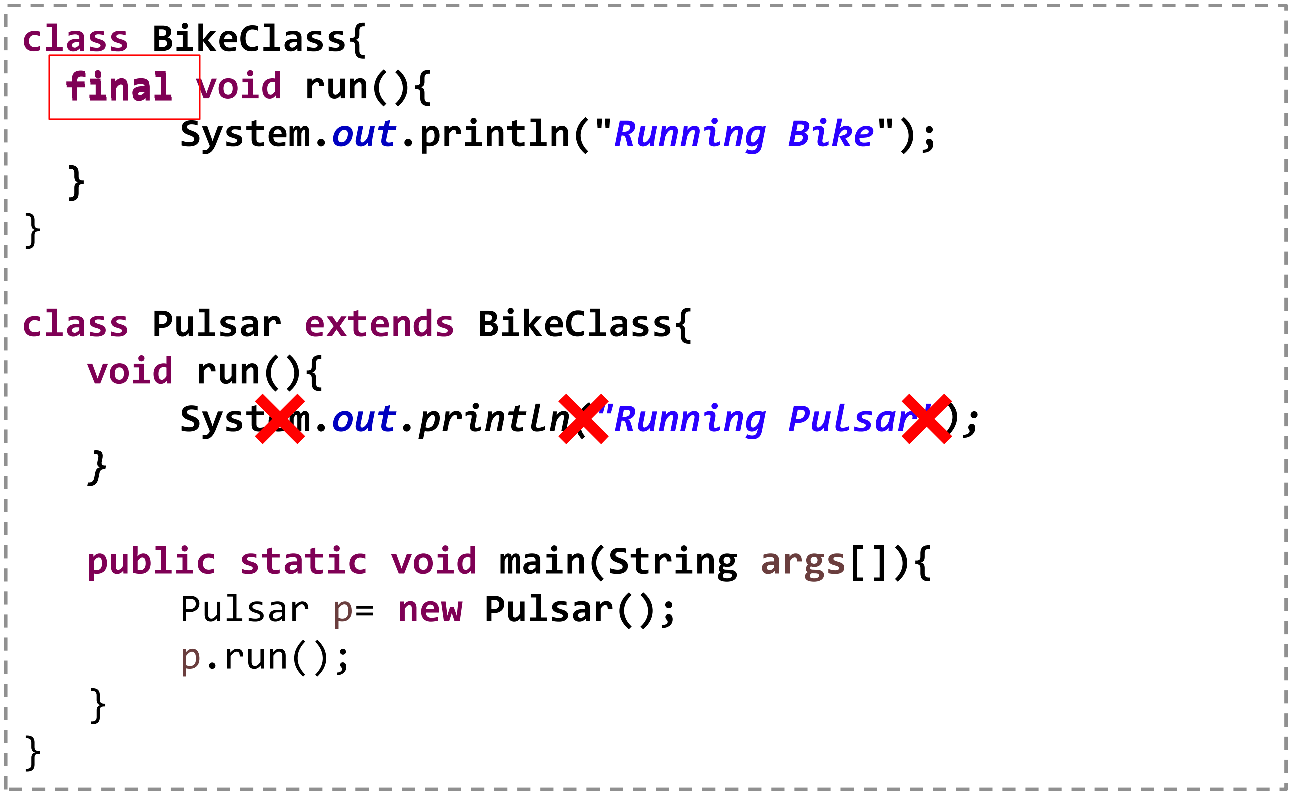
1) “final” as a variable

- Can **not change** the **value** of final **variable.**

```
public class FinalDemo {

 final int speedlimit=90;//final variable
 void run(){
       speedlimit=400;
 }
 public static void main(String args[]){
       FinalDemo obj=new  FinalDemo();
       obj.run();
 }
}
```

2) “final” as a method

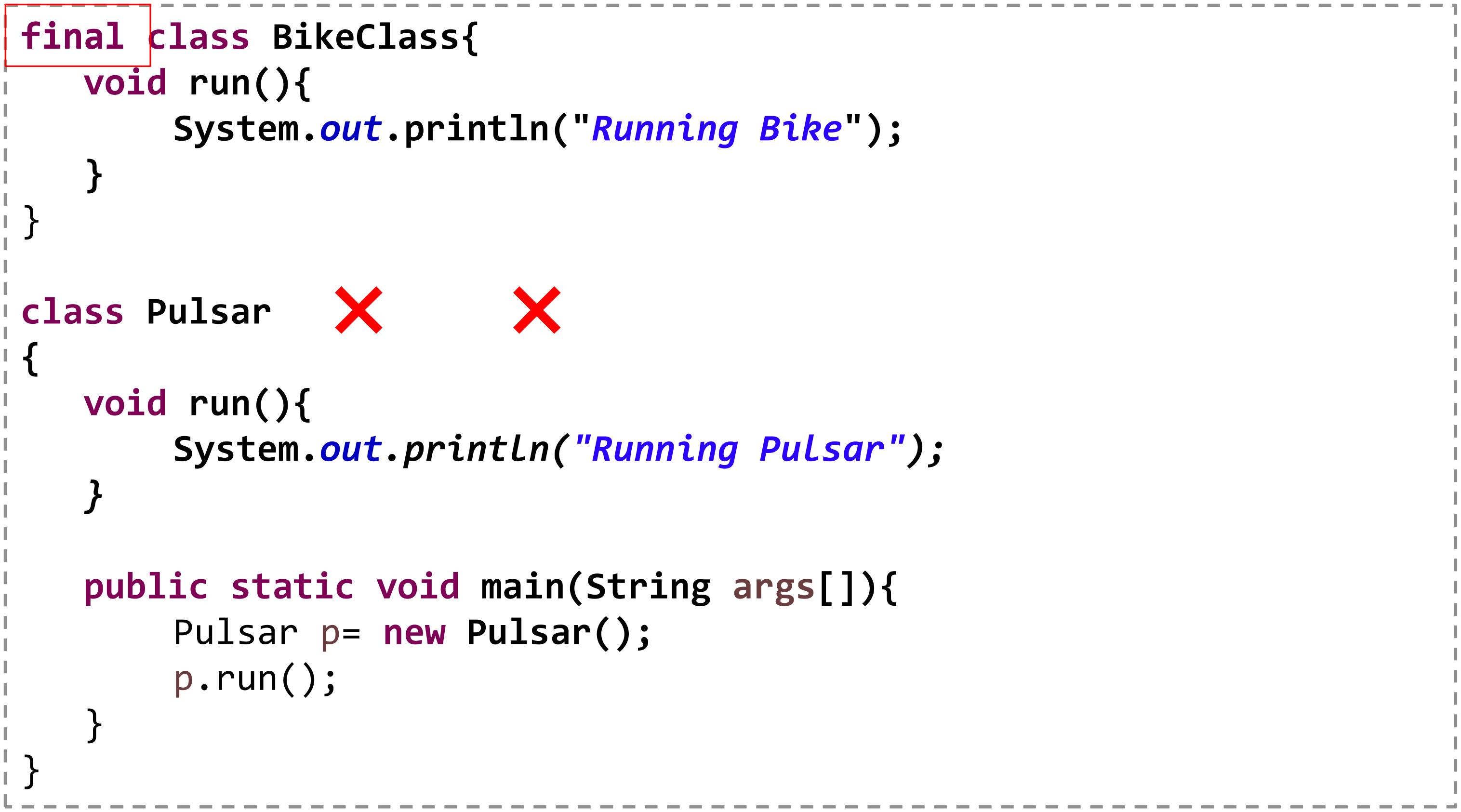
- If you make any **method** as **final**, you **cannot override** it.



```
class BikeClass{
  final void run(){
       System.out.println("Running Bike");
  }
}

class Pulsar extends BikeClass{
   void run(){
       System.out.println("Running Pulsar");
   }

   public static void main(String args[]){
       Pulsar p= new Pulsar();
       p.run();
   }
}
```

3) “final” as a Class

- If you make any **class** as **final**, you **cannot extend** it.



```
final class BikeClass{
    void run(){
          System.out.println("Running Bike");
    }
}

class Pulsar    ✗      ✗
{
    void run(){
          System.out.println("Running Pulsar");
    }

    public static void main(String args[]){
          Pulsar p= new Pulsar();
          p.run();
    }
}
```

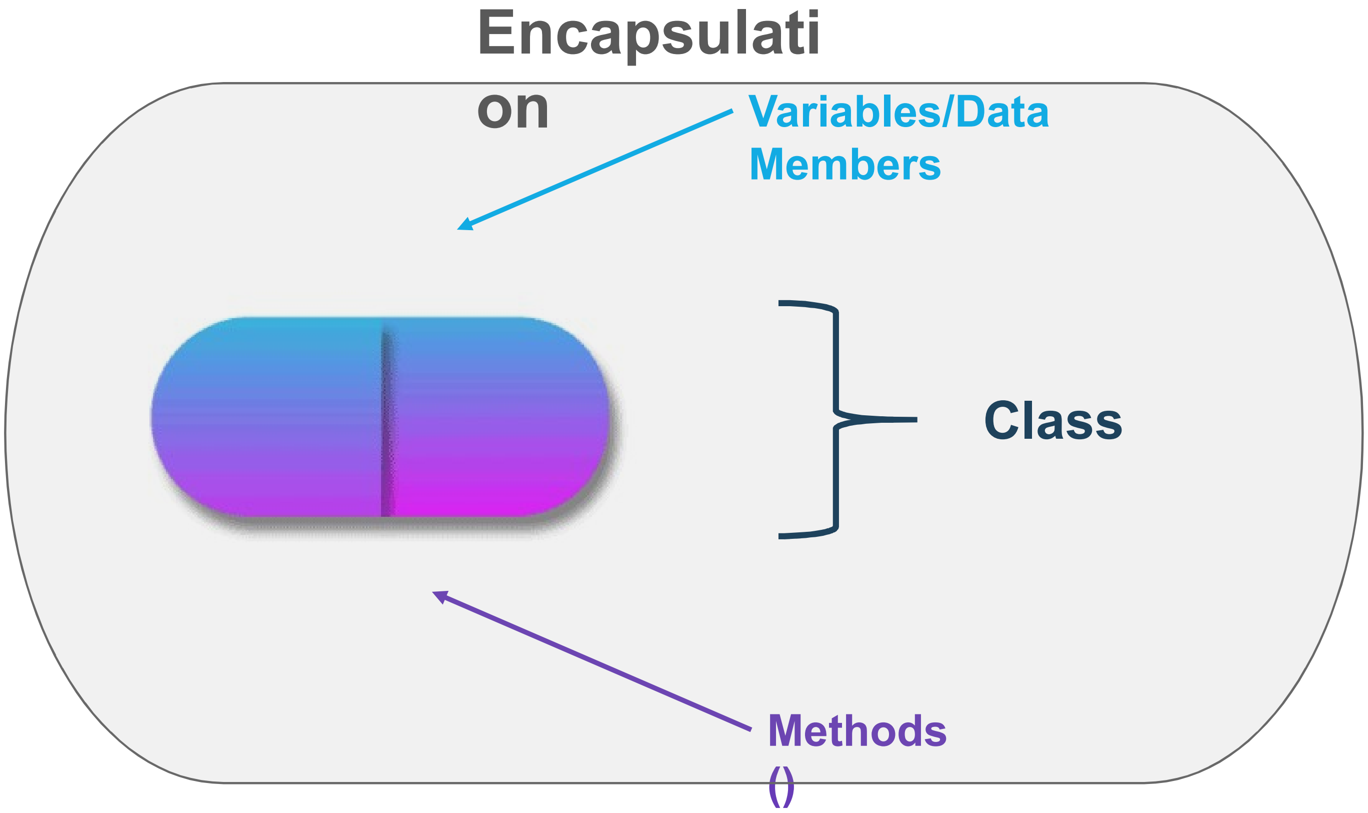
Encapsulation

Encapsulation

Program

```
Class
{
   Data Members(e.g. int a=10)
              +
   Methods(e.g. add() )

}
```

- The action of enclosing something in.
- In OOP, **encapsulation** refers to the bundling of data with the methods.



Encapsulati on

Variables/Data Members

Class

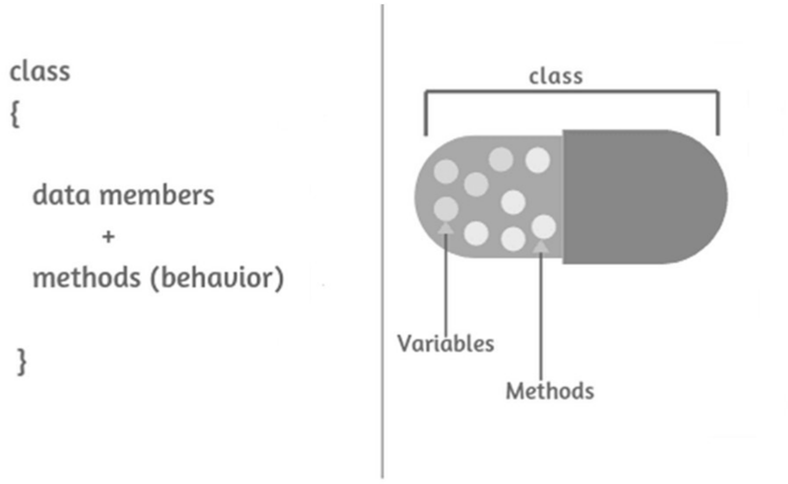
Methods ()

Encapsulation

- The wrapping up of data and functions into a single unit is known as **encapsulation**
- The insulation of the data from direct access by the program is called **data hiding** or **information hiding**.
- It is the process of enclosing one or more details from outside world through access right.

Advantages

- Protects an object from unwanted access
- It reduces implementation errors.
- Simplifies the maintenance of the application and makes the application easy to understand.
- Protection of data from accidental corruption.



```
class
{
   data members
         +
   methods (behavior)

}
```

Abstraction

Abstraction

- Data abstraction is also termed as information hiding.
- **Abstraction** is the concept of object-oriented programming that “represents” only essential attributes and “hides” unnecessary information.
- Abstraction is all about representing the simplified view and avoid complexity of the system.
- It only shows the data which is relevant to the user.
- In object-oriented programming, it can be implemented using Abstract Class.

Advantage:

- It reduces programming complexity.

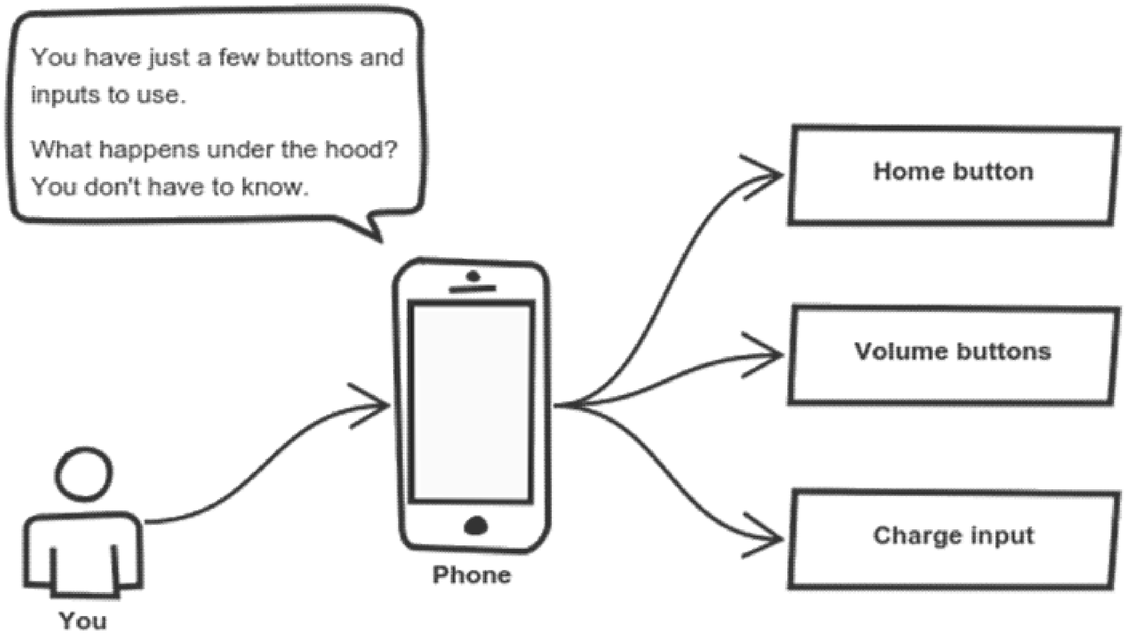
Example:

- A car is viewed as a car rather than its numerous individual components.

Using Abstraction

Without Abstraction

Abstraction

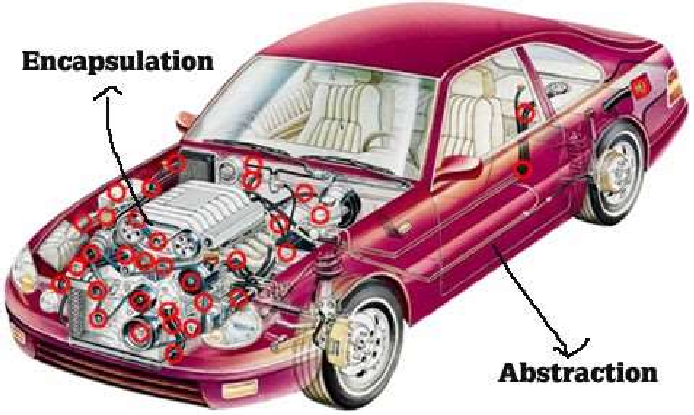


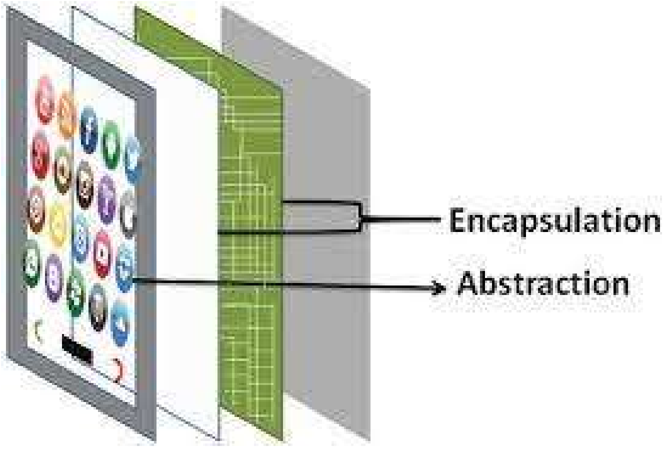
You have just a few buttons and inputs to use.

What happens under the hood? You don't have to know.

Abstraction vs. Encapsulation

Abstraction	Encapsulation
It means act of removing/ withdrawing something unnecessary.	It is act of binding code and data together and keep the data secure from outside
Applied at Designing stage.	Applied at Implementation
E.g. Interface and Abstract Class	E.g. Access Modifier (public, protected, private)
Purpose: Reduce code complexity	**Purpose**: Data protection





Implementing Abstraction

Abstract class

Abstract class

- **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 the object will do it.
- **Abstraction** is achieved using **Abstract classes** and **interfaces**.
- 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, we have to inherit it to another class and provide **implementations** of the abstract methods in it.

Abstract class (Example)

```java
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 s = new Swift();
		Baleno b = new Baleno();
		System.out.println(s.getAverage());
		System.out.println(b.getAverage());
	}
}
```

Output

```
22.5
23.2
```

Why Abstract Class?

- Sometimes, we need to define a superclass that declares the structure of a given abstraction without providing a complete implementation.
- The superclass will only define a generalized form, that will be shared by all the subclasses.
- The subclasses will fill the details of every method.
- When a superclass is unable to create a meaningful implementation for a method.

Points to remember for Abstract Class

- To declare a class abstract, you simply use the ***abstract*** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined.
- Cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

Interface

Interface

- An interface is similar to an abstract class with the following exceptions
  - **All** methods defined in an interface are **abstract**.
  - Interfaces doesn’t contain any logical implementation
  - Interfaces **cannot** contain **instance variables**. However, they can contain **public static final** variables (ie. constant class variables)
- Interfaces are declared using the "*interface*" keyword
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes using the "*implements*" keyword
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

Interface:Syntax

public or not used(default)

```
access interface name
{
      return-type method-name1(parameter-list);
      return-type method-name2(parameter-list);
      type final-varname1 = value;
      type final-varname2 = value;
      // ...
      return-type method-nameN(parameter-list);
      type final-varnameN = value;
}
```

Methods are without body(no implementation) and all methods are implicitly abstract.

implicitly final and static, cannot be changed by the implementing class, must
be initialized with a constant value.

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the ***implements*** clause in a class definition, and then create the methods defined by that interface.

```
access interface name
{
      return-type method-name1(parameter-list);
      type final-varname1 = value;
 }
```

```
access class classname
                 [extends superclass]
                     [implements interface [,interface...]]
{
// class-body
}
```

Interface (Example)

```
interface VehicleInterface {
    int a = 10;
    public void turnLeft();
    public void turnRight();
    public void accelerate();
    public void slowDown();
}
```

```
public class
    public s
    {
        CarCl
        c.tur
    }
}
```

Variable in i
are by d
public, stat

We have to provide implementation to all the methods of the interface

```
class CarClass implements VehicleInterface
{
    public void turnLeft() {
        System.out.println("Left");
    }
    public void turnRight() {
        System.out.println("Right");
    }
    public void accelerate() {
        System.out.println("Speed");
    }
    public void slowDown() {
        System.out.println("Brake");
    }
}
```

Output

```
Left
```

Interface (Example)

```
interface VehicleInterface {
    int a = 10;
    public void turnLeft();
    public void turnRight();
    public void accelerate();
    public void slowDown();
}
```

```
public class DemoInterface{
    public static void main(String[] a)
    {
VehicleInterface c = new CarClass();
        c.turnLeft();
    }
}
```

```
class CarClass implements VehicleInterface
{
    public void turnLeft() {
        System.out.println("Left");
    }
    public void turnRight() {
        System.out.println("Right");
    }
    public void accelerate() {
        System.out.println("Speed");
    }
    public void slowDown() {
        System.out.println("Brake");
    }
}
```

variable **c** is declared to be of the interface type ***VehicleInterface***, yet it was
assigned an instance of ***CarClass***.

Outpu
t

```
Left
```

Interface: Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.

```
interface VehicleInterface {
    int a = 10;
    public void turnLeft();
    public void turnRight();
    public void accelerate();
    public void slowDown();
}

public class DemoInterface{
    public static void main(String[] a)
    {
        CarClass c = new CarClass();
        c.turnLeft();
    }
}
```

```
abstract class CarClass implements VehicleInterface
    {
        public void turnLeft() {
            System.out.println("Left");
        }
        public void turnRight() {
            System.out.println("Right");
        }

        public void accelerate() {
            System.out.println("Speed");
        }
        public void slowDown() {
            System.out.println(“Brake");
        }


    }
```

Either class need to implement all the methods of Interface or declare that class as abstract if partial implementation is required.

Interface:Example

```java
interface StackIntf{
    public void
              push(int p);
    public int pop();
}
```

```java
class CreateStack implements StackIntf{
    int mystack[];
    int tos;
    CreateStack(int size){
        mystack= new int[size];
        tos=-1;
    }
    public void push(int p){
        if(tos==mystack.length-1){
            System.out.println("StackOverflow");
        }
        else{
            mystack[++tos]=p;
        }
    }
    public int pop(){
        if(tos<0){
            System.out.println("StackUnderflow");
            return 0;
        }
        else   return mystack[tos--];
    }
}
```

StackDemo.java

Interface:Example StackDemo.java

```java
class StackDemo{
   public static void main(String[] args) {
         CreateStack cs1= new CreateStack(5);
         CreateStack cs2= new CreateStack(8);
         for(int i=0;i<5;i++)
                     cs1.push(i);
         for(int i=0;i<8;i++)
                     cs2.push(i);

         System.out.println("MyStack1=");
         for(int i=0;i<5;i++)
               System.out.println(cs1.pop());

         System.out.println("MyStack2=");

         for(int i=0;i<8;i++)
               System.out.println(cs2.pop());
   }
}
```

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

InterfaceHierarchy.java

```java
interface A{
    void method1();
    void method2();
}
interface B extends A{
    void method3();
}
interface C extends A{
    void method4();
}
```

```java
class InterfaceHierarchy{
public static void main
          (String[] args) {
MyClass1 c1=new MyClass1();
MyClass2 c2=new MyClass2();
    c1.method1();
    c1.method2();
    c1.method3();
    c2.method1();
    c2.method2();
    c2.method4();
    }
}
```

```java
class MyClass1 implements B{
    public void method1(){
     System.out.println("inside MyClass1:method1()");}

    public void method2(){
      System.out.println("inside MyClass1:method2()");
    }

    public void method3(){
      System.out.println("inside MyClass1:method3()");
    }
}
```

```java
class MyClass2 implements C{
    public void method1(){
    System.out.println("inside MyClass2:method1()");}

    public void method2(){
    System.out.println("inside MyClass2:method2()");
    }

    public void method4(){
    System.out.println("inside MyClass2:method4()");
    }
}
```

Interface: Points to Remember

- Any number of classes can implement an **interface**.
- One class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.

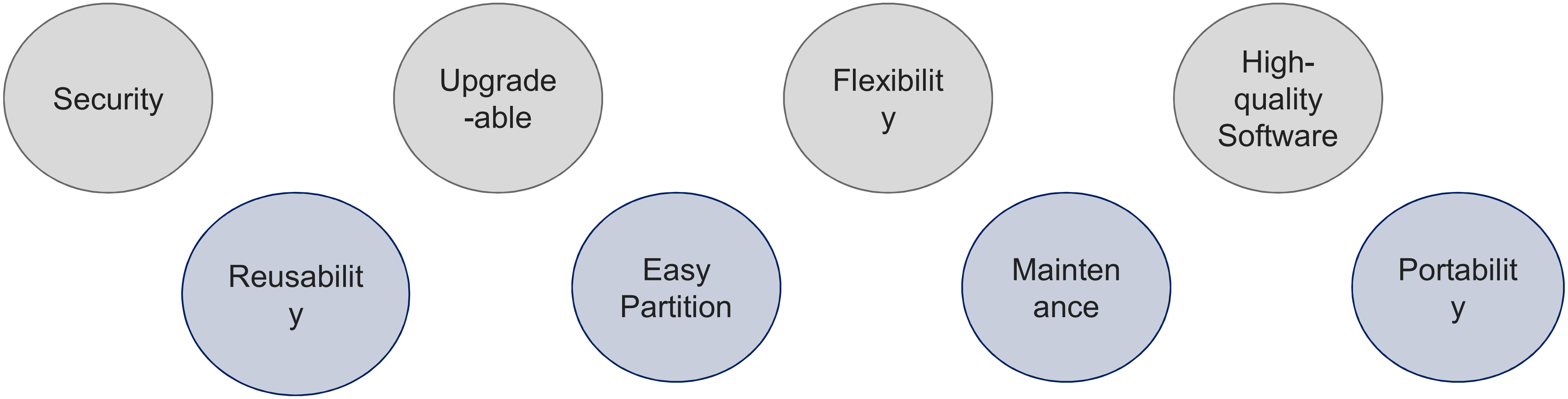
Abstract class vs. Interface

Abstract class	Interface
Abstract class doesn't support **multiple inheritance.**	Interface **supports multiple inheritance**.
Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Abstraction vs. Encapsulation

Advantages of Object-Oriented Programming

Advantages of Object-Oriented Programming



Security

Reusability

Upgrade-able

Easy Partition

Flexibility

Maintenance

High-quality Software

Portability

instanceof operator

- `instanceof` Operator
  - Syntax:

    (Object reference variable) instanceof (class/interface type)
  - Example:

    boolean result = name instanceof String;

Wrapper classes

- A Wrapper class is a class whose object **wraps** or **contains** a **primitive datatypes**.
- When we create an **object** to a wrapper class, it **contains** a **field** and in this field, we can store a primitive datatypes.
- In other words, we can **wrap** a **primitive** value into a wrapper **class object**.
- Use of wrapper class :
  - They **convert** primitive **datatypes** into **objects**.
  - The classes in **java.util** package handles **only objects** and hence wrapper classes help in this case also.
  - Data structures in the **Collection framework**, such as ArrayList and Vector, store **only objects** (reference types) and not primitive types.
  - An object is needed to support **synchronization** in **multithreading**.

Wrapper classes (Cont.)

Primitive datatype	Wrapper class	Example
byte	Byte	Byte b = new Byte((byte) 10);
short	Short	Short s = new Short((short) 10);
int	Integer	Integer i = new Integer(10);
long	Long	Long l = new Long(10);
float	Float	Float f = new Float(10.0);
double	Double	Double d = new Double(10.2);
char	Character	Character c = new Character('a');
boolean	Boolean	Boolean b = new Boolean(true);

- **Common Fields** (Except Boolean):
  - `MIN_VALUE` : will return the minimum value it can store.
  - `MAX_VALUE` : will return the maximum value it can store.

Parsing the String

- Using **wrapper class** we can **parse string** to any **primitive datatype (Except char)**.

```
byte b1 = Byte.parseByte("10");
short s = Short.parseShort("10");
int i = Integer.parseInt("10");
long l = Long.parseLong("10");
float f = Float.parseFloat("10.5");
double d = Double.parseDouble("10.5");
boolean b2 = Boolean.parseBoolean("true");
char c = Character.parseCharacter('a');
```

Note : for Integer class we have **parseInt** not **parseInteger**

BigInteger and BigDecimal

- The **BigInteger** class found in `java.math` package is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.
  - For example factorial of 100 contains 158 digits in it so we can’t store it in any primitive data type available.
  - There is no theoretical limit on the upper bound of the range because memory is allocated dynamically

```
import java.math.BigInteger;
public class DemoBigNumbers {
    public static void main(String[] args) {
        BigInteger bi = new BigInteger("1234567891234567891234567890");
        System.out.println(bi); // will return 1234567891234567891234567890
    }
}
```

- The **BigDecimal** class found in `java.math` package provides operation for arithmetic, comparison, hashing, rounding, manipulation and format conversion.
  - This method can handle very small and very big floating point numbers with great precision

```
BigDecimal bd = new BigDecimal("111111.000000000000000000025");
System.out.println(bd); //111111.000000000000000000025
```

String Class

- An object of the **String** class represents a string of characters.
- The String class belongs to the **java.lang** package, which does not require an import statement.
- Like other classes, **String** has constructors and methods.
- **String** class has **two operators**, **+ and +=** (used for concatenation).
- Empty String :
    - An empty String has no characters.

```
String word1 = "";
String word2 = new String();
```

Empty strings

    - Not the same as an uninitialized String.

```
String word1;
```

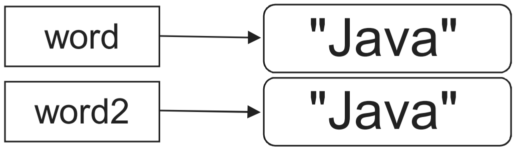
This is **null**

String Initialization

- Copy constructor creates a copy of an existing String.

Copy Constructor: Each variable points to a different copy of the String.

```
String word = new String("Java");
String word2 = new String(word);
```



word

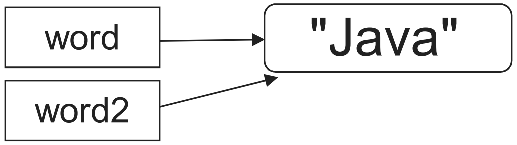
"Java"

word2

"Java"

Assignment: Both variables point to the same String.

```
String word = "Java";
String word2 = word;
```



word

"Java"

word2

String Immutability

Advantage

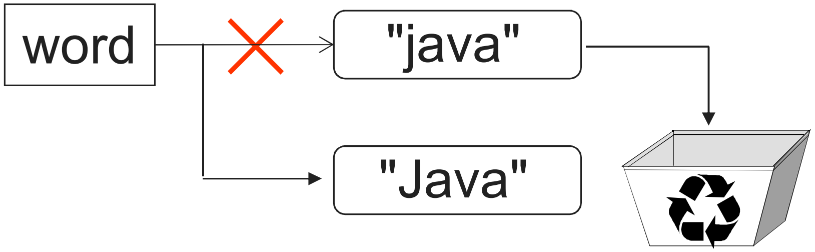
Convenient — Immutable objects are convenient because several references can point to the same object safely.

```
String name="DIET - Rajkot";
foo(name);
//Some operations on String name
name.substring(7,13);
bar(name);
/* we will be sure that the value of name
will be same for foo() as well as bar()
as String is immutable its value will be
same for both the functions.
 */
```

Disadvantage

Less efficient — you need to create a new string and throw away the old one even for small changes.

```
String word = "java";
char ch =
       Character.toUpperCase(
              word.charAt (0));
word =  ch + word.substring (1);
```



String Methods — length, charAt

int **length**();

- Returns the number of characters in the string

```
"Problem".length();
```

Returns:

7

int **charAt**(i);

- Returns the char at position i.

Character positions in strings starts from **0** – just like arrays.

```
"Window".charAt (2);
```

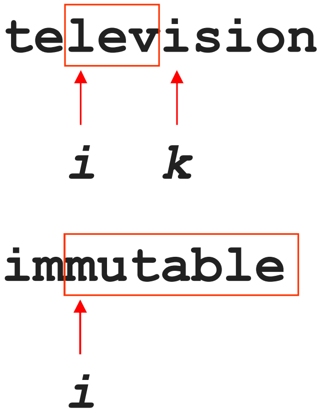
Returns:

'n'

String Methods — substring

We can obtain a portion of a string by use of substring(), It has two forms

1. String subs = word.**substring** (i, k);
   - returns the substring of chars in positions from **i** to **k-1**
2. String subs = word.**substring** (i);
   - returns the substring from the **i**-th char to the end



```
"television".substring(2,5);
"immutable".substring(2);
"rajkot".substring(9);
```

Returns:

“lev"

“mutable"

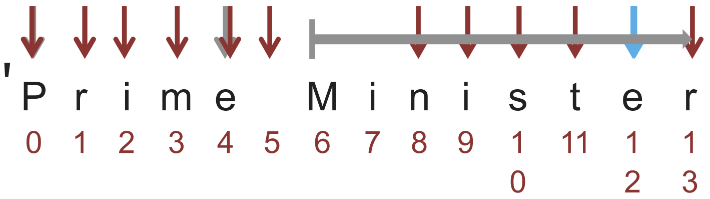
"" (empty string)

String Methods — Concatenation

```java
public class ConcatenationExample{
    public static void main(String[] args) {
        String word1 = "re";
        String word2 = "think";
        String word3 = "ing";
        int num = 2;
        String result = word1 + word2;
        // concatenates word1 and word2 "rethink"

        result = word1.concat(word2);
        // the same as word1 + word2 "rethink"

        result += word3;
        // concatenates word3 to result "rethinking"

        result += num;
        // converts num to String & joins it to result "rethinking2"
    }
}
```

String Methods — Find (indexOf)

String name = "Prime Minister";



name.**indexOf** ('P'); 0

name.**indexOf** ('e'); 4

name.**indexOf** ("Minister"); 6

name.**indexOf** ('e', 8); 12

(starts searching at position 8)

name.**indexOf** (“xyz"); -1

(not found)

name.**lastIndexOf** ('e'); 18

String Methods — Equality

boolean b = word1.**equals**(word2);

returns **true** if the string **word1** is equal to **word2**

```
b = "Raiders".equals("Raiders"); // will return true
b = "Raiders".equals("raiders"); // will return false
```

boolean b = word1.**equalsIgnoreCase**(word2);

returns **true** if the string **word1** matches **word2**, ignoring the case of the string.

```
b = "Raiders".equalsIgnoreCase("raiders"); // will return true
```

String Methods — Comparisons

int diff = word1.**compareTo**(word2);

returns the “difference” **word1 – word2**

int diff = word1.**compareToIgnoreCase**(word2);

returns the “difference” **word1 – word2**,
ignoring the case of the strings

- Usually programmers don’t care what the numerical “difference” of word1 - word2 is, what matters is if
  - the difference is negative (word1 less than word2),
  - zero (word1 and word2 are equal)
  - or positive (word1 grater than word2).
- Often used in conditional statements.

```
if(word1.compareTo(word2) > 0){
        //word1 grater than word2…
}
```

Comparison Examples

```
//negative differences
diff = "apple".compareTo("berry"); // a less than b
diff = "Zebra".compareTo("apple"); // Z less than a
diff = "dig".compareTo("dug"); // i less than u
diff = "dig".compareTo("digs"); // dig is shorter
```

```
//zero differences
diff = "apple".compareTo("apple"); // equal
diff = "dig".compareToIgnoreCase("DIG"); // equal
```

```
//positive differences
diff = "berry".compareTo("apple"); // b grater than a
diff = "apple".compareTo("Apple"); // a grater than A
diff = "BIT".compareTo("BIG"); // T grater than G
diff = "application".compareTo("app"); // application is longer
```

String Methods — trim & replace

trim() method

String word2 = **word1.trim()**;

- returns a new string formed from **word1** by removing white space at both ends,
- it does not affect whites space in the middle.

```
String word1 = "     Hello From Darshan ";
String word2 = word1.trim();
// word2 is "Hello From Darshan"
// no spaces on either end
```

replace() method

String word2 = word1.**replace**(oldCh, newCh);

➥ returns a new string formed from **word1** by replacing all occurrences of **oldCh** with **newCh**

```
String word1 = "late";
String word2 = word1.replace('l', 'h');
System.out.println(word2);
//Output : "hate"

String str1 = "Hello World";
String str2 =
str1.replace("World","Everyone");
System.out.println(str2);
// Output : "Hello Everyone"
```

String Methods — Changing Case

String word2 = word1.**toUpperCase**();

returns a new string formed from word1 by **converting** its characters to **upper** case

String word3 = word1.**toLowerCase**();

returns a new string formed from word1 by **converting** its characters to **lower** case

```
String word1 = "HeLLo";
String word2 = word1.toUpperCase(); // "HELLO"
String word3 = word1.toLowerCase(); // "hello"
```

StringBuffer

- The **java.lang.StringBuffer** class is a thread-safe, mutable sequence of characters.
- Following are the important points about StringBuffer:
  - A string buffer is like a String, but can be **modified** (**mutable**).
  - It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
  - [illegible]

S.N	Constructor & Description

StringBuffer Methods

Method	description
append(String s)	is used to append the specified string with this string.
insert(int offset, String s)	is used to insert the specified string with this string at the specified position.
replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
reverse()	is used to reverse the string.

- Remember : “StringBuffer” is mutable
  - As **StringBuffer** class is mutable we need not to replace the reference with a new reference as we have to do it with String class.

```
StringBuffer str1 = new StringBuffer("Hello Everyone");
str1.reverse();
// as it is mutable can not write str1 = str1.reverse();
// it will change to value of the string itself
System.out.println(str1);
// Output will be “enoyrevE olleH”
```

String Builder

- Java **StringBuilder** class is used to create **mutable string**.
- The Java StringBuilder class is same as StringBuffer class except that it is **non-synchronized**.
- It is available since JDK 1.5.
- It has similar methods as StringBuffer like append, insert, reverse etc…

ArrayList

- The java.util.**ArrayList** class provides resizable-array and implements the **List** interface.
- Following are the important points about **ArrayList**:
  - It implements all optional list operations and it also permits all elements, including null.
  - It provides methods to manipulate the size of the array that is used internally to store the

S.N.	Constructor & Description
1	ArrayList() This constructor is used to create an empty list with an initial capacity sufficient to hold 10 elements.
2	ArrayList(Collection<? extends E> c) This constructor is used to create a list containing the elements of the specified collection.
3	ArrayList(int initialCapacity) This constructor is used to create an empty list with an initial capacity.

ArrayList (method)

S.N.	Method & Description
1	void **add**(int index, E element) This method inserts the specified element at the specified position in this list.
2	boolean **addAll**(Collection<? extends E> c) This method appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator
3	void **clear**() This method removes all of the elements from this list.
4	boolean **contains**(Object o) This method returns true if this list contains the specified element.
5	E **get**(int index) This method returns the element at the specified position in this list.
6	int **indexOf**(Object o) This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

ArrayList (method) (cont.)

S.N.	Method & Description
7	boolean **isEmpty**() This method returns true if this list contains no elements.
8	int **lastIndexOf**(Object o) This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
9	boolean **remove**(Object o) This method removes the first occurrence of the specified element from this list, if it is present.
10	E **set**(int index, E element) This method replaces the element at the specified position in this list with the specified element.
11	int **size**() This method returns the number of elements in this list.
12	Object[] **toArray**() This method returns an array containing all of the elements in this list in proper sequence (from first to last element).

Thank You