Inheritance

- Type of association between objects:
- IS-A or Kind-of

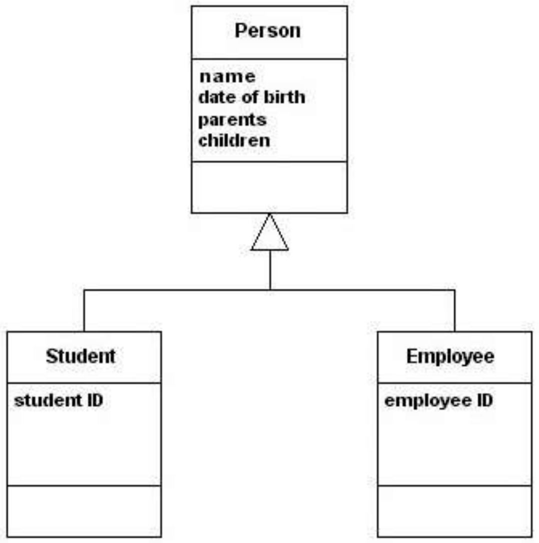
  Also known as Generalization or Specialization
- HAS-A or Part-of or contains-of

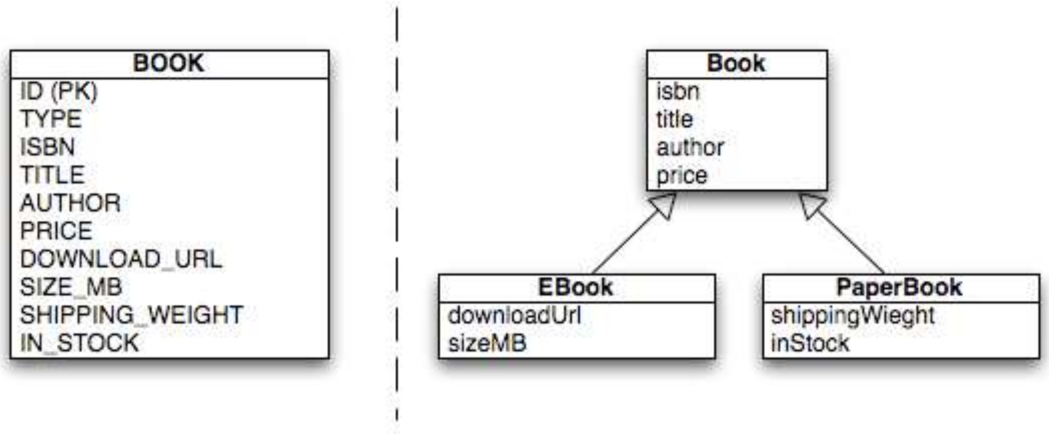
  Also known as Aggregation or Containership

IS-A (Inheritance)

- Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.
- **Inheritance** is a mechanism in which one object acquires some of the properties and behaviours of parent object.
- The idea behind inheritance is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Example: Inheritance

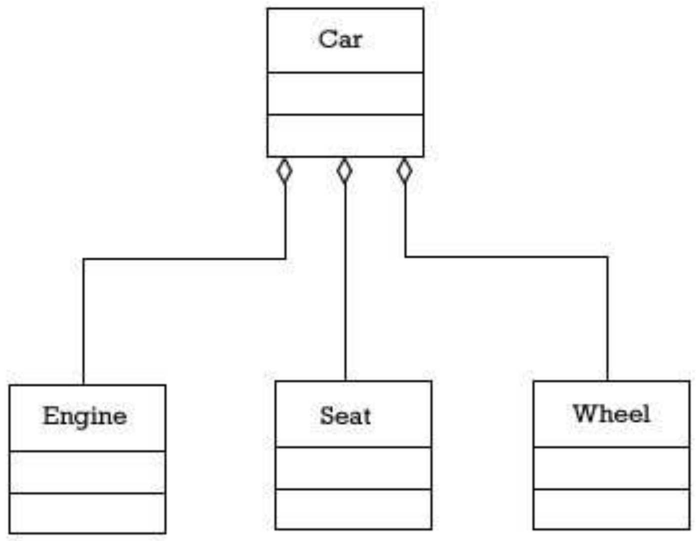


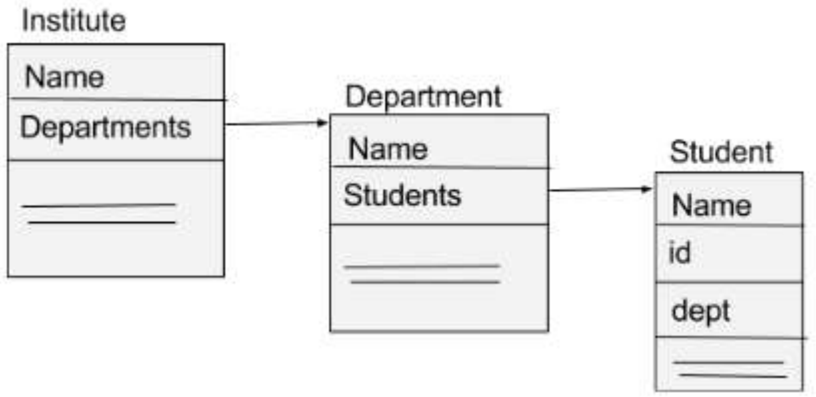


HAS-A (Aggregation)

- If a class contains an object of other class, it is known as Aggregation. Aggregation represents HAS-A relationship.
- It is a special form of Association where:
- It represents **Has-A** relationship.
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity

Example: Aggregation



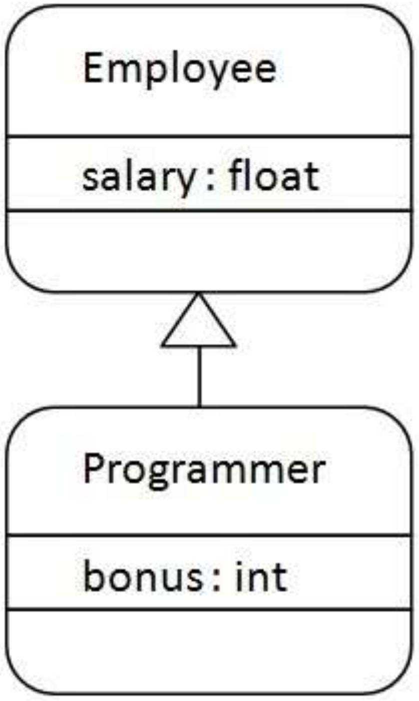


Why Inheritance and Aggregation

- **Why use inheritance**
  - For Method Overriding (so runtime polymorphism can be achieved).
  - For Code Reusability.
- Reuse is also best achieved by aggregation when there is no is-a relationship.

Toy Examples in Java

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

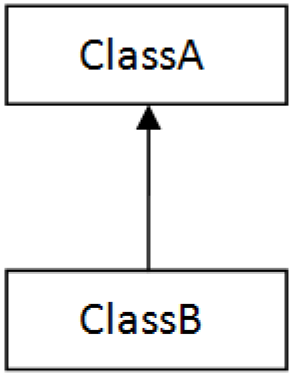


```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

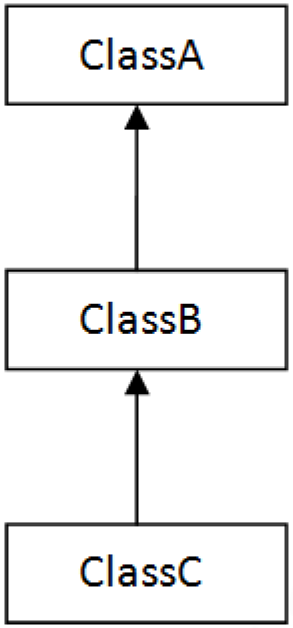
Output:

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

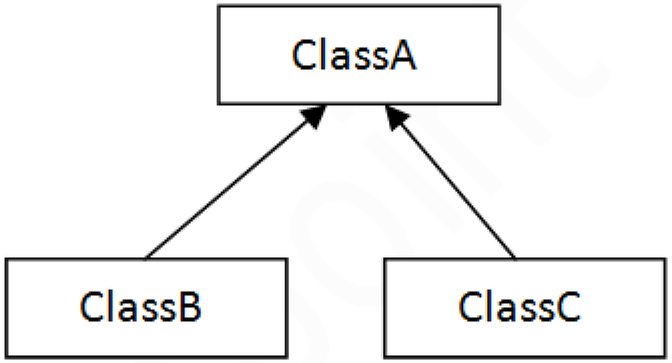
Types of Inheritance



1) Single

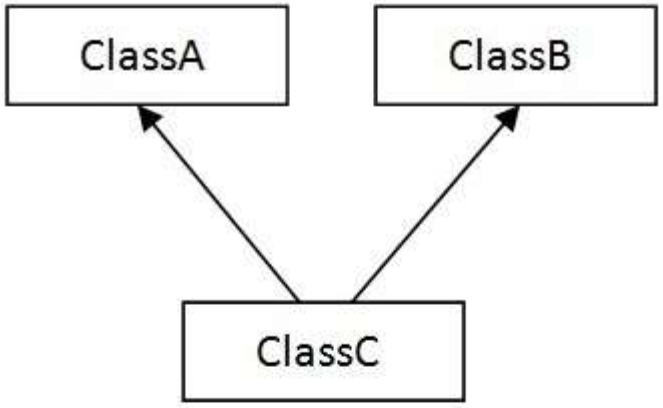


2) Multilevel

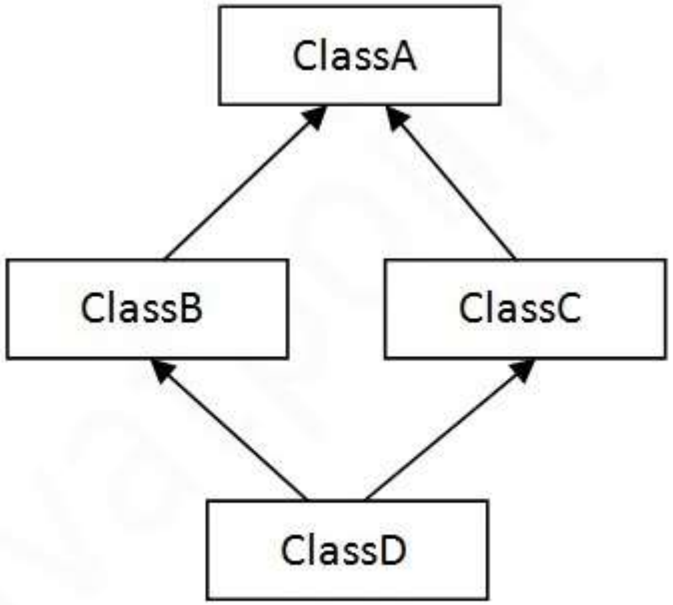


3) Hierarchical

Types of Inheritance…Contd



4) Multiple



5) Hybrid

.

Single Inheritance Example

```
class Animal
{                    void eat()
                    {
                      System.out.println("eating...");
                   }
}
class Dog extends Animal
{
        void bark(){System.out.println("barking...");       //eat(); }
}
class TestInheritance
{
         public static void main(String args[])
            {
                     Dog d=new Dog();
                    d.bark();
                    d.eat();
            }
}
Output:
barking...
eating...
```

Aggregation Example

```
public class Address
 {
   String city,state,country;
   public Address(String city, String state, String country)
     {
         this.city = city;
         this.state = state;
         this.country = country;
      }


}
```

```
public class Emp
 {
        int id;
       String name;
       Address address;
      public Emp(int id, String name,Address address)
       {
           this.id = id;
           this.name = name;
           this.address=address;
        }
      void display()
         {
              System.out.println(id+" "+name);
              System.out.println(address.city+" "+address.state+" “
            +address.country);
          }
}
```

```java
class Test
{
    public static void main(String[] args)
     {
            Address address1=new Address("gzb","UP","india");
            Address address2=new Address("gno","UP","india");

            Emp e=new Emp(111,"varun",address1);
            Emp e2=new Emp(112,"arun",address2);

           e.display();
           e2.display();

     }
}
```

- Design a discount system for a beauty saloon, which provides services and sells beauty products. It offers 3 types of memberships: Premium, Gold and Silver. Premium, gold and silver members receive a discount of 20%, 15%, and 10%, respectively, for all services provided.
- Customers without membership receive no discount. All members receives a flat 10% discount on products purchased (this might change in future).
- Identify the classes.
- Your System shall compute the total bill if a customer purchases $x of products and $y of services, for a visit. Also write a test program to exercise all the classes.

A shopping mall has four types of employees and their pay structure is given below:

- Salaried employees are paid a fixed weekly salary regardless of the number of hours worked. Hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours. Commission employees are paid a percentage of their sales. Salaried-commission employees receive a base salary plus a percentage of their sales.
- For the current pay period, the shopping mall has decided to reward salaried-commission employees by adding 10% to their base salaries. Develop a Java class hierarchy for the given case study:

Overloading in Java

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

- **What is the advantage?**

  We don’t have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2,… or sum2Int, sum3Int, … etc.

```
// Java program to demonstrate working of method
// overloading in Java.

public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y) {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z) {
         return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y) {
         return (x + y);
    }

    // Driver code
    public static void main(String args[]) {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

How Overloaded methods are resolved?

- Compiler follows the following sequence of steps to find the match from many function definitions for a particular call:
- **Exact match**: If any function definition contains same number and number of parameters as in function call, then compiler bind that definition with the call.
- For example:

```
System.out .println(s.sum(10, 20));
```

for this call the definition

```
public int sum(int x, int y) {
        return (x + y);
    }
```

Will be invoked by the compiler.

2. Integral Promotion

- If there is no exact match exist then compiler follows the Integral Promotion is done in the given chain:
- byte $\longrightarrow$ short $\longrightarrow$ int $\longrightarrow$ long
- float $\longrightarrow$ double

```
public int sum(int x, int y) {
        return (x + y);
    }
```

And the call is: System.out.println(s.sum(a, b));

where a and b are byte then by integral promotion **byte** will be converted to **int.**

Vice versa is possible only through explicit casting.

i.e. long can be converted to byte by type casting only.

Long $\rightarrow$ byte or int $\rightarrow$ byte

```
public int sum(byte x, byte y) {
        return (x + y);
    }
```

And the call is: System.out.println(s.sum(a, b));

Where a and b are long then we have to write:

System.out.println(s.sum((byte)a, ((byte)b));

This is explicit casting.

3. Internal Conversion

$$\begin{tikzcd}
\text{byte} \arrow[d] \arrow[dr] & \text{short} \arrow[dl] & \text{int} \arrow[dll] \arrow[dl] & \text{long} \arrow[dll] \\
\text{float} & \text{double} & &
\end{tikzcd}$$

- Such internal conversion is possible from
- byte $\longrightarrow$ float or byte $\longrightarrow$ double

```
e. g. public int sum(double x, double y) {
        return (x + y);
    }
```

- And the call is: System.out.println(s.sum(a, b));

where a and b are byte then by internal conversion **byte** will be converted to **double.**

User Defined Conversion(by calling constructor)

- In such case one class object can be converted to another class object but both the class must have perameterised constructor in it only then such conversion is possible.

e.g.
```
public Rational sum(Rational x, Rational y) {
        return Rational. add(x ,y);
 }
```

- And if the call is:System.out.println(s.sum(a, b));

where a and b are int then by user defined conversion **int** will be converted to **Rational.**

•If compiler fails to find any match after following these four steps then it reports syntax error

- **Can we overload methods on return type?**

  We **cannot** overload by return type.

- **Can we overload static methods?**

  ‘**Yes**’. We can have two ore more static methods with same name, but differences in input parameters.

- **Can we overload methods that differ only by static keyword?**

  We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same).

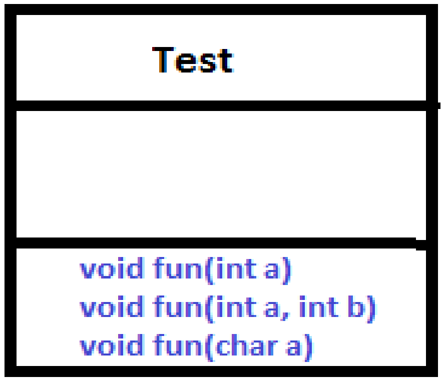
- **Can we overload main() in Java?**

```
import java.io.*;

public class Test {

    // Normal main()
    public static void main(String[] args) {
        System.out.println("Hi A1 main)");
        Test.main("Group A1");
    }

    // Overloaded main methods
    public static void main(String arg1) {
        System.out.println("Hi, " + arg1);
        Test.main("Dear A1","My Group");
    }
    public static void main(String arg1, String arg2) {
        System.out.println("Hi, " + arg1 + ", " + arg2);
    }
}
```

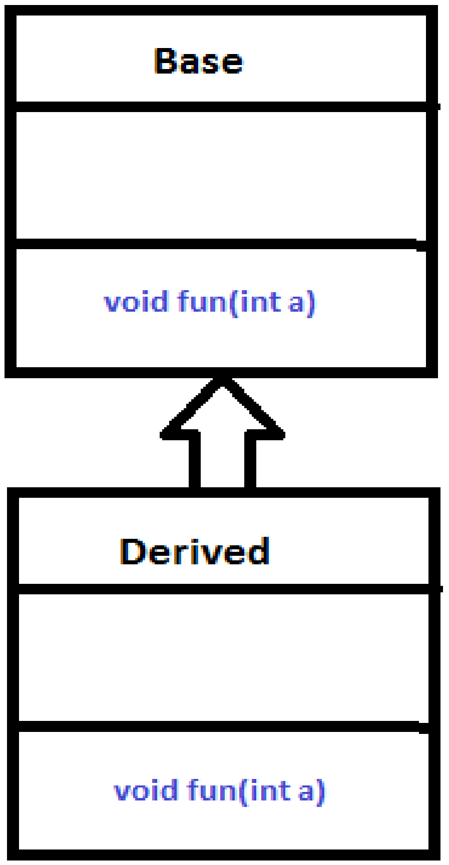
Difference between Overloading and Overriding?

- **Overloading** is about same function have different signatures.
- **Overriding** is about same function, same signature but different classes connected through inheritance.
- **Overloading** is an example of compiler time polymorphism and **overriding** is an example of run time polymorphism.

- Can we overload Constructors? Yes
- Can we override Constructors? No



Overloading



Overriding

Inheritance and Constructors in Java

- A subclass inherits all the members (fields, methods which are not private) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

- In Java, constructor of base class with no argument gets automatically called in derived class constructor.

```
class Base {
  Base() {
    System.out.println(″Base  Constructor Called ″);
  }
}

class Derived extends Base {
  Derived() {
    System.out.println(″Derived Constructor Called ″);
  }
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived();
  }}
```

- But, if we want to call parameterized constructor of base class, then we can call it using **super().**

- If the base class has only the parameterized constructor, then we have to define a parameterized constructor in sub class also.

```
class Base {
  int x;
  Base(int x) {
    this.x = x;
  }
}
 class Derived extends Base {
  int y;
  Derived(int x, int y) {
    super(x);  // calling of super class constructor explicitly
    this.y = y;
  }
  void Display() {
    System.out.println("x = "+x+", y = "+y);
  }
}
 public class Main {
  public static void main(String[] args) {
    Derived d = new Derived(10, 20);
    d.Display();
  }
}
```