Hierarchy

Is java support multiple inheritance?

- Java supports multiple interface inheritance.

```
interface A{  }
interface B{  }
interface C extends A, B{ } //OK: Multiple interface inheritance
```

- Java supports multiple interface implementation inheritance

```
interface A{  }
interface B{  }
class C implements A, B{ }  //OK: Multiple interface implementation inheritance
```

- Java do not support multiple implementation inheritance

```
class A{  }
class B{  }
class C extends A, B{  } //Not OK: Multiple Implementation inheritance
```

  - In C++, combination of two or more than two types of inheritance is called hybrid inheritance.
  - If we combine hierarchical inheritance and multiple inheritance then it becomes diamond inheritance, which creates some problems.
  - To avoid diamond problem, Java do not support multiple implementation inheritance / multiple class inheritance.
  - Conclusion: In Java, class can extend only one class.
- Consider following code:

```
class Object{
  public String toString( );
  public boolean equals( Object obj )
  public native int hashCode( );
  protected native Object clone( )throws CloneNotSupportedException;
  protected void finalize()throws Throwable;
  public final native Class<?> getClass( );
  public final void wait( )throws InterruptedException;
  public final native void wait( long timeOut )throws InterruptedException;
  public final void wait( long timeOut, int nanos )throws InterruptedException;
  public final void notify();
  public final void notifyAll();
}
class Person extends Object{
    //TODO
}
class Employee extends Person{
    //TODO
}
```

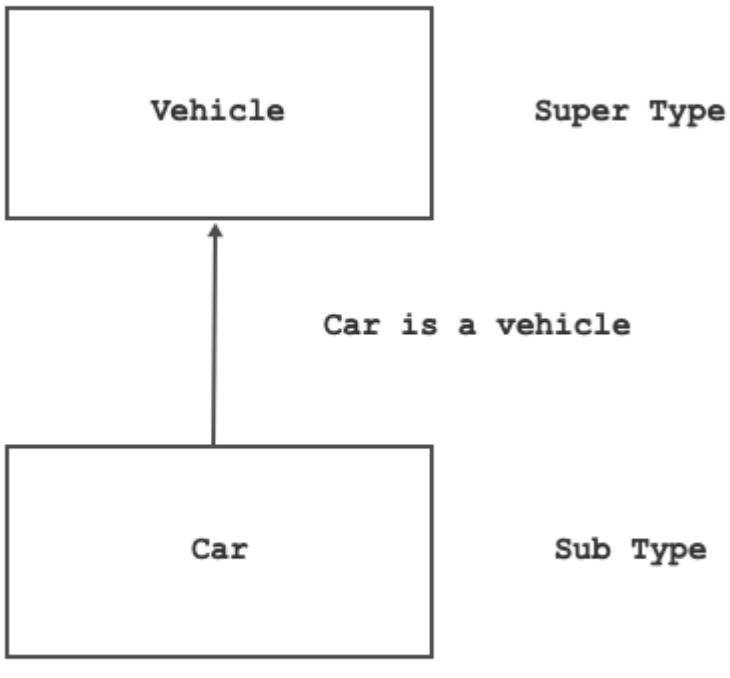
  - In above code, class Person will extend java.lang.Object class and class Employee will extend only Person class(Not java.lang.Object class).
  - class Person is direct super class and class java.lang.Object is indirect super class of Employee class.

Types of inheritance

Single inheritance

- In case of inheritance, if single super type is having single sub type then it is called as single inheritance.

- For example: Car is a vehicle.



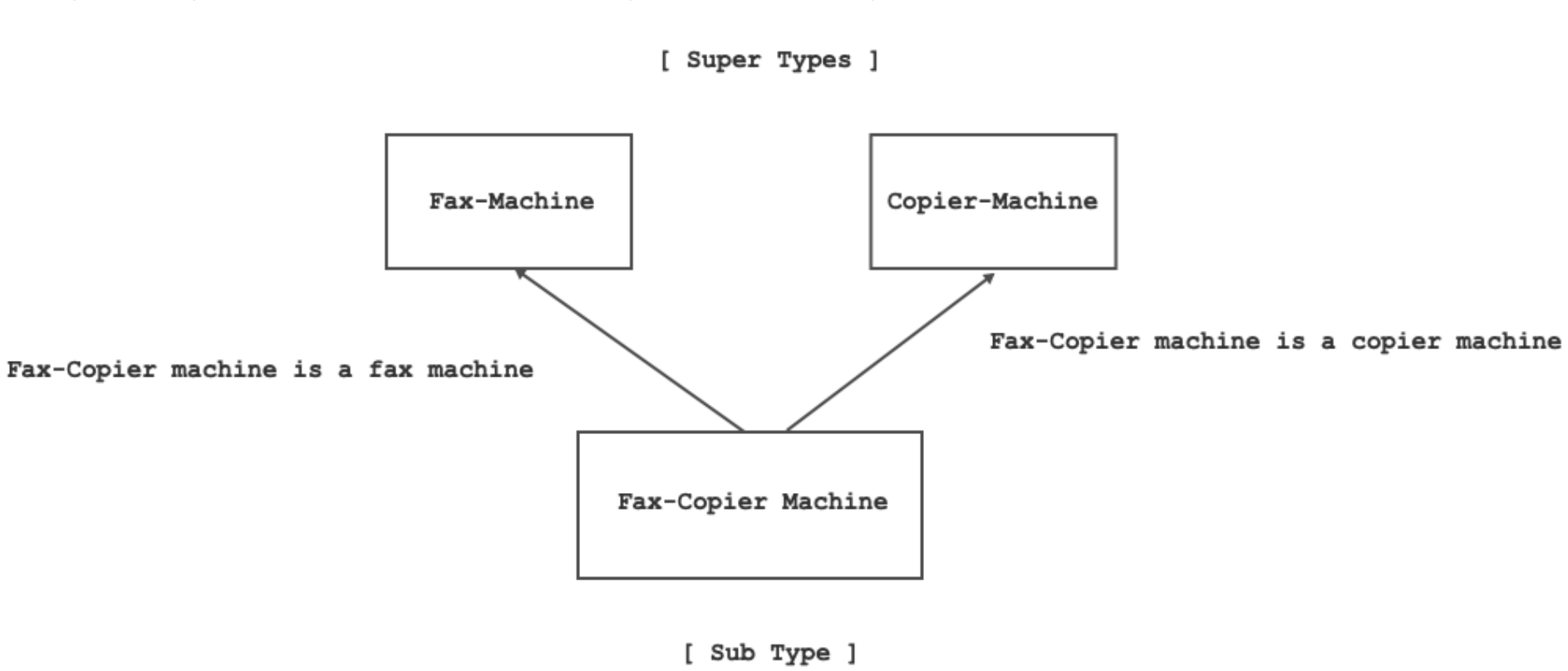
- Syntax:

```
class Vehicle{
  //TODO
}
class Car extends Vehicle{  //Single inheritance( implementation inheritance )
  //TODO
}
```

```
interface Iterable<T>{
  //TODO
}
interface Collection<E> extends Iterable<T>{  //Single inheritance( interface inheritance )
  //TODO
}
```

Multiple inheritance

- In case of inheritance, If multiple super types are having single sub type then it is called multiple inheritance.
- For Example: Fax-Copier machine is a fax machine and Fax-Copier machine is also Copier machine.



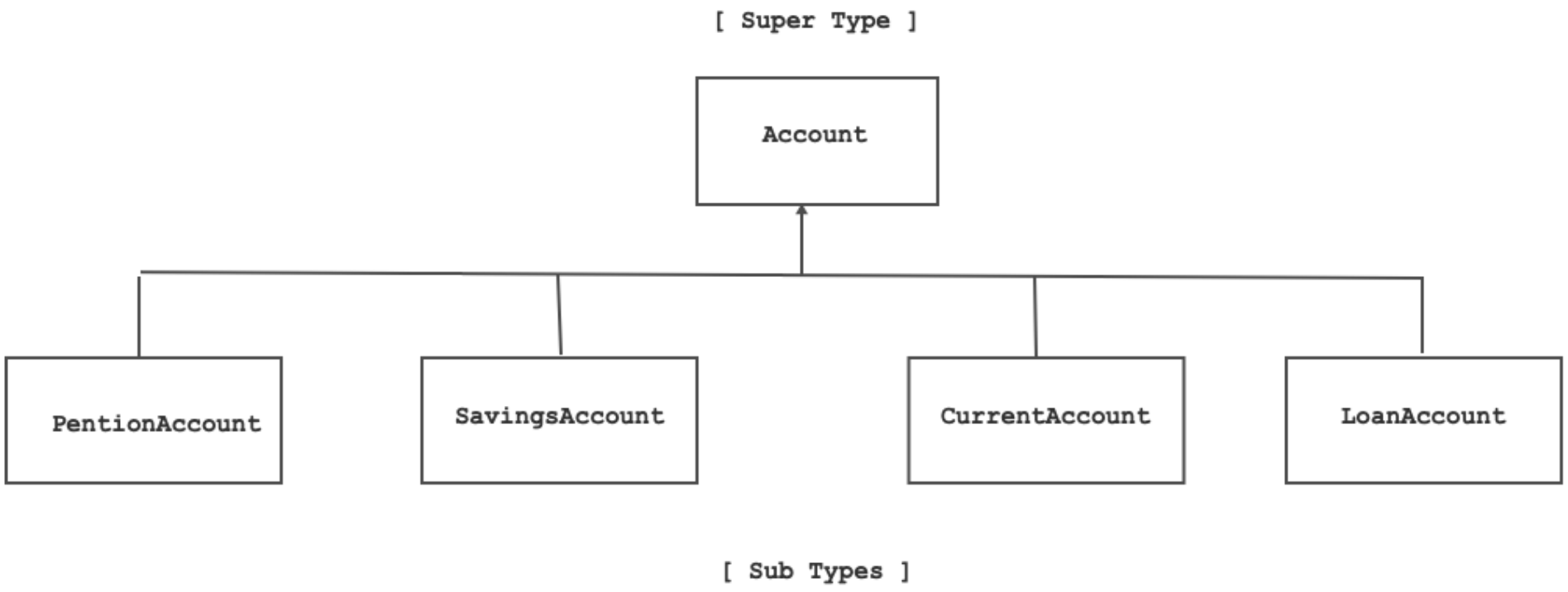
- Syntax:

```
class FaxMachine{ } //OK
class CopierMachine{  }//OK
class FaxCopierMachine extends FaxMachine, CopierMachine{ } //Not OK: Multiple Inheritance
```

```
interface List<E>{  }
interface Queue<E>{ }
class LinkedList implements List<E>, Queue<E>{  } //OK: Multiple Inheritance
```

Hierarchical inheritance

- In case of inheritance, If single super type is having multiple sub types then it is called as hierarchical inheritance.
- For Example: SavingsAccount is Account, CurrentAccount is a Account.



- Syntax:

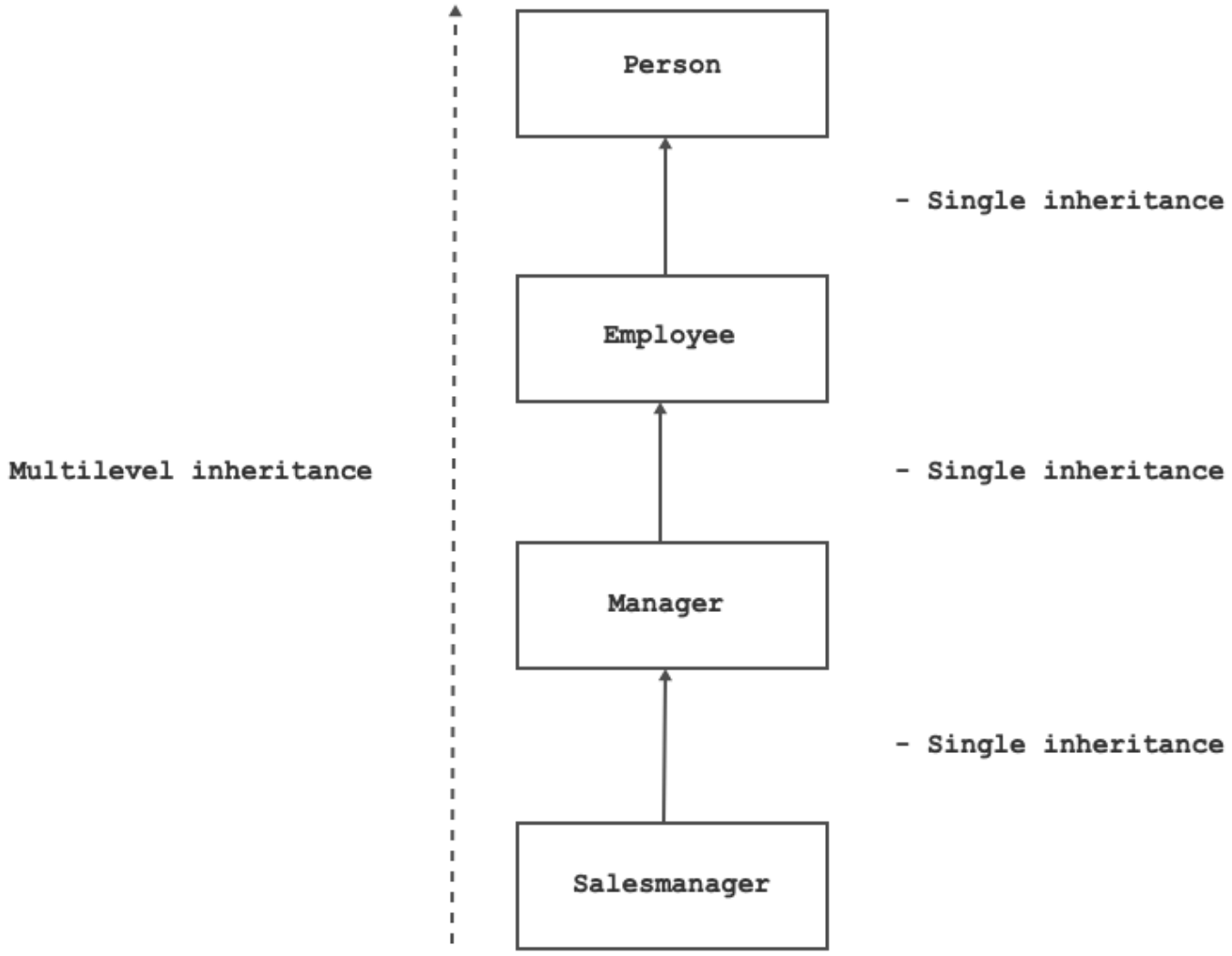
```
//Hierarchical inheritance
class Account{  }
class SavingsAccount extends Account{ } //Single inheritance
class CurrentAccount extends Account{ } //Single inheritance
```

```
interface Collection<E>{  }
interface List<Collection> extends <Collection<E>{  }
interface Queue<E> extends Collection<E>{  }
interface Set<E> extends Collection<E>{  }
```

Multilevel inheritance

- In case of inheritance, if single inheritance is having multiple levels then it is called as multilevel inheritance.

- For Example: Employee is a Person; Manager is a Employee; Salesanager is a Manager;



- Syntax:

```
//Multilevel inheritance
class Person{ };
class Employee extends Person{  }       //Single inheritance
class Manager extends Employee{  }      //Single inheritance
class SalesManager extends Manager{  }  //Single inheritance
```

```
//Multilevel inheritance
interface Iterable<T>{  }
interface Collection<E> extends Iterable<T>{  }
interface Queue<E> extends Collection<E>{  }
interface Deque<E> extends Queue<E>{  }
```

Syntax to use class and interface

- Interfaces: I1, I2, I3
- Classes: C1, C2, C3
    - I2 implements I1 //Not OK
    - I2 extends I1 //OK: Interface inheritance
    - I3 extends I1, I2 //OK: Multiple Interface inheritance
    - I1 extends C1; //Not OK
    - I1 implements C1; //Not OK
    - C1 extends I1; //Not OK
    - C1 implements I1; //OK: Interface implementation inheritance
    - C1 implements I1,I2; //OK: Multiple Interface implementation inheritance
    - C2 extends C1 implements I1,I2; //OK
    - C2 implements C1; //Not OK
    - C2 extends C1; // OK: Implementation inheritance
    - C3 extends C1,C2; // Not OK: Multiple Implementation inheritance

Access modifiers revision

- Below table will describe the access modifiers in Java:

	Same Package			Different Package	
Access Modifier	Same class	Sub class	Non Sub class	Sub Class	Non Sub class
private	A	NA	NA	NA	NA
package private	A	A	A	NA	NA
protected	A	A	A	A	NA
public	A	A	A	A	A

- Reference: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
- Private members(fields/methods/nested types) inherit into sub class. If we want to access value of private field inside sub class then we should use getter and setter methods of super class.

```
class A{
  private int num1;
  public A( ) {
    this.num1 = 10;
  }
  public int getNum1() {
    return this.num1;
  }
}
class B extends A{

}
public class Program {
  public static void main(String[] args) {
    B b = new B();
    //System.out.println(b.num1); //The field A.num1 is not visible
    System.out.println( b.getNum1() );    //10
  }
  public static void main1(String[] args) {
    A a = new A();
    //System.out.println(a.num1); //The field A.num1 is not visible
    System.out.println( a.getNum1() );
  }
}
```

Accessing members using super keyword

- According, clients requirement, if implementation of super class method is logically incomplete / partially complete then we should redefine method inside sub class. In other words, we should override method inside sub class.

```
public class Person {
  private String name;
  private int age;
  public Person() {
    this("",0);
  }
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public void printRecord( ) {
    System.out.println("Name    :   "+this.name);
    System.out.println("Age :   "+this.age);
  }
}

public class Employee extends Person{
  private int empid;
  private float salary;
  public Employee() {
```

```
    this("",0,0,0.0f);
  }
  public Employee( String name, int age, int empid, float salary ) {
    super( name, age );
    this.empid = empid;
    this.salary = salary;
  }
  public void printRecord() {   //overriden method
    System.out.println("Empid   :   "+this.empid);
    System.out.println("Salary  :   "+this.salary);
  }
}
public class Program {
  public static void main(String[] args) {
    Employee emp = new Employee("Sandeep", 39,3778, 45000.50f);
    emp.printRecord();  //Due to shadowing, Employee.printRecord() will call  //3778, 45000.50f
  }
}
```

- If name of super class method and sub class method is same and if we try to invoke such method on instance of sub class then preference will be given to the sub class method. It is called as method shadowing.

- If we want to use any member of super class inside method of sub class then we should use super keyword.

```
public class Employee extends Person{
  private int empid;
  private float salary;
  public Employee() {
    this("",0,0,0.0f);
  }
  public Employee( String name, int age, int empid, float salary ) {
    super( name, age );
    this.empid = empid;
    this.salary = salary;
  }
  public void printRecord() { //overriden method
    super.printRecord();
    System.out.println("Empid :   "+this.empid);
    System.out.println("Salary    :   "+this.salary);
  }
}
```

- According to client's requirement, if implementation of exisiting class is logically incomplete / partially complete then we should extend that class. In other words we should create sub class of that class i.e we should use inheritance.

Shadowing

- In case of local variable

```
class Complex{
  private int real;
  private int imag;
  public Complex( ) {
    this.real = 10;
    this.imag = 20;
  }
  public void setReal(int real) {
    real = real;    //Shadowing : Local variable is assigned to itself
  }
  public void setImag(int imag) {
    this.imag = imag;
  }
  @Override
  public String toString() {
    return this.real+"  "+this.imag;
  }
}
public class Program {
  public static void main(String[] args) {
    Complex c1 = new Complex( );    //10,20
    c1.setReal(50);
    System.out.println(c1.toString());  //10,20
```

```
  }
}
```

- In case of fields

```
class A{
  int num1 = 10;
  int num3 = 30;
}
class B extends A{
  int num2 = 20;
  int num3 = 40;
  public void printRecord( ) {
    System.out.println("Num1    :   "+num1);    //OK: 10
    System.out.println("Num1    :   "+this.num1);   //OK: 10
    System.out.println("Num1    :   "+super.num1);  //OK: 10

    System.out.println("Num2    :   "+num2);    //OK: 20
    System.out.println("Num2    :   "+this.num2);   //OK: 20

    System.out.println("Num3    :   "+num3);    //OK: 40    =>  Shadowing
    System.out.println("Num3    :   "+this.num3);   //OK: 40    =>  Shadowing
    System.out.println("Num3    :   "+super.num3);  //OK: 30
  }
}
public class Program {
  public static void main(String[] args) {
    B b = new B();
    b.printRecord();
  }
}
```

- In case method

```
class A{
  public void showRecord( ) {
    System.out.println("A.showRecord()");
  }
  public void printRecord( ) {
    System.out.println("A.printRecord()");
  }
}
class B extends A{
  public void displayRecord( ) {
    System.out.println("B.displayRecord()");
  }
  public void printRecord( ) {
    System.out.println("B.printRecord()");
  }
}
public class Program {
  public static void main(String[] args) {
    B b = new B();
    b.showRecord();       //A.showRecord()
    b.displayRecord();    //B.displayRecord()
    b.printRecord();  //B.printRecord()   => Shadowing
  }
}
```

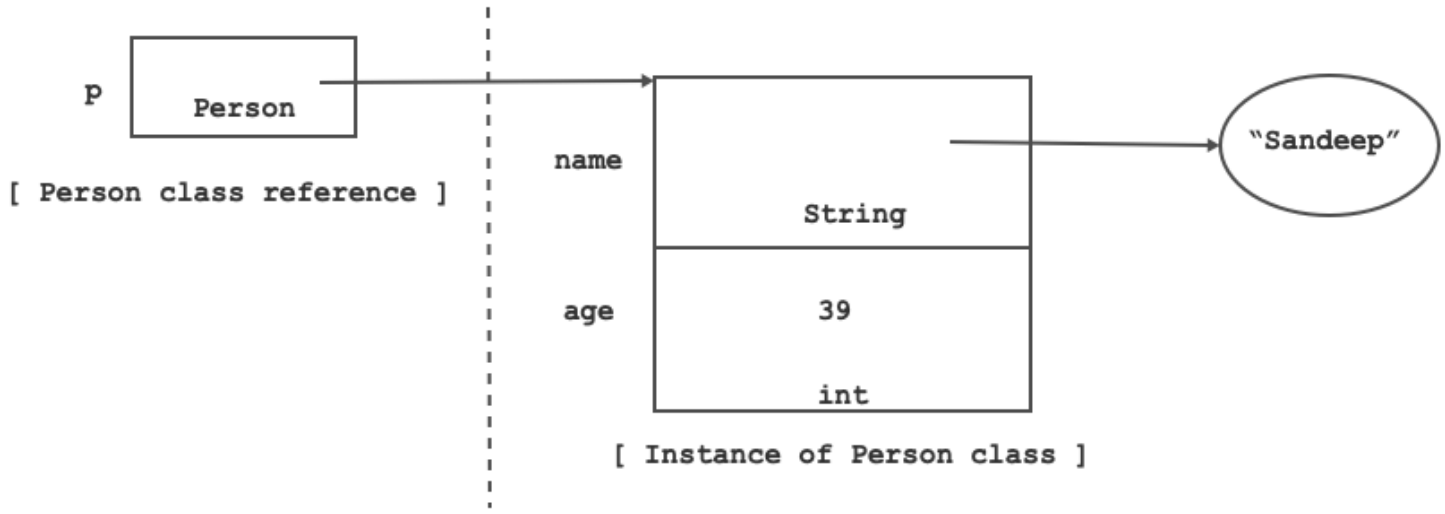
Upcasting and downcasting

```
class Person{
  String name;
  int age;
  public Person() {
    this.name = "";
    this.age = 0;
  }
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
```

```
  }
  public void showRecord( ) {
    System.out.println("Name  :   "+this.name);
    System.out.println("Age   :   "+this.age);
  }
}
class Employee extends Person{
  int empid;
  float salary;
  public Employee() {
    super();
    this.empid = 0;
    this.salary = 0.0f;
  }
  public Employee( String name, int age, int empid, float salary ) {
    super( name, age);
    this.empid = empid;
    this.salary = salary;
  }
  public void displayRecord( ) {
    super.showRecord();
    System.out.println("Empid :   "+this.empid);
    System.out.println("Salary    :   "+this.salary);
  }
}
```

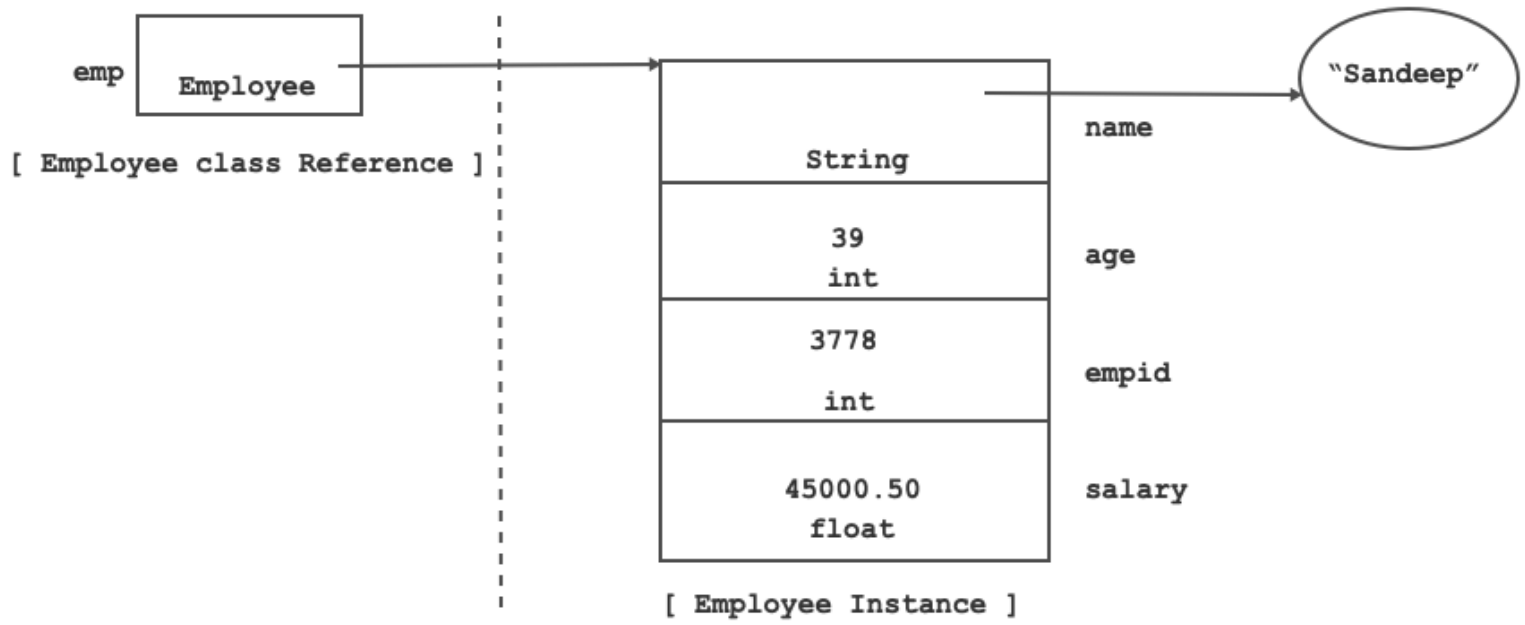
- Since members of sub class do not inherit into super class, using super class instance, we can access members of super class only.

```
  public static void main1(String[] args) {
  Person p = new Person();
  p.name = "Sandeep";
  p.age = 39;
  //p.empid = 3778;   //Not OK
  //p.salary = 45000.50f; //Not OK
  p.showRecord();
  //p.displayRecord();    //Not OK
}
```



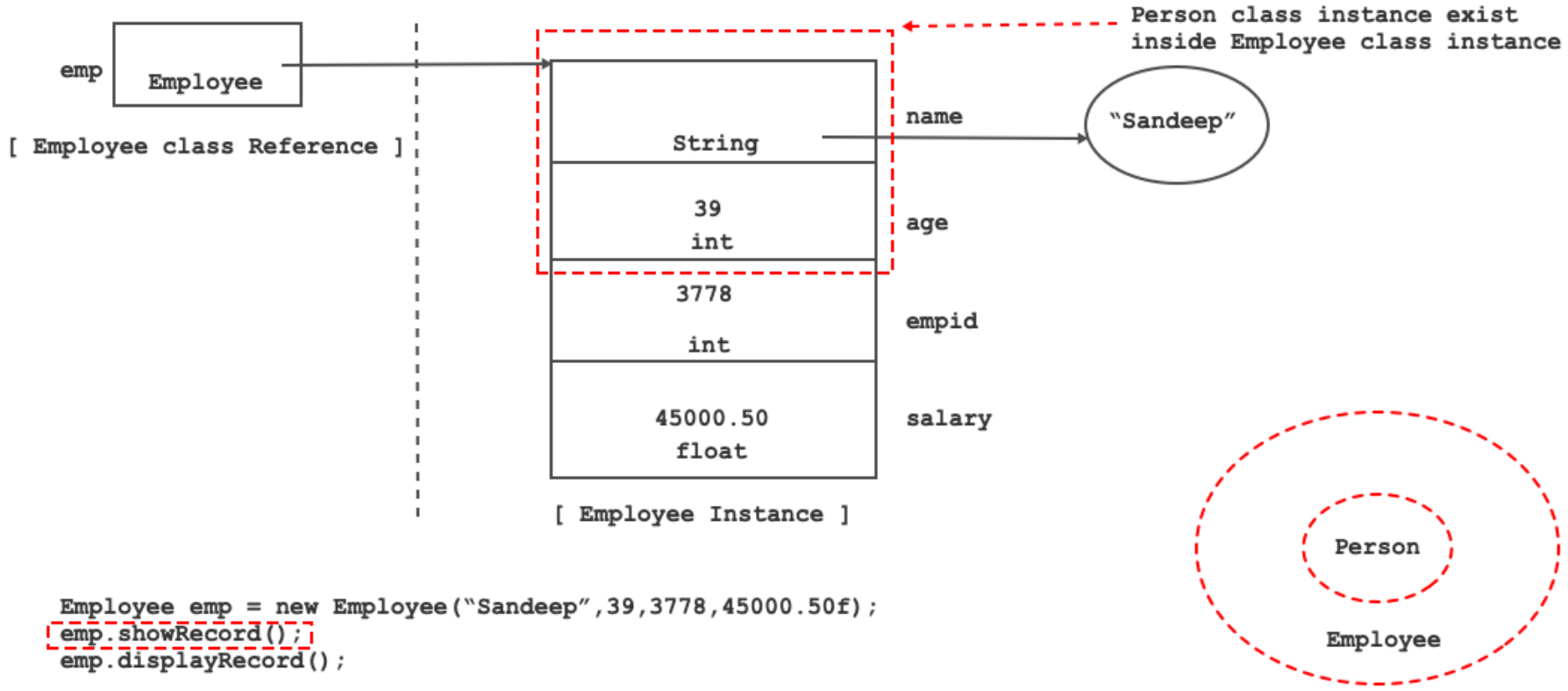
- Since members of super class inherit into sub class, using sub class instance we can access members of super class as well as sub class.

```
  public static void main(String[] args) {
  Employee emp = new Employee();
  emp.name = "Sandeep";
  emp.age = 39;
  emp.empid = 3778;
  emp.salary = 45000.50f;
  emp.showRecord();
  emp.displayRecord();
}
```




```
Employee emp = new Employee("Sandeep",39,3778,45000.50f);
emp.showRecord();
emp.displayRecord();
```

- Since members(fields, methods, nested type) of super class inherit into sub class, we can consider sub class instane as super class instance.
- Since sub class instance can be considered as super class instance, we can use it in place of super class instance.




```
Employee emp = new Employee("Sandeep",39,3778,45000.50f);
emp.showRecord();
emp.displayRecord();
```

```
Person p1 = new Person( );  //OK
Person p2 = p1; //OK
```

```
Employee e1 = new Employee(); //OK
Employee e2  = e1;  //OK
```

```
Employee emp =  new Employee();
Person p = (Person)emp; //OK:  Upcasting
Person p = emp; //OK:  Upcasting
```

```
Person p = new Employee();  //OK: Upcasting
```

- Since members of sub class do not inherit into super class, we can not consider super class instance as sub class instance.
- Since super class instance can not be considered as sub class instance, we can not use it in place of sub class instance.

```
Employee e1 = new Employee(); //OK
Employee e2 = e1; //OK
```

```
```

```
Person p1 = new Person();
Person p2 = p1;
```

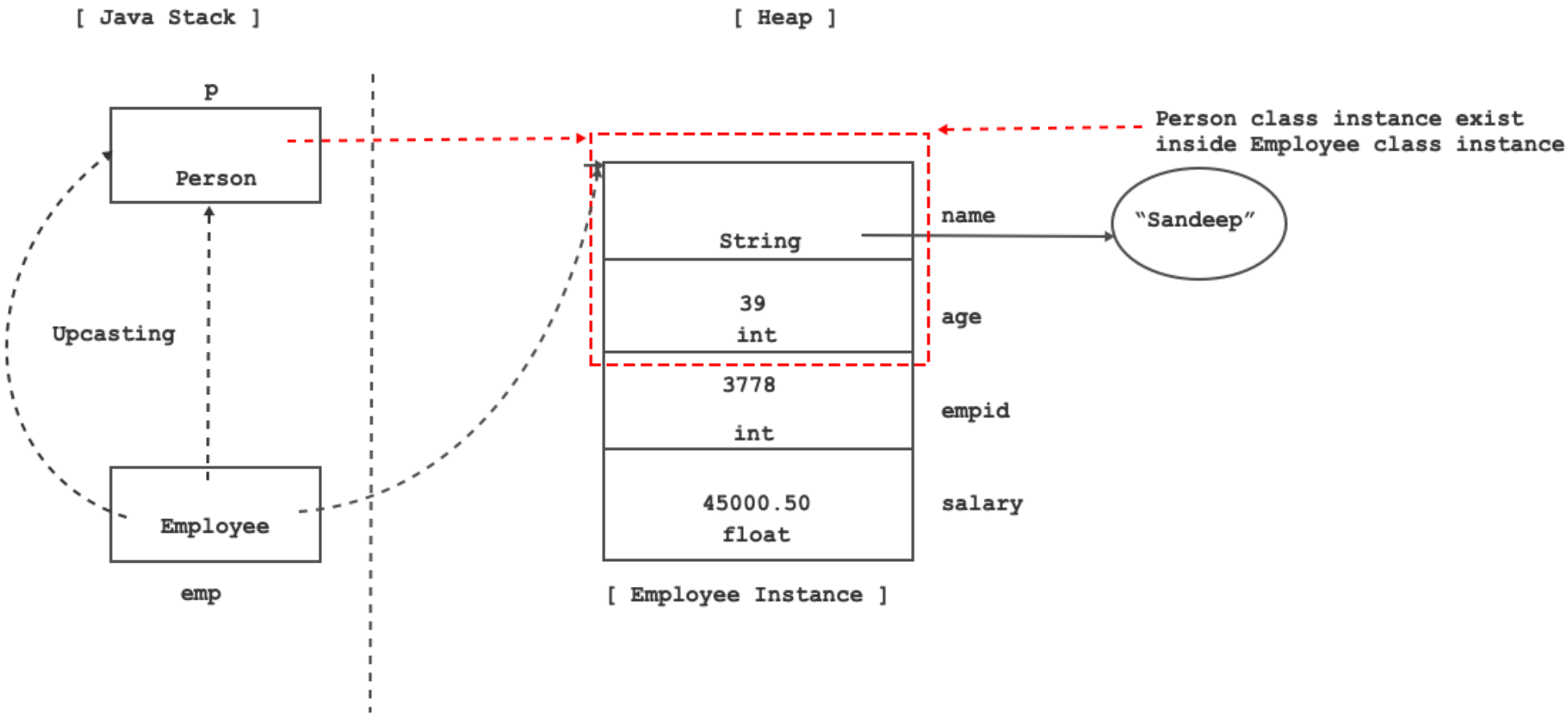
```
Person p = new Person();
Employee emp = (Employee)p; //Not OK
```

```
Employee emp = new Person();  //Not OK
```

Final conclusion:

- Person p = new Person(); //OK
- Person p = new Employee(); //OK
- Employee emp = new Employee(); //OK
- Employee emp = new Person(); //Not OK

Upcasting




```
Employee emp = new Employee("Sandeep", 39, 3778, 45000.50);
//Person p = ( Person )emp;   //Upcasting: OK
Person p = emp;   //Upcasting: OK
```

- Definition
  - Process of converting reference of sub class into reference of super class is called as upcasting.

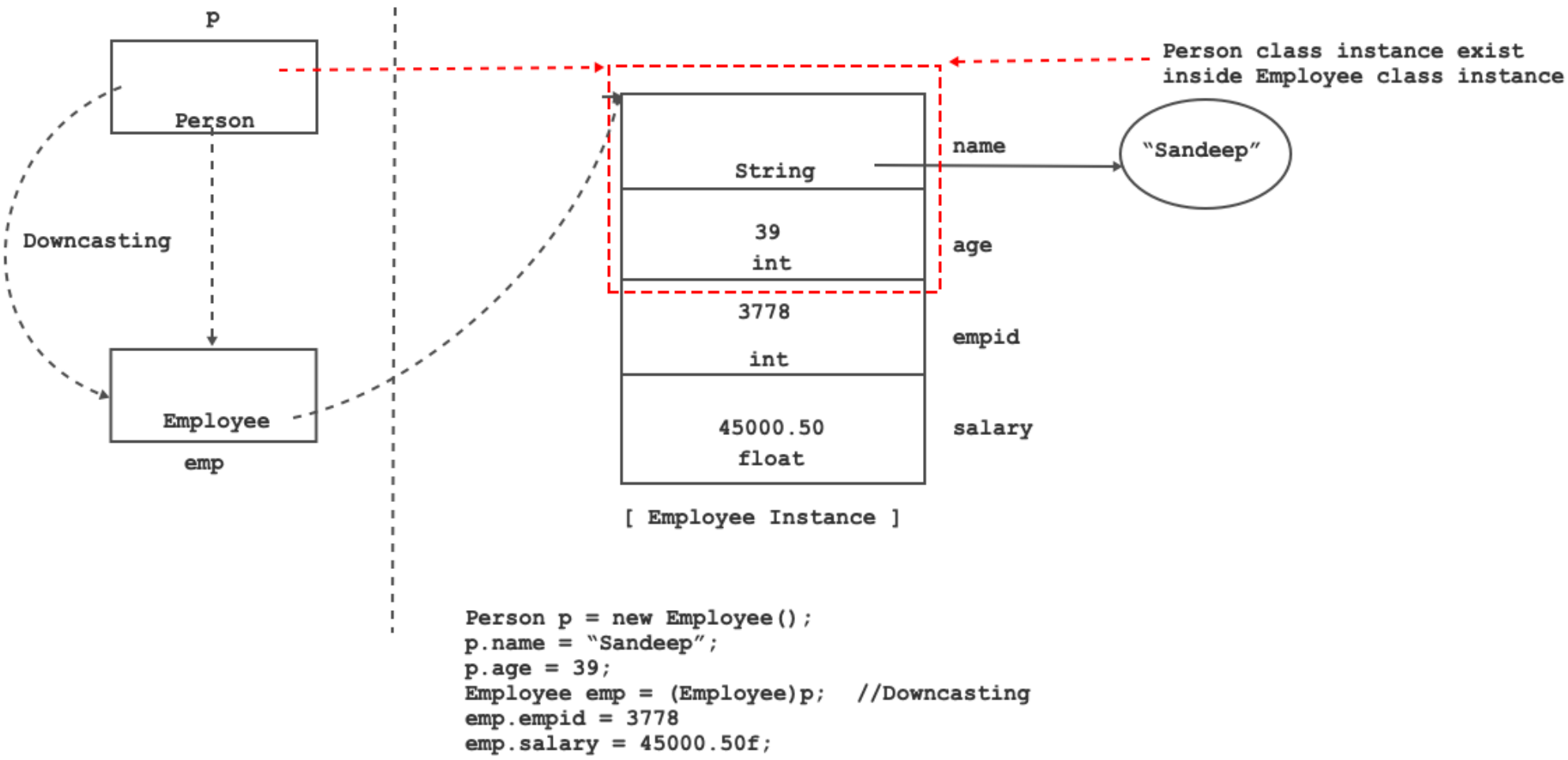
```
Employee emp = new Employee("Sandeep", 39, 3778, 45000.50);
//Person p = ( Person )emp;   //Upcasting: OK
Person p = emp;   //Upcasting: OK
```

  - Super class reference can contain reference of sub class instance. It is also called as upcasting.

```
Person p = new Employee("Sandeep", 39, 3778, 45000.50);
```

- If we want to minimize object/instance dependency in the code then we should use upcasting.

Downcasting



- Process of converting reference of super class into reference of sub class is called as downcasting.

```
Person p = new Employee();  //Upcasting
p.name = "Sandeep";
p.age = 39;
Employee emp = (Employee)p;   //Downcasting
emp.empid = 3778
emp.salary = 45000.50f;
```

- In case of upcasting, explicit type casting is optional but in case of dowcasting explicit typecasting is manadatory.

```
    public static void main(String[] args) {
    Person p = null;
    Employee emp = (Employee) p;    //Downcasting
    System.out.println( p);     //null
    System.out.println(emp);    //null
}
```

```
public static void main(String[] args) {
    Person p = new Employee();  //Upcasting
    Employee emp = (Employee) p;    //Downcasting: OK
}
```

```
public static void main(String[] args) {
    Person p = new Person();
    Employee emp = (Employee) p;    //Downcasting:  ClassCastException
}
```

- If JVM fails to do downcasting then it throws ClassCastException.

Method overriding

- If implementation of super class method is logically incomplete then we should redefine/override method inside sub class.
- Process of redefining, method of super class inside sub class is called as method overriding.
- In case of upcasting, process of calling method of sub class using reference of super class is called as dynamic method dispatch.

```

```

```
class Person{
    public void printRecord( ){
      System.out.println("Person.printRecord");
    }
}
class Employee extends Person{
    public void printRecord( ){
      System.out.println("Person.printRecord");
    }
}
class Program{
    public static void main(String[] args) {
      Person p = new Employee();  //Upcasting
      p.printRecord();  //Dynamic method dispatch
    }
}
```

- Note: In case of upcasting we can not access fields and non overriden methods of sub class. If we want to access it the we should do downcasting.

Rules of method overriding

- Below are the rules of method overriding
  - Access modifier of sub class method should be same or it should be wider.
  - Return type in sub class method should be same or it should be sub type.
  - Method name, number of paramaters and type of parameters in sub class method must be same.
  - Checked exception list in sub class method should be same or it should be sub set.
- Override is annotation declared in java.lang package. It helps developer to override method using above rules. If we make any mistake then it generates metadata for the compiler to generate error.

```
class Person{
    public void printRecord( ){
      System.out.println("Person.printRecord");
    }
}
class Employee extends Person{
    @Override
    public void printRecord( ){
      System.out.println("Person.printRecord");
    }
}
class Program{
    public static void main(String[] args) {
      Person p = new Employee();  //Upcasting
      p.printRecord();  //Dynamic method dispatch
    }
}
```