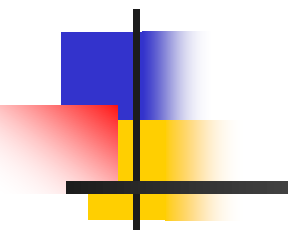


Object Oriented Programming (SEng 2202)



Chapter 03

Inheritance and Polymorphism

Objectives

After studying this chapter, students should be able to learn:

- Basics of Inheritance
- Types of Inheritance
- The *protected* modifier
- Super and final key word
- Polymorphism and its types
- Method Overloading vs. Overriding
- The Object class
- The Abstract class

- Object-oriented programming (OOP) is popular because:
 - It enables *reuse* of previous code saved as *classes*
- All Java classes are arranged in a hierarchy
 - **Object** is the *superclass* of all Java classes
- Inheritance and hierarchical organization capture idea:
 - One thing is a *refinement* or *extension* of another
- **Reusability** - building new classes by utilizing existing classes.

Inheritance - Introduction

- It is always good/“productive” if we are able to reuse something that already exists rather than creating the same all over again.
- This is achieved by creating new classes, reusing the data members and methods of existing classes.
- This mechanism of deriving a new class from existing/old class is called *“inheritance”*.
- The old class is known as **“base”** class, **“super”** class or **“parent”** class”; and the new class is known as **“sub”** class, **“derived”** class, or **“child”** class.

Examples

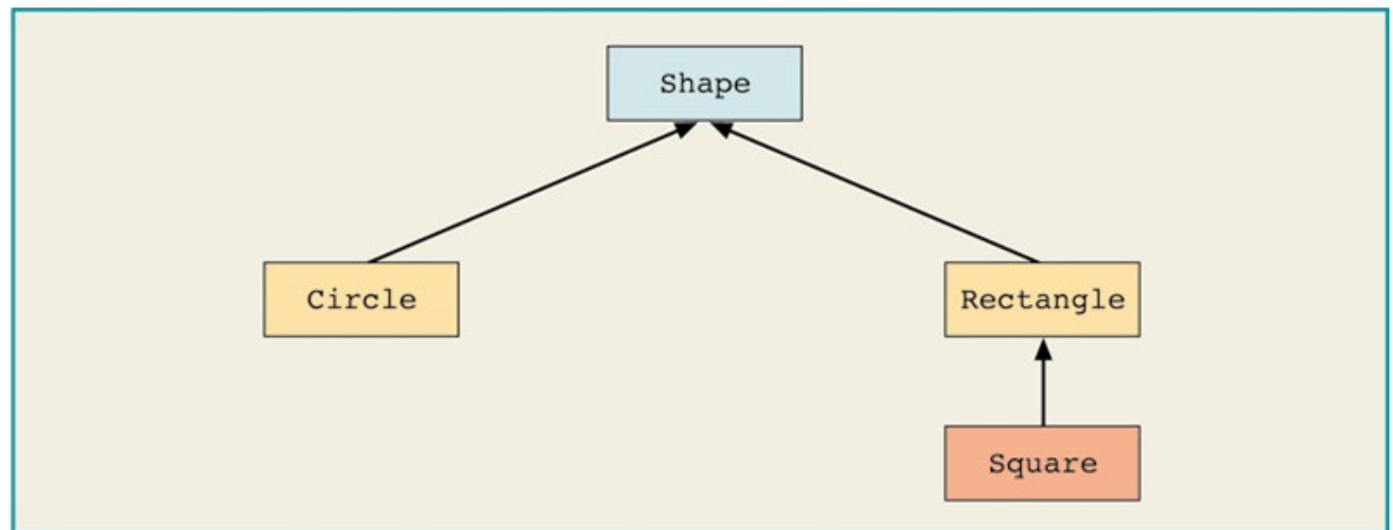
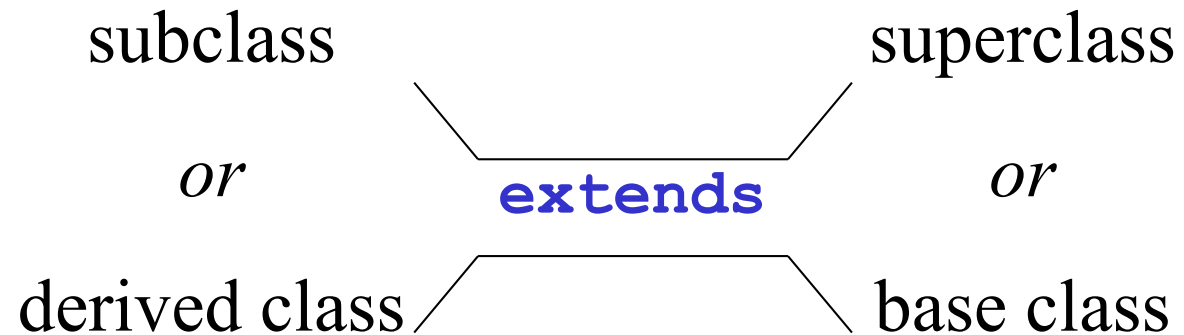
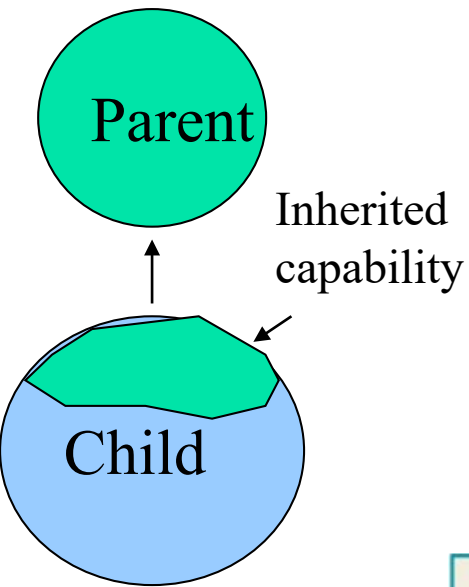


Figure Inheritance hierarchy

Examples – Contd.

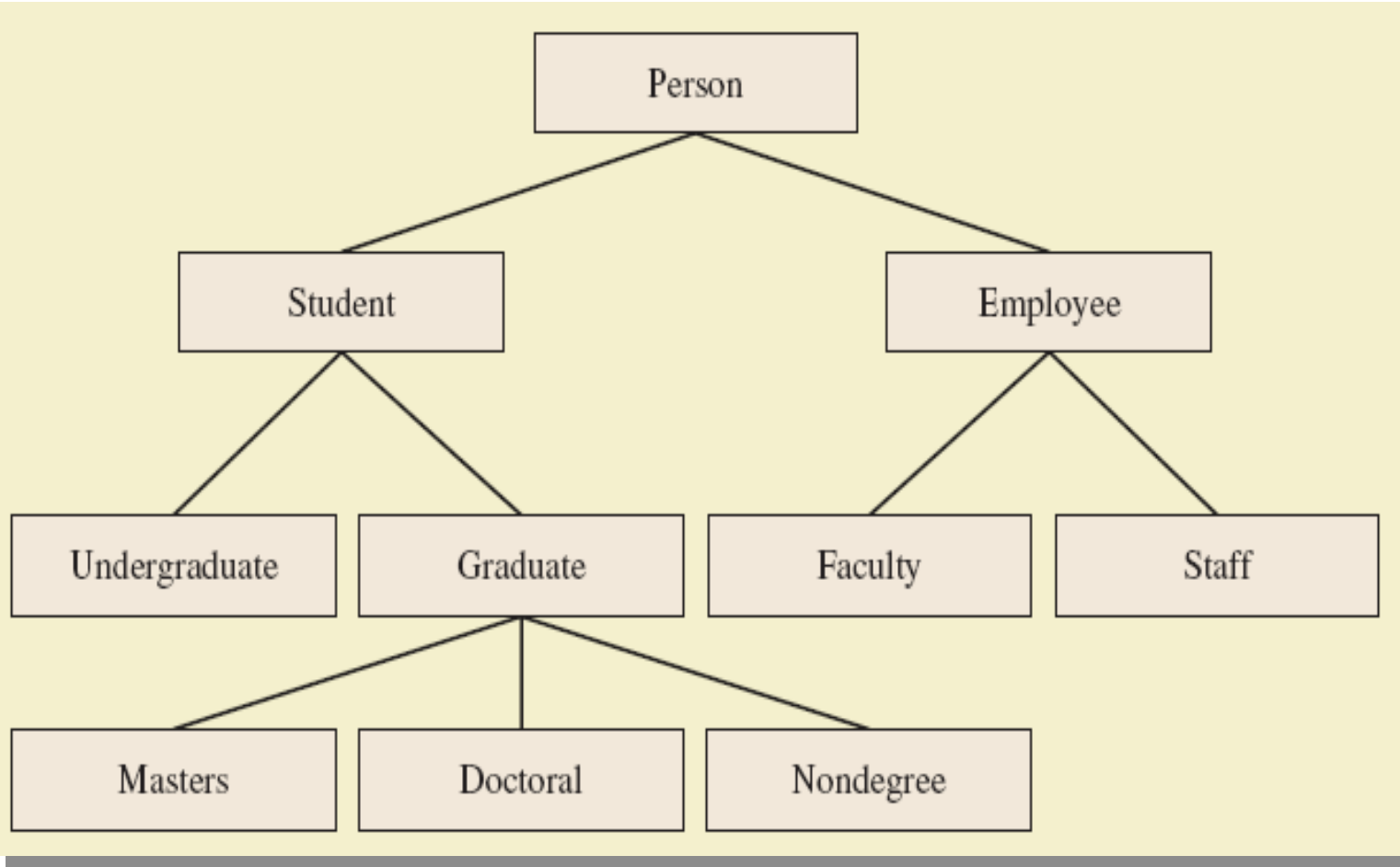
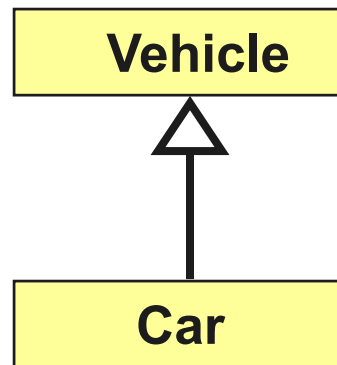


Figure : Class Hierarchies

- Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class.



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent.

Declaring Sub classes

- In Java, we use the reserved word **extends** to establish an inheritance relationship

```
class subclass-name extends superclass-name
{
    .....
    .....
}
```

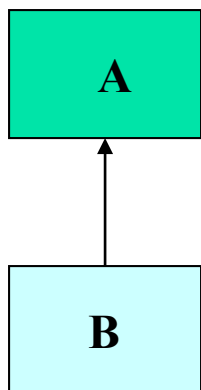
Example:

```
class Car extends Vehicle
{
    // class contents
}
```

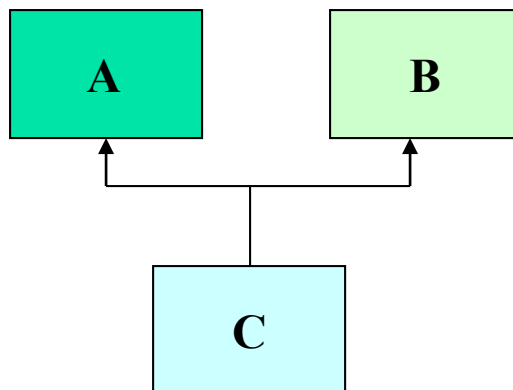
Types of Inheritance

- The different types of inheritance are:
 - Single inheritance (only one super class)
 - Multiple inheritance (several super classes, not supported by Java)
 - Hierarchical inheritance (one super class, many sub classes)
 - Multi-Level inheritance (derived from a derived class)
 - Hybrid inheritance (more than two types)
 - Multi-path inheritance (inheritance of some properties from two sources).

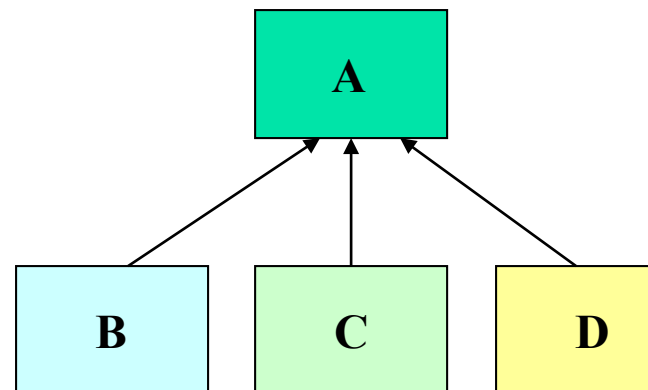
Types of Inheritance



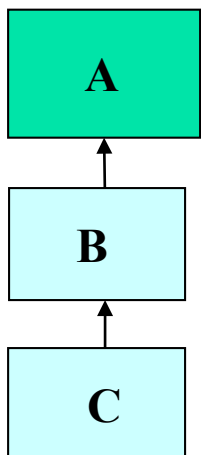
(a) Single Inheritance



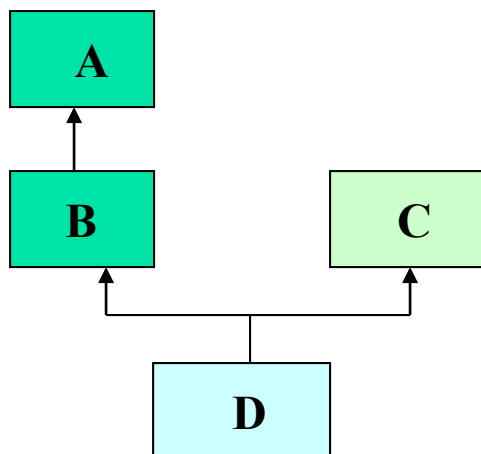
(b) Multiple Inheritance



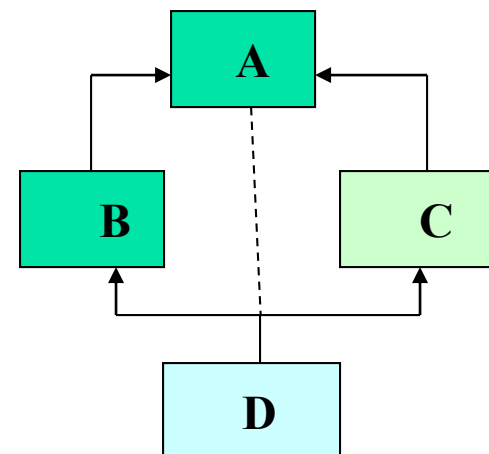
(c) Hierarchical Inheritance



(d) Multi-Level Inheritance



(e) Hybrid Inheritance



(f) Multipath Inheritance

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with **public** visibility are inherited; those with **private** visibility are not
- But **public** variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: **protected**

The protected Modifier

- The `protected` visibility modifier allows a member of a base class to be accessed in the child
 - `protected` visibility provides more encapsulation than `public` does
 - `protected` visibility is not as tightly encapsulated as `private` visibility

Modifier	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<code>none</code>	Y	Y	N	N
<code>private</code>	Y	N	N	N

Single Inheritance - Example

```
public class Person {  
    String name;  
    String permanentAddress;  
    int age;  
  
    void set_PermanentDetails(String name, String  
        permanentAddress, int age) {  
        this.name = name;  
        this.permanentAddress = permanentAddress;  
        this.age = age;  
    }  
  
    void get_PermanentDetails() {  
        System.out.println("Name : " + name);  
        System.out.println("Permanent Address : " +  
            permanentAddress);  
        System.out.println("Age : " + age);  
    }  
}
```

Person class contains general details of a person

Contd.

```
class Employ extends Person {
    int id;
    String companyName;
    String companyAddress;
    Employ(int id, String name, String permanentAddress, int age,
           String companyName, String companyAddress) {
        this.id = id;
        set_PermanentDetails(name, permanentAddress, age);
        this.companyName = companyName;
        this.companyAddress = companyAddress;
    }
    void get_EmployDetails() {
        System.out.println("Employ Id : " + id);
        get_PermanentDetails();
        System.out.println("Company Name : " + companyName);
        System.out.println("Company Address : " + companyAddress);
    }
}
```

Employ contains company details of a person. Reuse the general details of the person in its sub class.

Contd.

```
class InherDemo {  
    public static void main(String args[]) {  
        Employ e1 = new Employ(101,"Abebe", "Addis Ababa, Ethiopia",  
                                29, "ASTU", "Adama, Ethiopia");  
        e1.get_EmployDetails();  
    }  
}
```

Output:

Employ Id : 101
Name : Abebe
Permanent Address : Addis Ababa, Ethiopia
Age :29
Company Name : ASTU
Company Address : Adama, Ethiopia

Multilevel Inheritance - Example

```
public class Student {  
  
    int id;  
    Scanner in = new Scanner(System.in);  
  
    void readdatal() {  
        System.out.print("Enter your Id: ");  
        id = in.nextInt();  
    }  
  
    void printdata1() {  
        System.out.println("Id = " + id);  
    }  
}
```

Contd.

```
class Exam extends Student {

    int m1, m2, m3;
    Scanner in = new Scanner(System.in);

    void readdata2() {
        readdata1();
        System.out.print("Enter 3 subjects marks:");
        m1 = in.nextInt();
        m2 = in.nextInt();
        m3 = in.nextInt();
    }

    void printdata2() {
        printdata1();
        System.out.println("Subject1 marks=" + m1);
        System.out.println("Subject2 marks=" + m2);
        System.out.println("Subject3 marks=" + m3);
    }
}
```

Contd.

```
class Results extends Exam {

    int avg;

    void printdata3() {
        printdata2();
        avg = (m1 + m2 + m3) / 3;
        if (avg < 35) {
            System.out.println("Grade: F");
        } else if (avg >= 35 && avg < 50) {
            System.out.println("Grade: D");
        } else if (avg >= 50 && avg < 60) {
            System.out.println("Grade: C");
        } else if (avg >= 60 && avg < 75) {
            System.out.println("Grade: B");
        } else if (avg >= 75) {
            System.out.println("Grade: A");
        }
    }
}
```

Contd.

```
class MultiInh {  
  
    public static void main(String sad[]) {  
        Results obj = new Results();  
        obj.readdata2();  
        obj.printdata3();  
    }  
}
```

Output:

```
Enter your Id: 101  
Enter 3 subjects marks:75 82 95  
Id = 101  
Subject1 marks=75  
Subject2 marks=82  
Subject3 marks=95  
Grade: A
```

The use of “super” keyword

- ‘super’ is a keyword used to refer to hidden variables of super class from sub class.
 - **super.a=a;**
- It is used to call a constructor of super class from constructor of sub class which should be first statement.
 - **super(a,b);**
- It is used to call a super class method from sub class method to avoid redundancy of code
 - **super.addNumbers(a, b);**

“super” and hiding

- Why is super needed to access super-class members?
- When a sub-class declares the variables or methods with the same names and types as its super-class:

```
class A {  
    int i = 1;  
}  
class B extends A {  
    int i = 2;  
    System.out.println("i is " + i);  
}
```

- The re-declared variables/methods hide those of the super-class.

“super” - Example

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

Contd.

- Although the `i` variable in `B` hides the `i` variable in `A`, `super` allows access to the hidden variable of the super-class:

```
class UseSuper {  
  
    public static void main(String args[]) {  
  
        B subOb = new B(1, 2);  
  
        subOb.show();  
  
    }  
  
}
```

Multiple Inheritance

- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of **interfaces** gives us aspects of multiple inheritance without the overhead.

Using final keyword with Inheritance



- **final** keyword is used declare constants which can not change its value of definition.
- final variables can not change its value.
- final methods can not be Overridden or Overloaded
- final Classes can not be extended or inherited

Preventing Overriding with final

- A method declared final cannot be overridden in any sub-class:

```
class A {  
    final void meth() {  
        System.out.println("This is a final method.");  
    }  
}
```

This class declaration is illegal:

```
class B extends A {  
    void meth() {  
        System.out.println("Illegal!");  
    }  
}
```

- A class declared final cannot be inherited – has no sub-classes.
`final class A { ... }`
- This class declaration is considered illegal:
`class B extends A { ... }`
- Declaring a class final implicitly declares all its methods final.
- It is illegal to declare a class as both abstract and final.

The Benefits of Inheritance

- Code reusability:- Inheritance automates the process of reusing the code of the super classes in the subclasses.
 - With inheritance, an object can inherit its more general properties from its parent object, and that saves the redundancy in programming.
- Code maintenance:- Organizing code into hierarchical classes makes its maintenance and management easier.
- Implementing OOP:- Inheritance helps to implement the basic OOP philosophy to adapt computing to the problem and not the other way around, because entities (objects) in the real world are often organized into a hierarchy.
- Increased Reliability (resulting from reuse and sharing of well-tested code)

- Aggregation in Java is a relationship between two classes that is best described as a "has-a" and "whole/part" relationship.
- It is a more specialized version of the association relationship. The aggregate class contains a reference to another class and is said to have ownership of that class.
- Each class referenced is considered to be *part-of* the aggregate class.
- For example, if you imagine that a Student class that stores information about individual students at a school. Now assume a Subject class that holds the details about a particular subject
- If the Student class is defined to contain a Subject object then it can be said that the Student object *has-a* Subject object.
- The Subject object also makes up *part-of* the Student object after all, there is no student without a subject to study. The Student object, therefore, owns the Subject object.

Contd...

```
■ public class Subject {  
    private String name;  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getName()  
    {  
        return name;  
    }  
  
}  
  
public class Student {  
    private Subject[] studyAreas = new Subject[10];  
    //the rest of the Student class  
}
```

What is the relation ship b/n Address and Employee class?

Polymorphism

- Polymorphism came from the two Greek words ‘poly’ means many and morph means forms i.e. *many forms*
- If the same method has ability to take more than one form to perform several tasks then it is called *polymorphism*.
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time.
- It is of two types: *Static Polymorphism and Dynamic Polymorphism*.
- **Static Polymorphism:**
 - The polymorphism exhibited at compile time is called Static polymorphism.
 - Here the compiler knows which method is called at the compilation. This is also called compile time polymorphism or static binding.
 - Achieving method overloading & method overriding using private, static and final methods is an example of Static Polymorphism.

Static Polymorphism

```
class Animal
{static void move ()
    { System.out.println ("Animals can move");
    }
}
class Dog extends Animal
{
    static void move ()
    { System.out.println ("Dogs can walk and run");
    }
}
public class StaticPoly
{ public static void main(String args[])
    { Animal.move ();
      Dog.move ();
    }
}
```

Polymorphism – Contd.

■ **Dynamic Polymorphism:**

- The polymorphism exhibited at run time is called dynamic polymorphism.
- In this dynamic polymorphism a method call is linked with method body at the time of execution by JVM.
- Java compiler does not know which method is called at the time of compilation. This is also known as dynamic binding or run time polymorphism.
- Method overloading and method overriding are examples of Dynamic Polymorphism in Java.
- Provided that the methods are instance method.

Dynamic Polymorphism - overloading

```
class Sample
{ void add(int a,int b)
  {
    System.out.println ("sum of two="+ (a+b));
  }
  void add(int a,int b,int c)
  {
    System.out.println ("sum of three="+ (a+b+c));
  }
}

class OverLoad
{ public static void main(String[] args)
  { Sample s=new Sample ( );
    s.add (20, 25);
    s.add (20, 25, 30);
  }
}
```

Method Overriding

- Writing two or more methods in super & sub classes with same name and same signatures is called method overriding.
- In method overriding JVM executes a method depending on the type of the object.

```
class Animal
{
void move()
{
    System.out.println ("Animals can move");
}
}
class Dog extends Animal
{ void move()
{
    System.out.println ("Dogs can walk and run");
}
}
```

Contd.

```
public class OverRide
{ public static void main(String args[])
{
    Animal a = new Animal (); // Animal reference and object
    Animal b = new Dog (); // Animal reference but Dog object
    a.move(); // runs the method in Animal class
    b.move(); //Runs the method in Dog class
}
}
```

Overloading Vs Overriding

- Overloading deals with multiple methods in the same class with the same name but **different signatures**
- Overloading lets you define a similar operation in different ways for **different data**

- Overriding deals with two methods, one in a parent class and one in a child class, with the **same** and **signature**
- Overriding lets you define a similar operation in different ways for different **object types**

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies.
- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we define the `toString` method, we are actually overriding an inherited definition

- The `toString` method in the `Object` class is defined to **return a string** that contains the **name** of the object's **class** along with some other information
- The `equals` method of the `Object` class **returns true** if two references are **aliases**
- We can override `equals` in any class to define equality in some more appropriate way
- As we've seen, the `String` class defines the `equals` method to return **true** if two `String` objects contain the **same characters**
- The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version

The Abstract Class

- A method with method body is called **concrete method**. In general any class will have all concrete methods.
- A method without method body is called **abstract method**. A class that contains **abstract method** is called **abstract class**. i.e. An abstract class is a class with **zero or more abstract methods**
- It is possible to implement the abstract methods differently in the subclasses of an abstract class.
- These different implementations will help the programmer to perform different tasks depending on the need of the sub classes.
- Moreover, the common members of the abstract class are also shared by the sub classes.

- The abstract methods and abstract class should be declared using the keyword **abstract**.
- We cannot create objects to abstract class because it is having incomplete code.
- Whenever an abstract class is created, subclass should be created to it and the abstract methods should be implemented in the subclasses, then we can create objects to the subclasses.
- Abstract class reference can be used to refer to the objects of its sub classes.
- Abstract class references cannot refer to the individual methods of sub classes.
- A class cannot be both '**abstract**' & '**final**' simultaneously
e.g.: **final abstract class A** **// invalid**

Example – Abstract class

```
abstract class Figure
{
    double dim1;
    double dim2;
    Figure (double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    abstract double area () ;// area is an abstract method
}
class Rectangle extends Figure
{
    Rectangle (double a, double b)
    {
        super(a,b) ;
    }
    double area () // override area for rectangle
    {
        System.out.println ("Inside Area of Rectangle.");
        return dim1 * dim2;
    }
}
```

Contd.

```
class Triangle extends Figure
{ Triangle (double a, double b)
  { super (a, b);
  }
  double area() // override area for right triangle
  { System.out.println ("Inside Area of Triangle.");
    return dim1 * dim2 / 2;
  }
}

class AbstractAreas
{ public static void main(String args[])
  { // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    System.out.println("Area is " + r.area());
    System.out.println("Area is " + t.area());
  }
}
```

Review Questions

1. What is inheritance? What is a superclass? What is a subclass?
2. Which class is at the top of the class hierarchy in Java?
3. What are the constructor issues surrounding inheritance?
4. What is method overriding? What is polymorphism? How are they related?
5. What is a final method? What is a final class? What is final variable?