

CS2833: Modular Software Development

OOP Concepts

Department of Computer Science and Engineering
University of Moratuwa

Object Oriented Programming Concepts

1. Encapsulation

2. Inheritance

3. Abstraction

4. Polymorphism

1. Encapsulation

Encapsulation

- Encapsulation hides the implementation details
 - > **Data shielding**



Content is encapsulated



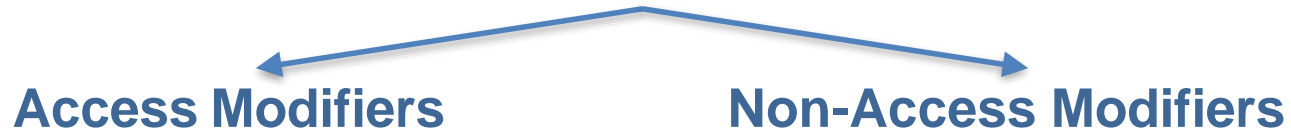
Content is Open

Encapsulation

Encapsulation Benefits

- ~~Ensures~~ that structural changes remain local:
 - Changing the class internals does not affect any code outside of the class
- Encapsulation allows adding some logic when accessing client's data
 - E.g. validation on modifying a property value
- Hiding implementation details reduces complexity -> easier maintenance
- Fields can be made read-only or write-only

Modifiers



public

protected

default

private

final

abstract

static

strictfp

synchronize

native

transit

Volatile

Access Modifiers

- Specifies the accessibility or scope of a field, method, constructor, or class.
- We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

Access Modifiers	Class	Package	Subclass		Global
			Same Package	Different Package	
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Example 1

```
class A{  
    private void display() {  
        System.out.println("Inside display method");  
    }  
}
```

```
class B{  
    public static void main(String args[]) {  
        A obj = new A();  
        //trying to access private method of  
another class  
        obj.display();  
    }  
}
```

```
error: display() has private access in A  
        obj.display();
```


Example 2

```
package p1;
```

```
//Class A is having Default access modifier
```

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

```
//Java program to illustrate error while using class from  
different package with default modifier
```

```
package p2;  
import p1.*;
```

```
//This class is having default access modifier
```

```
class B{  
    public static void main(String args[]){  
        //accessing class Geek from package p1  
        A obj = new A();  
        obj.display();  
    }  
}
```

Compile time error

Example 3

```
package p1;

public class A{
    protected void display(){
        System.out.println("Inside display method");
    }
}
```

```
package p2;
import p1.*; //importing all classes in package p1

//Class B is subclass of A
class B extends A{
    public static void main(String args[])
    {
        B obj = new B();
        obj.display();
    }
}
```

Inside display method

Example 4

```
package p1;
public class A{
    public void display(){
        System.out.println("Inside display method");
    }
}
```

```
package p2;
import p1.*;
class B{
    public static void main(String args[]){
        A obj = new A;
        obj.display();
    }
}
```

Inside display method

Encapsulation - Example

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

```
Student.java
//A fully encapsulated class.
package A;

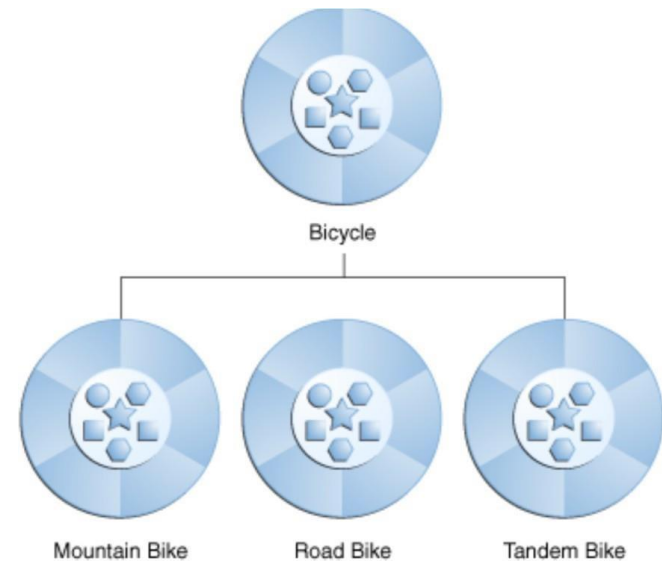
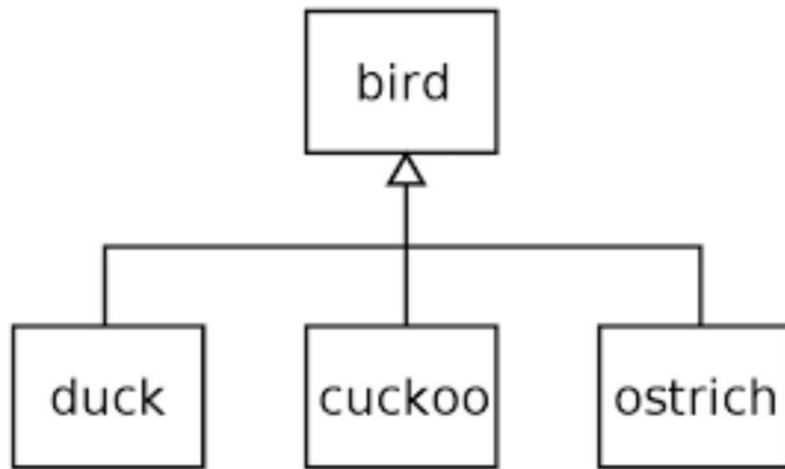
public class Student{
    //private data member
    private String name;
    //getter method for name
    public String getName(){
        return name;
    }
    //setter method for name
    public void setName(String name){
        this.name=name
    }
}
```

```
Test.java
//A Java class to test the encapsulated class.
package A;
class Test{
    public static void main(String[] args){
        //creating instance of the encapsulated class
        Student s=new Student();
        //setting value in the name member
        s.setName("John Doe");
        //getting value of the name member
        System.out.println(s.getName());
    }
}
```

Output: John Doe

2. Inheritance

Inheritance



Inheritance

- The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called **inheritance**.
- The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from another class.
- **Child Class:**
 - The class that extends the features of another class is known as child class, sub class or derived class.
- **Parent Class:**
 - The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Inheritance - Example

```
class Employee{
    float salary=40000;

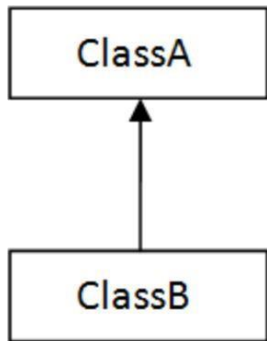
    int calc(int OT){
        return salary + OT*500;
    }
}

class Programmer extends Employee{
    int bonus=10000;

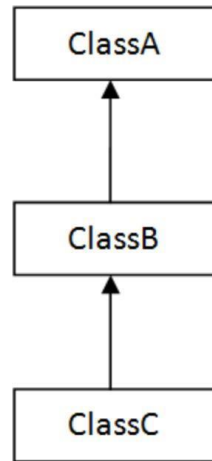
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary:"+p.calc(5));
        System.out.println("Bonus of Programmer:"+p.bonus);

    }
}
```

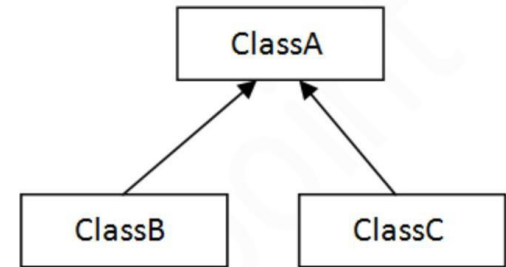

Types of Inheritance



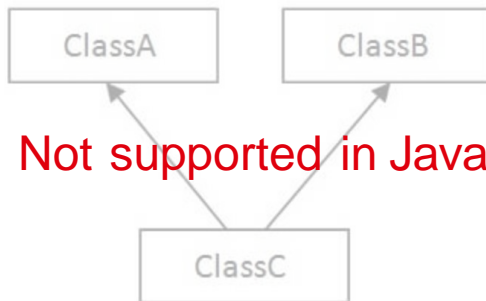
1) Single



2) Multilevel

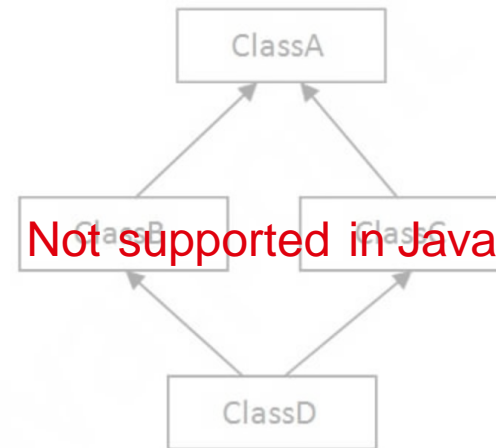


3) Hierarchical



Not supported in Java

4) Multiple



Not supported in Java

5) Hybrid

Subclass

- A subclass inherits all of the default (same package only), protected and public members of its parent
- **Fields (attributes)**
 - The inherited fields can be used directly, just like any other fields.
 - Can declare a field in the subclass with the same name as the one in the super-class, thus **hiding** it (not recommended).
 - Can declare new fields in the subclass that are not in the super class.

Subclass

- **Methods**
 - Inherited methods can be used directly as they are
 - Can have a new instance method in the subclass that has the same signature as the one in the super class, thus **overriding** it.
 - Can have a new static method in the subclass that has the same signature as the one in the super class, thus **hiding** it.
 - Can declare new methods that are not in the super class.
- Can write a constructor that invokes the constructor of the super class
 - Implicitly
 - By using the keyword super

```

Public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}

```

```

Public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }
}

```

```

public static void main(String[] args) {
    Animal myCat = new Cat();
    Cat myCat2 = new Cat();
    myCat.testClassMethod();
    myCat2.testClassMethod();
    myCat.testInstanceMethod();
    myCat2.testInstanceMethod();
}
}

```

- **Overriding vs hiding**

Output:

The static method in Animal
 The static method in Cat
 The instance method in Cat
 The instance method in Cat

3. Abstraction

Abstraction

- Abstraction – we know what an object does and how to use it, but not how it does what it does
 - Ignoring irrelevant features, properties, or functions and emphasizing the relevant ones to a given project
- **Abstract Class**
 - An abstract class is a class that is declared abstract
 - Abstract classes can not be instantiated, but can be sub-classed
 - Abstract classes are used only as super classes in inheritance hierarchies
 - Abstract class are used to share a common design

Abstract Methods

- If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes
- Can be included ONLY in an abstract class
- Any child class must either override the abstract method or declare itself abstract

Abstract Class

- Parent class contains the common functionality of a collection of child classes
- Impose **what** should be there but not **how** it should be done

```
public abstract class Animal {  
    //declare fields  
    //declare non - abstract methods  
    public abstract void makeNoise();  
}
```


4. Polymorphism

Polymorphism

- The ability of an **object** to take on many forms
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class
- In Java polymorphism is divided into two types:
 1. Compile time Polymorphism
 2. Runtime Polymorphism

Compile time Polymorphism

- This type of polymorphism is achieved by function **overloading**
 - Same method name
 - Different Parameters

```
class Dog {  
    public void Bark() {  
        System.out.println("Woof");  
    }  
  
    public void Bark(String sound) {  
        System.out.println(sound);  
    }  
}
```

Run time Polymorphism

- This type of polymorphism is achieved by function **overriding**
 - Same method name
 - Same Parameters

```
class Dog {  
    public void Bark() {  
        System.out.println("Woof");  
    }  
}
```

```
class Hound extends Dog {  
    public void Sniff() {  
        System.out.println("Sniff");  
    }  
  
    public void Bark() {  
        System.out.println("Woof");  
    }  
}
```

Questions?

Thank You