Design Patterns Course

Agenda

Introduction

Benefits of design patters

Three main groups of design patterns:

- Creational
- Structural
- Behavioral

OOP Principles

- Abstract
- Encapsulation
- Inheritance
- Polymorphism

Design Patterns - Introduction

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.

A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design Patterns - Introduction

Design patterns can speed up the development process by providing tested, proven development paradigms.

Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

In addition, patterns are also allows developers to communicate using well-known, well understood names for software interactions.

Design Patterns – Three Main Groups

Design Patterns are categorized in three main groups:

- Creational
- Structural
- Behavioral

Creational design patterns

This design patterns is all about class instantiation.

Examples:

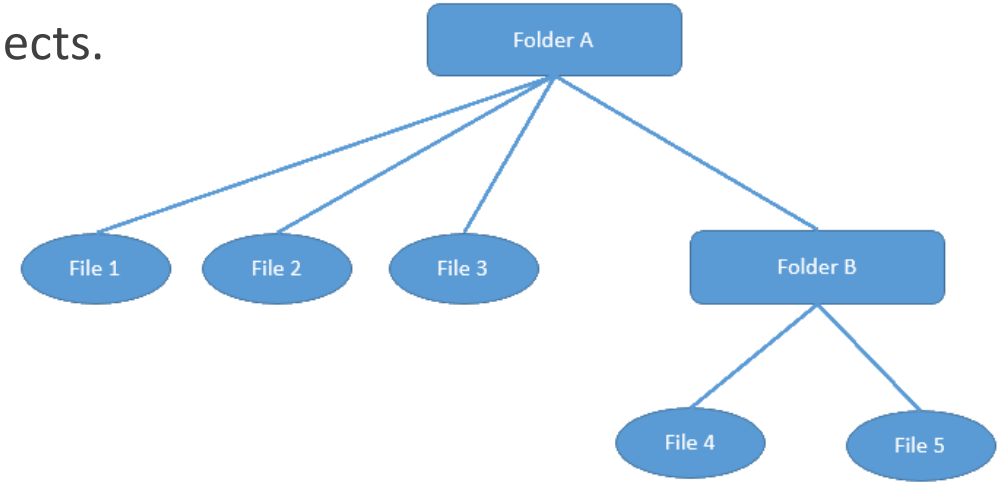
* Singleton - Ensure a class has only one instance, and provide a global point of access to it

Structural Design Patterns:

These design patterns are related to class and object composition. Structural object-patterns define ways to compose objects to obtain new functionality.

For example:

* Composite - A tree structure of simple and composite objects.
  - For example File system structure



Behavioral design pattern:

This design pattern is about class's object's communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

For example:

Observer - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The "View" part of Model-View-Controller. It could be also considered as a way of notifying change to a number of classes.

MVC – Model View Controller

Command

Visitor

OOP Principles

The object-oriented paradigm supports four major principles:

* abstraction
* encapsulation
* inheritance
* polymorphism

Abstraction

Real word example:

when you drive your car you do not have to be concerned with the exact internal working of your car.

What you are concerned with is interacting with your car via its interfaces like:

- steering wheel
- brake pedal
- accelerator pedal

The knowledge you have of your car is abstract.

Abstraction – cont’

Abstraction is to hide information that is not relevant or rather show only relevant information.

Abstraction can be seen in two ways:

- Data abstraction
- Control abstraction

Abstraction – Data abstraction

Data abstraction is the way to create complex data types from multiple smaller data types.

Example:

An Employee class can be a complex object of having various small associations.

```
public class Employee
{
    private Department department;
    private Address address;
    private Education education;
    //So on...
}
```

If we want to fetch information of a employee, we ask from Employee object.

Abstract – Control abstract

Control abstraction is achieved by hiding the sequence of actions for a complex task – inside a simple method call.

The logic to perform the task can be hidden from client and could be changed in future without impacting the client code.

```
public class EmployeeManager
{
    public Address getPrefferedAddress(Employee e)
    {
        //Get all addresses from database
        //Apply logic to determine which address is preferred
        //Return address
    }
}
```

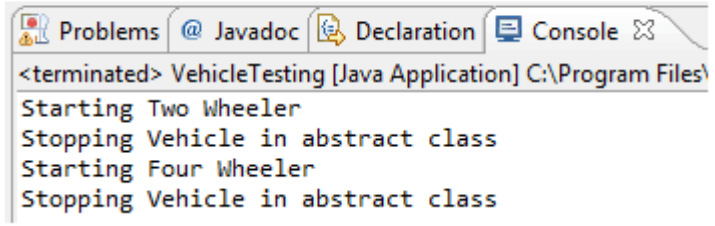
Tomorrow if we want to change the logic inside *getPrefferedAddress*() method, the client will be NOT affected.

Abstract - example

```java
package oopsconcept;
public abstract class VehicleAbstract {
    public abstract void start();
    public void stop(){
        System.out.println("Stopping Vehicle in abstract class");
    }
}
class TwoWheeler extends VehicleAbstract{
    @Override
    public void start() {
        System.out.println("Starting Two Wheeler");
w   }
}
class FourWheeler extends VehicleAbstract{
    @Override
    public void start() {
        System.out.println("Starting Four Wheeler");
    }
}
```

```java
package oopsconcept;
public class VehicleTesting {
    public static void main(String[] args) {
        VehicleAbstract my2Wheeler = new TwoWheeler();
        VehicleAbstract my4Wheeler = new FourWheeler();
        my2Wheeler.start();
        my2Wheeler.stop();
        my4Wheeler.start();
        my4Wheeler.stop();
    }
}
```

Output :



Encapsulation

The whole idea behind encapsulation is to hide the implementation details from users.

If a data member is *private* it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

Data can only be accessed by *public* methods thus making the private fields and their implementation hidden for outside classes. That's why encapsulation is known as **data hiding**.

Encapsulation - example

```
public class EncapsulationDemo{
    private int ssn;
    private String empName;
    private int empAge;

    //Getter and Setter methods
    public int getEmpSSN(){
        return ssn;
    }

    public String getEmpName(){
        return empName;
    }

    public int getEmpAge(){
        return empAge;
    }

    public void setEmpAge(int newValue){
        empAge = newValue;
    }

    public void setEmpName(String newValue){
        empName = newValue;
    }

    public void setEmpSSN(int newValue){
        ssn = newValue;
    }
}
public class EncapsTest{
    public static void main(String args[]){
        EncapsulationDemo obj = new EncapsulationDemo();
        obj.setEmpName("Mario");
        obj.setEmpAge(32);
        obj.setEmpSSN(112233);
        System.out.println("Employee Name: " + obj.getEmpName());
        System.out.println("Employee SSN: " + obj.getEmpSSN());
        System.out.println("Employee Age: " + obj.getEmpAge());
    }
}
```

Output:

```
Employee Name: Mario
Employee SSN: 112233
Employee Age: 32
```

Encapsulation – cont’

All three data members (or data fields) are *private* which cannot be accessed directly.

These fields can be accessed via *public* methods only.

Fields *empName*, *ssn* and *empAge* are made hidden data fields using encapsulation technique of OOPs.

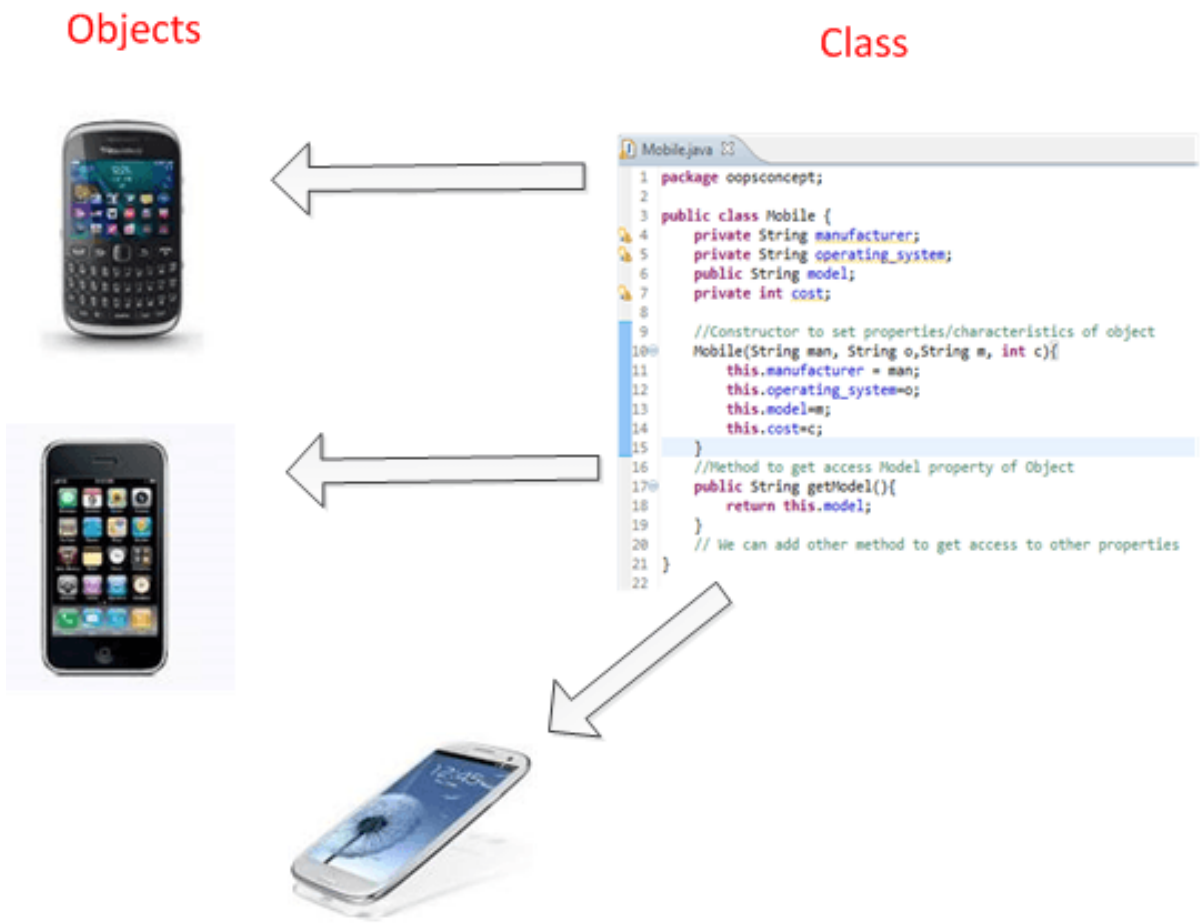
Inheritance

The ability to create classes that share the attributes and methods of existing classes, but with more specific features.

Inheritance is mainly used for code reusability. So you are making use of already written the classes and further extending on that.

Deriving a new class from existing class, it’s called as Inheritance.

Inheritance – cont’



Inheritance – cont’

view plain copy to clipboard print ?

```
01. package oopsconcept;
02. public class Mobile {
03.     private String manufacturer;
04.     private String operating_system;
05.     public String model;
06.     private int cost;
07.     //Constructor to set properties/characteristics of object
08.     Mobile(String man, String o,String m, int c){
09.         this.manufacturer = man;
10.         this.operating_system=o;
11.         this.model=m;
12.         this.cost=c;
13.     }
14.     //Method to get access Model property of Object
15.     public String getModel(){
16.         return this.model;
17.     }
18.     // We can add other method to get access to other properties
19. }
```

```
01. package oopsconcept;
02. public class Blackberry extends Mobile{
03.     //Constructor to set properties/characteristics of object
04.     Blackberry(String man, String o,String m, int c){
05.                     super(man, o, m, c);
06.                 }
07.     public String getModel(){
08.         return "This is Blackberry-"+ model;
09.     }
10. }
```

```
01. package oopsconcept;
02. public class Android extends Mobile{
03.         //Constructor to set properties/characteristics of object
04.         Android(String man, String o,String m, int c){
05.                 super(man, o, m, c);
06.             }
07.         //Method to get access Model property of Object
08.         public String getModel(){
09.             return "This is Android Mobile- " + model;
10.         }
11. }
```

Polymorphism

Polymorphism definition is that Poly means many and morphos means forms.

It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context.

For example, in English, the verb “run” means different things if you use it with “a footrace,” a “business,” or “a computer.” You understand the meaning of “run” based on the other words used with it.

There are two types of Polymorphism available in Java:

- Static Polymorphism (compile time polymorphism/ Method overloading)
- Dynamic Polymorphism (run time polymorphism/ Method Overriding)

Static Polymorphism

Also known as compile time polymorphism/ Method overloading

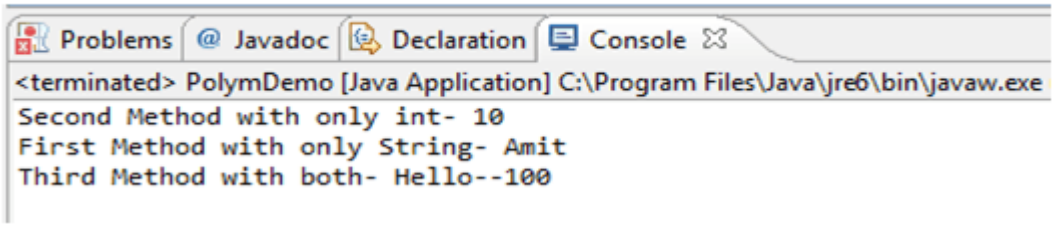
The ability to execute different method implementations by altering the argument used with the method name (method overloading).

In below program, we have three print methods each with different arguments. When you properly overload a method, you can call it providing different argument lists, and the appropriate version of the method executes.

Polymorphism - method overloading

```
01. package oopsconcept;
02. class Overloadsample {
03.     public void print(String s){
04.         System.out.println("First Method with only String- "+ s);
05.     }
06.     public void print (int i){
07.         System.out.println("Second Method with only int- "+ i);
08.     }
09.     public void print (String s, int i){
10.         System.out.println("Third Method with both- "+ s + "--" + i);
11.     }
12. }
13. public class PolymDemo {
14.     public static void main(String[] args) {
15.         Overloadsample obj = new Overloadsample();
16.         obj.print(10);
17.         obj.print("Amit");
18.         obj.print("Hello", 100);
19.     }
20. }
```

Output :



Dynamic Poymorphism

Dynamic Polymorphism, also known as run time polymorphism/ Method Overriding.

Method overriding is a feature which you get when you implement inheritance in your program.

A simple example can be from real world e.g. *Animal*.

An application can have *Animal* class, and its specialized sub classes like *Cat* and *Dog*. These subclasses will override the default behavior provided by Animal class + some of its own specific behavior.

Dynamic Polymorphism - Example

```
public class Animal {
    public void makeNoise()
    {
        System.out.println("Some sound");
    }
}

class Dog extends Animal{
    public void makeNoise()
    {
        System.out.println("Bark");
    }
}

class Cat extends Animal{
    public void makeNoise()
    {
        System.out.println("Meawoo");
    }
}
```

```
public class Demo
{
    public static void main(String[] args) {
        Animal a1 = new Cat();
        a1.makeNoise(); //Prints Meowoo

        Animal a2 = new Dog();
        a2.makeNoise(); //Prints Bark
    }
}
```

Now which makeNoise() method will be called, depends on type of actual instance created on runtime.

Dynamic Polymorphism – Example 2

view plain copy to clipboard print ?

```
01. package oopsconcept;
02. public class Mobile {
03.     private String manufacturer;
04.     private String operating_system;
05.     public String model;
06.     private int cost;
07.     //Constructor to set properties/characteristics of object
08.     Mobile(String man, String o,String m, int c){
09.         this.manufacturer = man;
10.         this.operating_system=o;
11.         this.model=m;
12.         this.cost=c;
13.     }
14.     //Method to get access Model property of Object
15.     public String getModel(){
16.         return this.model;
17.     }
18.     // We can add other method to get access to other properties
19. }
```

```
01. package oopsconcept;
02. public class Android extends Mobile{
03.         //Constructor to set properties/characteristics of object
04.         Android(String man, String o,String m, int c){
05.                 super(man, o, m, c);
06.             }
07.         //Method to get access Model property of Object
08.         public String getModel(){
09.             return "This is Android Mobile- " + model;
10.         }
11. }
```

```
01. package oopsconcept;
02. public class Blackberry extends Mobile{
03.     //Constructor to set properties/characteristics of object
04.     Blackberry(String man, String o,String m, int c){
05.                     super(man, o, m, c);
06.                 }
07.     public String getModel(){
08.         return "This is Blackberry-"+ model;
09.     }
10. }
```

Dynamic Polymorphism – Example 2

```
01. package oopsconcept;
02. public class OverridingDemo {
03.     public static void main(String[] args) {
04.         //Creating Object of SuperClass and calling getModel Method
05.         Mobile m = new Mobile("Nokia", "Win8", "Lumia",10000);
06.         System.out.println(m.getModel());
07.         //Creating Object of Sublcass and calling getModel Method
08.         Android a = new Android("Samsung", "Android", "Grand",30000);
09.         System.out.println(a.getModel());
10.         //Creating Object of Sublcass and calling getModel Method
11.         Blackberry b = new Blackberry("BlackB", "RIM", "Curve",20000);
12.         System.out.println(b.getModel());
13.     }
14. }
```

Output:

```
<terminated> OverridingDemo [Java Application] C:\
Lumia
This is Android Mobile- Grand
This is Blackberry-Curve
```

Class Exercise

Copy the examples of OOP principles (abstraction, encapsulation, inheritance, polymorphism) to your local IDE (Eclipse) and try to run it.

Debug every program with step-by-step breakpoints and observe the object creation.