# Design Patterns

**Vũ Thị Hồng Nhạn**

(vthnhan@vnu.edu.vn)

Dept. Software Engineering, FIT, UET,

Vietnam National Univ., Hanoi

# Content

- Benefits of design patterns
- 3 Categories of design patterns
  - Creational design patterns
  - Structural design patterns
  - Behavioral design patterns

# Materials

- Bruce Eckel, *Thinking in Patterns*
- Erich Gamma, *Design Patterns – Elements of Reusable Object-Oriented Software*

# Introduction

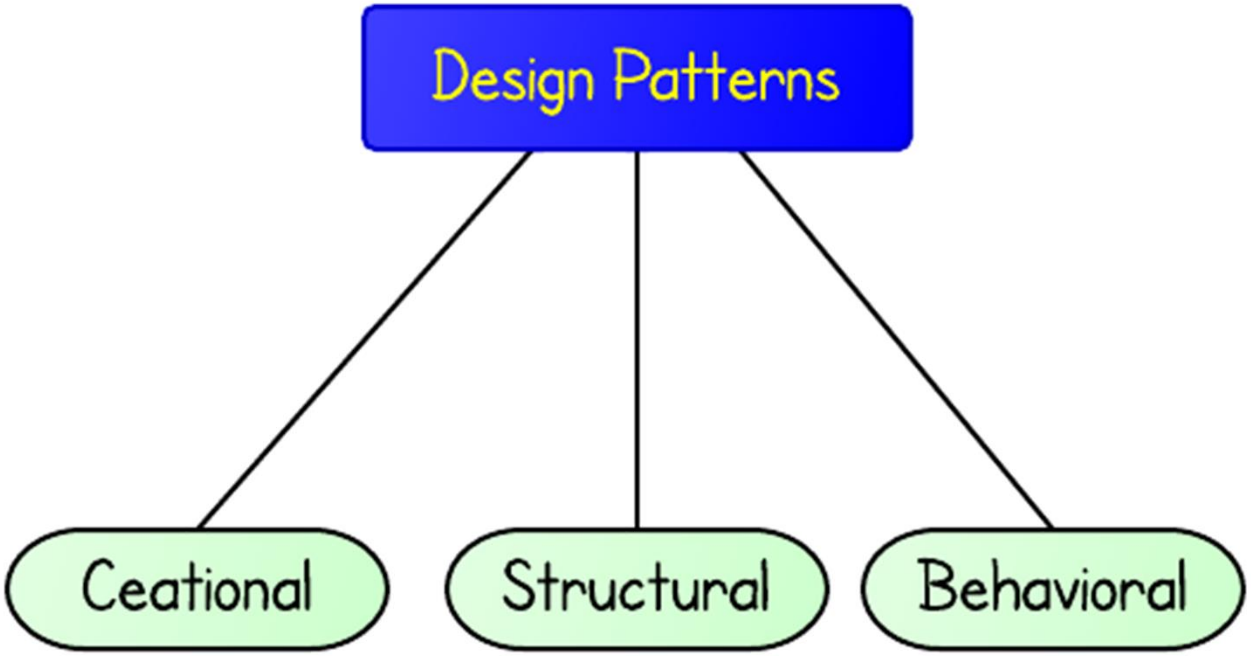
## Benefit system developers

- Helping construct reliable software using proven architectures and accumulated industry expertise
- **Prompting** design reuse in future systems
- Helping to **identify** common mistakes and pitfalls that occur when building systems
- Helping **to design systems** independently of the language in which they will ultimately implemented
- **Establishing** a common design vocabulary among developers
- **Shortening** the design phase in the software-development process

# Introduction…

- Design patterns are **neither** classes nor objects
- Rather, designers use design patterns to construct **sets** of classes and objects
- To use design patterns effectively, designers must **familiarize with** the most popular and effective patterns used in the software-engineering industry
- This lesson introduces several popular design patterns in Java, but these design patterns can be implemented in any OO language (e.g.,C++)

# Three categories

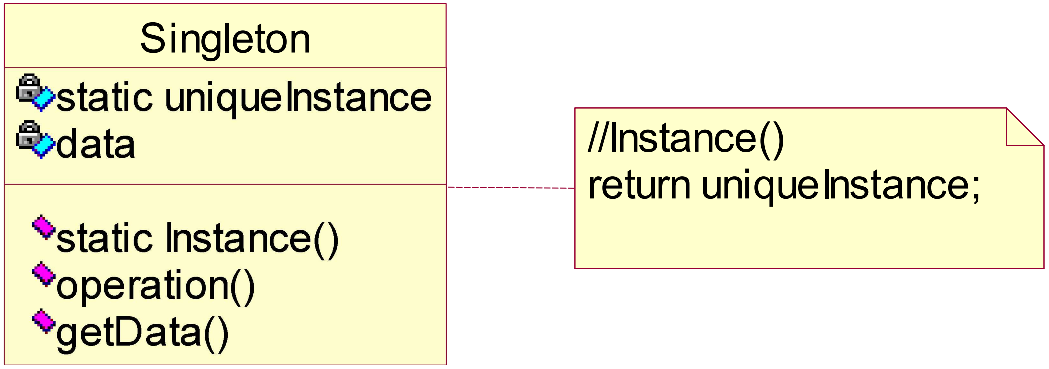


# Three categories…

| 1. Creational | 2. Structural | 3. Behavioral |
|---|---|---|
| Singleton, | Adapter, | Memento, |
| Factory method , | Composite, | State, |
| Abstract factory, | Decorator, | Chain of responsibility, |
| Prototype | Proxy, | Command, |
| | Bridge, | Observer, |
| | Façade | Strategy, |
| | | Template method, |
| | | Iterator |

# 1. Creational

## Singleton



Singleton

static uniqueInstance
data

static Instance()
operation()
getData()

```
//Instance()
return uniqueInstance;
```

- Occasionally, a system should contain **exactly** one object of a class
  - i.e., **once** the program instantiates that object, the program should **not** be allowed to create additional objects of that class
  - E.g., some systems connect to a database using only one object that manages database connections, which ensures that other objects cannot initialize unnecessary connections that would slow the system

# 1. Creational

## Example using singleton

```
public final class Singleton{ //”final” implies: subclasses that provide multiple
                              //instantiations cannot be created

private static final Singleton singleton = new Singleton(); // just one instance  of
    this class is created and provided  to clients

  private Singleton(){
    System.err.println("Singleton object created");// ”private constructor” means
    that only this class can use this constructor;
  }

  public static Singleton getInstance(){ //return a copy of the reference
    return singleton;
  }
}
```

# 1. Creational

## Example using singleton…

```
public class SingletonTest{
    public static void main(String[] args){
        Singleton firstSingleton;
        Singleton secondSingleton;

        firstSingleton = Singleton.getInstance();
        secondSingleton=Singleton.getInstance();

        if(firstSingleton==secondSingleton)
            System.err.println("1st and 2nd singleton "
                                        + "refers to the same Singleton object");
    }
}
```

2 references to Singleton object

# Remark

## ❖ Private Constructor

- A **private constructor** does not allow a class to be subclassed
- A **private constructor** does not allow to create an object outside the class
- If all the constant methods are there in our class we can use a **private constructor**
- If all the methods are **static** then we can use a **private constructor**
- If we try to **extend a class** which is having private constructor **compile time error will occur**

# 1. Creational

## Factory method

- The sole purpose of the factory method is to create **objects** by allowing the system to determine **which class** to instantiate **at run time**

- E.g., Designing a system that **opens** an image for a specific file
  - Several different image formats exist (e.g., GIF, JPEG)
  - we can **use** the method ImageCreate() of class java.awt.Component to create **an Image object**
  - Assume we want to create an JPEG image and GIF image **in an object** of a Component **subclass** (e.g. a JPanel object)
  - We pass *the name of the image file* to method ImageCreate(), which returns **an Image object** that stores the image data

# 1. Creational

## Factory method…

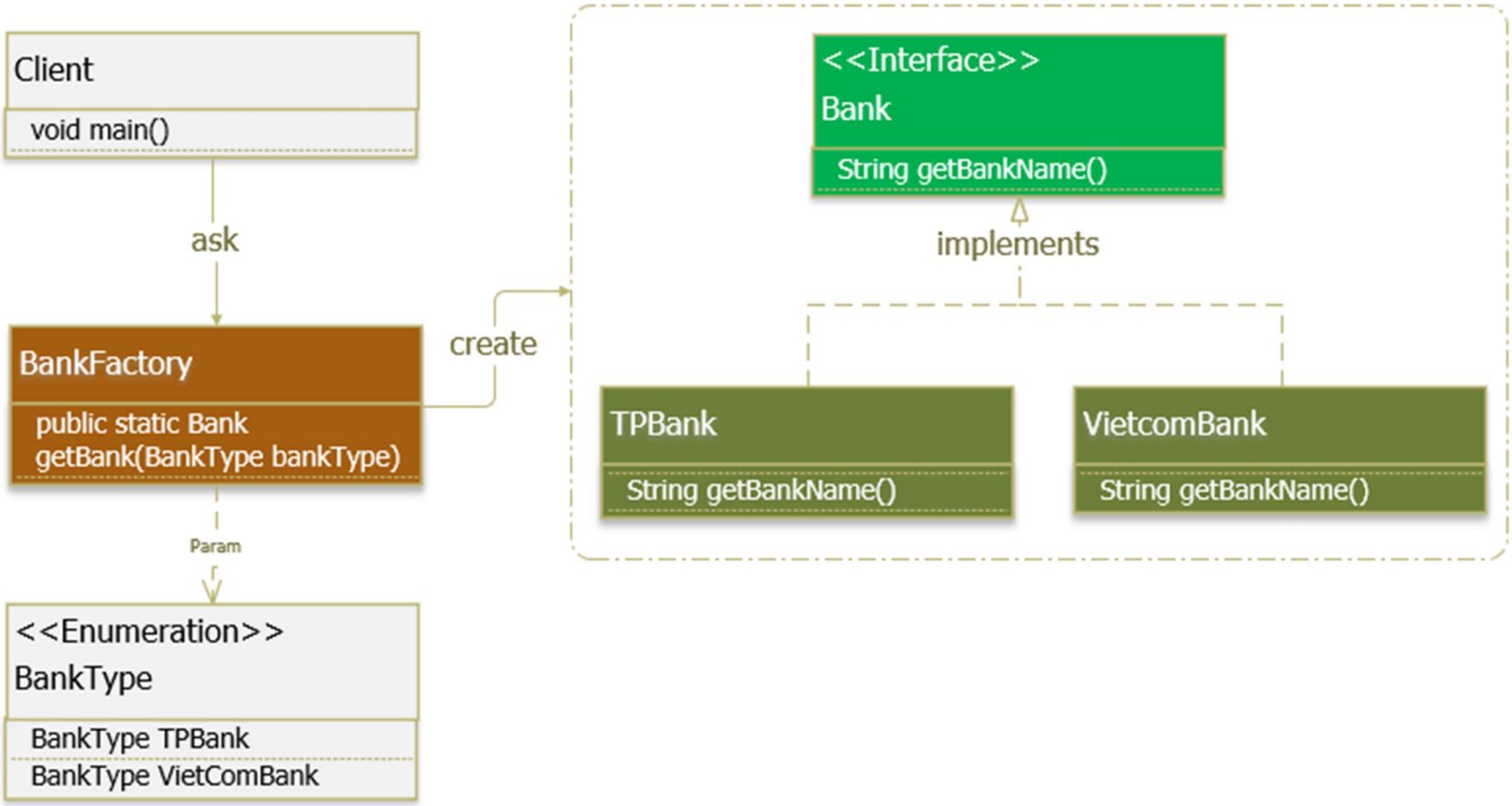
- We can create 2 image objects, **each** of which contains data for 2 images having entirely different structures (JPEG image can hold up to 16.7 million colors, whereas GIF image can hold  up to only 256)
- a GIF image can contain transparent pixel that are not rendered on screen, whereas a JPEG image **cannot** contain transparent pixels

❖ **Class Image** is an **abstract class** that represent an image we can display on the screen

❖ Using the parameters passed by the programmer, **method** createImage() determines the specific **Image** subclass  from which to instantiate the Image object

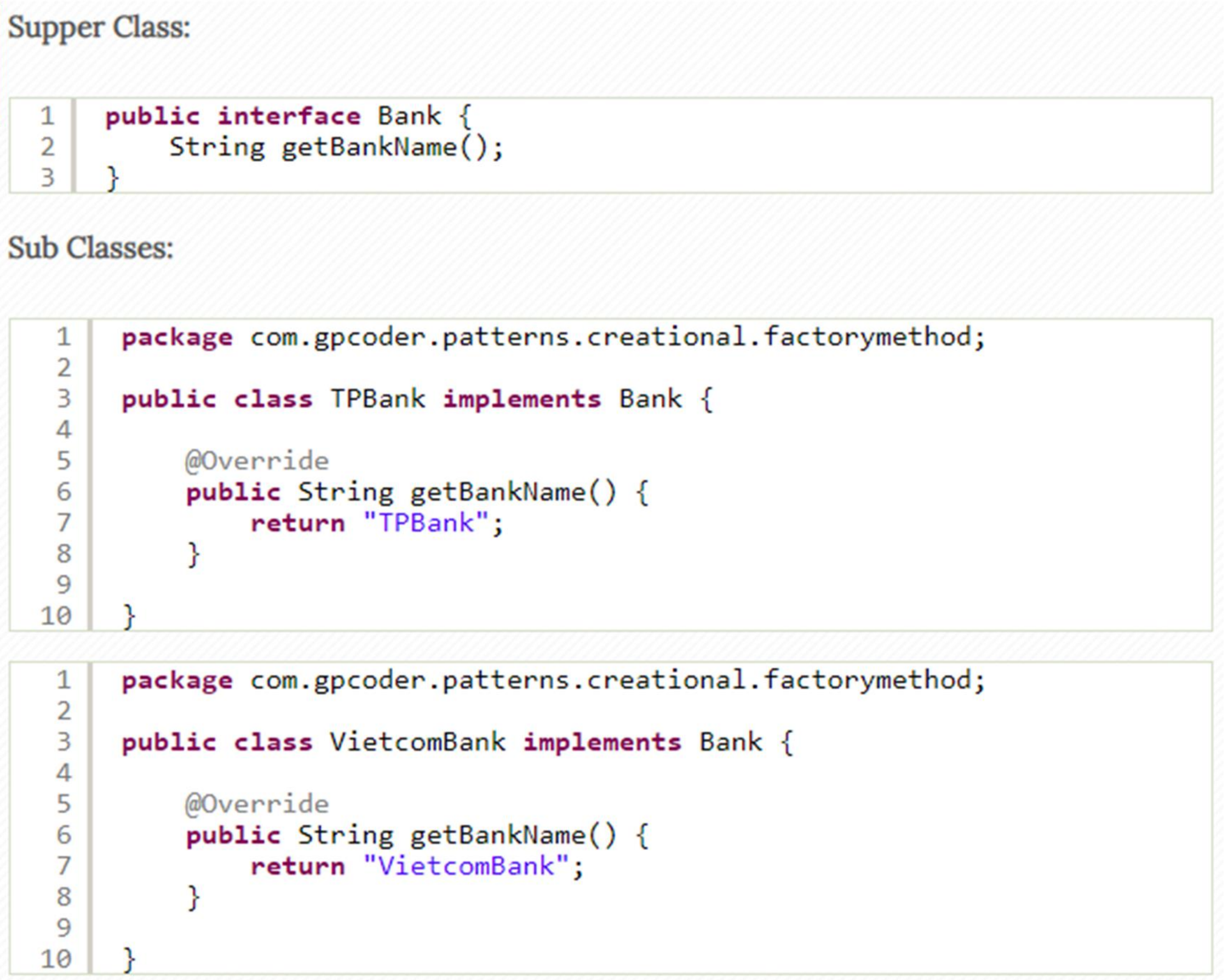
# Factory

## Eg.


Client
void main()
ask
BankFactory
public static Bank getBank(BankType bankType)
Param
<<Enumeration>>
BankType
BankType TPBank
BankType VietComBank
create
<<Interface>>
Bank
String getBankName()
implements
TPBank
String getBankName()
VietcomBank
String getBankName()

# Factory

## Eg.



Supper Class:

```java
public interface Bank {
    String getBankName();
}
```

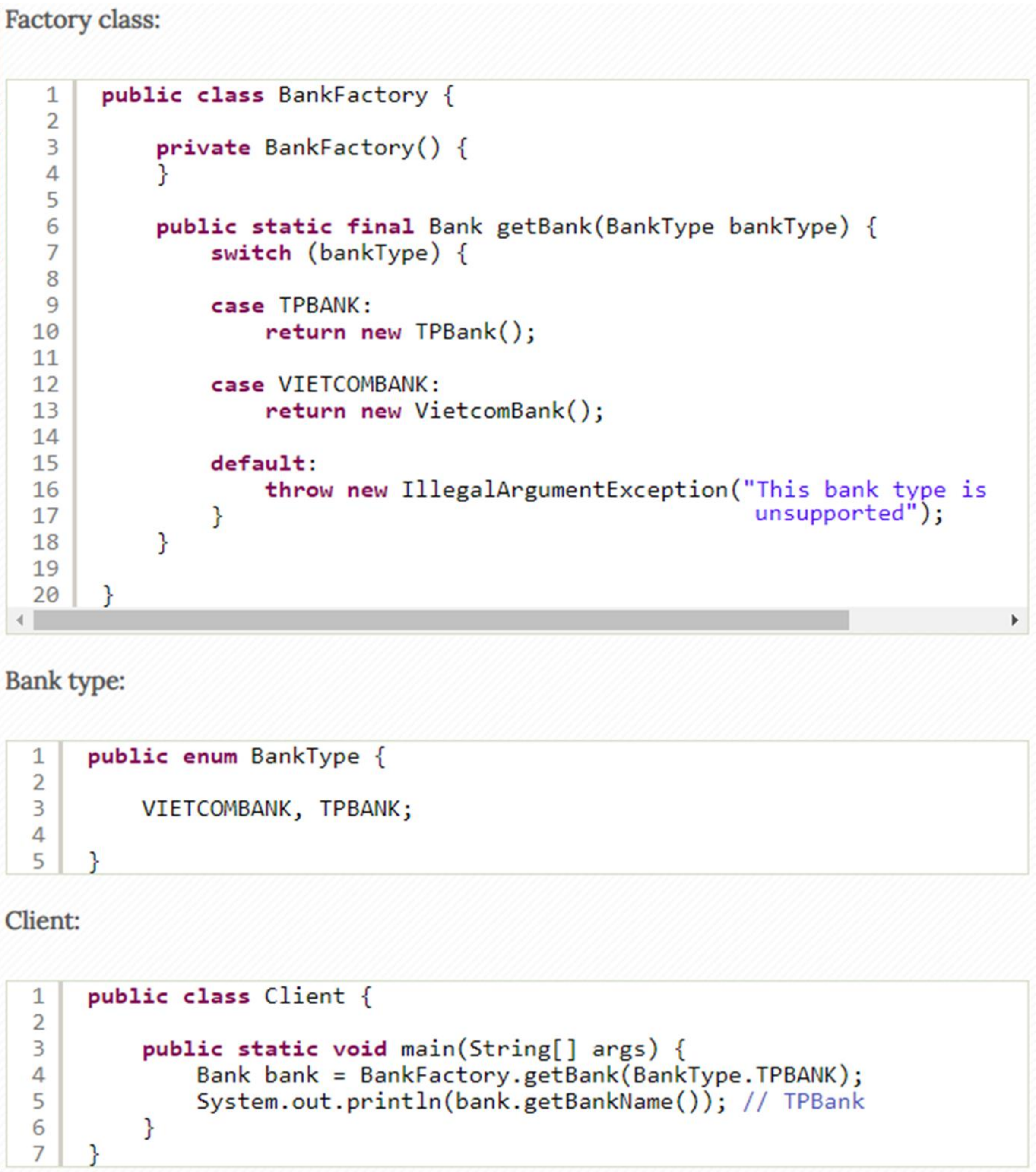
Sub Classes:

```java
package com.gpcoder.patterns.creational.factorymethod;

public class TPBank implements Bank {

    @Override
    public String getBankName() {
        return "TPBank";
    }

}
```

```java
package com.gpcoder.patterns.creational.factorymethod;

public class VietcomBank implements Bank {

    @Override
    public String getBankName() {
        return "VietcomBank";
    }

}
```

# Factory

## Eg.



Factory class:

```java
public class BankFactory {

    private BankFactory() {
    }

    public static final Bank getBank(BankType bankType) {
        switch (bankType) {

        case TPBANK:
            return new TPBank();

        case VIETCOMBANK:
            return new VietcomBank();

        default:
            throw new IllegalArgumentException("This bank type is unsupported");
        }
    }

}
```

Bank type:

```java
public enum BankType {

    VIETCOMBANK, TPBANK;

}
```

Client:

```java
public class Client {

    public static void main(String[] args) {
        Bank bank = BankFactory.getBank(BankType.TPBANK);
        System.out.println(bank.getBankName()); // TPBank
    }
}
```

# 1. Creational

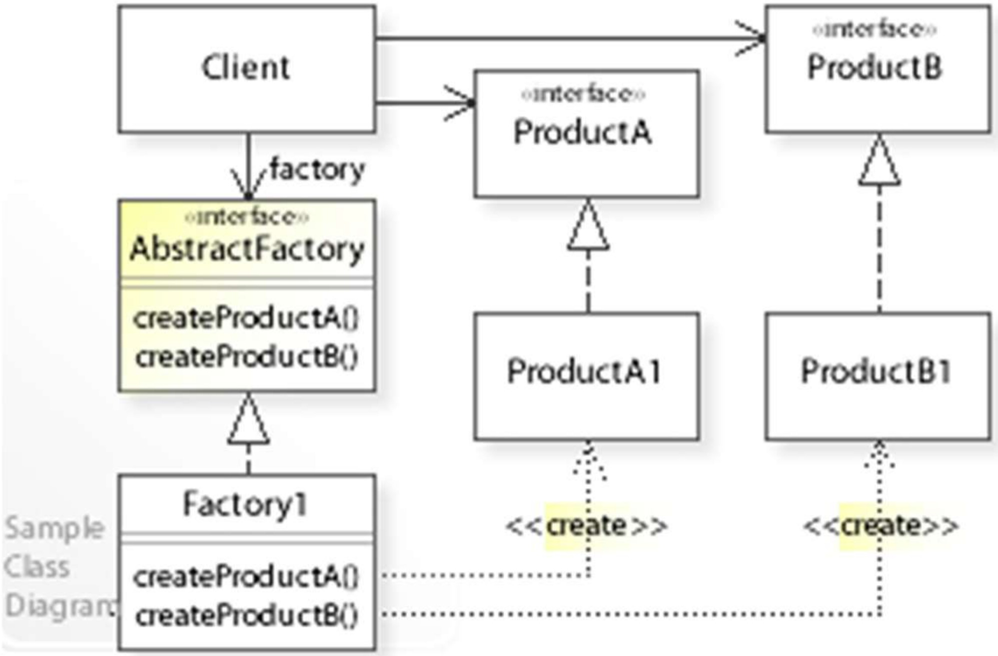
## Abstract factory

- Like Factory method design pattern, abstract factory design pattern allows a system **to determine the subclass** from which to instantiate an object at run time

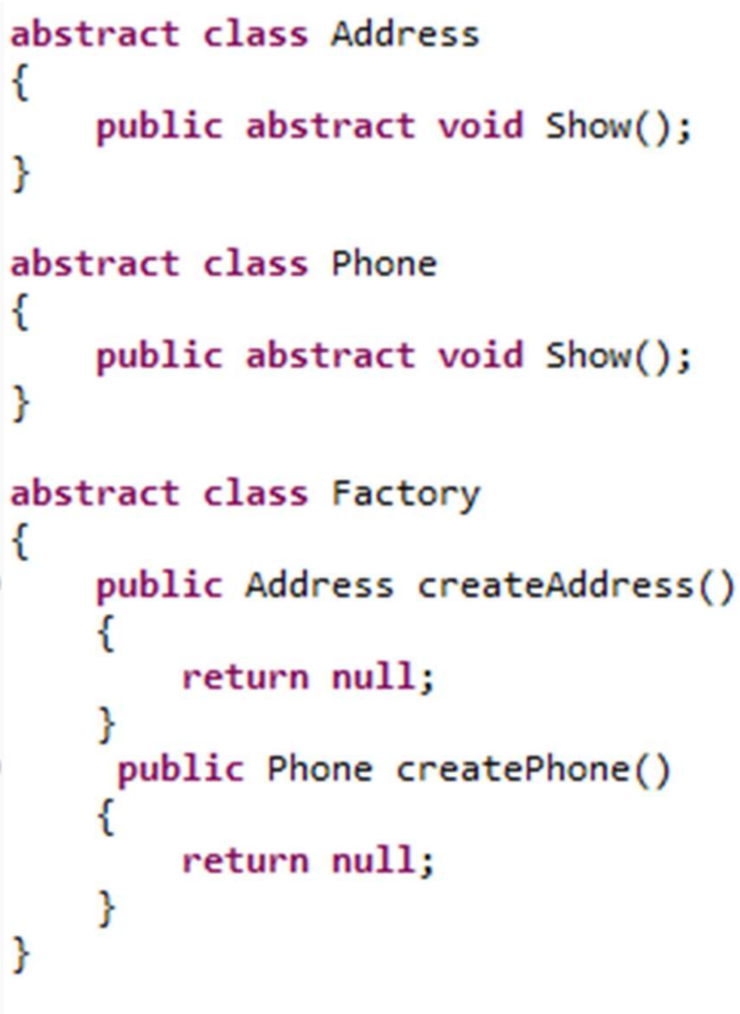
- **However**, abstract factory uses **an object** known as **factory** that uses *an interface* to instantiate objects
  - A factory creates a product, in this case, **that product** is an object of a **subclass** determined **at run time**

# Abstract factory

- Solve problems
  - How can an application be independent of how its objects are created
  - How can a class be independent of how the objects it requires are created
  - How can families of related or dependent objects be created?

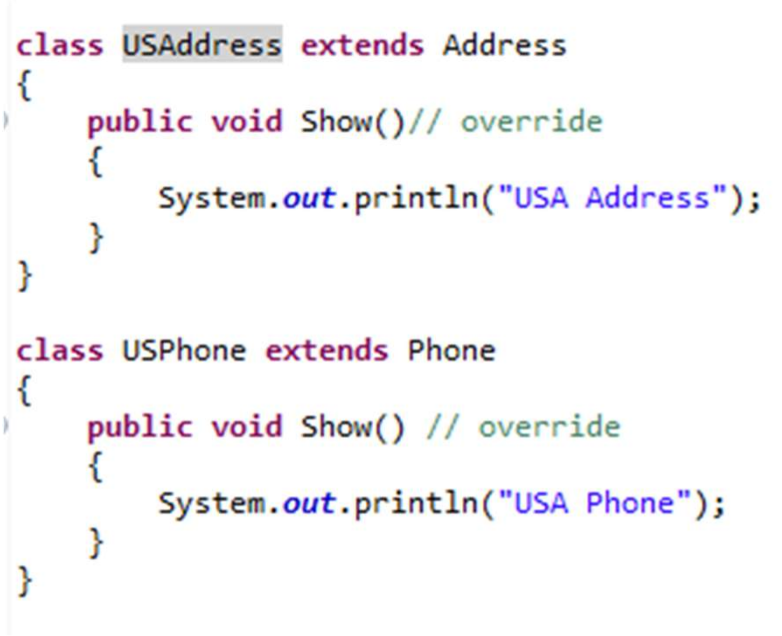


# E.g. 2

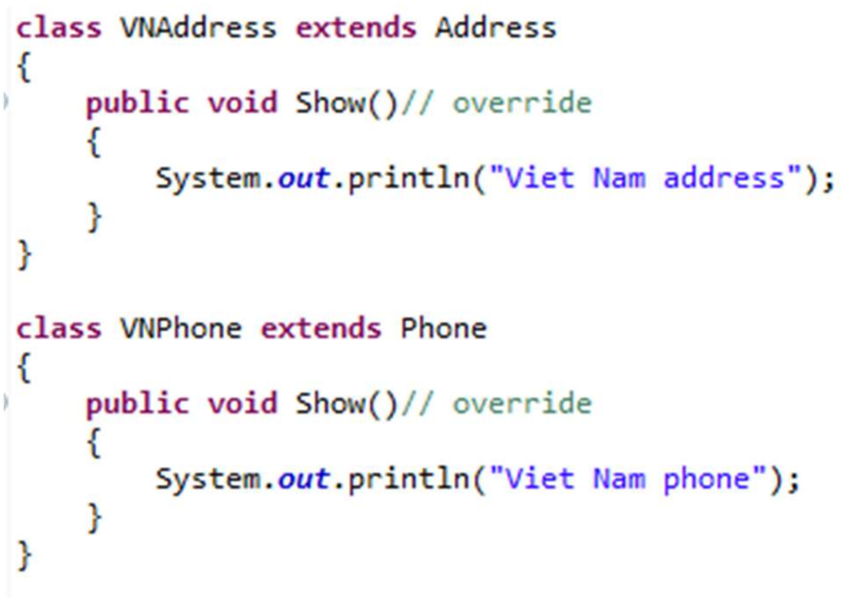


```
abstract class Address
{
    public abstract void Show();
}

abstract class Phone
{
    public abstract void Show();
}

abstract class Factory
{
    public Address createAddress()
    {
        return null;
    }
     public Phone createPhone()
    {
        return null;
    }
}
```

# Eg.

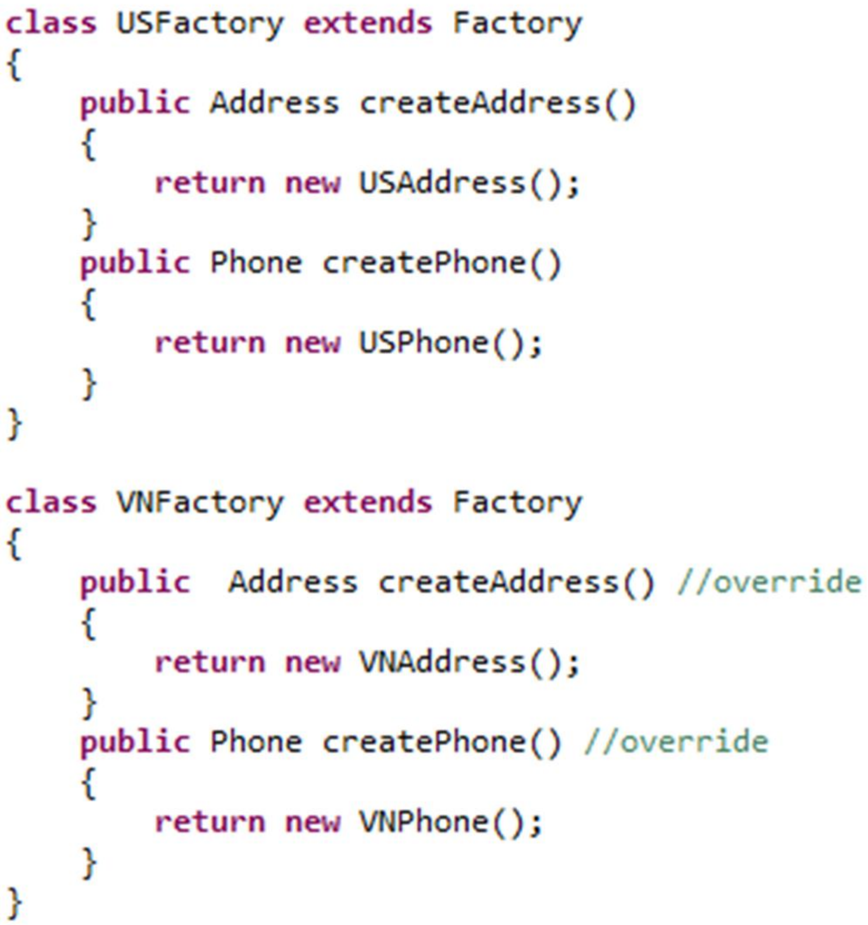


```
class USAddress extends Address
{
    public void Show()// override
    {
        System.out.println("USA Address");
    }
}

class USPhone extends Phone
{
    public void Show() // override
    {
        System.out.println("USA Phone");
    }
}
```



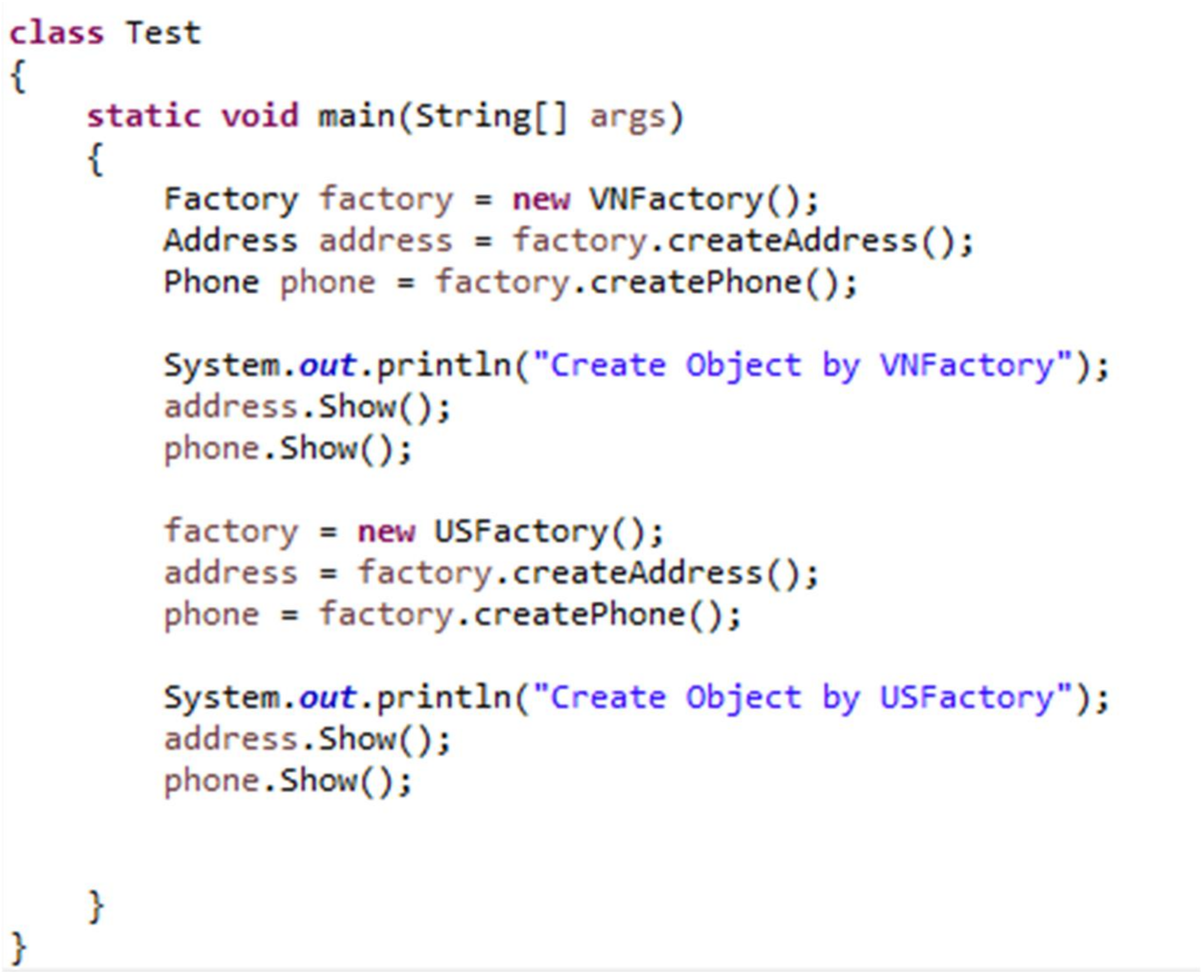
```
class VNAddress extends Address
{
    public void Show()// override
    {
        System.out.println("Viet Nam address");
    }
}

class VNPhone extends Phone
{
    public void Show()// override
    {
        System.out.println("Viet Nam phone");
    }
}
```

# Eg.



```
class USFactory extends Factory
{
    public Address createAddress()
    {
        return new USAddress();
    }
    public Phone createPhone()
    {
        return new USPhone();
    }
}

class VNFactory extends Factory
{
    public  Address createAddress() //override
    {
        return new VNAddress();
    }
    public Phone createPhone() //override
    {
        return new VNPhone();
    }
}
```

# Eg.



```
class Test
{
    static void main(String[] args)
    {
        Factory factory = new VNFactory();
        Address address = factory.createAddress();
        Phone phone = factory.createPhone();

        System.out.println("Create Object by VNFactory");
        address.Show();
        phone.Show();

        factory = new USFactory();
        address = factory.createAddress();
        phone = factory.createPhone();

        System.out.println("Create Object by USFactory");
        address.Show();
        phone.Show();


    }
}
```

# 1. Creational

## Abstract factory…

- Java socket library in package **java.net** uses the Abstract Factory
- A socket describes a connections (or a stream of data) between 2 processes
- **Class Socket** references an object of a **SocketImpl** subclass
- **Class Socket** also contains a **static** reference to an object implementing interface SocketImpFactory
- The **Socket constructor** invokes method createSocketImpl of interface SocketImplFactory to create the SocketImpl object
  - The object that implements interface SocketImplfactory is **the factory,**
  - and an object of a SocketImpl subclass is the **product** of that factory

# 1. Creational

## Abstract factory…

- The system cannot specify the **SocketImpl** **subclass** from which to instantiate **until** run time, because the system has no knowledge of that type of Socket implementation required

- Method createSocketImpl decides the the SocketImpl subclass from which to instantiate the object at run time

# 2. Structural design pattern

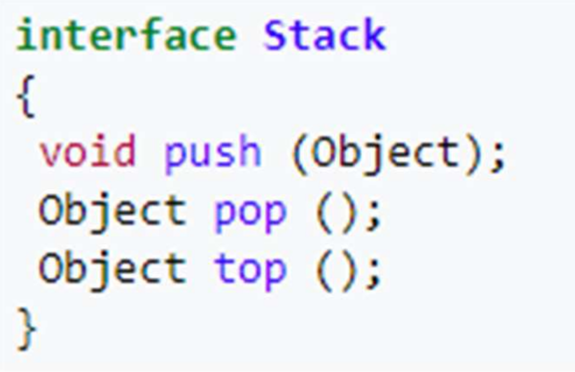
- Describe common way to organize classes and objects in a system
- 3 out of 7 structural design patterns will be introduced
  - Adapter
  - Composite
  - Decorator

# 2. Structural

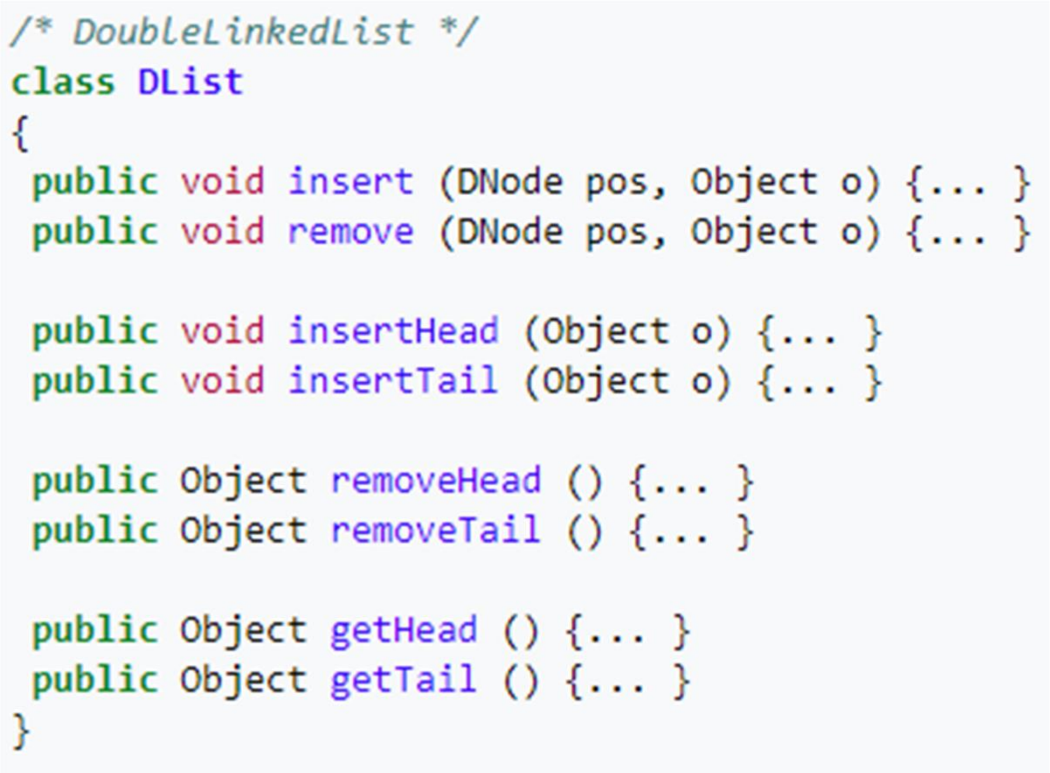
## Adapter

- Helps objects with **incompatible interface** collaborate with one another
- Provides an object with a **new** interface that adapts to another object's interface, allowing both objects to collaborate with one another
- Java provides **several classes** that use the Adapter design pattern
  - **Objects** of the concrete subclasses of these classes **act** as adapters between objects that *generate certain events* and objects that *handle the events*
  - e.g., a **MouseAdapte**r adapts an object that generates **MouseEvent** to an object that handle **MouseEvents**

# E.g.

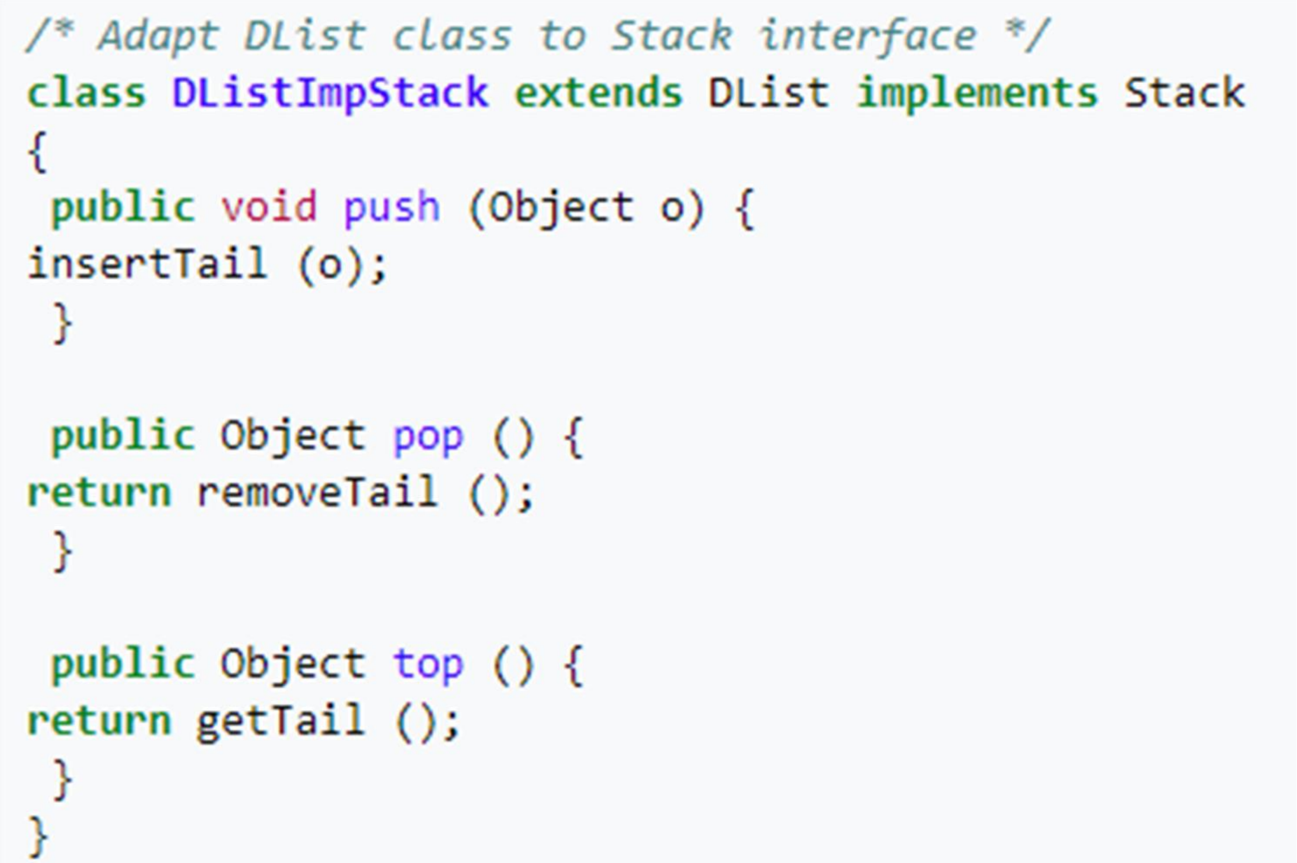


```
interface Stack
{
 void push (Object);
 Object pop ();
 Object top ();
}
```



```
/* DoubleLinkedList */
class DList
{
 public void insert (DNode pos, Object o) {... }
 public void remove (DNode pos, Object o) {... }

 public void insertHead (Object o) {... }
 public void insertTail (Object o) {... }

 public Object removeHead () {... }
 public Object removeTail () {... }

 public Object getHead () {... }
 public Object getTail () {... }
}
```

# E.g.



```
/* Adapt DList class to Stack interface */
class DListImpStack extends DList implements Stack
{
 public void push (Object o) {
insertTail (o);
 }

 public Object pop () {
return removeTail ();
 }

 public Object top () {
return getTail ();
 }
}
```

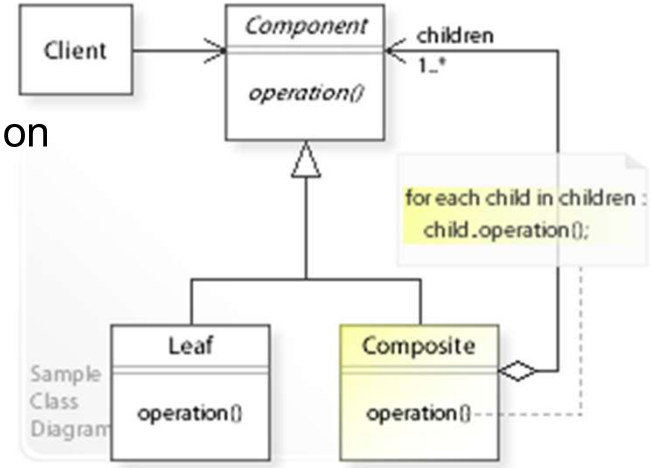
# Composite

- Provides a way for designers to organize and manipulate objects
- Designers often organize components into hierarchical structure
  - Occasionally, a structure contains objects from several different classes (e.g., a directory contains files & directories)
- In *the composite design pattern*, each component in a hierarchical structure implements the **same** interface **or** extends a common superclass
  - this ensures that clients can traverse all elements uniformly in the structure
  - Using this pattern, a client traversing the structure doesn't have to determine each component type

# 2. Structural

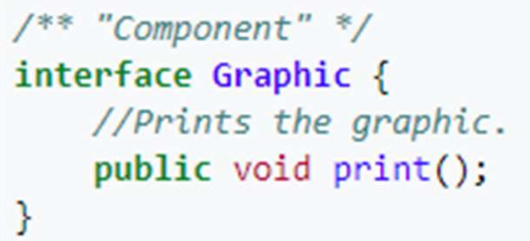
## Composite

- Component
  - Is the abstraction for all components
  - Declares the interface for objects in the composition
- Leaf
  - Represents leaf objects in the composition
  - Implements all Component methods
- Composition
  - Represents a composite Component (component having children)
  - Implements methods to manipulate children
  - Implements all Component methods,

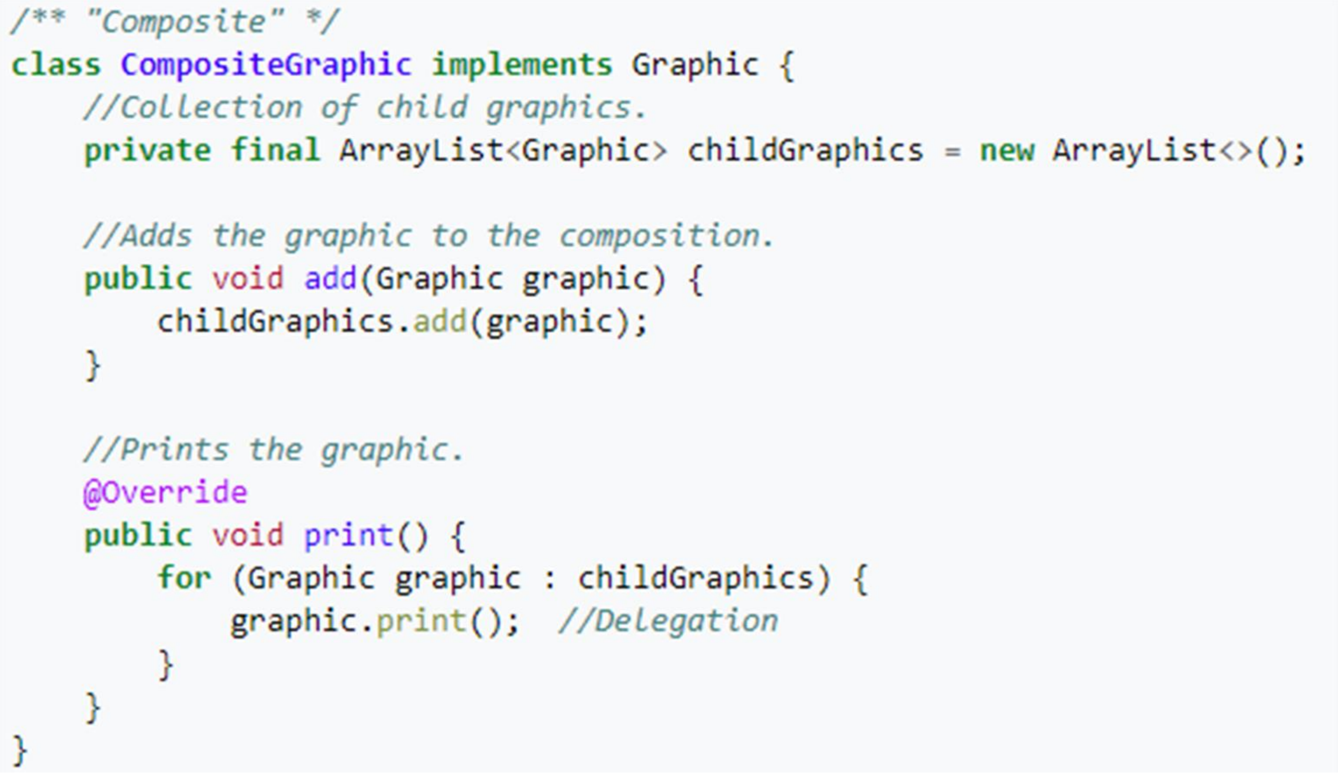


# 2. Structural

## Composite



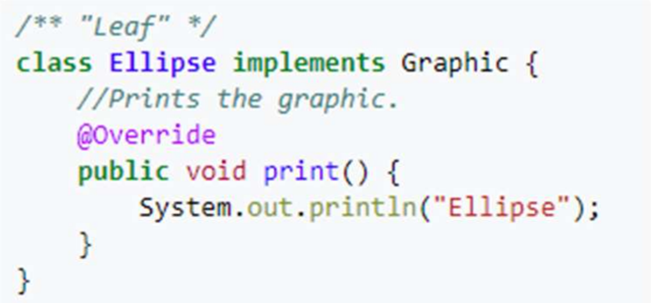
```
/** "Component" */
interface Graphic {
    //Prints the graphic.
    public void print();
}
```



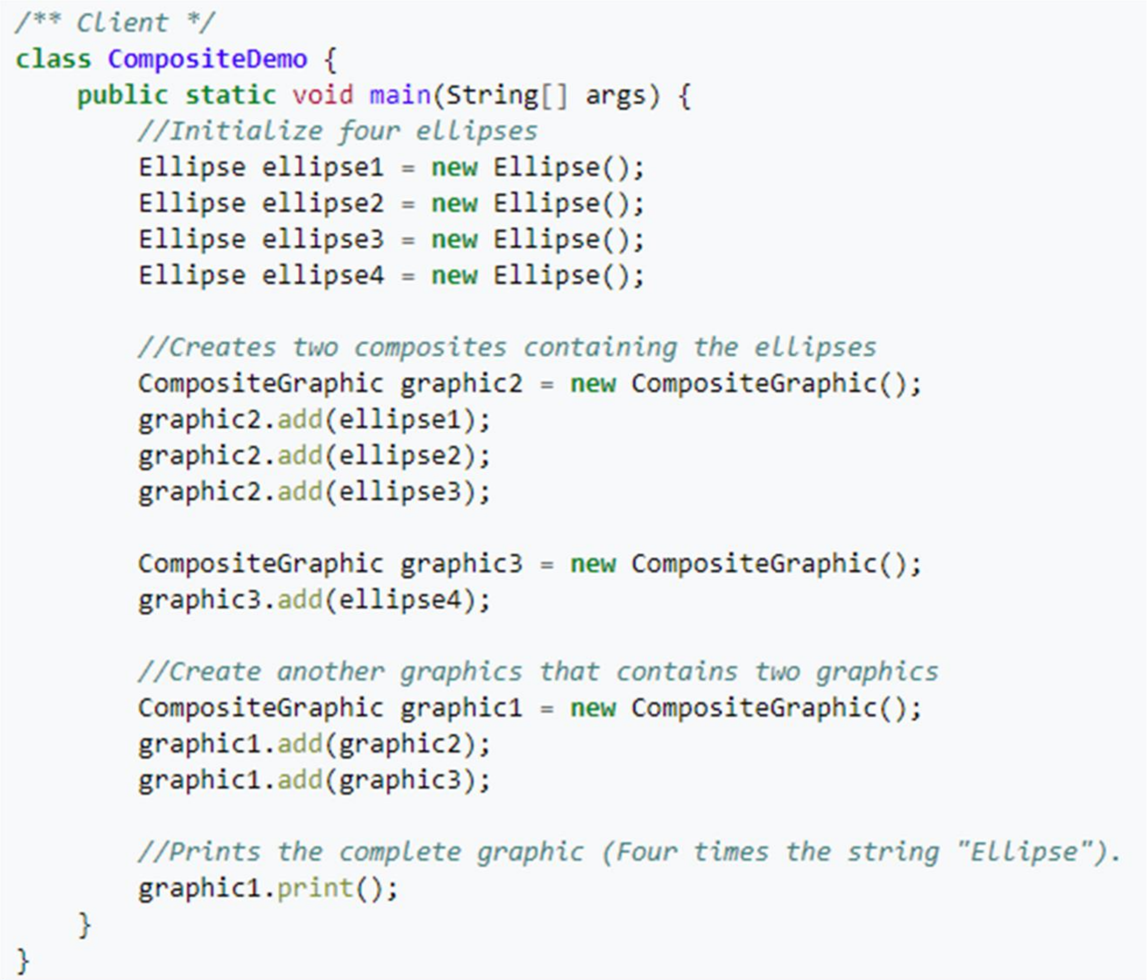
```
/** "Composite" */
class CompositeGraphic implements Graphic {
    //Collection of child graphics.
    private final ArrayList<Graphic> childGraphics = new ArrayList<>();

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    //Prints the graphic.
    @Override
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();  //Delegation
        }
    }
}
```

# 2. Structural

## Composite



```java
/** "Leaf" */
class Ellipse implements Graphic {
    //Prints the graphic.
    @Override
    public void print() {
        System.out.println("Ellipse");
    }
}
```

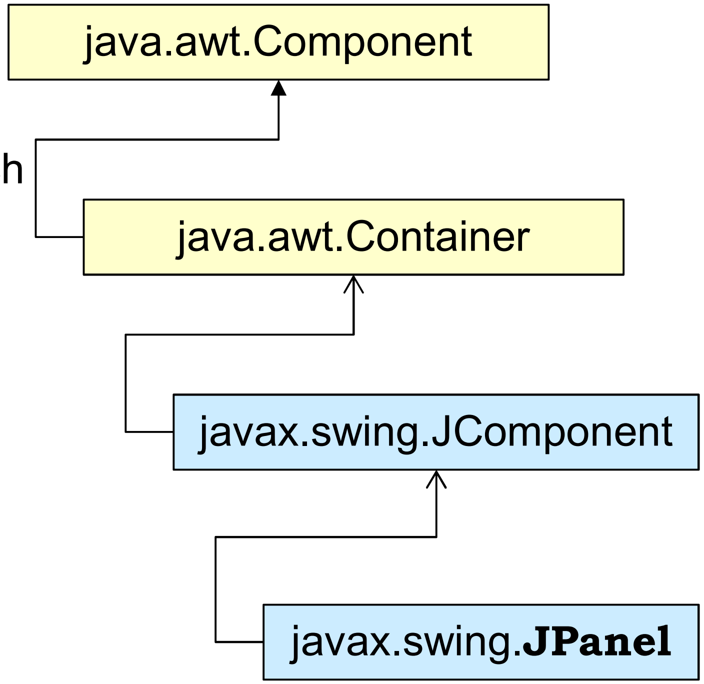


```java
/** Client */
class CompositeDemo {
    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Creates two composites containing the ellipses
        CompositeGraphic graphic2 = new CompositeGraphic();
        graphic2.add(ellipse1);
        graphic2.add(ellipse2);
        graphic2.add(ellipse3);

        CompositeGraphic graphic3 = new CompositeGraphic();
        graphic3.add(ellipse4);

        //Create another graphics that contains two graphics
        CompositeGraphic graphic1 = new CompositeGraphic();
        graphic1.add(graphic2);
        graphic1.add(graphic3);

        //Prints the complete graphic (Four times the string "Ellipse").
        graphic1.print();
    }
}
```

# 2. Structural

## Composite...

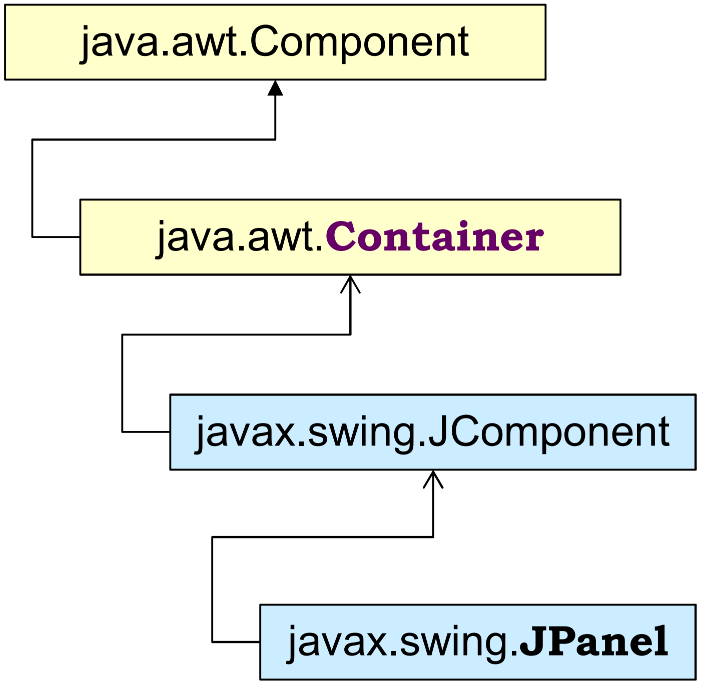
- Java GUI components use the Composite design pattern
- consider the Swing component class **JPanel**, which extends class JComponent
- class **Jcomponent** extends class java.awt.Container, which extends java.awt.Component
- class **Container** provides method **add,** which appends a Component object (or Component subclass)
  - So a JPanel object may be added to any object of a Component subclass,
  - and any object from a Component subclass may be added to that JPanel object



# 2. Structural

## Composite...

- a **JPanel** object can contain **any** GUI component while remaining unaware of that component's specific type
- A client, such as JPanel object, can traverse all components uniformly in hierarchy
  - e.g., if **JPanel** object calls method repaint of **superclass Container,** method repaint displays the **JPanel object** as well **as all components** added to the JPanel object
  - The method repaint doesn't have to determine each component's **style** because all components inherit from superclass Container which contains method repaint



# 2. Structural

## Decorator

- Allows an object to gain additional functionality dynamically

```
public class CreateSequentialFile {
        ...//open a file
        output= new ObjectOutputStream(new FileOutputStream(fileName))
}
```

- allows a FileOutputStream object, which write bytes to a file, to gain the functionality of an ObjectOutputStream, which provides methods for writing **entire objects** to an OutputStream
- class CreateSequentialFile appears to “**wrap**” an ObjectOutputStream around a FileOutputStream object

# 2. Structural

## Decorator…

- We can dynamically add the behavior of an ObjectOutputStream to a FileOutputStream **prevents** the need for a separate class called ObjectFileOutputStream, which would implement the behavior of both classes

- → Using this pattern, designers **don't** have to create separate, unnecessary classes to add responsibilities to objects of a given class

# 2. Structural

## Decorator...

- Can simplify a system's structure
- E.g., we want to enhance the I/O performance of the previous example using a BufferedOutputStream. Using decorator design pattern we can write

```
output = new ObjectOutputStream(new BufferedOutputStream(new
                                                FileOutputStream(fileName)));
```

- We can combine objects in this manner because
  - ObjectOutputStream, BufferedOutputStream and FileOutputStream extends abstract superclass OutputStream
  - and **each subclass constructor** takes an OutputStream object as a parameter

# 2. Structural

## Decorator...

- if the stream objects in package java.io did not use the Decorator design pattern, package java.io would have to provide classes *BufferedFileOutputStream, ObjectBufferedOutputStream, ObjectBufferedFileOutputStream* and *ObjectFileOutputStream*
- → consider how many classes we would have to create if we combined even more stream objects without applying the Decorator pattern

# Summary

- Get familiar with most of the **important concepts** of OOP
- **Plan, design,** and **create** an entire Java program from scratch, as well as creating an **interactive experience** by reading use input and writing it to file