Design Patterns

Lecture 08 – Part 01

Jayathma Chathurangani
ejc@ucsc.cmb.ac.lk

OUTLINE

1. Introduction
2. Creational Patterns
3. Structural Patterns
4. Behavioural Patterns

TOPIC 01

Introduction

1.1 What is Gang of Four (GOF)?

- n 1994, four authors **Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides** published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.
- These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.
  - Program to an interface not an implementation
  - Favor object composition (has-a) over inheritance (is-a)

1.2 Definition of Design Patterns

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.
- The pattern is not a specific piece of code, but a general concept for solving a particular problem.
- You can follow the pattern details and implement a solution that suits the realities of your own program.

1.3 Design Patterns Explained Further

- Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution.
- The code of the same pattern applied to two different programs may be different.
- An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal.
- On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.
- Pattern consist of
  - **Intent** of the pattern briefly describes both the problem and the solution.
  - **Motivation** further explains the problem and the solution the pattern makes possible.
  - **Structure** of classes shows each part of the pattern and how they are related.
  - **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

1.4 Uses of Design Patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation.
- 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.
- Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.
- Patterns allow developers to communicate using well-known, well understood names for software interactions.
- Common design patterns can be improved over time, making them more robust than ad-hoc designs.

1.5 AntiPattern

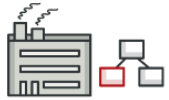
- AntiPatterns, like their design pattern counterparts, define an industry vocabulary for the common defective processes and implementations within organizations.
- An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.
- The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context.
- A key goal of development AntiPatterns is to describe useful forms of software refactoring.
- ***Software refactoring*** is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance.
- In most cases, the goal is to transform code without impacting correctness..

1.6 Classification of Patterns

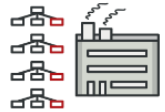
- Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed.
- Analogy to road construction: you can make an intersection safer by either installing some traffic lights or building an entire multi-level interchange with underground passages for pedestrians.
- The most basic and low-level patterns are often called ***idioms***. They usually apply only to a single programming language.
- The most universal and high-level patterns are ***architectural patterns***. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.
- In addition, all patterns can be categorized by their intent, or purpose. We three main groups of patterns:
  - **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
  - **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
  - **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

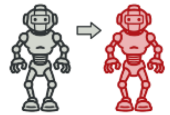


Factory Method

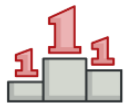


Abstract Factory

Builder



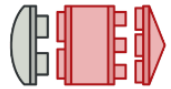
Prototype



Singleton

Structural patterns

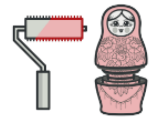
These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



Adapter

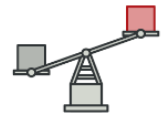
Bridge

Composite

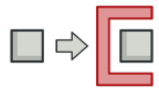


Decorator

Facade



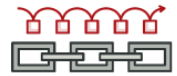
Flyweight



Proxy

Behavioral patterns

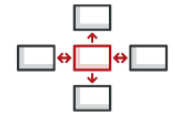
These patterns are concerned with algorithms and the assignment of responsibilities between objects.



Chain of Responsibility

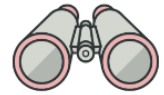
Command

Iterator

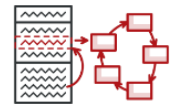


Mediator

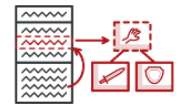
Memento



Observer



State



Strategy

Template Method

Visitor

1.7 SOLID Principles

- SOLID principles are an object-oriented approach that are applied to software structure design. It is conceptualized by Robert C. Martin (also known as Uncle Bob).
- These five principles have changed the world of object-oriented programming, and also changed the way of writing software. It also ensures that the software is modular, easy to understand, debug, and refactor.

S.O.L.I.D. Class Design Principles

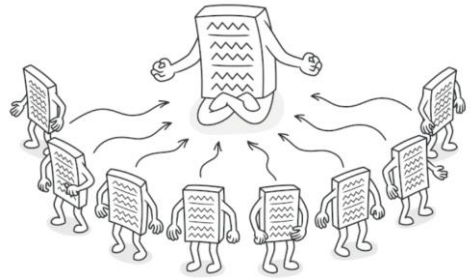
Principle Name	What it says? howtodoinjava.com
Single Responsibility Principle	One class should have one and only one reasonability
Open Closed Principle	Software components should be open for extension, but closed for modification
Liskov's Substitution Principle	Derived types must be completely substitutable for their base types
Interface Segregation Principle	Clients should not be forced to implement unnecessary methods which they will not use
Dependency Inversion Principle	Depend on abstractions, not on concretions

TOPIC 02

Creational Patterns

Creational design patterns provide solution to instantiate a object in the best possible way for specific situations.

2.1 Singleton Pattern



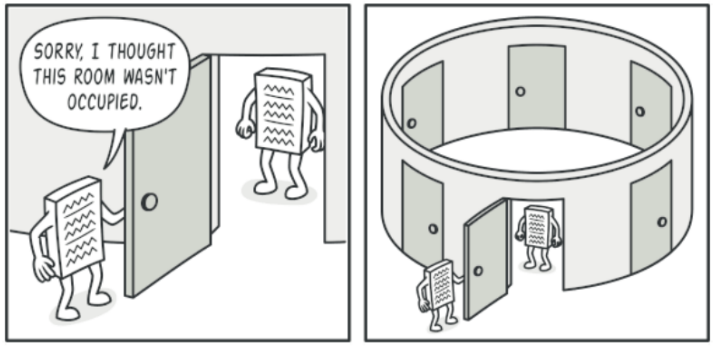
Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

❖ Introduction

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
- The singleton class must provide a global access point to get the instance of the class.
- Singleton pattern is used for **logging, drivers objects (When using external resource and hardware interface), caching and thread pool**.
- Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc.
- Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

Reference: https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples

2.1 Singleton Pattern (Continued)



Clients may not even realize that they're working with the same object all the time.

❖ Problem

1. **Ensure that a class has just a single instance.** Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.
   - Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.
2. **Provide a global access point to that instance**. Remember those global variables that you used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.
   - Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

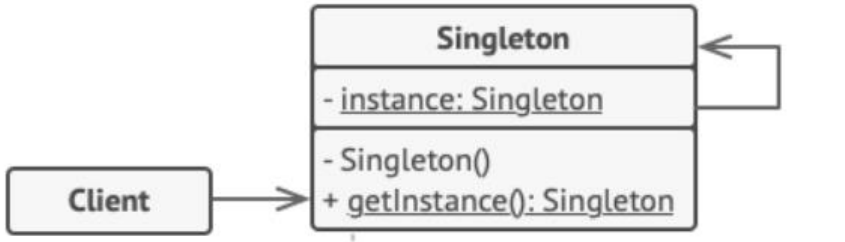
2.1 Singleton Pattern (Continued)

❖ Solution

- All implementations of the Singleton have these two steps in common:
  - Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
  - Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.
- If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

2.1 Singleton Pattern (Continued)

❖ Structure

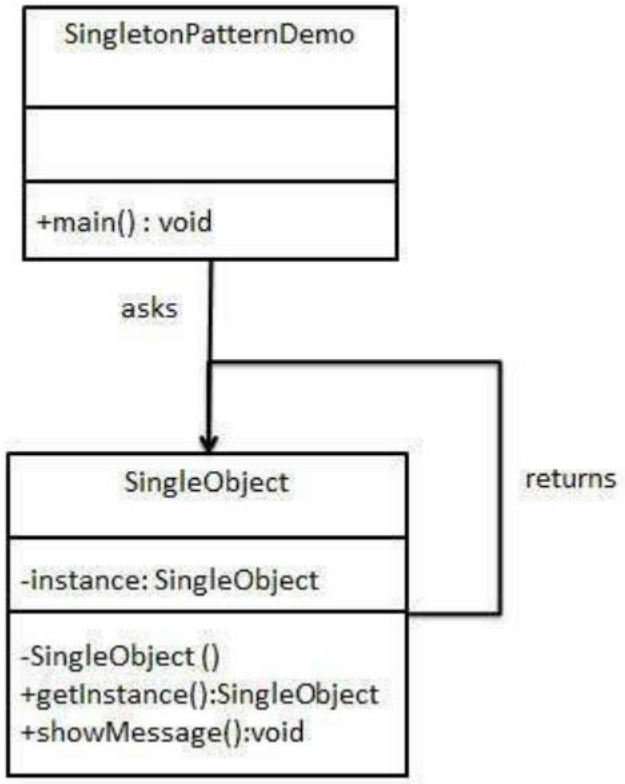


1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

2.1 Singleton Pattern (Continued)

❖ Implementation



➢ **Implementation –**

- We are going to create a SingletonObject class which has its constructor as private and a static instance of itself.
- SingletonObject class provides a static method to get its static instance to outside world. SingletonPatternDemo, our demo class, will use SingletonObject class to get a SingletonObject object.

2.1 Singleton Pattern (Continued)

❖ Implementation (Eager Initialization)

Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Verify the output.

```
Hello World!
```

2.1 Singleton Pattern (Continued)

❖ Applicability

Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.

The Singleton pattern disables all other means of creating objects of a class except for the special creation method. This method either creates a new object or returns an existing one if it has already been created.

Use the Singleton pattern when you need stricter control over global variables.

Unlike global variables, the Singleton pattern guarantees that there's just one instance of a class. Nothing, except for the Singleton class itself, can replace the cached instance.

Note that you can always adjust this limitation and allow creating any number of Singleton instances. The only piece of code that needs changing is the body of the `getInstance` method.

2.1 Singleton Pattern (Continued)

❖ Pros and Cons

- ✓ You can be sure that a class has only a single instance.
- ✓ You gain a global access point to that instance.
- ✓ The singleton object is initialized only when it's requested for the first time.

- ✗ Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.
- ✗ The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- ✗ The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- ✗ It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

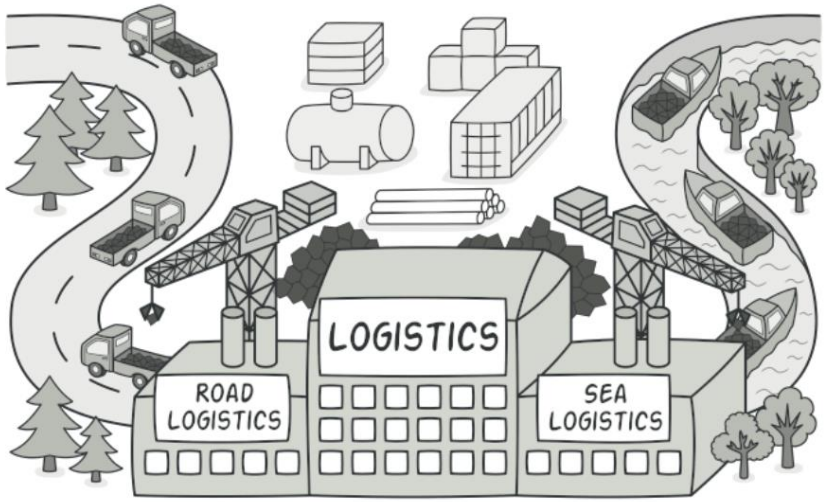
Further Reading:
https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples

2.2 Factory Pattern

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

❖ Introduction

- The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.
- This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



2.2 Factory Pattern (Continued)

❖ Problem

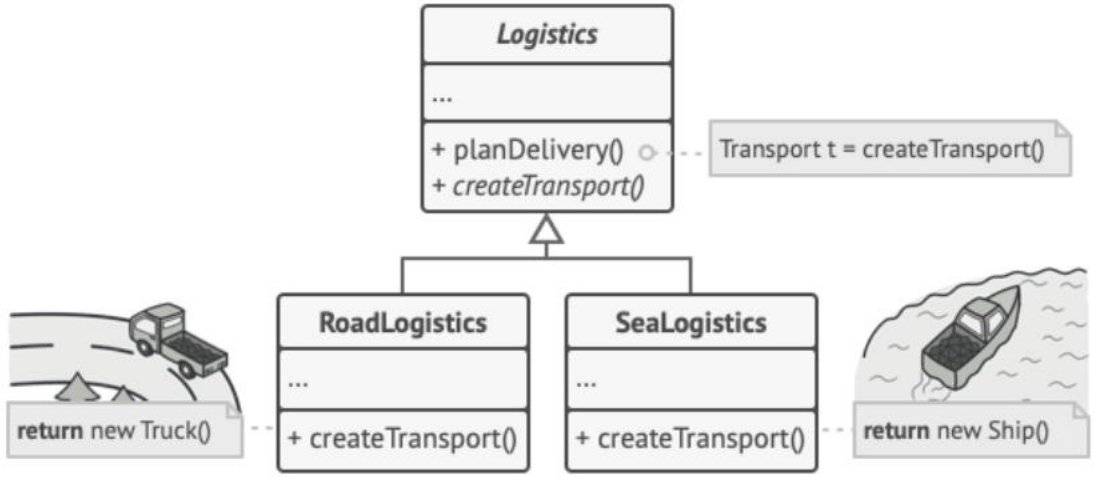
- Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
- After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.
- Great news, right? But how about the code? At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.
- As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.

Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

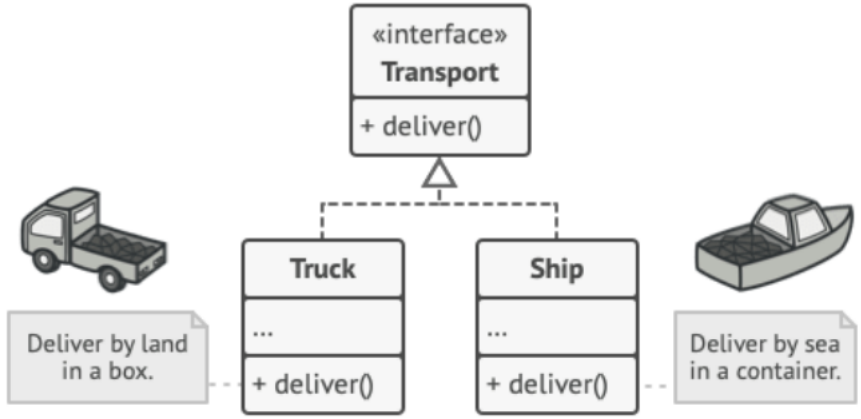
2.2 Factory Pattern (Continued)

❖ Solution

- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.
- Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as products.



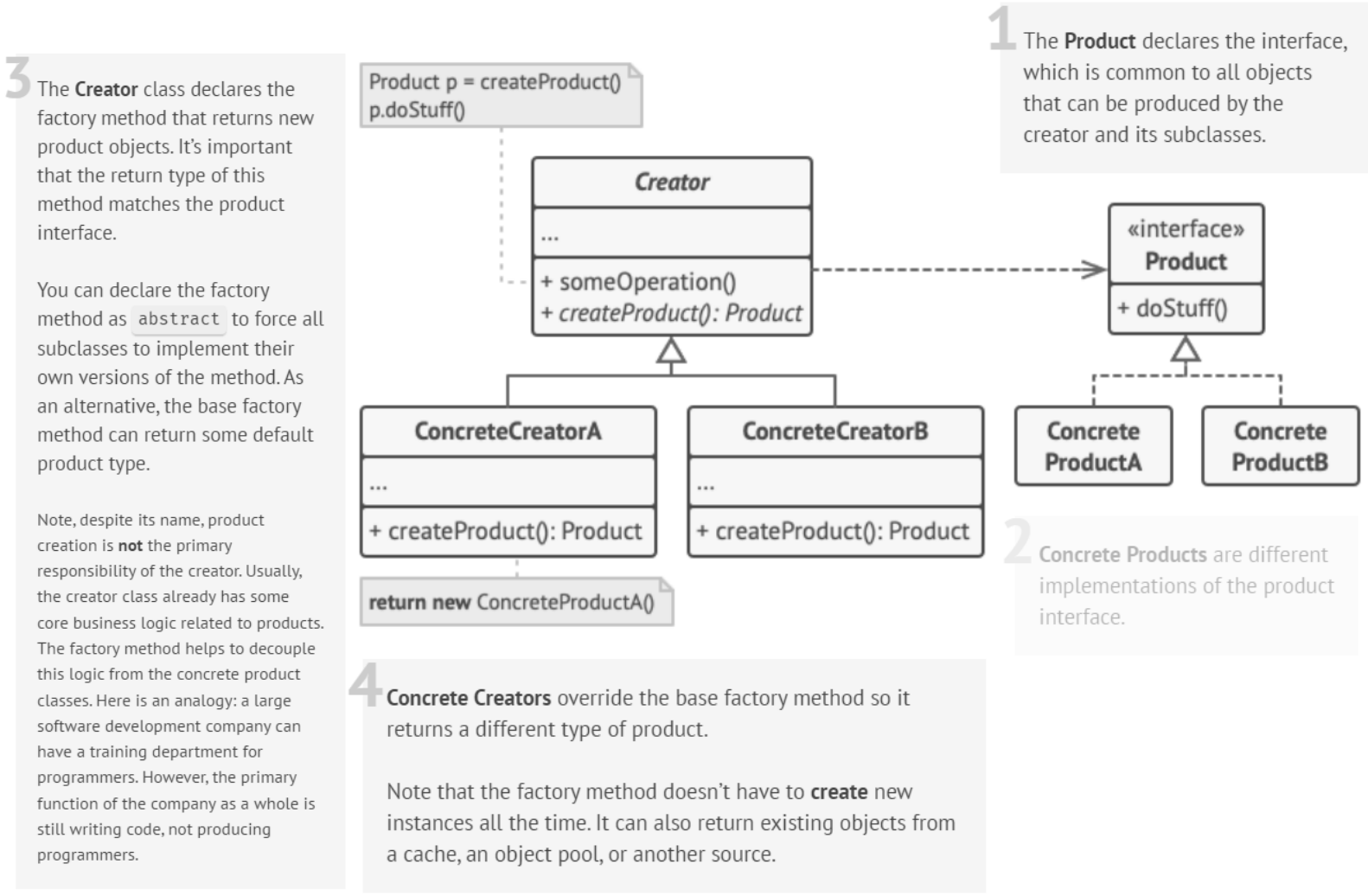
Subclasses can alter the class of objects being returned by the factory method.



All products must follow the same interface.

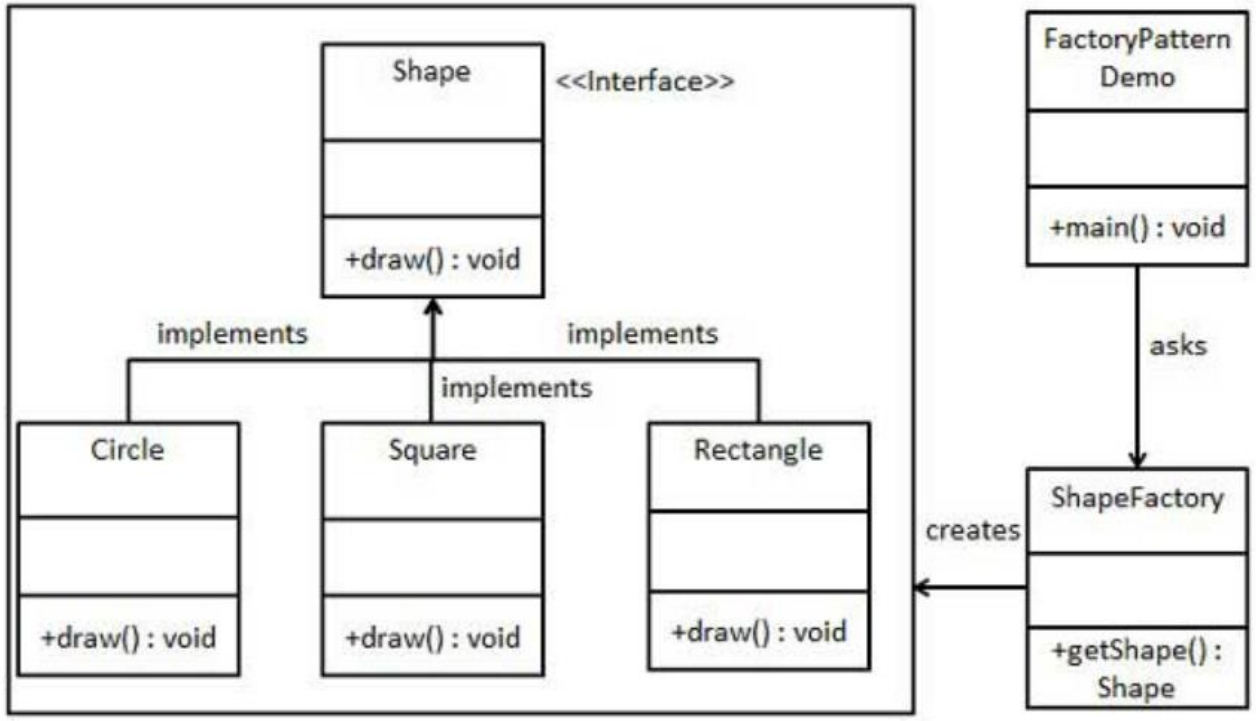
2.2 Factory Pattern (Continued)

❖ Structure



2.2 Factory Pattern (Continued)

❖ **Implementation**



➢ **Implementation –**

- We are going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.
- FactoryPatternDemo, our demo class, will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.

2.2 Factory Pattern (Continued)

❖ Implementation

Step 1

Create an interface.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

2.2 Factory Pattern (Continued)

❖ Implementation - Continued

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

2.2 Factory Pattern (Continued)

❖ **Extra**

- Super class in factory design pattern can be an interface, abstract class or a normal java class.
- The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.
- This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.
- Some important points about Factory Design Pattern method are;
  - We can keep Factory class Singleton or we can keep the method that returns the subclass as static.
  - Notice that based on the input parameter, different subclass is created and returned.

2.2 Factory Pattern (Continued)

❖ Applicability

Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.

The Factory Method separates product construction code from the code that actually uses the product. Therefore it's easier to extend the product construction code independently from the rest of the code.

For example, to add a new product type to the app, you'll only need to create a new creator subclass and override the factory method in it.

Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.

Inheritance is probably the easiest way to extend the default behavior of a library or framework. But how would the framework recognize that your subclass should be used instead of a standard component?

The solution is to reduce the code that constructs components across the framework into a single factory method and let anyone override this method in addition to extending the component itself.

Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

You often experience this need when dealing with large, resource-intensive objects such as database connections, file systems, and network resources.

2.2 Factory Pattern (Continued)

❖ Pros and Cons

- ✓ You avoid tight coupling between the creator and the concrete products.
- ✓ *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- ✓ *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.

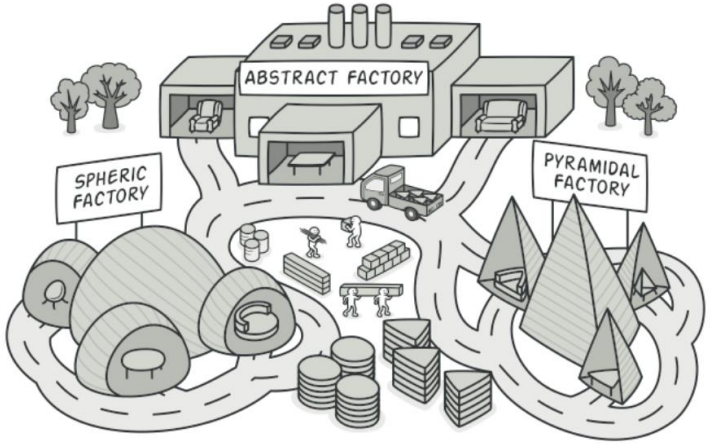
- ✗ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

2.3 Abstract Factory Pattern

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

❖ Introduction

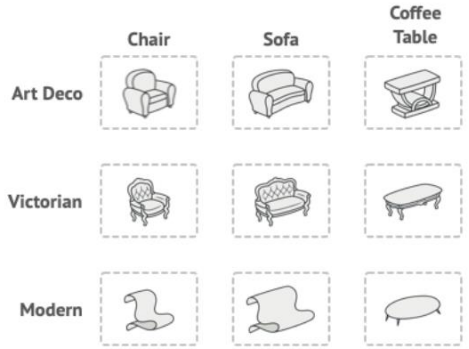
- Abstract Factory pattern is similar to Factory pattern and it’s a factory of factories.
- If you are familiar with the factory design pattern in java, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and the factory class uses if-else or switch statements to achieve this.
- In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class.



2.3 Abstract Factory Pattern (Continued)

❖ Problem

- Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:
  1. A family of related products, say: Chair + Sofa + CoffeeTable.
  2. Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco.
- You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.
- Also, you don't want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn't want to change the core code each time it happens.



Product families and their variants.

A Modern-style sofa doesn't match Victorian-style chairs.

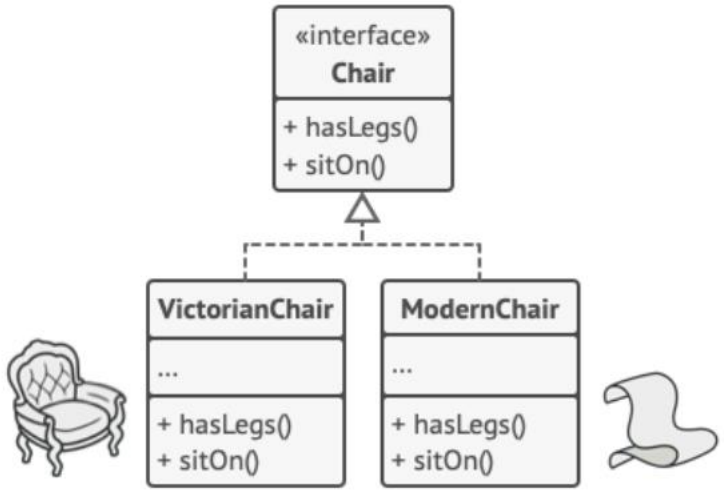
2.3 Abstract Factory Pattern (Continued)

❖ **Solution**

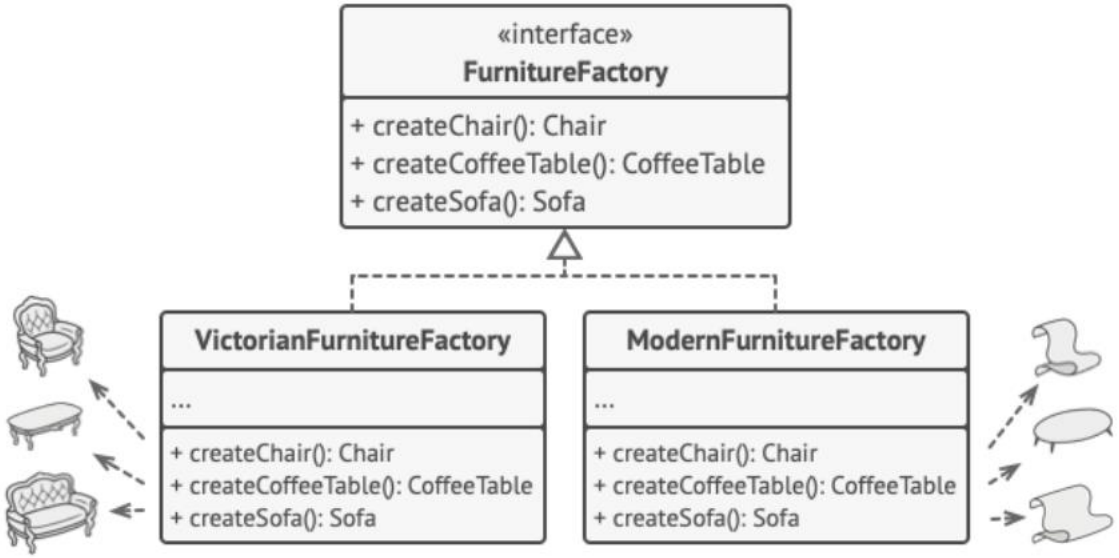
- The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces. For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.
- The next move is to declare the Abstract Factory—an interface with a list of creation methods for all products that are part of the product family (for example, createChair, createSofa and createCoffeeTable). These methods must return abstract product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable and so on.
- Now, how about the product variants? For each variant of a product family, we create a separate factory class based on the AbstractFactory interface. A factory is a class that returns products of a particular kind. For example, the ModernFurnitureFactory can only create ModernChair, ModernSofa and ModernCoffeeTable objects.
- The client code has to work with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.

2.3 Abstract Factory Pattern (Continued)

❖ **Solution - Continued**



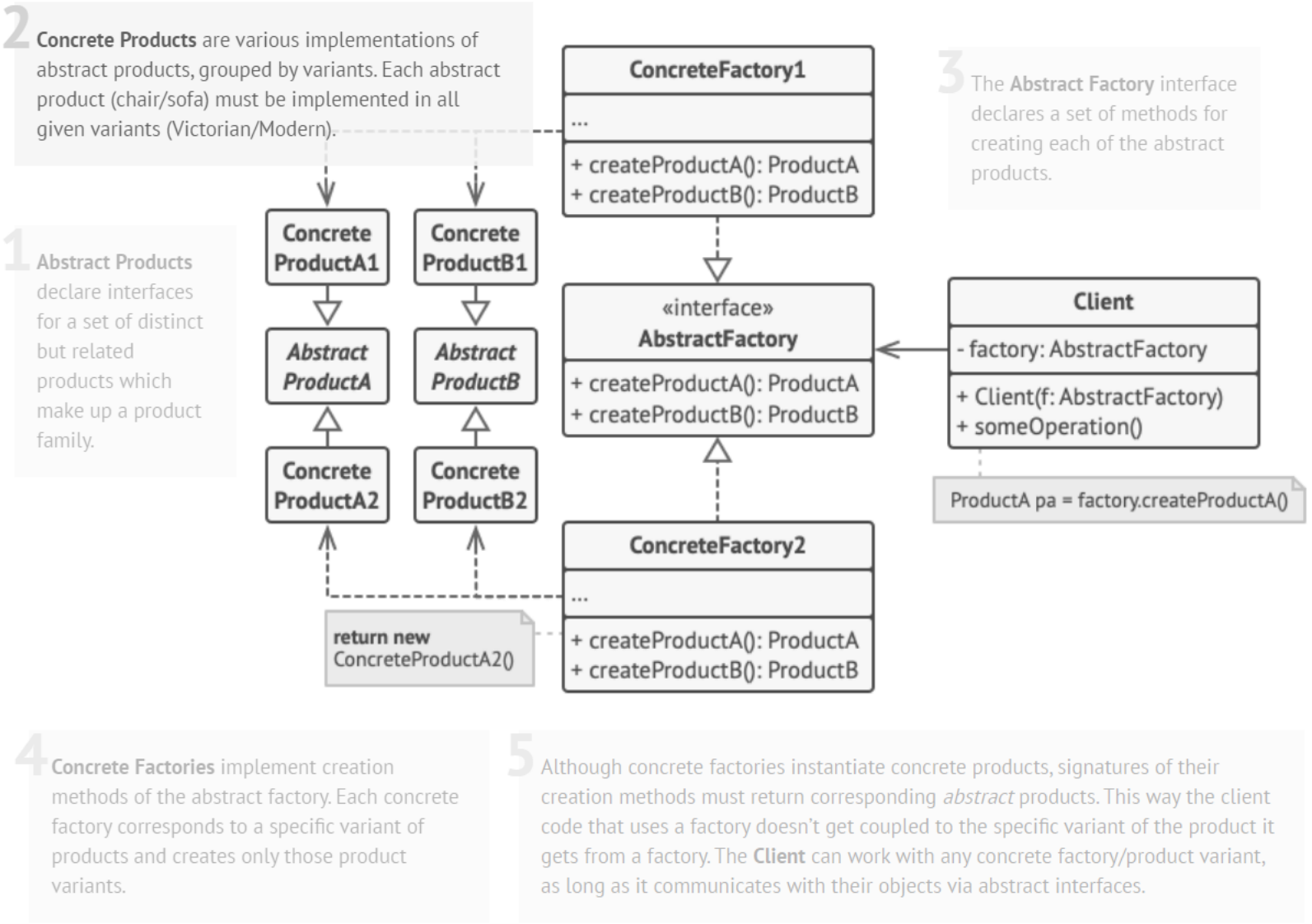
All variants of the same object must be moved to a single class hierarchy.



Each concrete factory corresponds to a specific product variant.

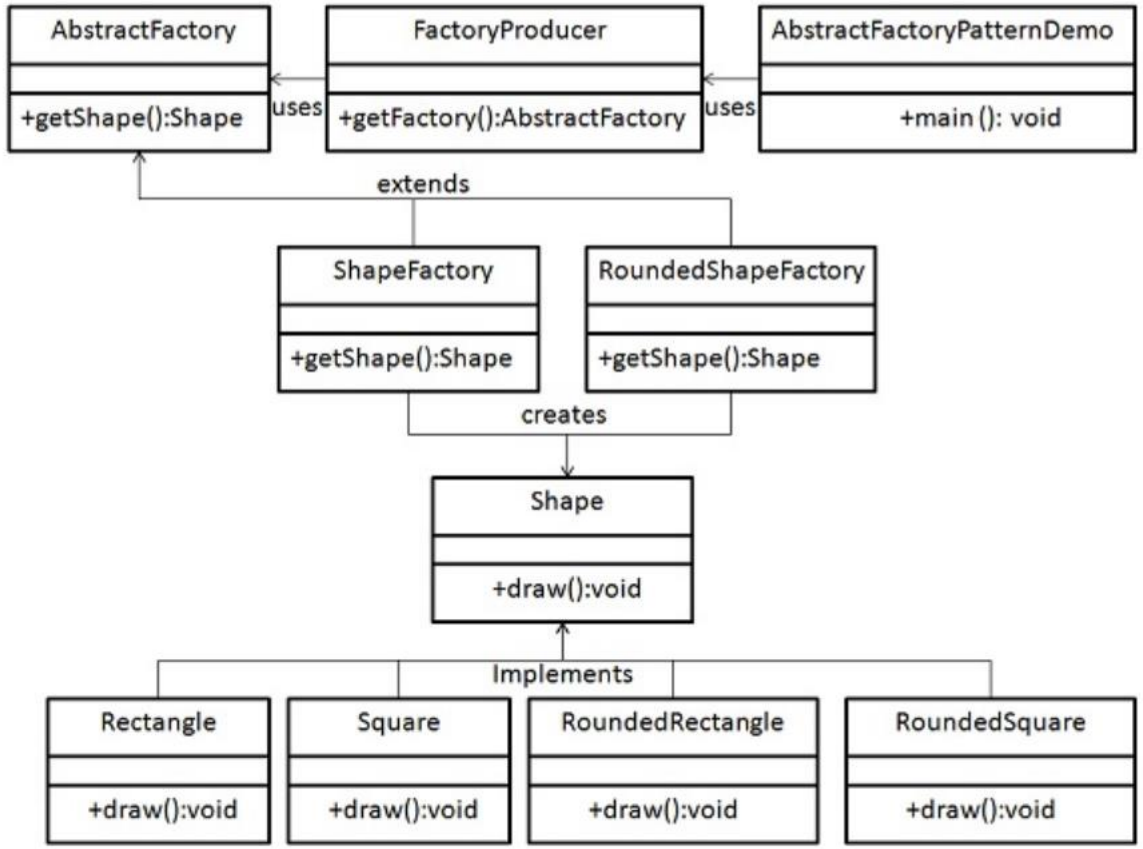
2.3 Abstract Factory Pattern (Continued)

- **Structure**



2.3 Abstract Factory Pattern (Continued)

❖ **Implementation**



➢ **Implementation –**

- We are going to create a Shape and Color interfaces and concrete classes implementing these interfaces. We create an abstract factory class AbstractFactory as next step. Factory classes ShapeFactory and ColorFactory are defined where each factory extends AbstractFactory. A factory creator/generator class FactoryProducer is created.
- AbstractFactoryPatternDemo, our demo class, uses FactoryProducer to get an AbstractFactory object. It will pass information (CIRCLE / RECTANGLE / SQUARE for Shape) to AbstractFactory to get the type of object it needs. It also passes information (RED / GREEN / BLUE for Color) to AbstractFactory to get the type of object it needs.

2.3 Abstract Factory Pattern (Continued)

❖ Implementation - Continued

Step 1

Create an interface for Shapes.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

RoundedRectangle.java

```
public class RoundedRectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedRectangle::draw() method.");
   }
}
```

RoundedSquare.java

```
public class RoundedSquare implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedSquare::draw() method.");
   }
}
```

Rectangle.java

```
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square class is missing

2.3 Abstract Factory Pattern (Continued)

❖ Implementation - Continued

Step 3

Create an Abstract class to get factories for Normal and Rounded Shape Objects.

AbstractFactory.java

```
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

Step 4

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

RoundedShapeFactory.java

```
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

2.3 Abstract Factory Pattern (Continued)

❖ Implementation - Continued

Step 5

Create a Factory generator/producer class to get factories by passing an information such as Shape

FactoryProducer.java

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

Verify the output.

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

Step 6

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
      //get an object of Shape Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
      //get an object of Shape Square
      Shape shape2 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape2.draw();
      //get shape factory
      AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
      //get an object of Shape Rectangle
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape3.draw();
      //get an object of Shape Square
      Shape shape4 = shapeFactory1.getShape("SQUARE");
      //call draw method of Shape Square
      shape4.draw();

   }
}
```

2.3 Abstract Factory Pattern (Continued)

❖ Applicability

Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

The Abstract Factory provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.

2.3 Abstract Factory Pattern (Continued)

❖ **Pros and Cons**

- ✓ You can be sure that the products you're getting from a factory are compatible with each other.
- ✓ You avoid tight coupling between concrete products and client code.
- ✓ *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
- ✓ *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.

- ✗ The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.

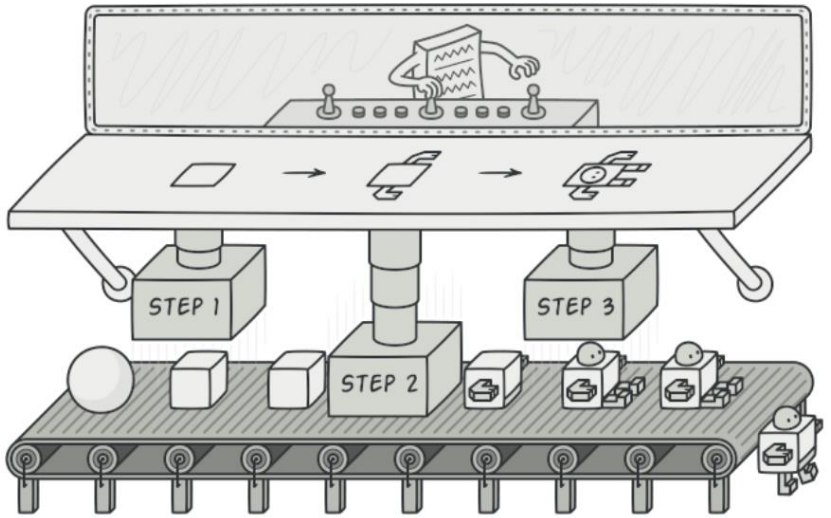
We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be **inconsistent** until unless all the attributes are set explicitly. Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

2.4 Builder Pattern

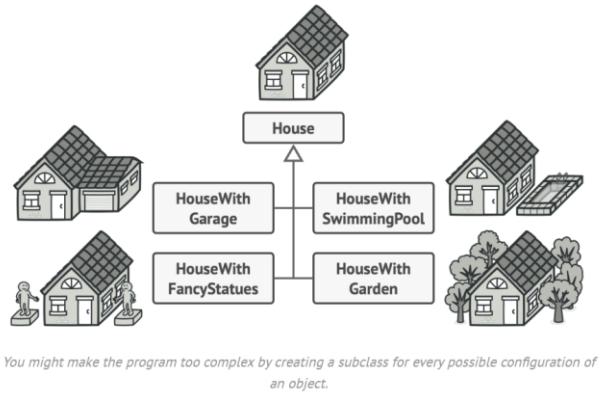
Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

❖ Introduction

- This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.
- Builder pattern solves the issue with a large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.



2.4 Builder Pattern (Continued)



You might make the program too complex by creating a subclass for every possible configuration of an object.

❖ Problem

- Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.

Example

- Let's think about how to create a House object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?
- The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.
- There's another approach that doesn't involve breeding subclasses. You can create a giant constructor right in the base House class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.
- In most cases most of the parameters will be unused, making the constructor calls pretty ugly.

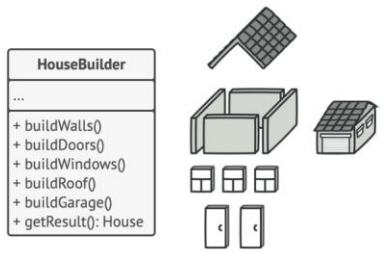
2.4 Builder Pattern (Continued)

❖ Solution

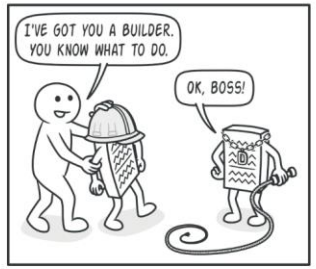
- The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called ***builders***.

In our example,

- Imagine a builder that builds everything from wood and glass, a second one that builds everything with stone and iron and a third one that uses gold and diamonds.
- By calling the same set of steps, you get a regular house from the first builder, a small castle from the second and a palace from the third.
- However, this would only work if the client code that calls the building steps is able to interact with builders using a common interface.
- You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called ***director***. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.



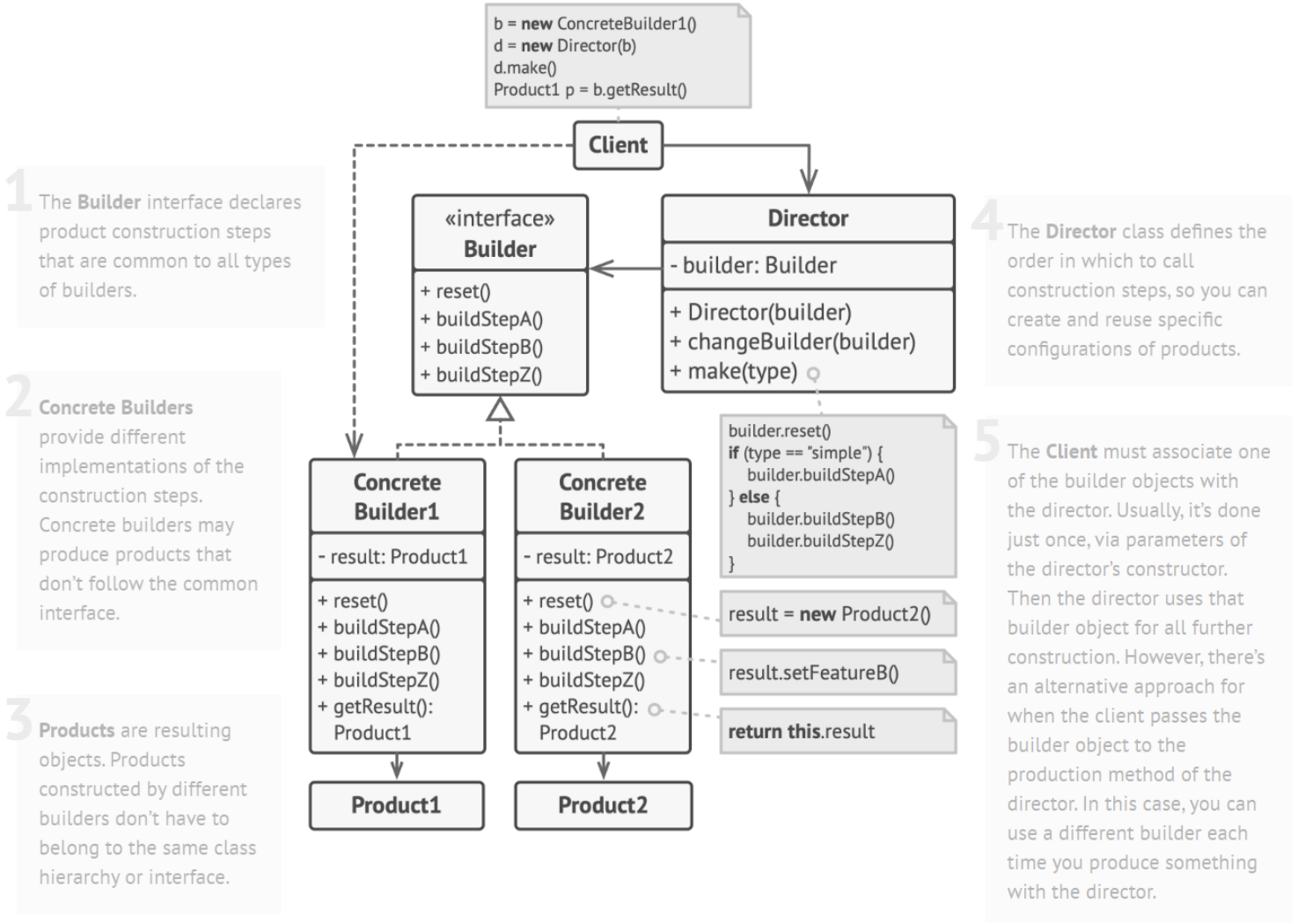
The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.



The director knows which building steps to execute to get a working product.

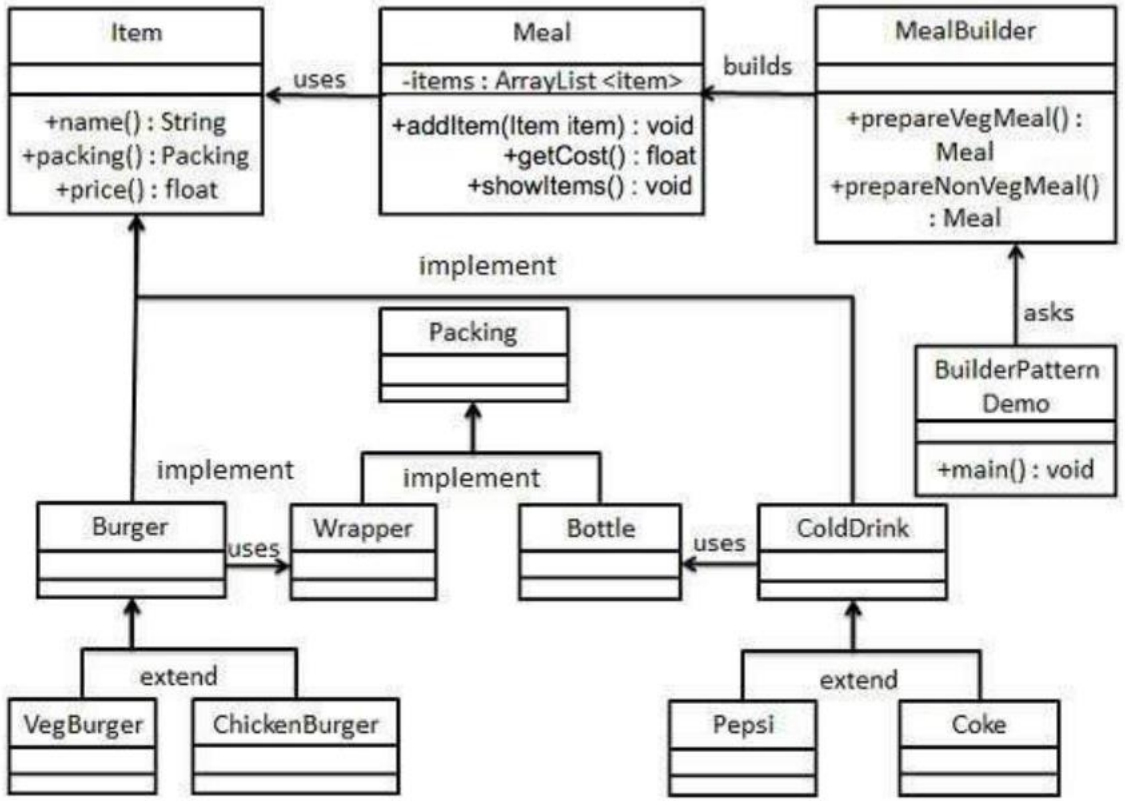
2.4 Builder Pattern (Continued)

❖ **Structure**



2.4 Builder Pattern (Continued)

❖ **Implementation**



➢ **Implementation –**

- We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.
- We are going to create an Item interface representing food items such as burgers and cold drinks and concrete classes implementing the Item interface and a Packing interface representing packaging of food items and concrete classes implementing the Packing interface as burger would be packed in wrapper and cold drink would be packed as bottle.
- We then create a Meal class having ArrayList of Item and a MealBuilder to build different types of Meal objects by combining Item. BuilderPatternDemo, our demo class, will use MealBuilder to build a Meal.

- **Step-1:** Create an interface Item representing food item and packing: **Item.java**, **Packing.java**
- **Step-2:** Create concrete classes implementing the Packing interface: **Wrapper.java, Bottle.java**
- **Step-3:** Create abstract classes implementing the item interface providing default functionalities: **Burger.java**, **ColdDrink.java**
- **Step-4:** Create concrete classes extending Burger and ColdDrink classes: **VegBurger.java**, **ChickenBurger.java**, **Coke.java**, **Pepsi.java**
- **Step-5:** Create a Meal class having Item objects defined above: **Meal.java**
- **Step-6:** Create a MealBuilder class, the actual builder class responsible to create Meal objects: **MealBuilder.java**
- **Step-7:** BuiderPatternDemo uses MealBuider to demonstrate builder pattern: **BuilderPatternDemo.java**
- **Step-8:** Verify the output.

2.4 Builder Pattern (Continued)

❖ Implementation - Continued

Step 1

Create an interface Item representing food item and packing.

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```
public interface Packing {
   public String pack();
}
```

Step 2

Create concrete classes implementing the Packing interface.

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

2.4 Builder Pattern (Continued)

❖ Implementation - Continued

Step 3

Create abstract classes implementing the item interface providing default functionalities.

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

Step 4

Create concrete classes extending Burger and ColdDrink classes

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

2.4 Builder Pattern (Continued)

❖ Implementation - Continued

Step 5

Create a Meal class having Item objects defined above.

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

Step 6

Create a MealBuilder class, the actual builder class responsible to create Meal objects.

MealBuilder.java

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

Step 7

BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

Verify the output.

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0


Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

2.4 Builder Pattern (Continued)

❖ Applicability

Use the Builder pattern to get rid of a "telescoping constructor".

Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters.

```
class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }
    // ...
```

Creating such a monster is only possible in languages that support method overloading, such as C# or Java.

The Builder pattern lets you build objects step by step, using only those steps that you really need. After implementing the pattern, you don't have to cram dozens of parameters into your constructors anymore.

Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).

The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.

The base builder interface defines all possible construction steps, and concrete builders implement these steps to construct particular representations of the product. Meanwhile, the director class guides the order of construction.

Use the Builder to construct Composite trees or other complex objects.

The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.

A builder doesn't expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.

2.4 Builder Pattern (Continued)

❖ Pros and Cons

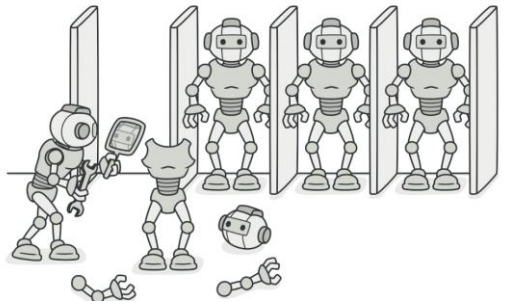
- ✓ You can construct objects step-by-step, defer construction steps or run steps recursively.
- ✓ You can reuse the same construction code when building various representations of products.
- ✓ *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.
- ✗ The overall complexity of the code increases since the pattern requires creating multiple new classes.

2.5 Prototype Pattern

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

❖ Introduction

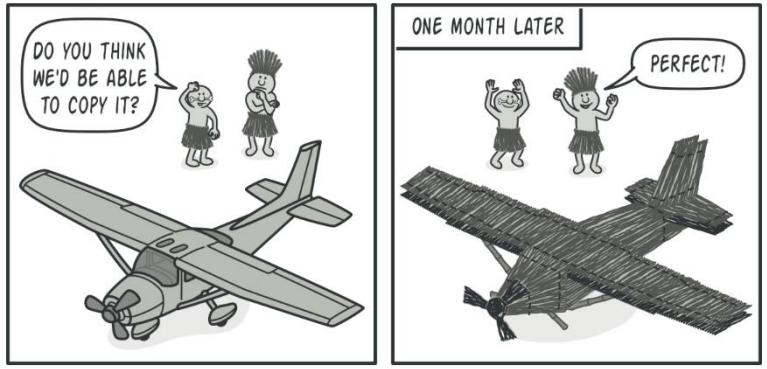
- The prototype pattern is used when the Object creation is **a costly affair and requires a lot of time and resources and you have a similar object already existing**.
- So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object.
- Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class.
- However whether to use the shallow or deep copy of the Object properties depends on the requirements and it's a design decision.



2.5 Prototype Pattern (Continued)

❖ Problem

- Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.
- Nice! But there's a catch. **Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.**
- There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes dependent on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete class, when, for example, a parameter in a method accepts any objects that follow some interface.



*Copying an object "from the outside" **isn't** always possible.*

2.5 Prototype Pattern (Continued)

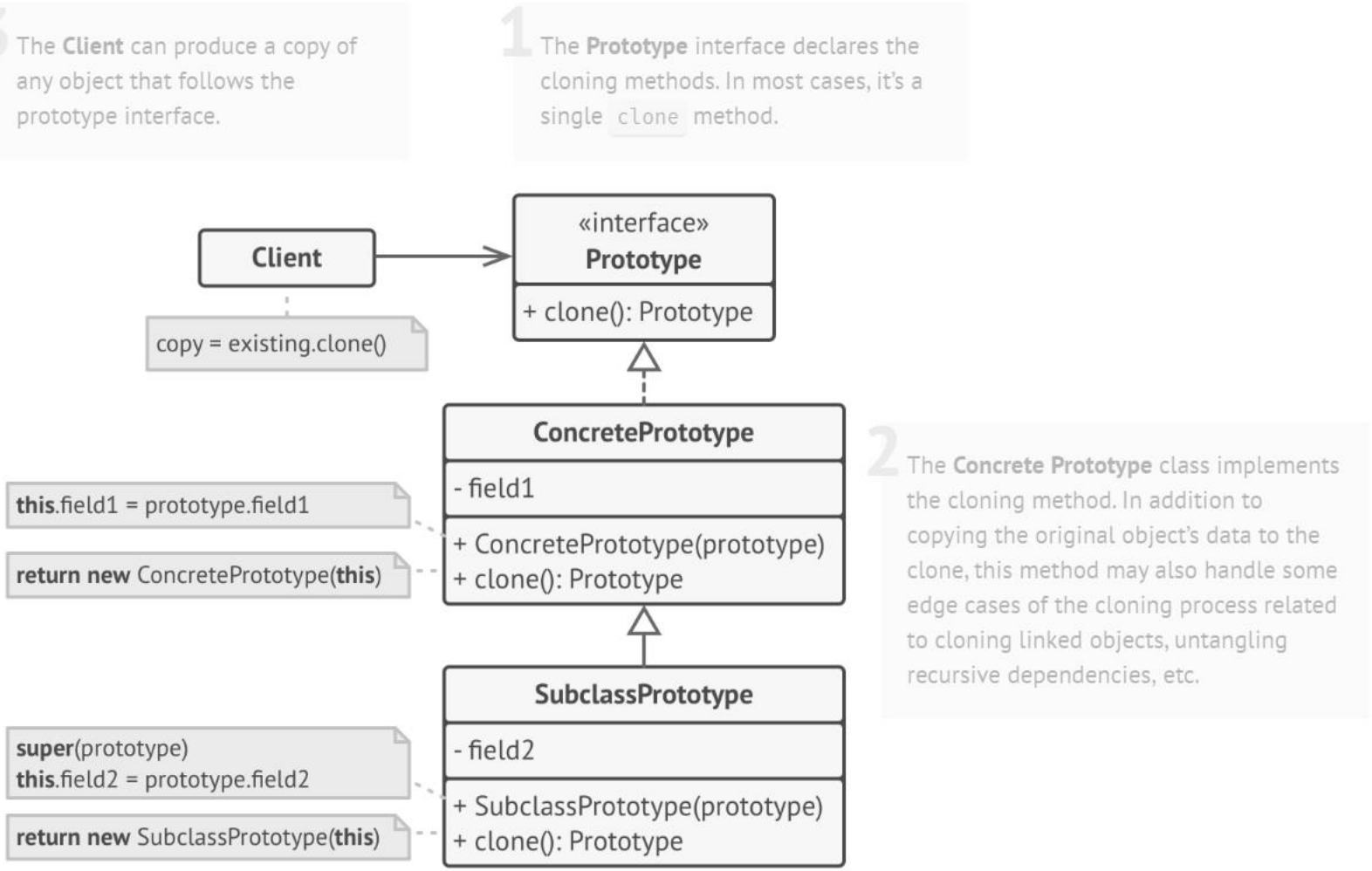
Pre-built prototypes can be an alternative to subclassing.

❖ Solution

- The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single clone method.
- The implementation of the clone method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.
- An object that supports cloning is called a prototype. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.
- Here's how it works: you create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch.
- Prototype pattern refers to creating duplicate object while keeping performance in mind.

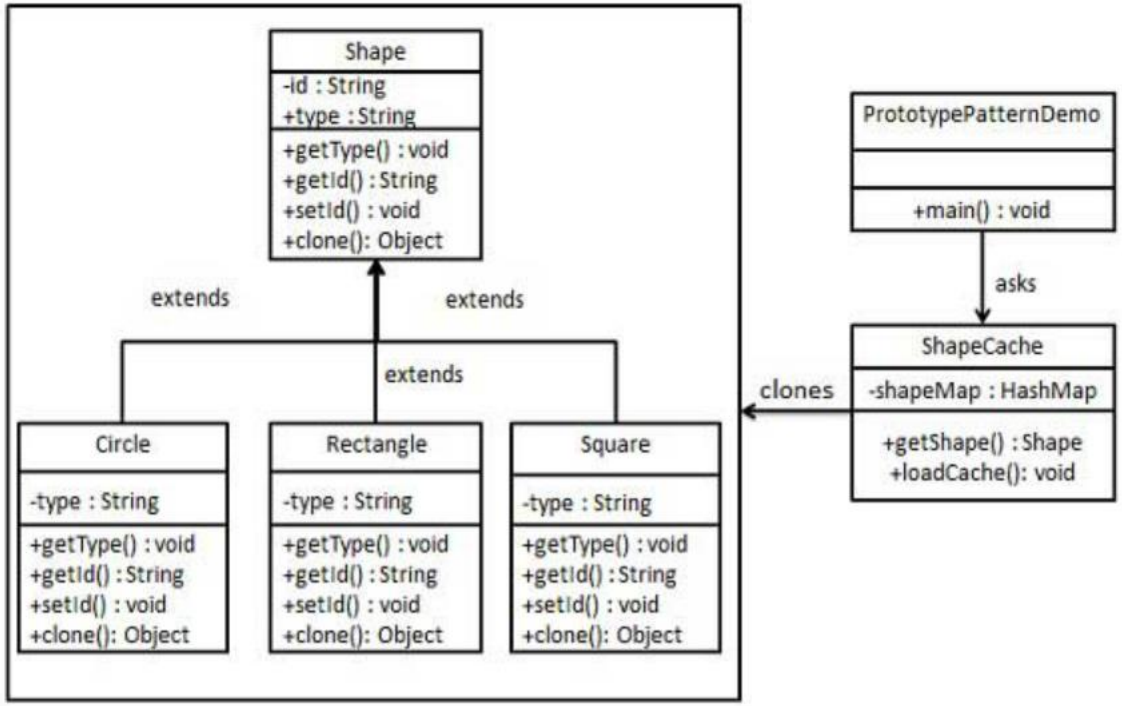
2.5 Prototype Pattern (Continued)

❖ **Structure**



2.5 Prototype Pattern (Continued)

❖ **Implementation**



➢ **Implementation –**

- We are going to create an abstract class Shape and concrete classes extending the Shape class. A class ShapeCache is defined as a next step which stores shape objects in a Hashtable and returns their clone when requested.

- **Step-1:** Create an abstract class implementing Clonable interface: **Shape.java**
- **Step-2:** Create concrete classes extending the above class: **Rectangle.java**, **Square.java**, **Circle.java**
- **Step-3:** Create a class to get concrete classes from database and store them in a Hashtable: **ShapeCache.java**
- **Step-4:** PrototypePatternDemo uses ShapeCache class to get clones of shapes stored in a Hashtable: **PrototypePatternDemo.java**
- **Step-5:** Verify the output.

2.5 Prototype Pattern (Continued)

❖ Implementation - Continued

Step 1

Create an abstract class implementing *Clonable* interface.

Shape.java

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Step 2

Create concrete classes extending the above class.

Rectangle.java

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

```
// The java.lang.Object class contains a clone() method that returns a bitwise copy of the current object.
// It particular only instances of classes that implement the Cloneable interface can be cloned.
// Trying to clone an object that does not implement the Cloneable interface throws
// a CloneNotSupportedException
```

2.5 Prototype Pattern (Continued)

❖ Implementation - Continued

Step 3

Create a class to get concrete classes from database and store them in a *Hashtable*.

ShapeCache.java

```
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

Step 4

PrototypePatternDemo uses *ShapeCache* class to get clones of shapes stored in a *Hashtable*.

PrototypePatternDemo.java

```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

Verify the output.

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

2.5 Prototype Pattern (Continued)

❖ Applicability

Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.

This happens a lot when your code works with objects passed to you from 3rd-party code via some interface. The concrete classes of these objects are unknown, and you couldn't depend on them even if you wanted to.

The Prototype pattern provides the client code with a general interface for working with all objects that support cloning. This interface makes the client code independent from the concrete classes of objects that it clones.

Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.

Suppose you have a complex class that requires a laborious configuration before it can be used. There are several common ways to configure this class, and this code is scattered through your app. To reduce the duplication, you create several subclasses and put every common configuration code into their constructors. You solved the duplication problem, but now you have lots of dummy subclasses.

The Prototype pattern lets you use a set of pre-built objects configured in various ways as prototypes. Instead of instantiating a subclass that matches some configuration, the client can simply look for an appropriate prototype and clone it.

2.5 Prototype Pattern (Continued)

❖ Pros and Cons

- ✓ You can clone objects without coupling to their concrete classes.
- ✓ You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- ✓ You can produce complex objects more conveniently.
- ✓ You get an alternative to inheritance when dealing with configuration presets for complex objects.
- ✗ Cloning complex objects that have circular references might be very tricky.

Recap…

Thank You!