2

Creational Design Patterns

Objectives

After completing this lesson, you should be able to do the following:

- Explore different Patterns available in Creational DP
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern
- Singleton Pattern

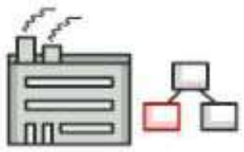
Course Roadmap

Java Design Patterns

- Lesson 1: Introduction to Design Patterns
- **Lesson02: Creational Design Patterns** — **You are here!**
- Lesson03: Structural Design Patterns
- Lesson04: Behavioral Design Patterns
- Lesson 5: Most Useful Design Patterns

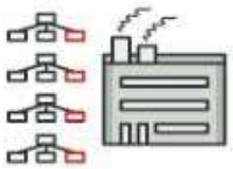
Creational Design Patterns

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



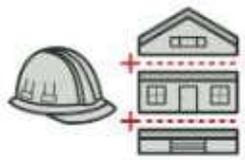
Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



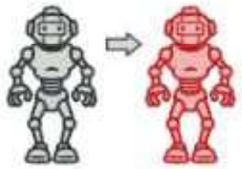
Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.



Builder

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.

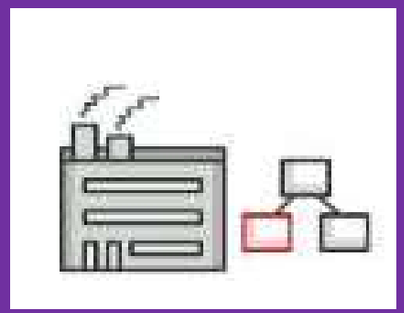


Prototype

Lets you copy existing objects without making your code dependent on their classes.

Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.



Factory Method

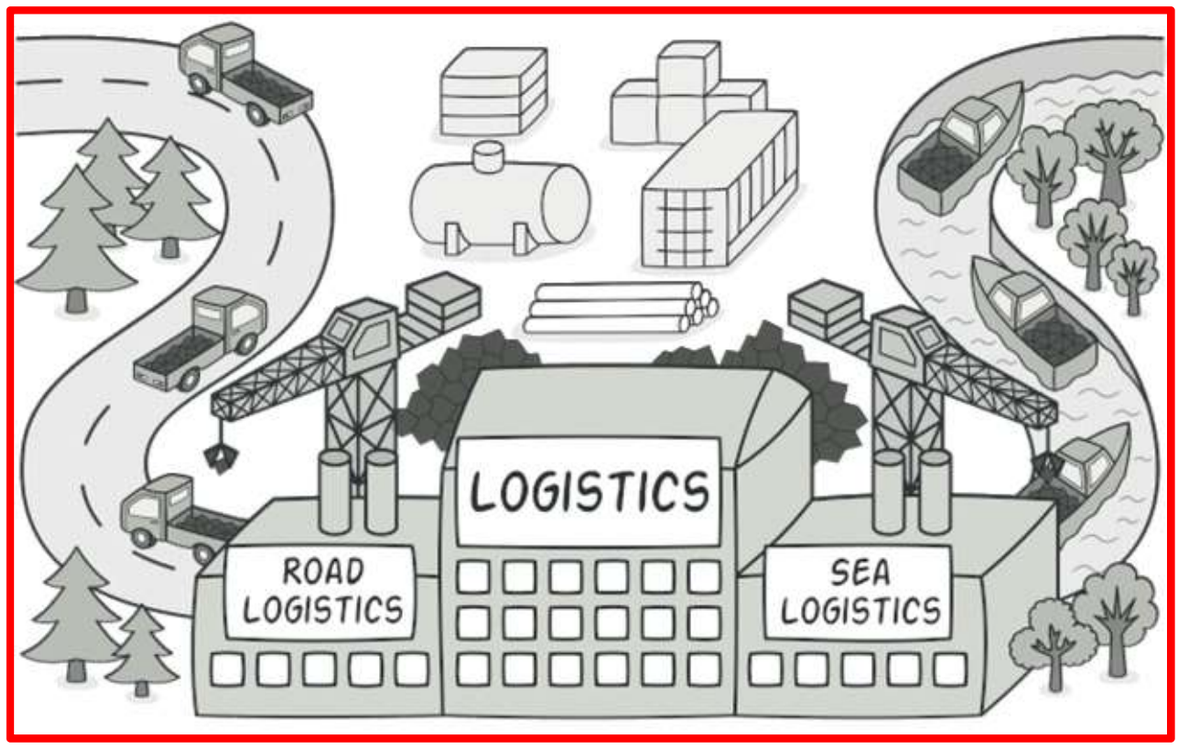
Introduction

- ***Factory method*** is a creational design pattern which solves the problem of creating product objects without specifying their concrete classes.

- The Factory Method defines a method, which should be used for creating objects instead of using a direct constructor call (new operator). Subclasses can override this method to change the class of objects that will be created.

Factory Method [Also known as: Virtual Constructor]

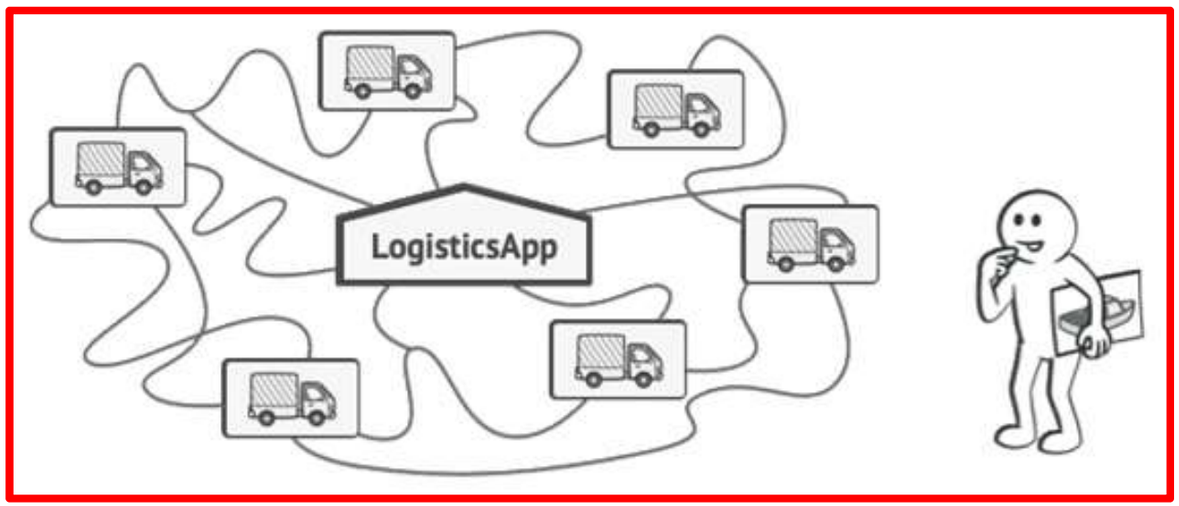
- **Intent**
  - **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



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.



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

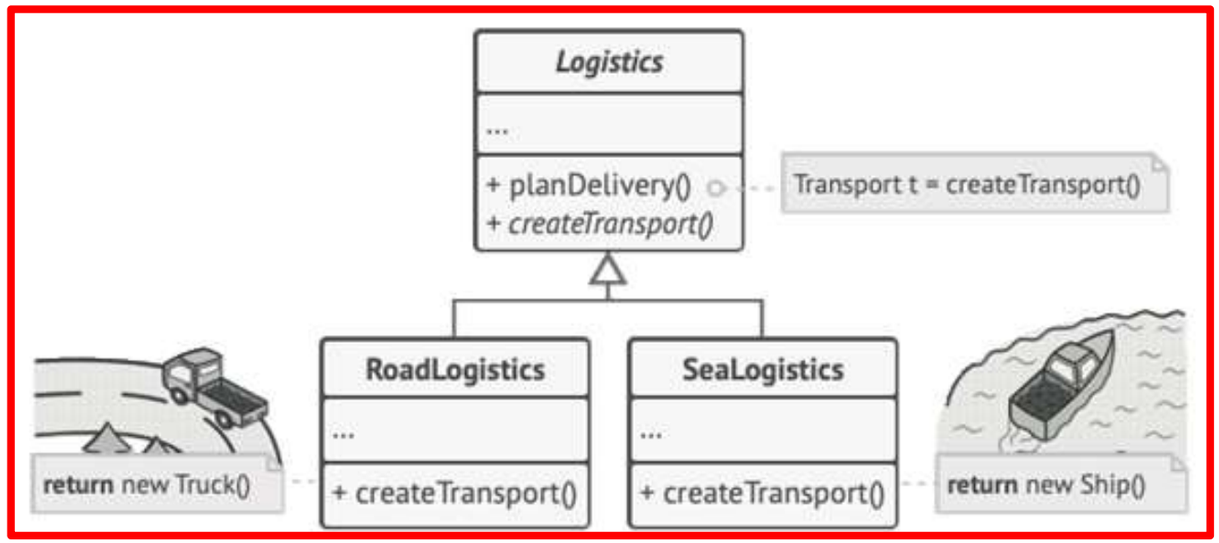
- Great news, right? But how about the code? At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of 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.

Solution

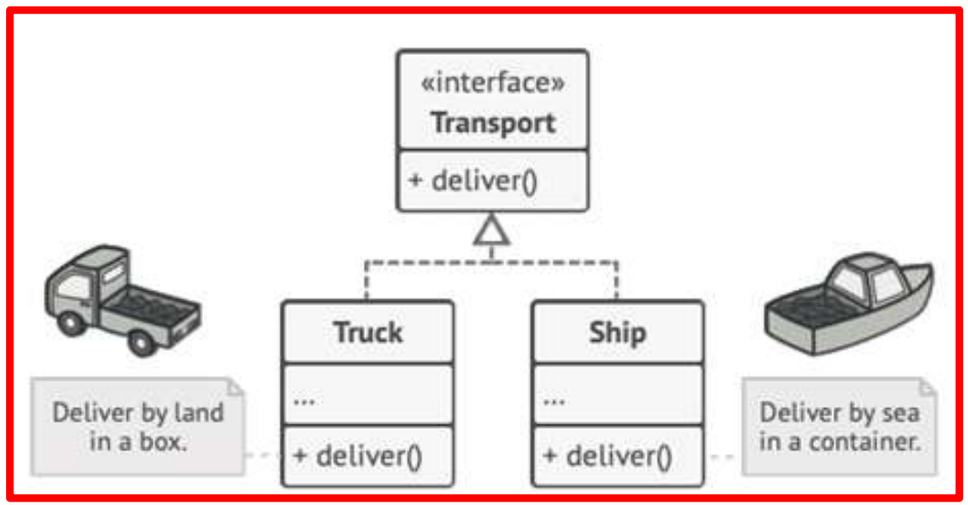
- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.

- 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.

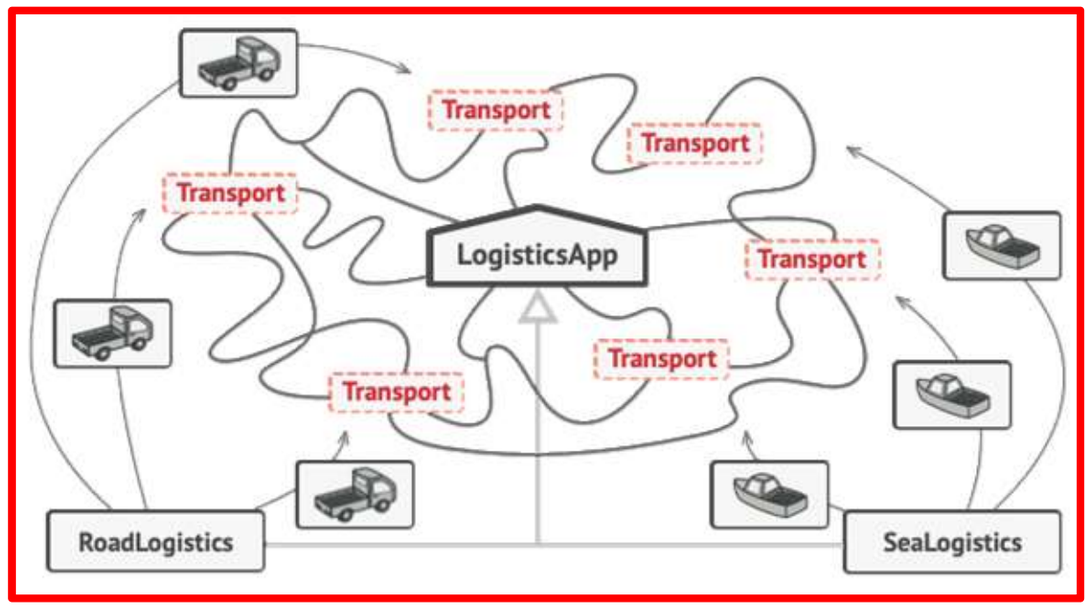


- *Subclasses can alter the class of objects being returned by the factory method.*

- For example, both Truck and Ship classes should implement the Transport interface, which declares a method called deliver.
- Each class implements this method differently: trucks deliver cargo by land, ships deliver cargo by sea.
- The factory method in the ***RoadLogistics*** class returns **truck** objects, whereas the factory method in the ***SeaLogistics*** class returns **ships**.



- The code that uses the factory method (often called the client code) doesn't see a difference between the actual products returned by various subclasses.
- The client treats all the products as abstract Transport. The client knows that all transport objects are supposed to have the deliver method, but exactly how it works isn't important to the client.



- *As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.*

Usage in Java

Complexity: ★☆☆

Popularity: ★★★

Usage examples: The Factory Method pattern is widely used in Java code. It's very useful when you need to provide a high level of flexibility for your code.

The pattern is present in core Java libraries:

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`
- `java.net.URLStreamHandlerFactory#createURLStreamHandler(String)` (Returns different singleton objects, depending on a protocol)
- `java.util.EnumSet#of()`
- `javax.xml.bind.JAXBContext#createMarshaller()` and other similar methods.

Example

```
public abstract class WorkStation {

    public abstract String getRAM();
    public abstract String getSSD();
    public abstract String getCPU();

    @Override
    public String toString() {
       return "Details [ RAM : " + this.getRAM() + ", SSD : " + this.getSSD() +
              ", CPU : " + this.getCPU() + " ]";
    }

}
```

Example

```
public class Laptop extends WorkStation {

    private String ram;
    private String ssd;
    private String cpu;

    public Laptop() {

    }

    public Laptop(String ram, String ssd, String cpu) {
        this.ram = ram;
        this.ssd = ssd;
        this.cpu = cpu;
    }
```

Example

```
@Override
        public String getRAM() {
            return this.ram;
        }

        @Override
        public String getSSD() {
            return this.ssd;
        }

        @Override
        public String getCPU() {
            return this.cpu;
        }

}
```

Example

```
public class Server extends WorkStation {

    private String ram;
    private String ssd;
    private String cpu;

    public Server() {

    }

    public Server(String ram, String ssd, String cpu) {
        this.ram = ram;
        this.ssd = ssd;
        this.cpu = cpu;
    }
```

Example

```
@Override
        public String getRAM() {
            return this.ram;
        }

        @Override
        public String getSSD() {
            return this.ssd;
        }

        @Override
        public String getCPU() {
            return this.cpu;
        }

}
```

Example

```
public class WorkStationFactory {

    // Factory Method [ Factory is a Place Where Products are Created ],
    // In Java Context [ Factory method Creates the Requried Objects ]
    public static WorkStation getWorkStation(String givenType, String givenRam,
                                             String givenSsd, String givenCpu) {

        if("Laptop".equalsIgnoreCase(givenType)) {
            return new Laptop(givenRam, givenSsd, givenCpu);
        }
        else if("Server".equalsIgnoreCase(givenType)) {
            return new Server(givenRam, givenSsd, givenCpu);
        }
        return null;
    }
}
```

Example

```
public class FactoryMethodPatternTest {

    public static void main(String[] args) {

        //Laptop laptopRef = new Laptop();
        WorkStation laptopRef = WorkStationFactory
                .getWorkStation("Laptop", "8 GB", "480 GB", "2.4 GHz");

        WorkStation serverRef = WorkStationFactory
                .getWorkStation("Server", "32 GB", "1000 TB", "3.0 GHz");

        System.out.println("Factory Laptop Config Details  : " + laptopRef);
        System.out.println("Factory Server Configt Details : " + serverRef);

    }
}
```

Applicability

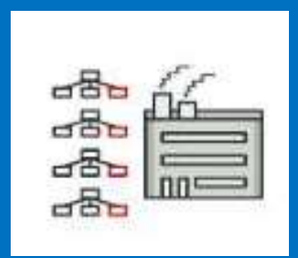
1. Use the Factory Method when you don’t know beforehand the exact types and dependencies of the objects your code should work with.

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

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

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.



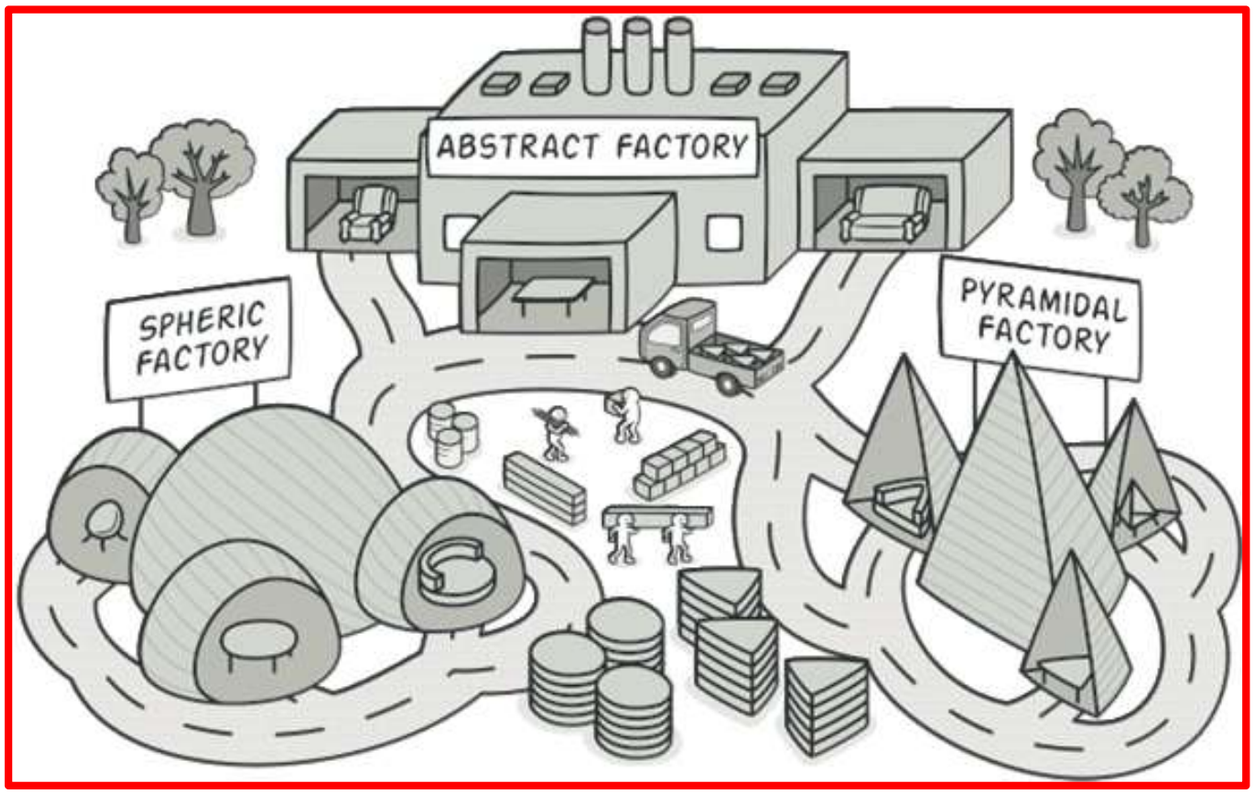
Abstract Factory Method

Introduction

- **Abstract Factory** is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes.
- The abstract factory pattern is similar to the factory pattern and is a factory of factories.
- In **Factory Method** class that returns the different subclasses based on the input provided and the factory class uses if-else or switch statements to achieve this.
- In the abstract factory pattern, we get rid of if-else block and have a factory class for each subclass and then an abstract factory class that will return the subclass based on the input factory class.

Intent

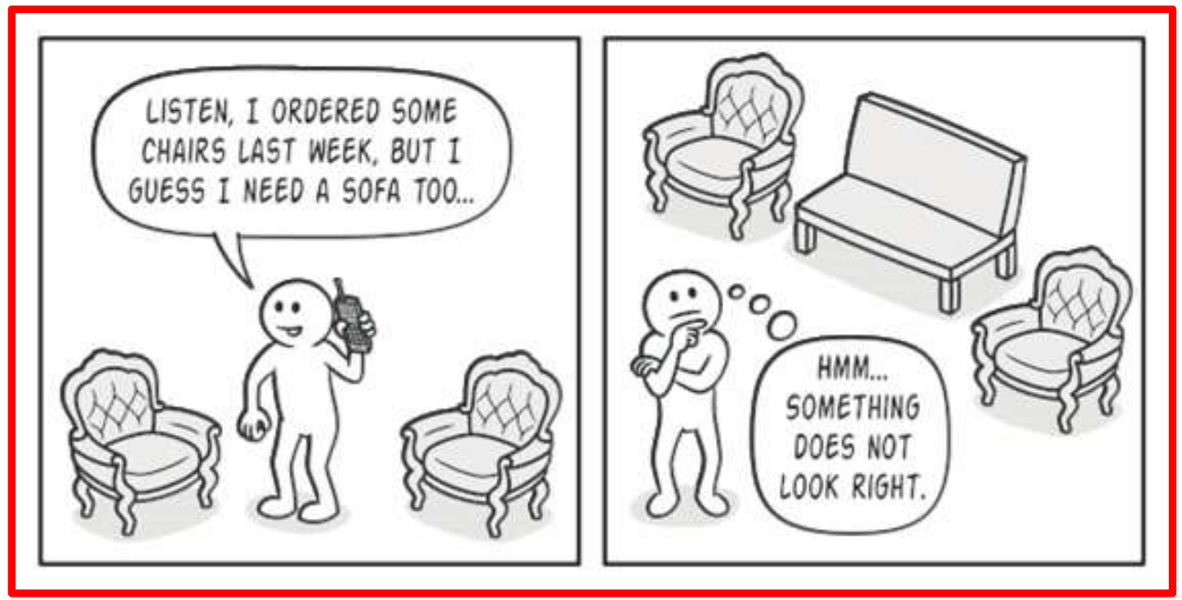
- **Abstract Factory** is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.



Problem

- Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:
- A family of related products, say: Chair + Sofa + CoffeeTable.
- Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco.

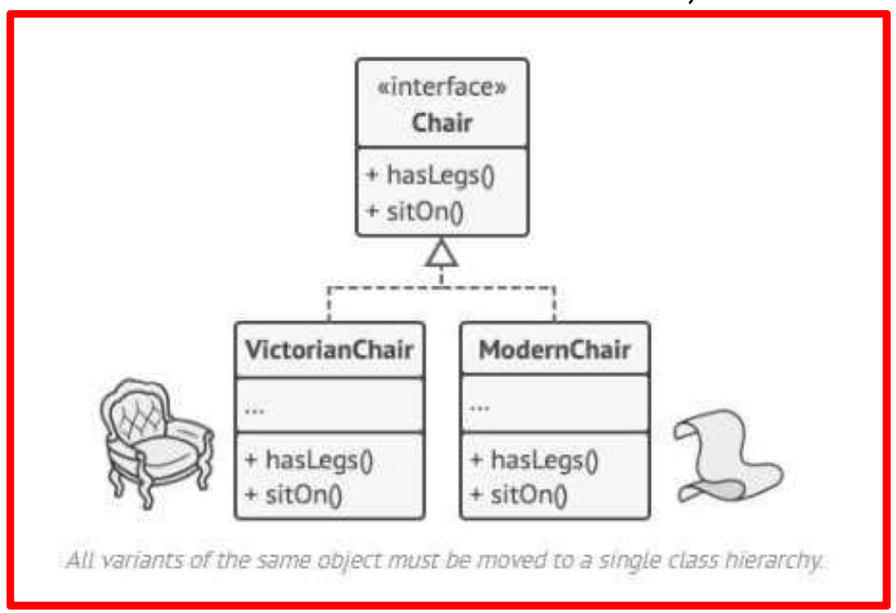
- You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.



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

Solution

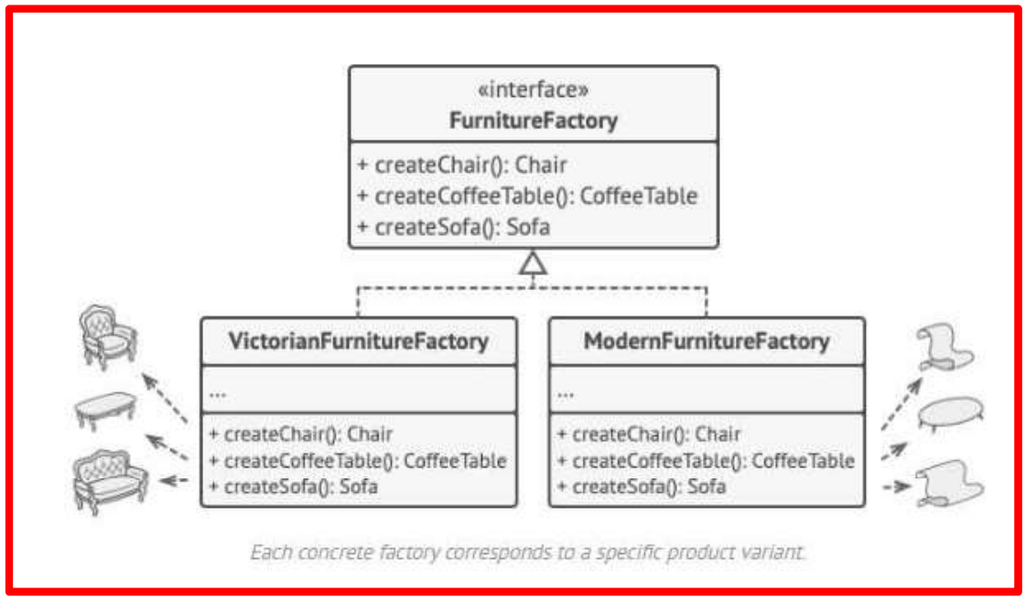
- The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table).
- Then you can make all variants of products follow those interfaces. For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.



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

Next Step

- 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.



Usage in Java

Complexity: ★★☆

Popularity: ★★★

Usage examples: The Abstract Factory pattern is pretty common in Java code. Many frameworks and libraries use it to provide a way to extend and customize their standard components.

Here are some examples from core Java libraries:

- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

Identification: The pattern is easy to recognize by methods, which return a factory object. Then, the factory is used for creating specific sub-components.

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

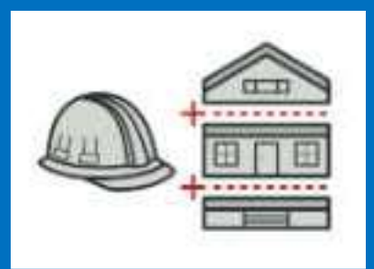
Applicability

1. 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.

2. Consider implementing the Abstract Factory when you have a class with a set of Factory Methods that blur its primary responsibility.

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

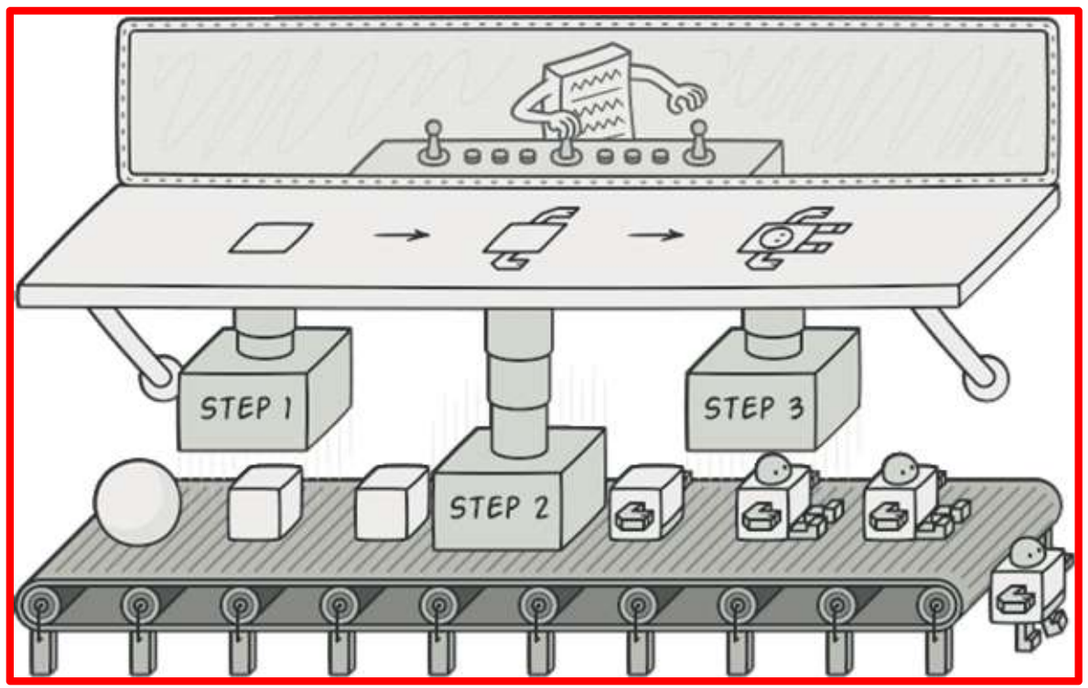
Introduction

- **Builder** is a creational design pattern, which allows constructing complex objects step by step.

- Unlike other creational patterns, Builder doesn't require products to have a common interface. That makes it possible to produce different products using the same construction process.

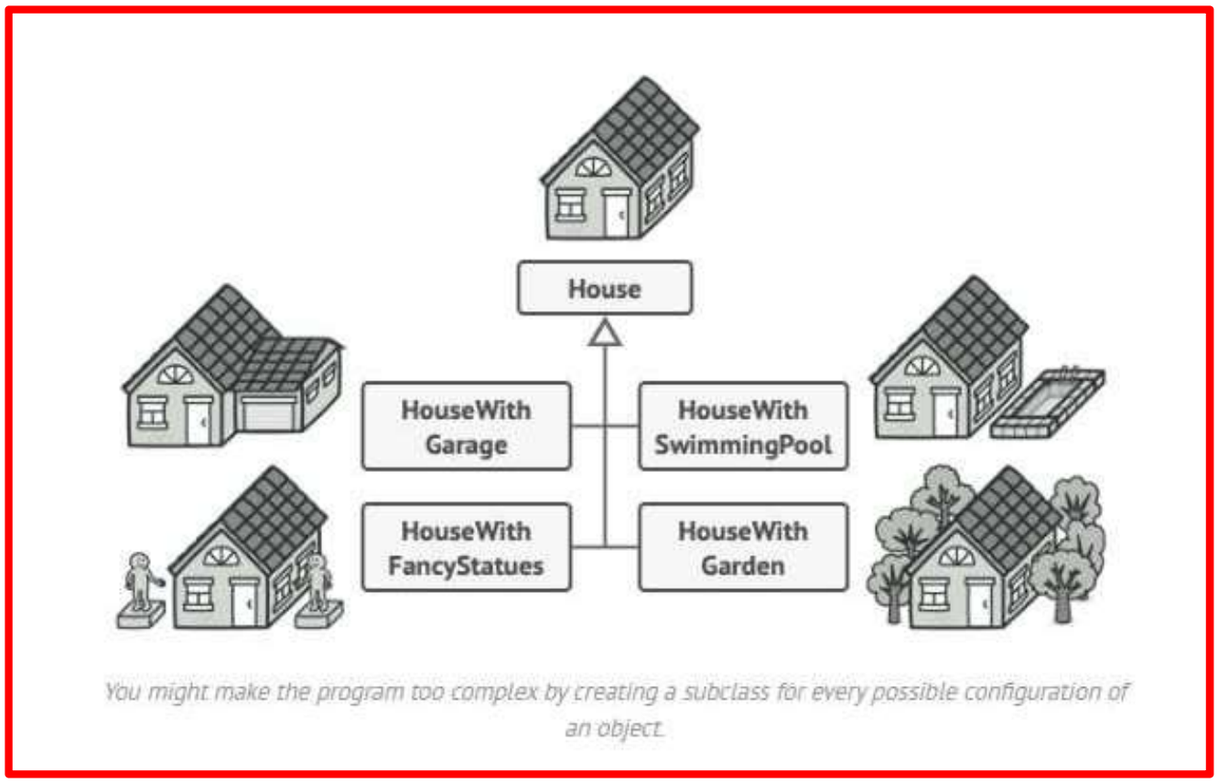
Intent

- **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.



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.

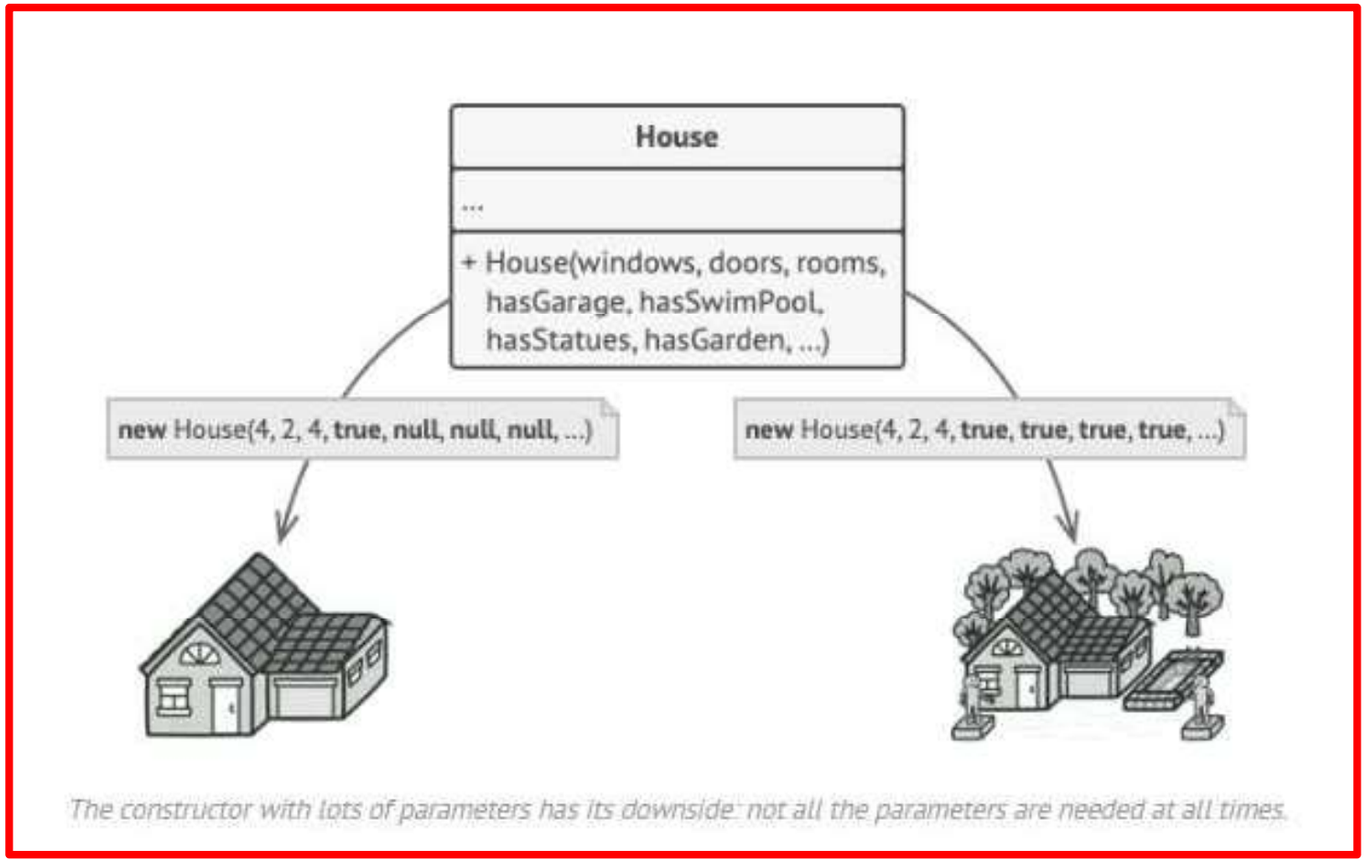


You might make the program too complex by creating a subclass for every possible configuration of an object.

- For 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)?

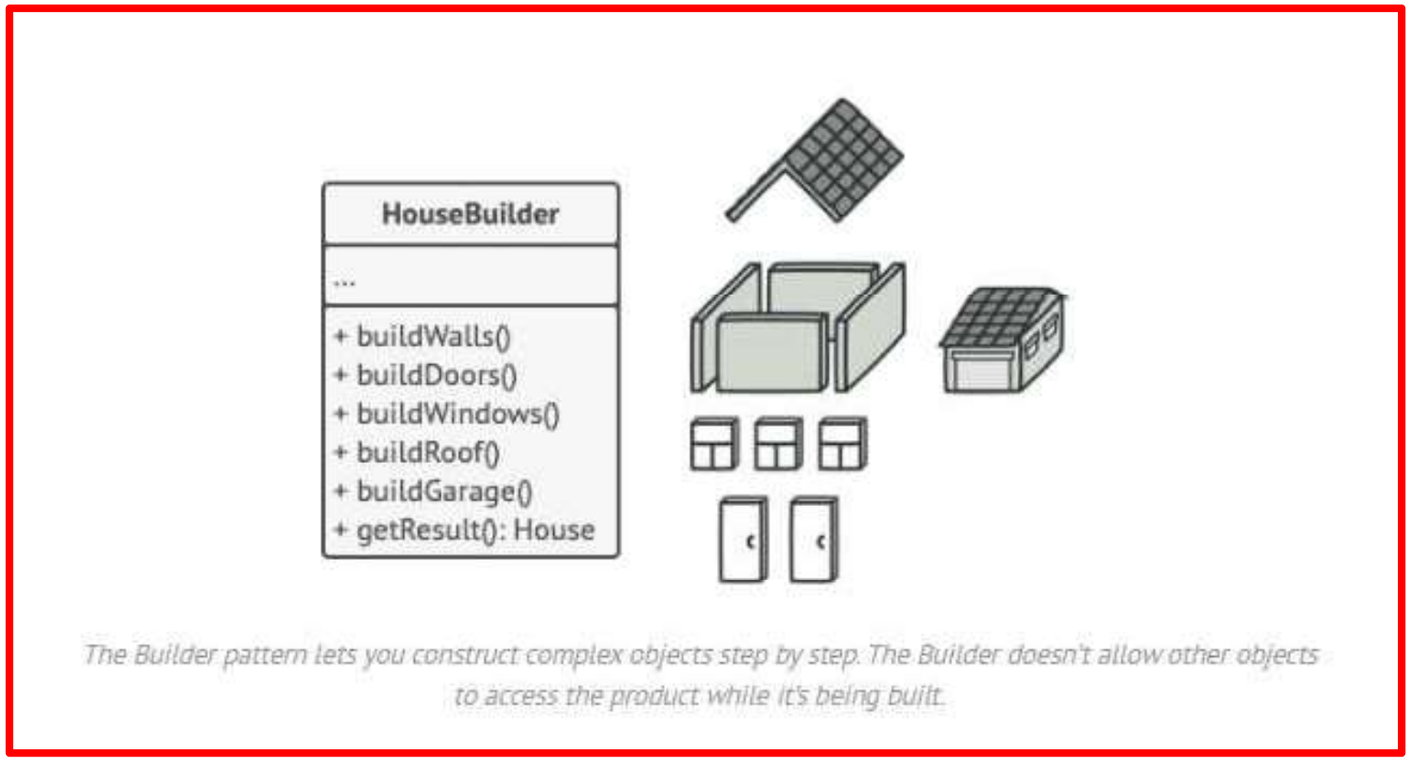
- The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.

- 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.



The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

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



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.

Usage in Java

Complexity: ★★☆

Popularity: ★★★

Usage examples: The Builder pattern is a well-known pattern in Java world. It's especially useful when you need to create an object with lots of possible configuration options.

Builder is widely used in Java core libraries:

- `java.lang.StringBuilder#append()` (`unsynchronized`)
- `java.lang.StringBuffer#append()` (`synchronized`)
- `java.nio.ByteBuffer#put()` (also in `CharBuffer` , `ShortBuffer` , `IntBuffer` , `LongBuffer` , `FloatBuffer` and `DoubleBuffer`)
- `javax.swing.GroupLayout.Group#addComponent()`
- All implementations `java.lang.Appendable`

Identification: The Builder pattern can be recognized in a class, which has a single creation method and several methods to configure the resulting object. Builder methods often support chaining (for example, `someBuilder.setValueA(1).setValueB(2).create()`).

Applicability

1. Use the Builder pattern to get rid of a “telescoping constructor”.

2. 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).

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.

Summary

In this lesson, you should have learned how to:

- Explore different Patterns available in Creational DP
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern
- Singleton Pattern