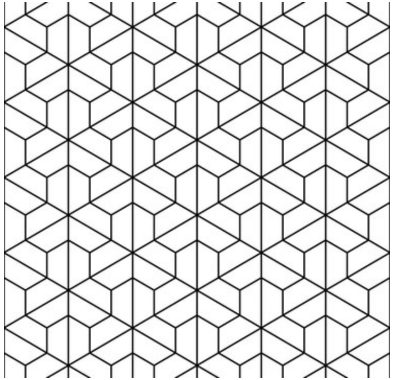
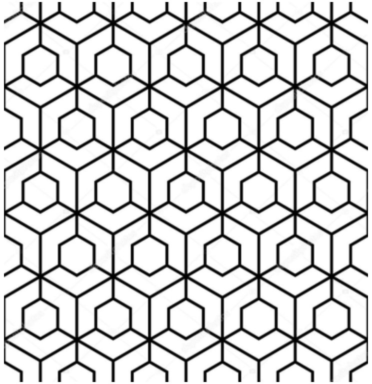
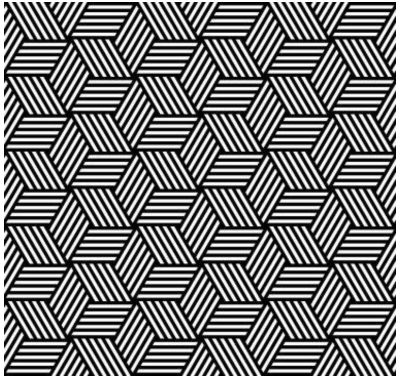
Design Patterns
(credit refactoring.guru)

Patterns





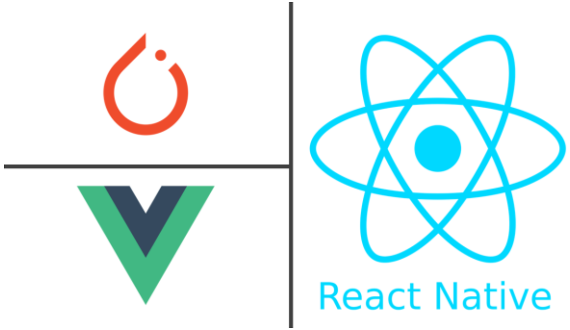


What is a Design Pattern?

A design pattern is a general **repeatable** solution to a **commonly occurring** problem in software design. It's a description or **template** for how to solve a problem that can be used in different situations. It's not a finished design that can be turned directly into code but a guide for solving specific problems in a particular way.

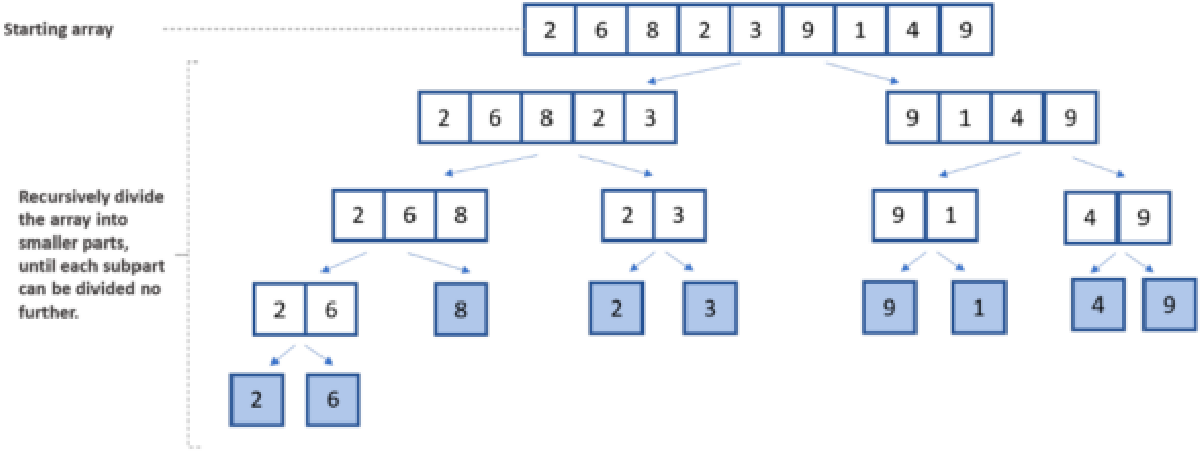
What is NOT a Design Pattern?

- **Frameworks**: A software framework is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications. Unlike design patterns, which are more abstract and used to solve design problems, frameworks are **concrete implementations** used to solve a specific type of problem or provide a specific type of functionality.



What is NOT a Design Pattern? (Cont.)

- **Algorithms**: An algorithm is a step-by-step procedure to solve a specific problem, whereas a design pattern is a general solution to a design problem. Algorithms are about computation and how to perform tasks, while design patterns are about design and structure.

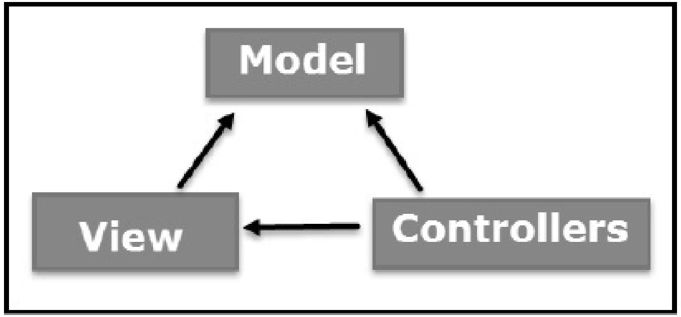


Starting array

Recursively divide the array into smaller parts, until each subpart can be divided no further.

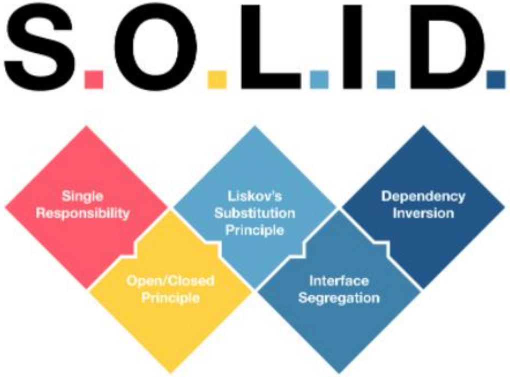
What is NOT a Design Pattern? (Cont.)

- **Architectural Patterns**: Architectural patterns are patterns for the overall layout of an application, defining the high-level structure of the software. Examples include MVC (Model-View-Controller), MVVM (Model-View-ViewModel), and Microservices. On the other hand, design patterns are lower level, providing solutions to common problems in software design.



What is NOT a Design Pattern? (Cont.)

- **Principles and Concepts**: Principles and concepts like SOLID, DRY (Don't Repeat Yourself), and KISS (Keep It Simple, Stupid) are often confused with design patterns. These are general guidelines used in software development, while design patterns are more specific and provide solutions to common design problems.



Why use design pattern?

- **Reusable**: Solutions that can be re-applied to common problems in software design. Reduce the time and effort required to solve similar problems.
- **Communicate Clearly**: Design patterns create a standard terminology, enabling developers to communicate more efficiently.
- **Best Practices:** By using design patterns, you're leveraging the collective experience of skilled software developers. They encapsulate best practices based on these developers' experiences.

Why use design pattern(Cont.)

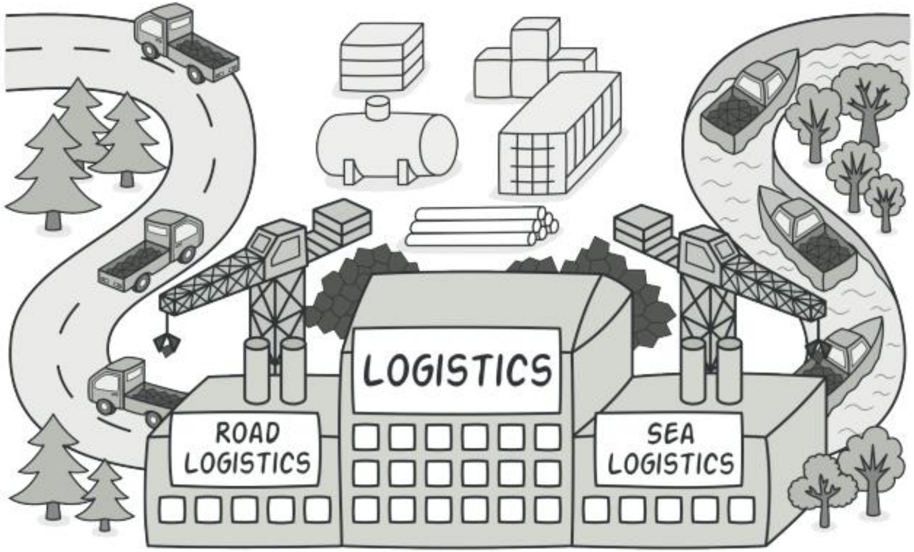
- **Improves Code Readability**: Patterns can often make the system easier to understand and maintain because they represent solutions that have been refined and improved over time.
- **Scalability**: Design patterns often allow code to be more easily extended or scaled, making the system more robust and adaptable to change.
- **Avoids Common Pitfalls**: Because design patterns represent solutions to common problems, they also help developers avoid common pitfalls in software design.

Basic Types of patterns.

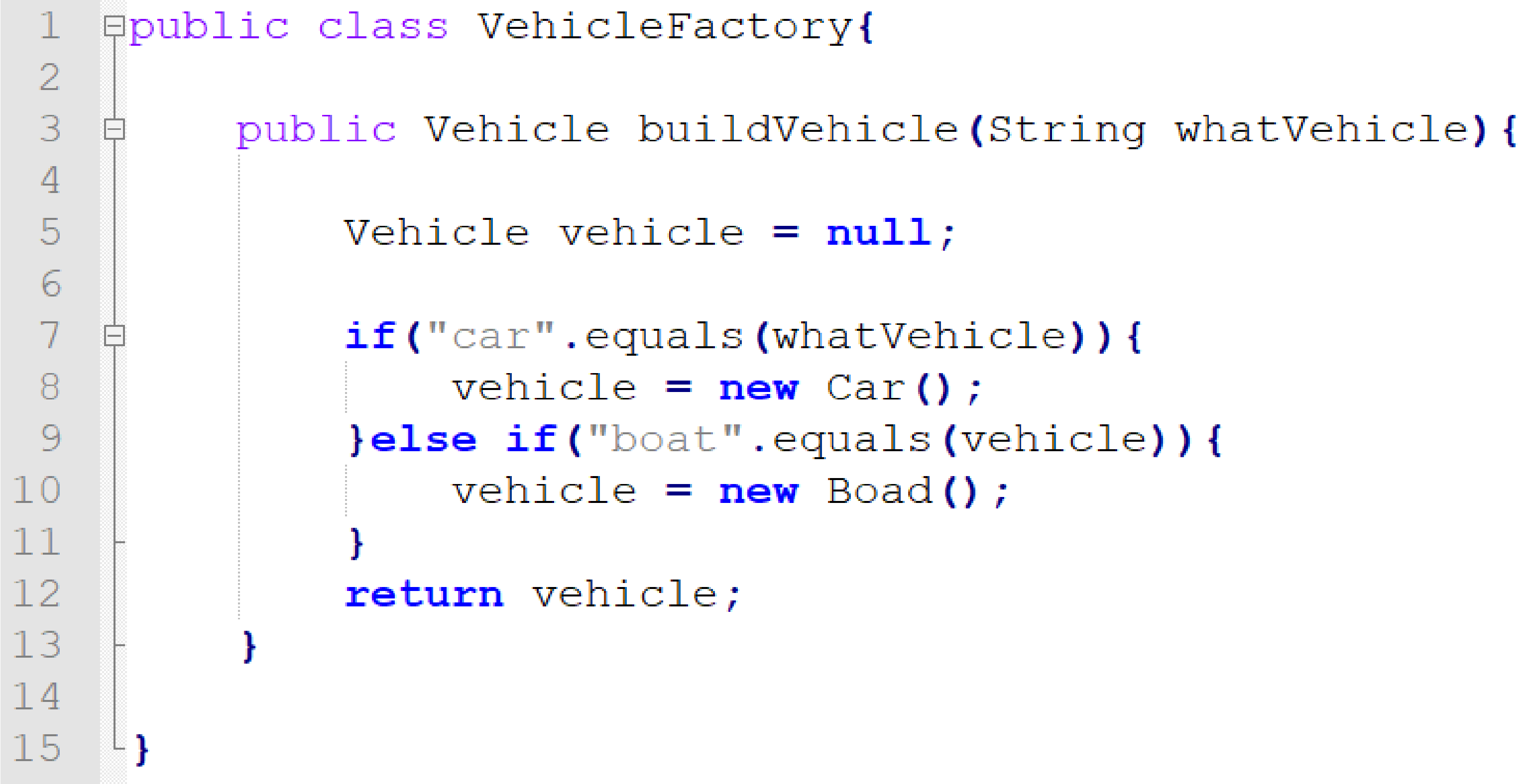
- **Creational patterns**: These concern object creation mechanisms, trying to create objects in a manner suitable to the situation.
- **Structural patterns**: These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.
- **Behavioral patterns**: These are specifically concerned with communication between objects.

Creational Pattern – Factory Method

The Factory Method pattern provides an interface for **creating objects in a super class, but allows subclasses to alter the type of objects** that will be created. In other words, it's a way to delegate the instantiation logic to child classes.



The problem - Why to use Factory Method

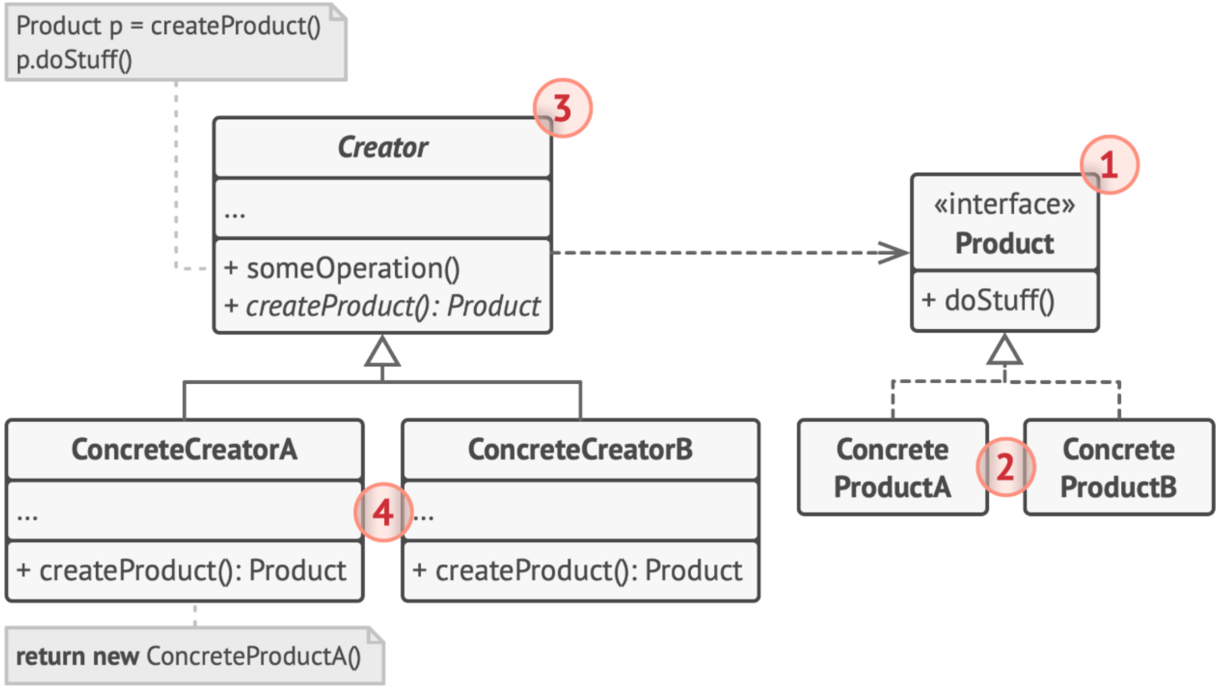


```
public class VehicleFactory{

    public Vehicle buildVehicle(String whatVehicle){

        Vehicle vehicle = null;

        if("car".equals(whatVehicle)){
            vehicle = new Car();
        }else if("boat".equals(vehicle)){
            vehicle = new Boad();
        }
        return vehicle;
    }

}
```

Factory Method – When to use

- **When you don't know beforehand the exact types and dependencies of the objects your code should work with.** The Factory Method serves as an interface to create objects in a superclass, but it lets subclasses alter the type of objects they will create.
- **When you want to provide users of your library or framework with a way to extend its internal components.** In this case, you can provide a factory method that users override to create customized components.
- **When you want to introducing new products without breaking existing code**

Factory Method – UML



Factory Method – Participants

- **Product**

  defines the interface of objects the factory method creates.
- **ConcreteProduct A,B**

  implements the Product interface.
- **Creator**

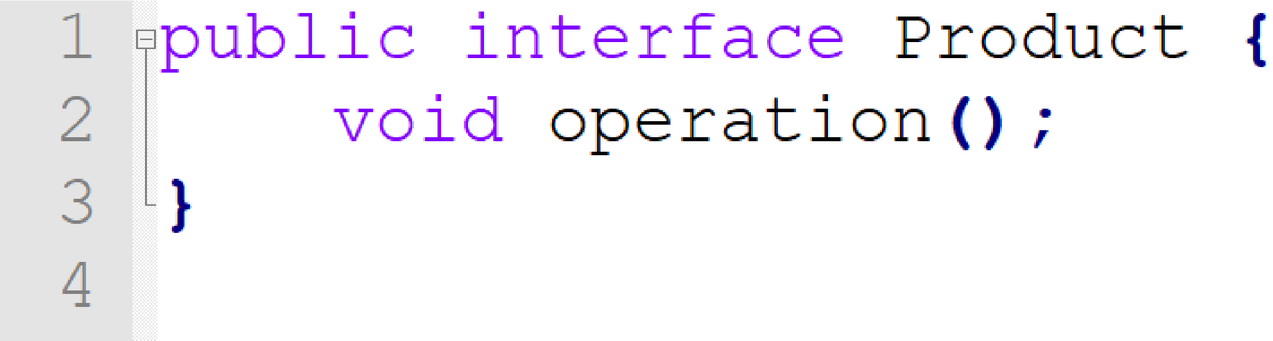
  Abstract class declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.

  may call the factory method to create a Product object.
- **ConcreteCreator**

  overrides the factory method to return an instance of a ConcreteProduct.

Factory Method – Implementation step 1

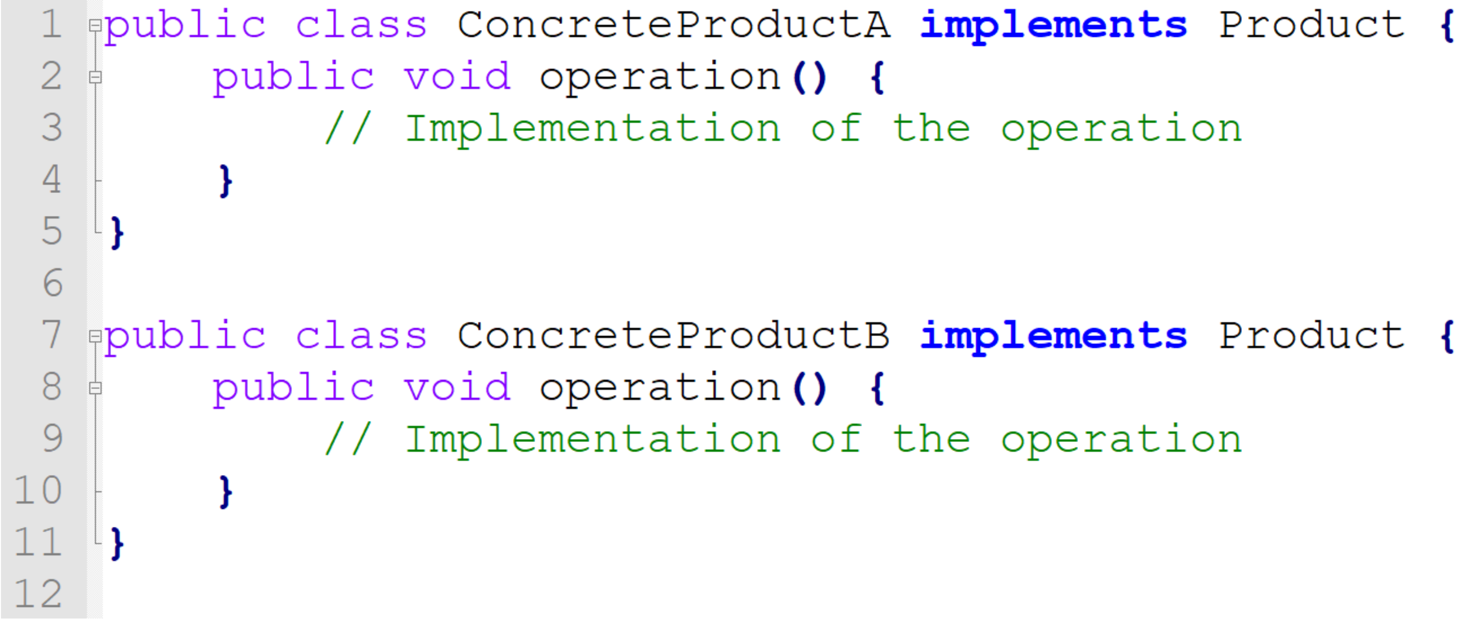
Create a Product Interface: This defines the operations that all the concrete products must implement.



```java
public interface Product {
    void operation();
}
```

Factory Method – Implementation step 2

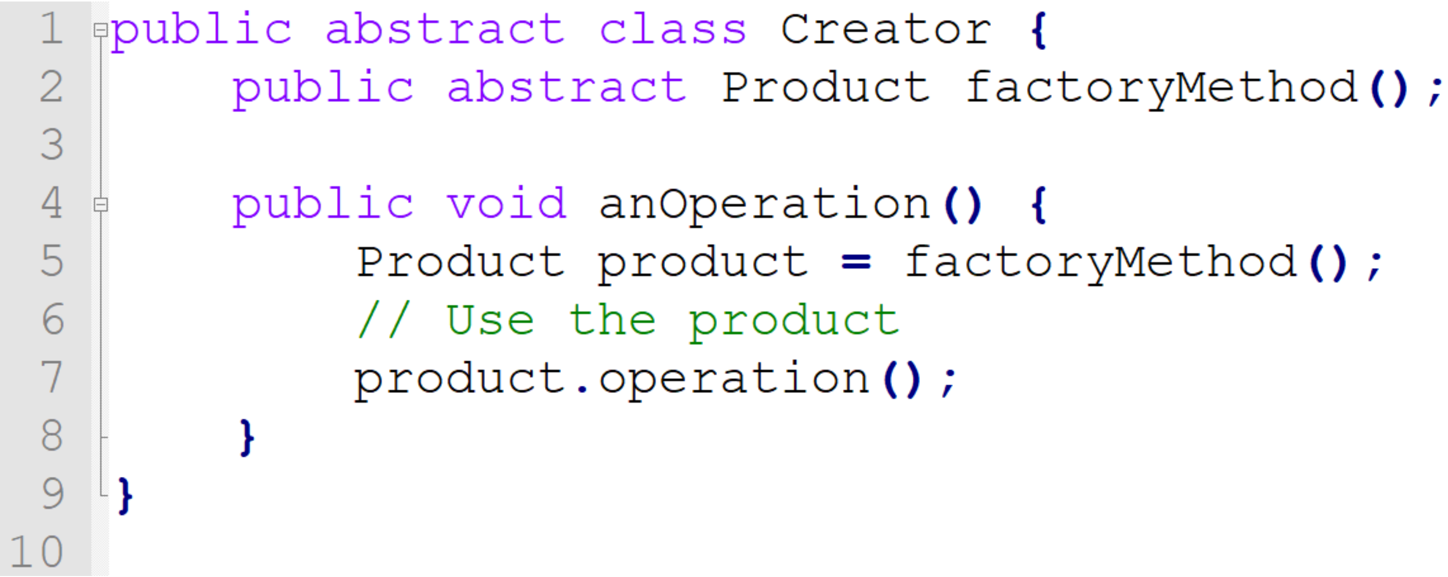
Implement Concrete Products: These are classes that implement the Product interface. Each class will implement the operations in a different way.



```java
public class ConcreteProductA implements Product {
    public void operation() {
        // Implementation of the operation
    }
}

public class ConcreteProductB implements Product {
    public void operation() {
        // Implementation of the operation
    }
}

```

Factory Method – Implementation step 3

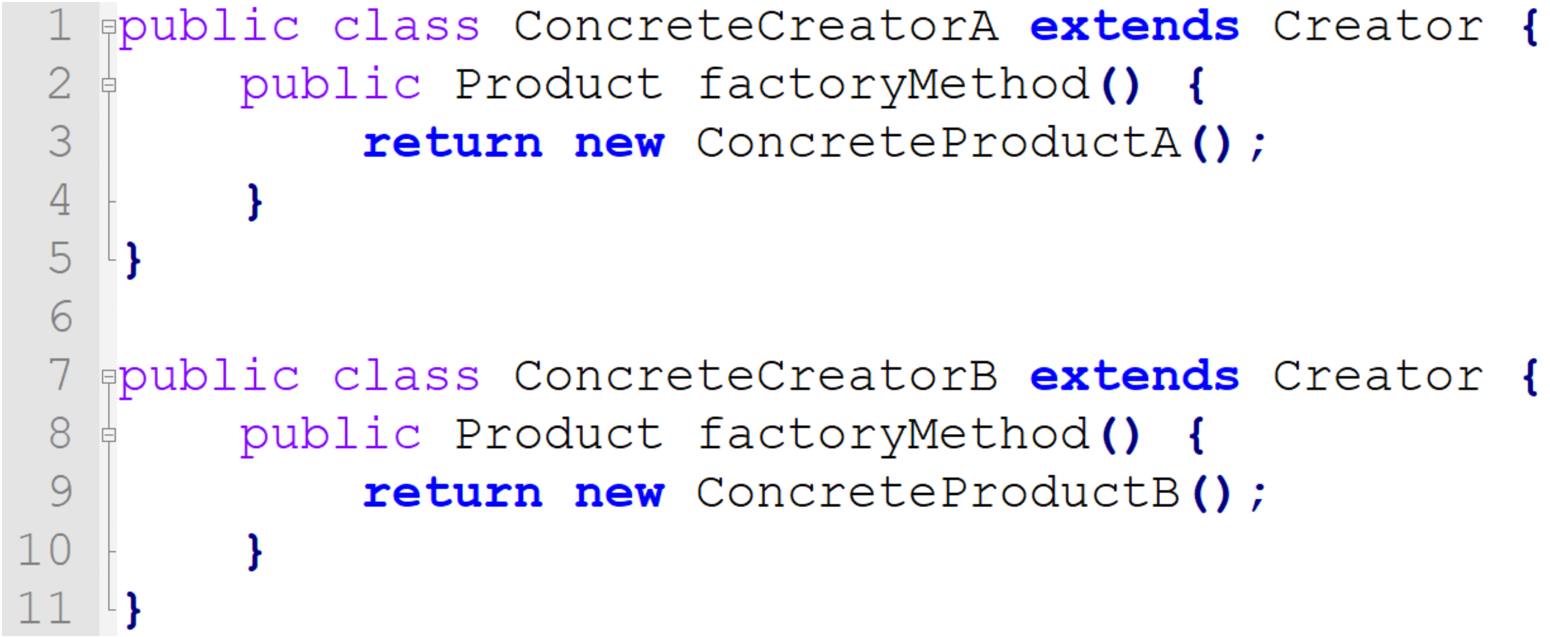
Create a Creator Abstract Class: This class declares the factory method, which returns an object of type Product. The Creator also defines a default implementation of the factory method that returns a default ConcreteProduct object. This class can also contain code that uses the Product.



```java
public abstract class Creator {
    public abstract Product factoryMethod();

    public void anOperation() {
        Product product = factoryMethod();
        // Use the product
        product.operation();
    }
}

```

Factory Method – Implementation step 4

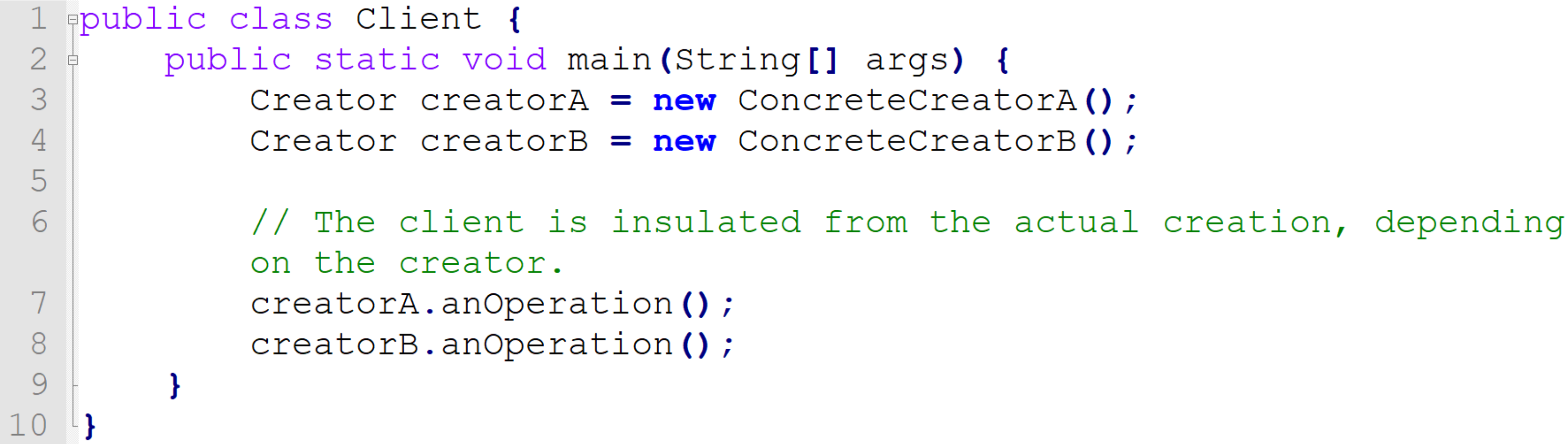
Create Concrete Creators: These classes override the factory method to change the resulting product's type.



```java
public class ConcreteCreatorA extends Creator {
    public Product factoryMethod() {
        return new ConcreteProductA();
    }
}

public class ConcreteCreatorB extends Creator {
    public Product factoryMethod() {
        return new ConcreteProductB();
    }
}
```

Factory Method – Implementation step 5

Use the Factory Method: The client code calls the creator's factory method instead of creating products directly with a constructor call (**new**).



```
public class Client {
    public static void main(String[] args) {
        Creator creatorA = new ConcreteCreatorA();
        Creator creatorB = new ConcreteCreatorB();

        // The client is insulated from the actual creation, depending on the creator.
        creatorA.anOperation();
        creatorB.anOperation();
    }
}
```

Factory Method – Pros and Cons

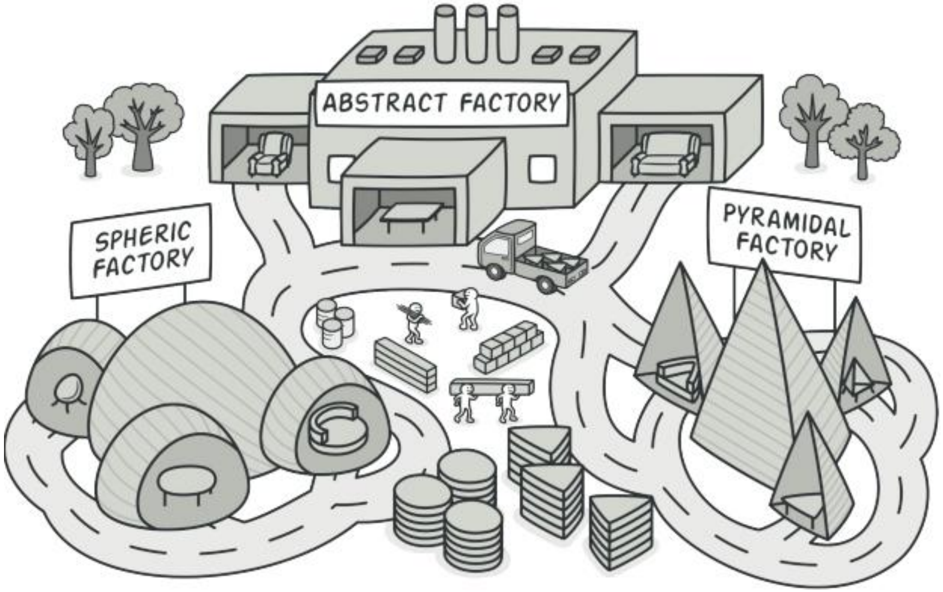
- The tight coupling between the creator and the concrete products is minimized.
- Adhering to the Single Responsibility Principle, the product creation logic can be consolidated into a single location, enhancing the maintainability of the code.
- In line with the Open/Closed Principle, the introduction of new product types won't disrupt the existing client code.
- Implementing this pattern may complexify the code due to the addition of many subclasses. It's best applied within an established creator class hierarchy.

Factory Method – Use case in Java

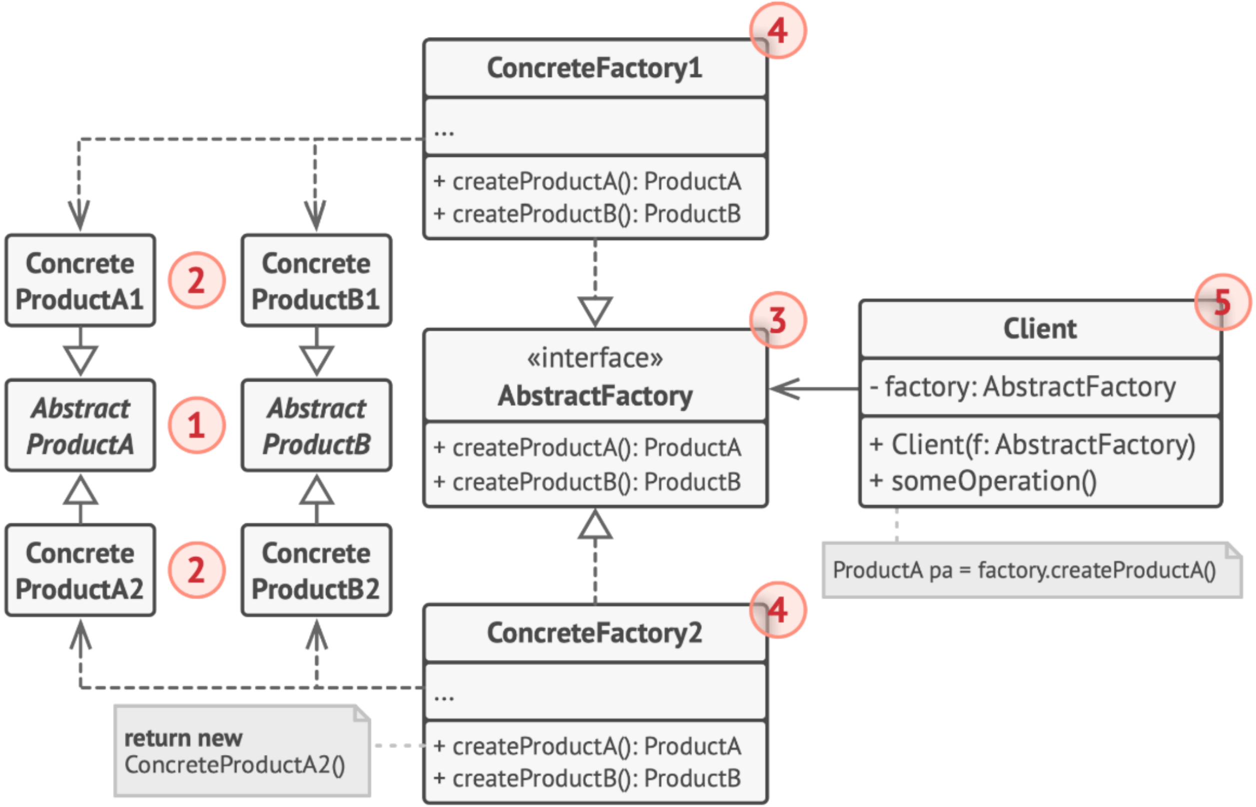
- java.util.Calendar#getInstance()
- java.util.ResourceBundle#getBundle()
- java.text.NumberFormat#getInstance()
- java.nio.charset.Charset#forName()

Creational Pattern – Abstract Factory

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



Abstract Factory – UML


4
ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
2
Concrete ProductB1
3
«interface»
AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
5
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
Abstract ProductA
1
Abstract ProductB
ProductA pa = factory.createProductA()
Concrete ProductA2
2
Concrete ProductB2
4
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new
ConcreteProductA2()

Abstract Factory – Participants

- **AbstractFactory**

  - declares an interface for operations that create abstract product objects.

- **ConcreteFactory**

  - implements the operations to create concrete product objects.

- **AbstractProduct**

  - declares an interface for a type of product object.

- **ConcreteProduct**

  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.

- **Client**

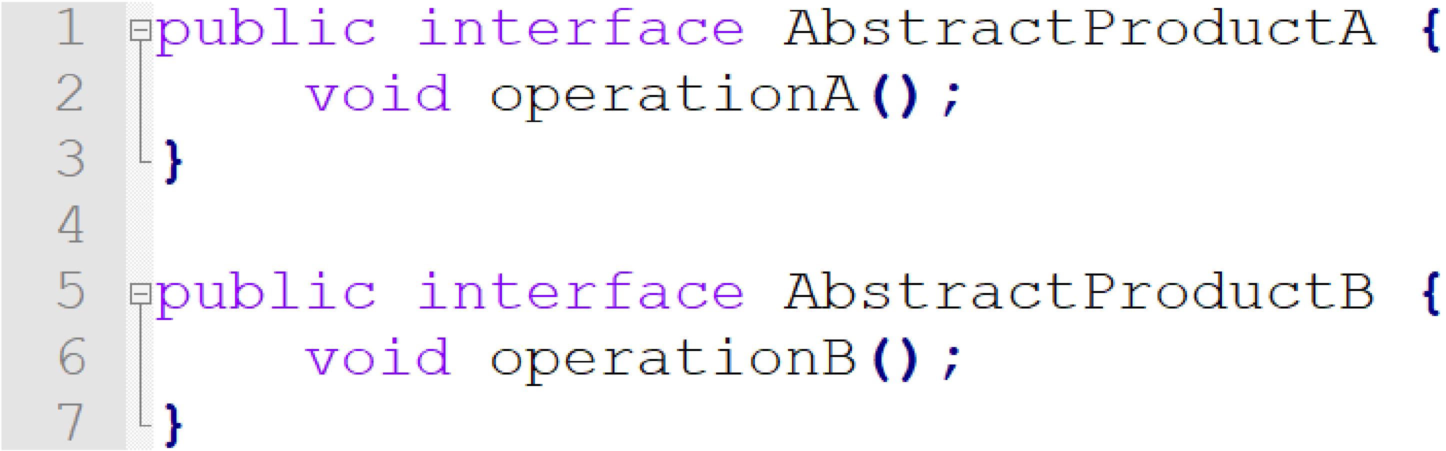
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory – When to use

- **Platform Independence**: When your system needs to be independent of how its products are created, composed, and represented. The pattern is widely used in systems that need to be platform agnostic.
- **Switching Product Families at Runtime**: When a system is expected to create different families of related products and to switch between them at runtime. This is facilitated by Abstract Factory's ability to encapsulate product creation and to make it interchangeable.
- **Consistent Object Composition**: When you have families of related products, and you want to ensure that a family of products used together is compatible. This can be useful in situations where products from one family work well together and mixing products from different families can lead to incorrect behavior.
- **Encapsulation of Product Creation**: If you want to hide the specifics of product creation and representation, Abstract Factory allows you to do this, providing an interface for creating a family of products and leaving the specifics to the concrete factory classes.
- **Future Extensibility**: If you anticipate the introduction of new types of products or families of products in the future, Abstract Factory allows for easy extension.

Abstract Factory – Implementation step 1

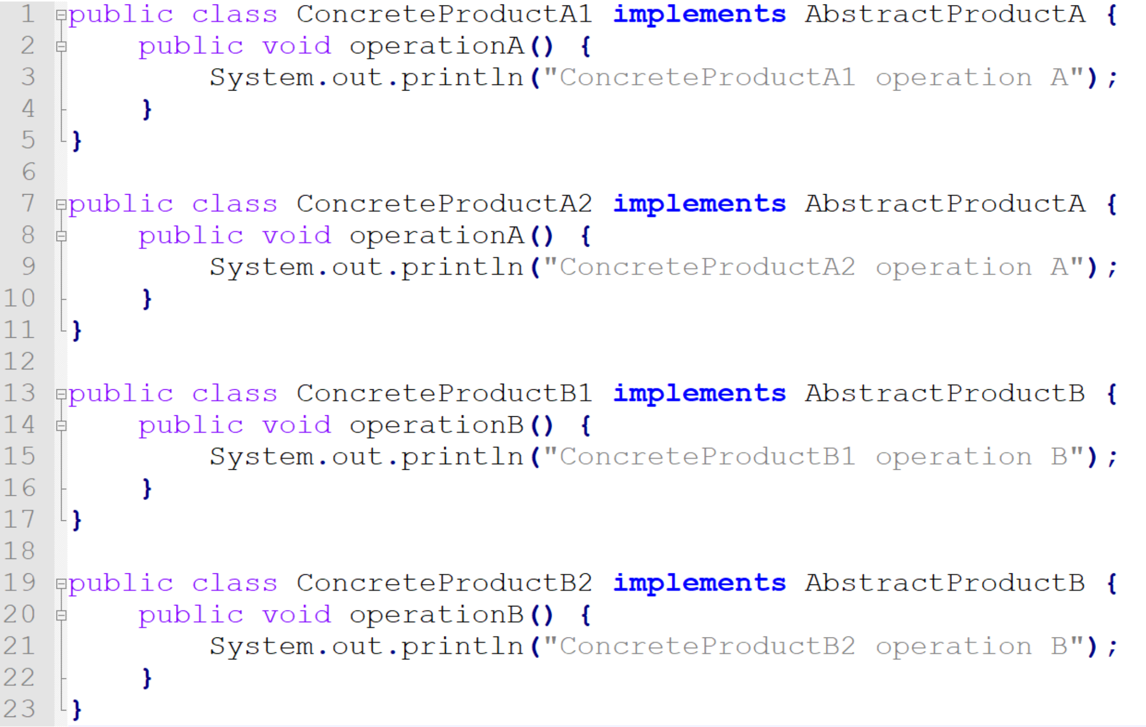
Define Abstract Product Interfaces: These are the different types of products that our factories will produce.



```
public interface AbstractProductA {
        void operationA();
}

public interface AbstractProductB {
        void operationB();
}
```

Abstract Factory – Implementation step 2

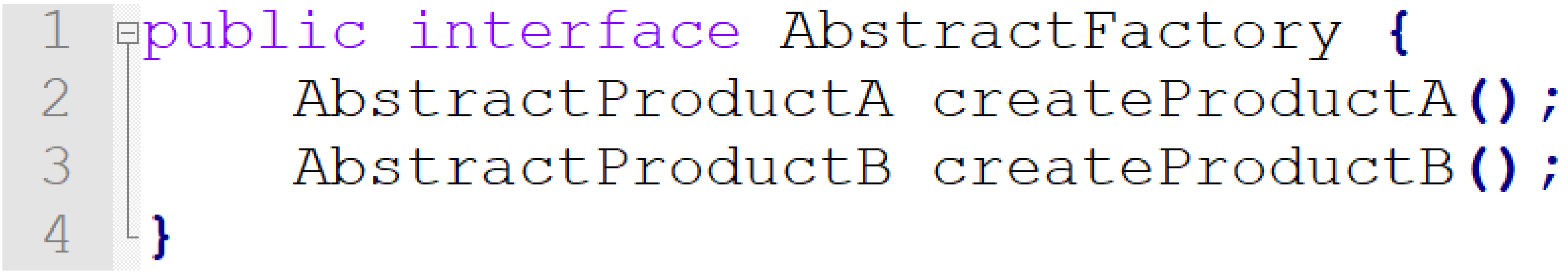
Create Concrete Product Classes: These classes will implement the product interfaces.



```java
public class ConcreteProductA1 implements AbstractProductA {
    public void operationA() {
        System.out.println("ConcreteProductA1 operation A");
    }
}

public class ConcreteProductA2 implements AbstractProductA {
    public void operationA() {
        System.out.println("ConcreteProductA2 operation A");
    }
}

public class ConcreteProductB1 implements AbstractProductB {
    public void operationB() {
        System.out.println("ConcreteProductB1 operation B");
    }
}

public class ConcreteProductB2 implements AbstractProductB {
    public void operationB() {
        System.out.println("ConcreteProductB2 operation B");
    }
}
```

Abstract Factory – Implementation step 3

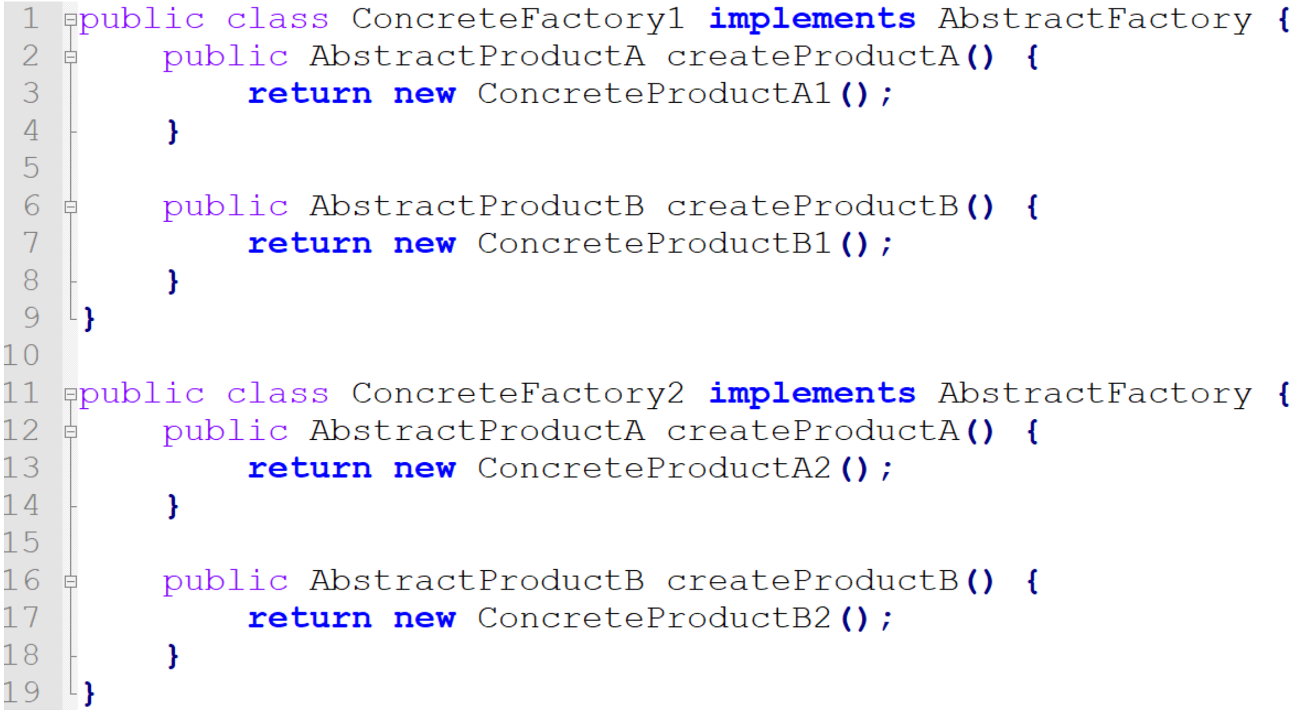
Define Abstract Factory Interface: This interface declares methods for creating objects of product classes.



```
public interface AbstractFactory {
    AbstractProductA createProductA();
    AbstractProductB createProductB();
}
```

Abstract Factory – Implementation step 4

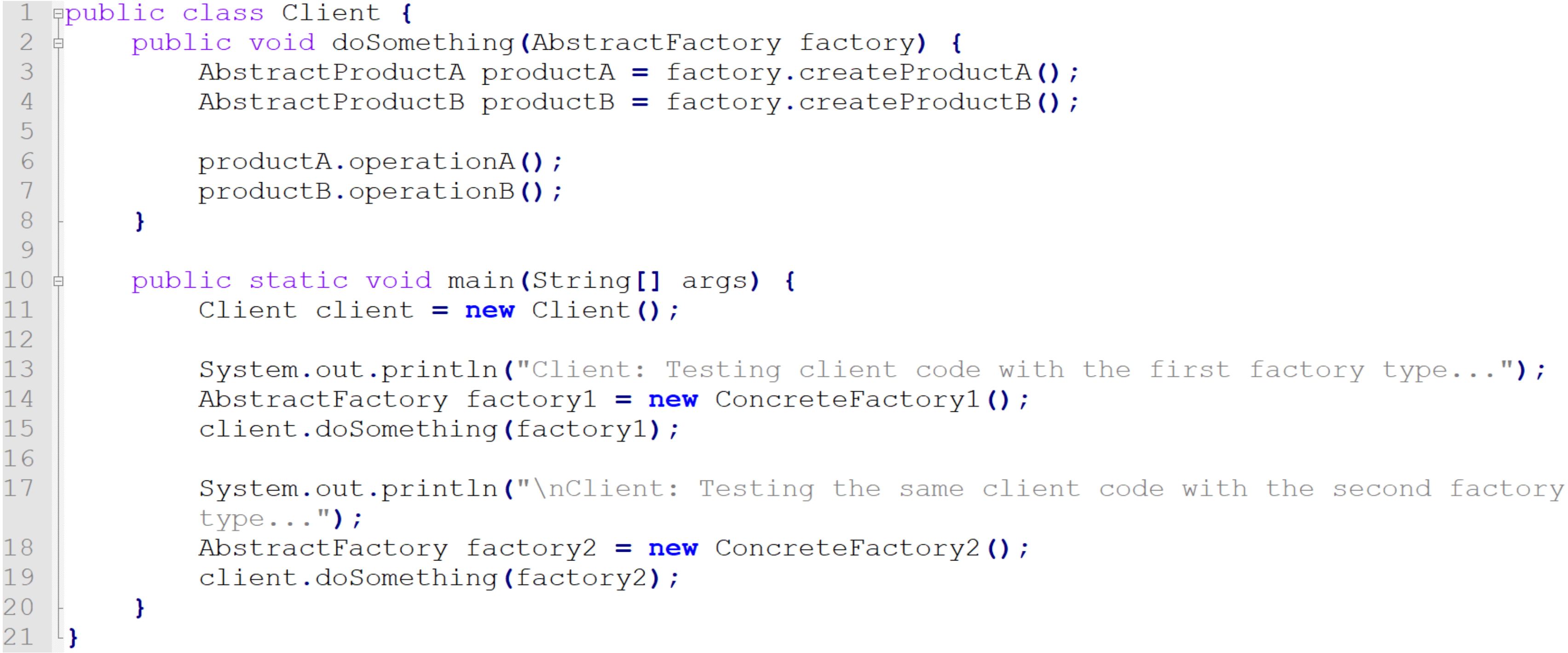
Create Concrete Factory Classes: These classes implement the Abstract Factory interface and return the concrete products.



```
public class ConcreteFactory1 implements AbstractFactory {
    public AbstractProductA createProductA() {
        return new ConcreteProductA1();
    }

    public AbstractProductB createProductB() {
        return new ConcreteProductB1();
    }
}

public class ConcreteFactory2 implements AbstractFactory {
    public AbstractProductA createProductA() {
        return new ConcreteProductA2();
    }

    public AbstractProductB createProductB() {
        return new ConcreteProductB2();
    }
}
```

Abstract Factory – Implementation step 5

Client Code: The client code uses the factory object to create the products.



```java
public class Client {
    public void doSomething(AbstractFactory factory) {
        AbstractProductA productA = factory.createProductA();
        AbstractProductB productB = factory.createProductB();

        productA.operationA();
        productB.operationB();
    }

    public static void main(String[] args) {
        Client client = new Client();

        System.out.println("Client: Testing client code with the first factory type...");
        AbstractFactory factory1 = new ConcreteFactory1();
        client.doSomething(factory1);

        System.out.println("\nClient: Testing the same client code with the second factory type...");
        AbstractFactory factory2 = new ConcreteFactory2();
        client.doSomething(factory2);
    }
}
```

Abstract Factory – Pros and Cons

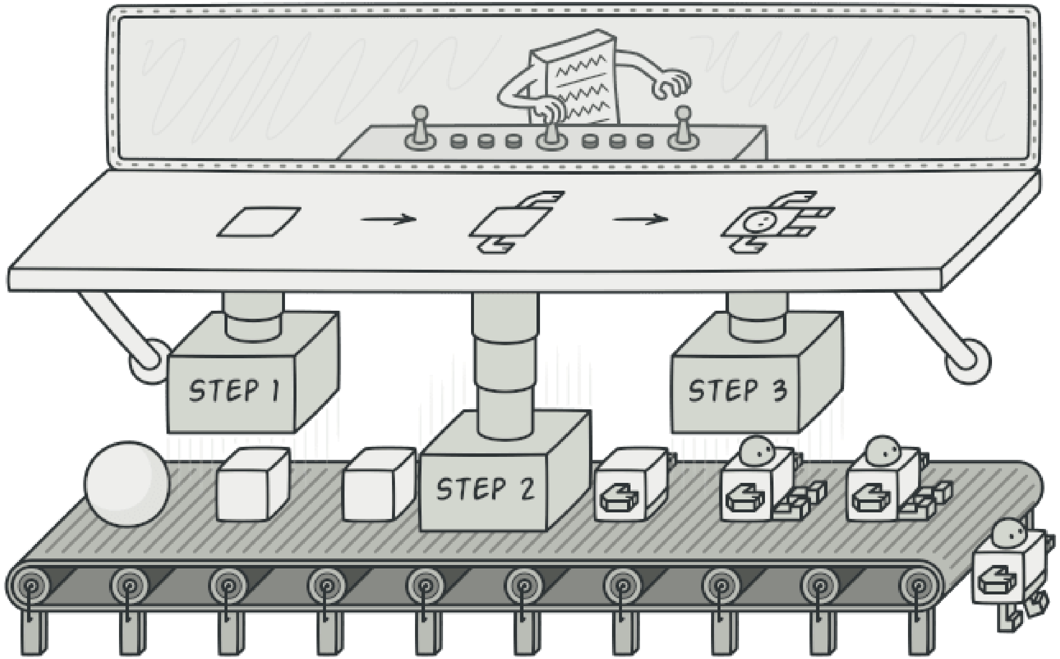
- You can be sure that the products you’re getting from a factory are compatible with each other.
- You avoid tight coupling between concrete products and client code.
- *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
- *Open/Closed 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.

Abstract Factory – Use case in Java

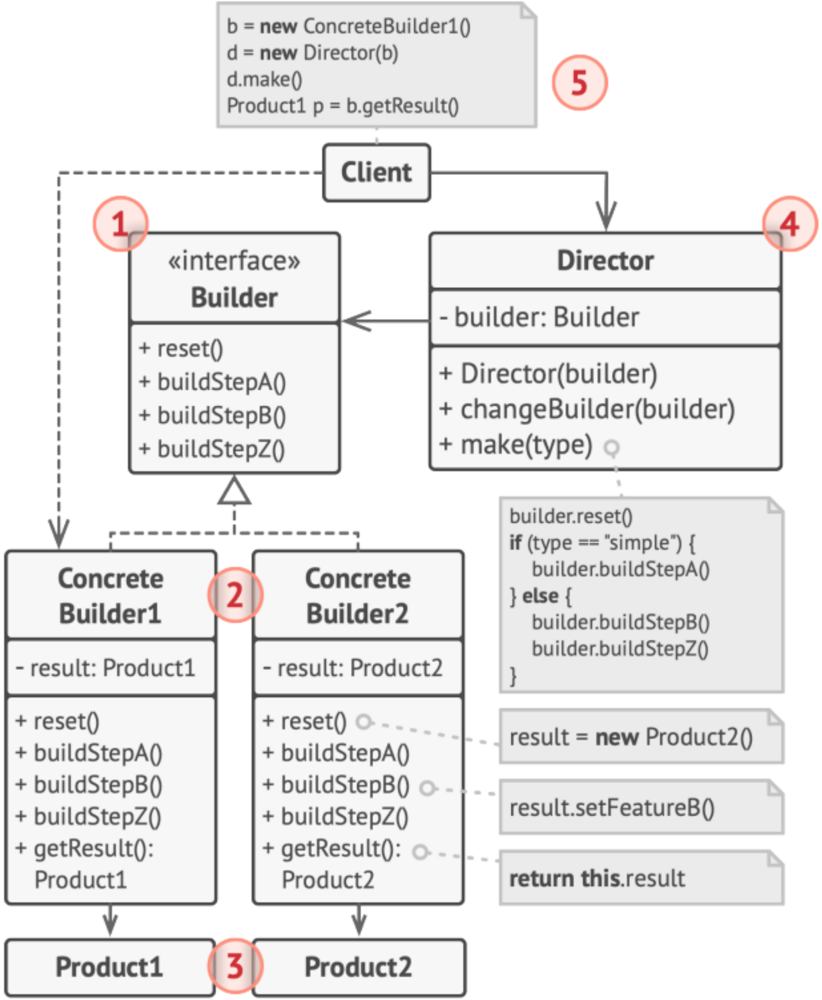
- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

Creational Pattern – Builder

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**.



Builder – UML



```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

5

Client

1

«interface»
Builder

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()

4

Director

- builder: Builder

+ Director(builder)
+ changeBuilder(builder)
+ make(type)

```
builder.reset()
if (type == "simple") {
    builder.buildStepA()
} else {
    builder.buildStepB()
    builder.buildStepZ()
}
```

2

Concrete Builder1

- result: Product1

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult(): Product1

Concrete Builder2

- result: Product2

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult(): Product2

```
result = new Product2()
```

```
result.setFeatureB()
```

```
return this.result
```

3

Product1

Product2

Builder – Participants

- **Builder**

  - Specifies an abstract interface for creating parts of a Product object.

- **ConcreteBuilder**

  - Constructs and assembles parts of the product by implementing the Builder interface.
  - Defines and keeps track of the representation it creates.
  - Provides an interface for retrieving the product.

- **Director**

  - Constructs an object using the Builder interface.

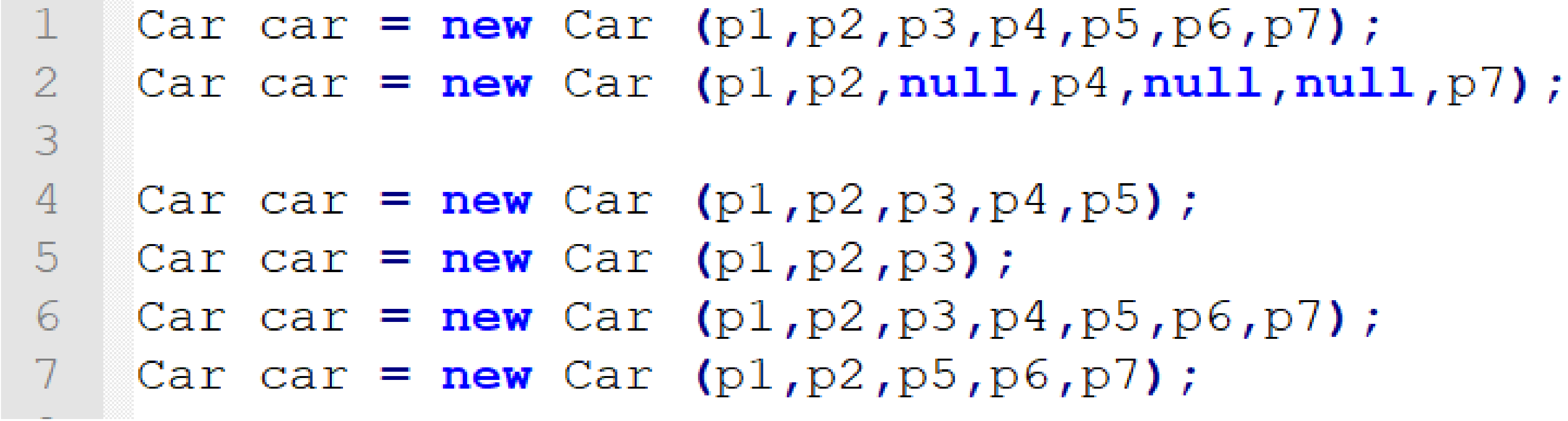
- **Product**

  - Represents the complex object under construction.

Builder – When to use

- **Complex Objects**: The pattern is most valuable when you need to create a complex object that has many parts or configuration options. The builder allows you to produce different types and representations of an object using the same construction code.
- **Composition of Objects**: If the object is made up of a composite of other objects and the object needs to be constructed in a step-by-step fashion.
- **Clear Code**: When you want to keep your code clear, with the responsibilities of construction separated from the representation of the complex object.
- **Immutability**: The Builder pattern can be used to build an immutable object. Once all the parts of the object are created, the object itself can be created as an immutable object.
- **Parameter Overloading**: If a class has many constructors with a large number of parameters, it's better to use the Builder pattern.

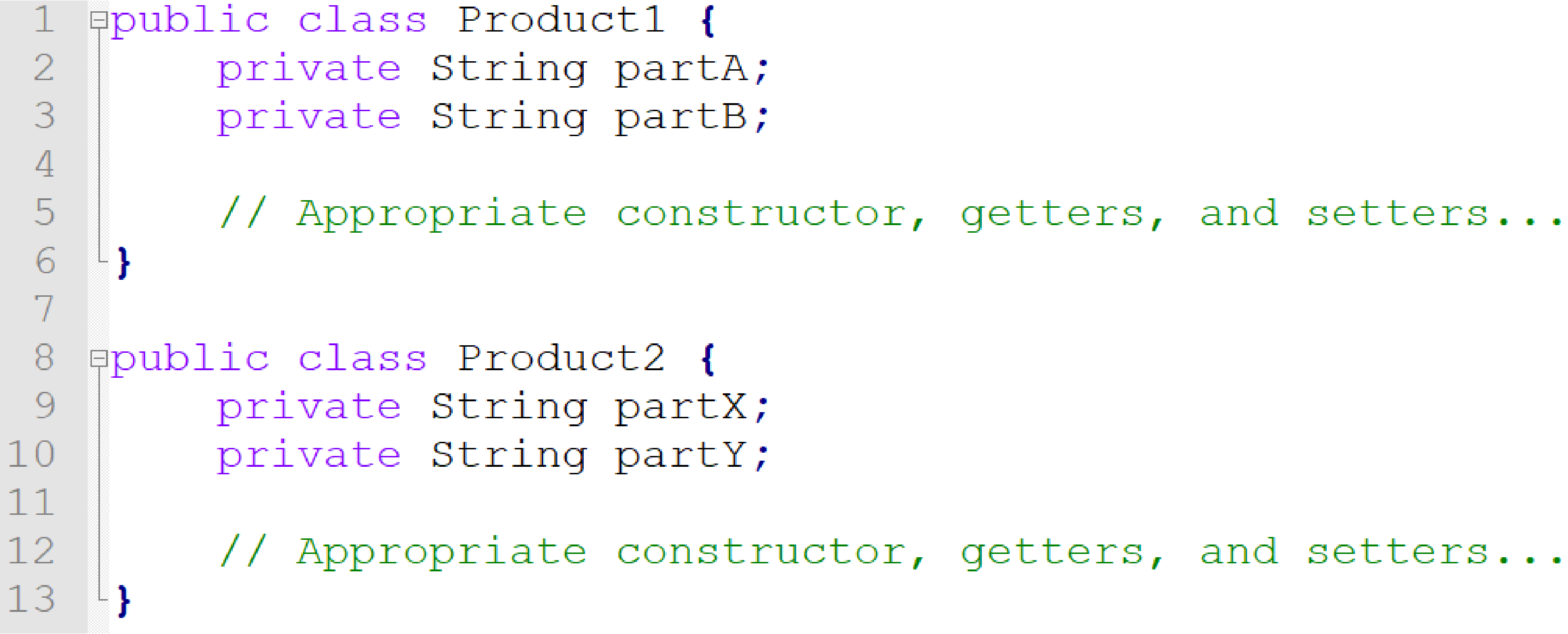
Builder – Why to use



```
Car car = new Car (p1,p2,p3,p4,p5,p6,p7);
Car car = new Car (p1,p2,null,p4,null,null,p7);

Car car = new Car (p1,p2,p3,p4,p5);
Car car = new Car (p1,p2,p3);
Car car = new Car (p1,p2,p3,p4,p5,p6,p7);
Car car = new Car (p1,p2,p5,p6,p7);
```

Builder – Implementation step 1

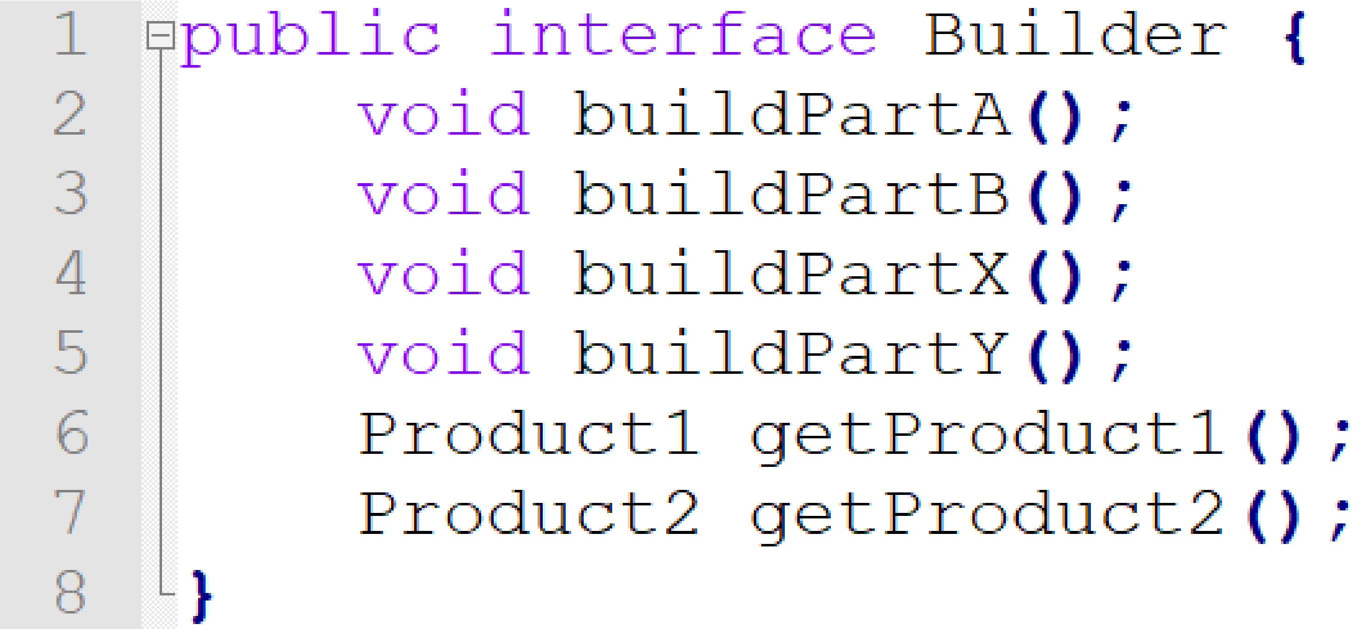
Product Classes (Product1, Product2): These are the complex objects that we're trying to build.



```
public class Product1 {
    private String partA;
    private String partB;

    // Appropriate constructor, getters, and setters...
}

public class Product2 {
    private String partX;
    private String partY;

    // Appropriate constructor, getters, and setters...
}
```

Builder – Implementation step 2

Builder Interface: This is an interface specifying the "contract" for creating parts of a complex object.



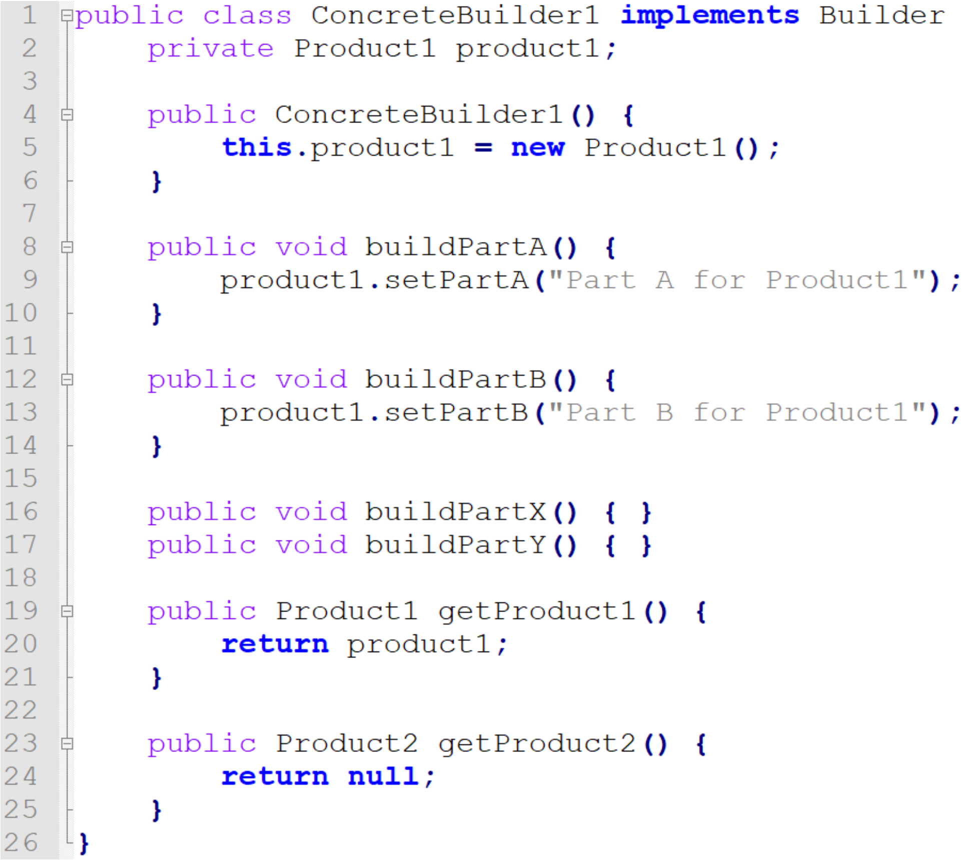
```java
public interface Builder {
    void buildPartA();
    void buildPartB();
    void buildPartX();
    void buildPartY();
    Product1 getProduct1();
    Product2 getProduct2();
}
```

Builder – Implementation step 3

Concrete Builder Classes (ConcreteBuilder1):

Some problem here ?

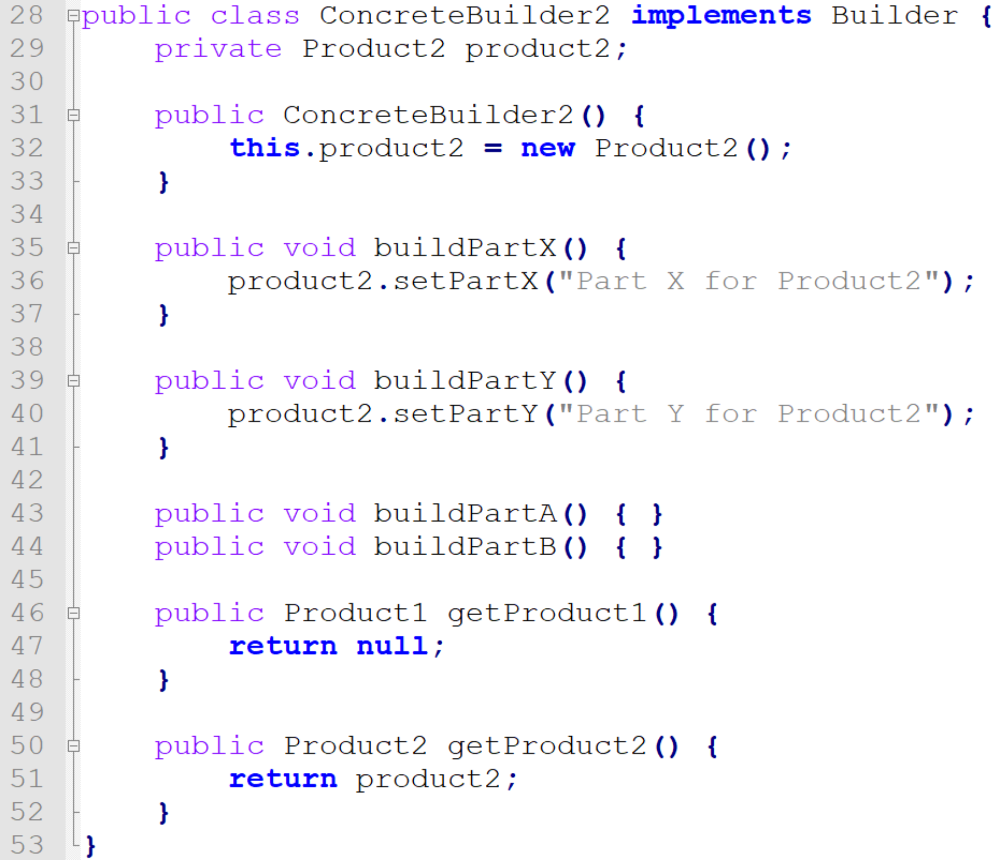
These are concrete builders implementing the **Builder** interface, one for each product.



```
public class ConcreteBuilder1 implements Builder
    private Product1 product1;

    public ConcreteBuilder1() {
        this.product1 = new Product1();
    }

    public void buildPartA() {
        product1.setPartA("Part A for Product1");
    }

    public void buildPartB() {
        product1.setPartB("Part B for Product1");
    }

    public void buildPartX() { }
    public void buildPartY() { }

    public Product1 getProduct1() {
        return product1;
    }

    public Product2 getProduct2() {
        return null;
    }
}
```

Builder – Implementation step 3

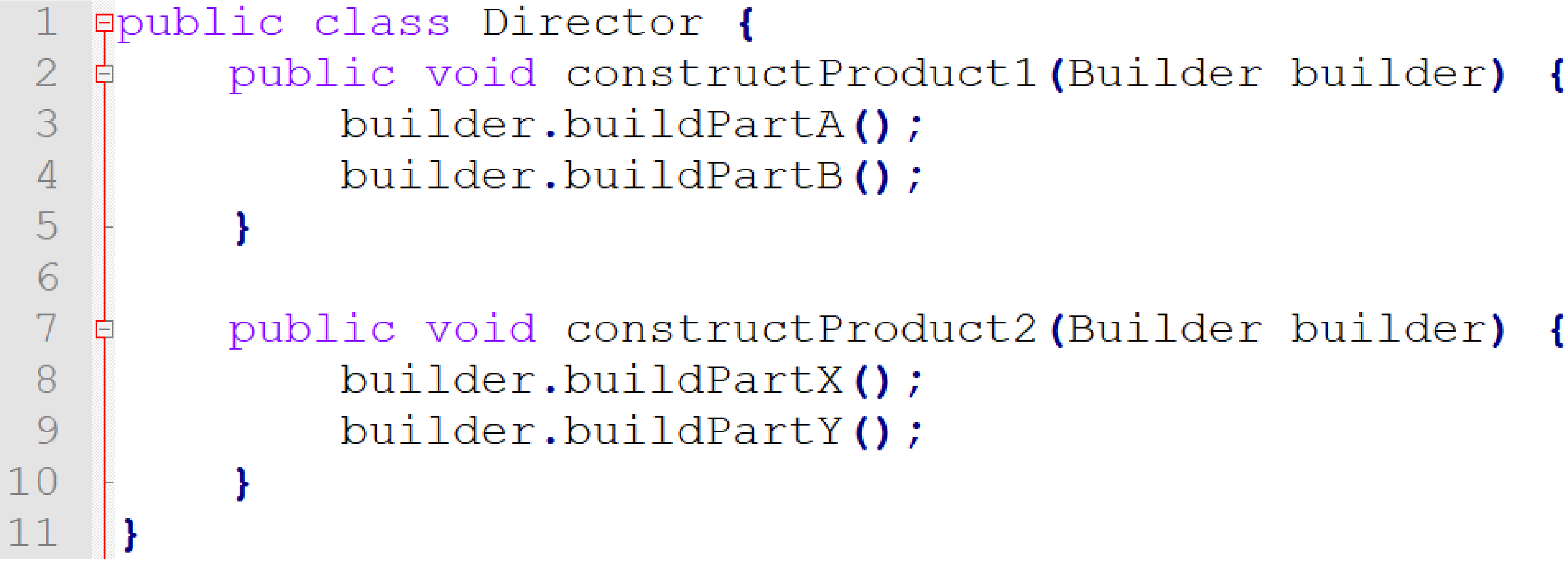
Concrete Builder Classes (ConcreteBuilder2):



```java
public class ConcreteBuilder2 implements Builder {
    private Product2 product2;

    public ConcreteBuilder2() {
        this.product2 = new Product2();
    }

    public void buildPartX() {
        product2.setPartX("Part X for Product2");
    }

    public void buildPartY() {
        product2.setPartY("Part Y for Product2");
    }

    public void buildPartA() { }
    public void buildPartB() { }

    public Product1 getProduct1() {
        return null;
    }

    public Product2 getProduct2() {
        return product2;
    }
}
```

Builder – Implementation step 4

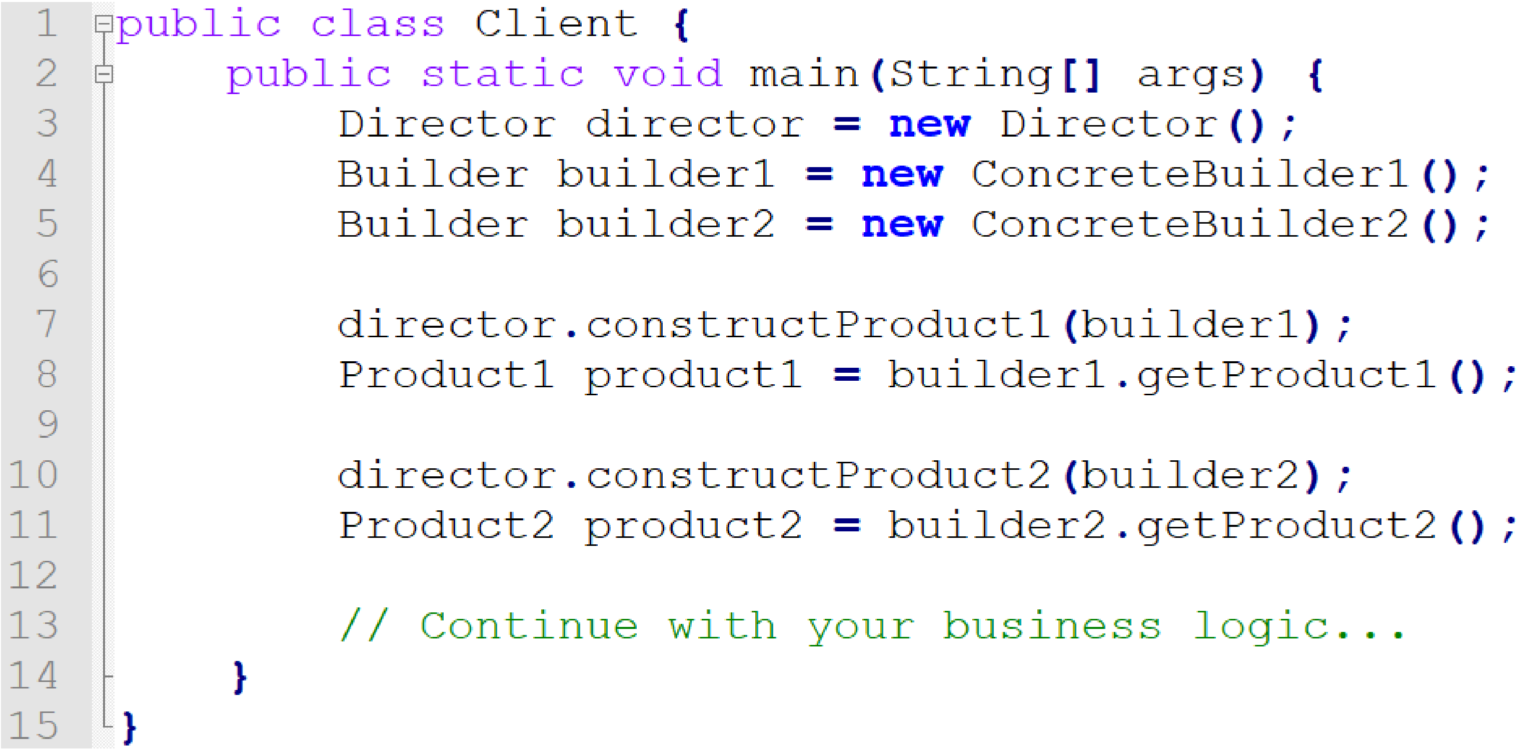
Director Class: The **Director** class defines the order in which to execute the building steps.



```java
public class Director {
    public void constructProduct1(Builder builder) {
        builder.buildPartA();
        builder.buildPartB();
    }

    public void constructProduct2(Builder builder) {
        builder.buildPartX();
        builder.buildPartY();
    }
}
```

Builder – Implementation step 5

Client Class: This class uses the **Director** and **Builder** to construct the products.



```java
public class Client {
    public static void main(String[] args) {
        Director director = new Director();
        Builder builder1 = new ConcreteBuilder1();
        Builder builder2 = new ConcreteBuilder2();

        director.constructProduct1(builder1);
        Product1 product1 = builder1.getProduct1();

        director.constructProduct2(builder2);
        Product2 product2 = builder2.getProduct2();

        // Continue with your business logic...
    }
}
```

Builder – Pros and Cons

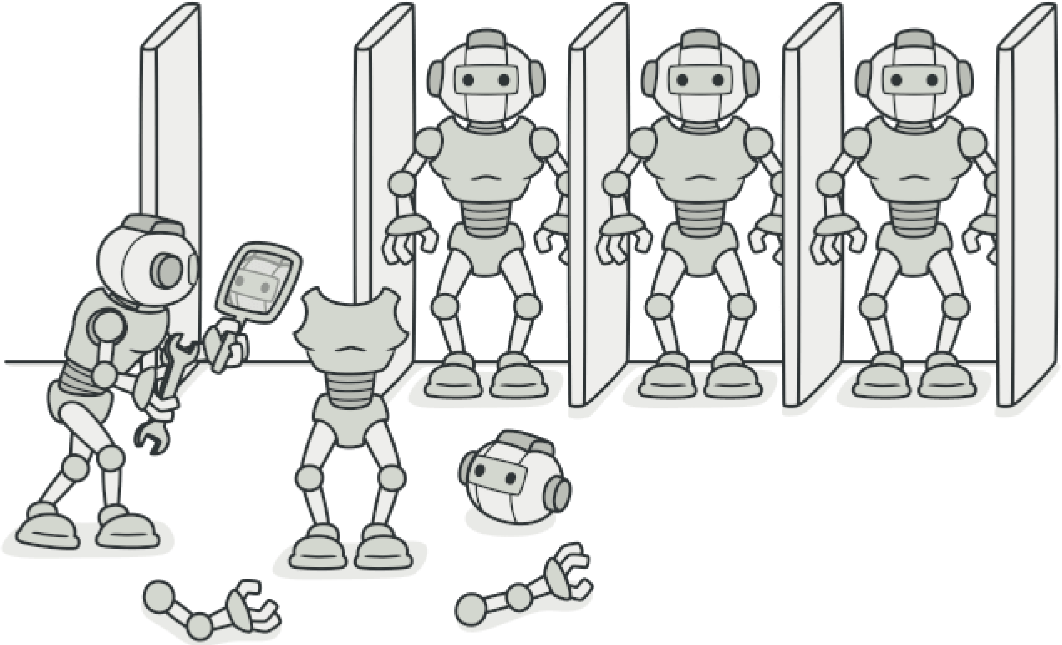
- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.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

Builder – Use case in Java

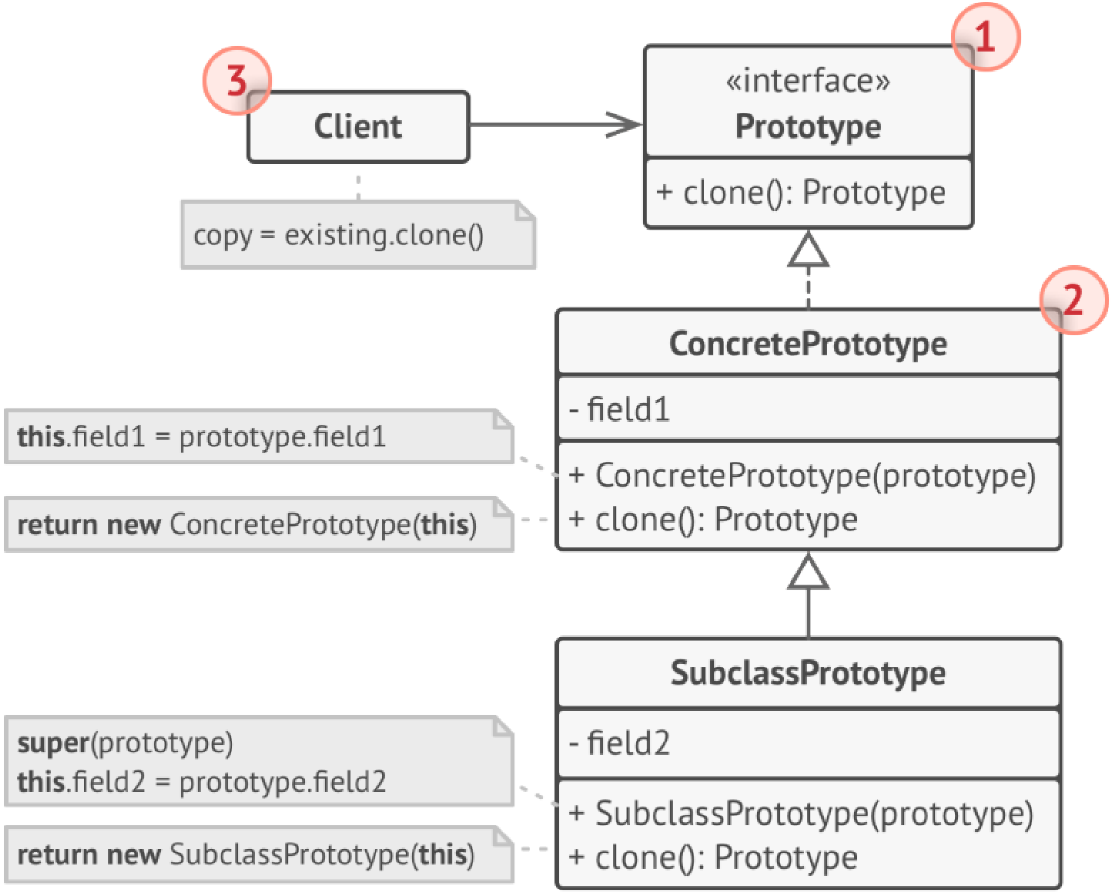
- java.lang.StringBuilder#append() (unsynchronized)
- java.lang.StringBuffer#append() (synchronized)
- java.nio.ByteBuffer#put() (also in CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
- javax.swing.GroupLayout.Group#addComponent()

Creational Pattern – Prototype

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



Prototype – UML


3
Client
copy = existing.clone()
1
«interface»
Prototype
+ clone(): Prototype
2
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1
return new ConcretePrototype(this)
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)

Prototype – Participants

- **Prototype**

  - Declare an interface for clone itself

- **ConcreteClass**

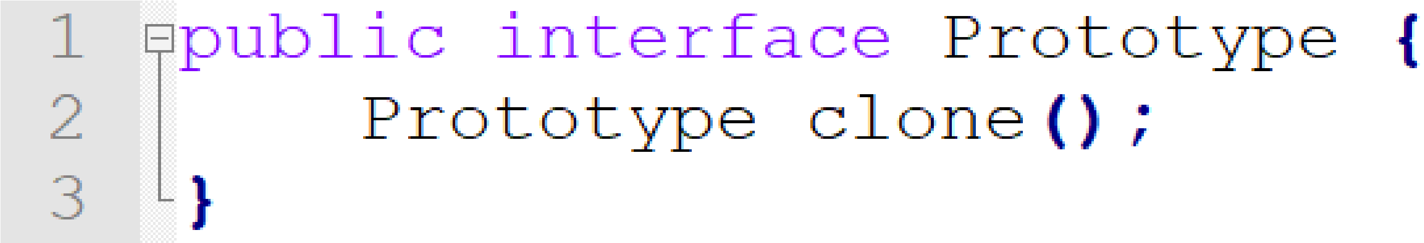
  - Implements an operation for cloning itself

- **Client**

  - Creates a new object by asking a prototype to clone itself

Prototype – Implementation step 1

Prototype Interface/Abstract Class: Declare an interface or an abstract class with a **clone()** method. This will be the base Prototype.

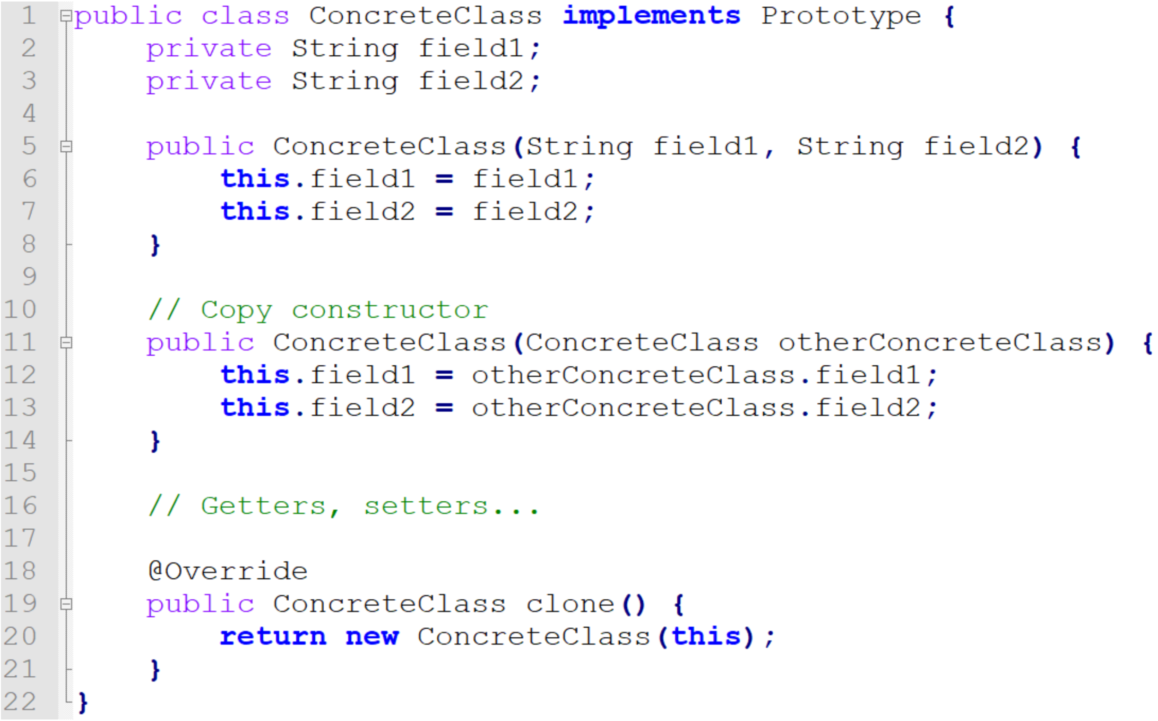


```java
public interface Prototype {
    Prototype clone();
}
```

Prototype – Implementation step 2

Concrete Prototype Classes:

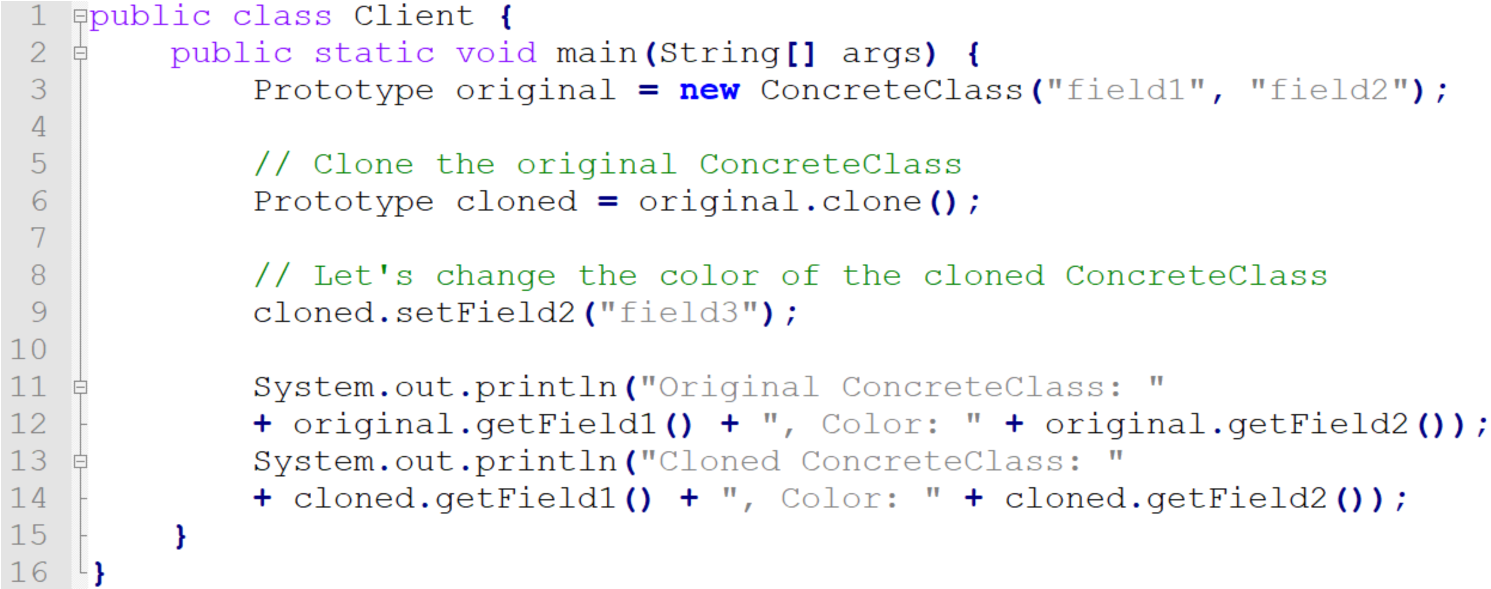
Create one or more classes that implement the **Prototype** interface. In the **clone()** method, create a new instance of the class and copy over its data members from the existing instance.



```java
public class ConcreteClass implements Prototype {
    private String field1;
    private String field2;

    public ConcreteClass(String field1, String field2) {
        this.field1 = field1;
        this.field2 = field2;
    }

    // Copy constructor
    public ConcreteClass(ConcreteClass otherConcreteClass) {
        this.field1 = otherConcreteClass.field1;
        this.field2 = otherConcreteClass.field2;
    }

    // Getters, setters...

    @Override
    public ConcreteClass clone() {
        return new ConcreteClass(this);
    }
}
```

Prototype – Implementation step 3

Client Code: Use the **clone()** method instead of calling **new** to create a new object. The client code doesn't need to be aware of the concrete classes of objects it works with, as long as they implement the **clone()** method.



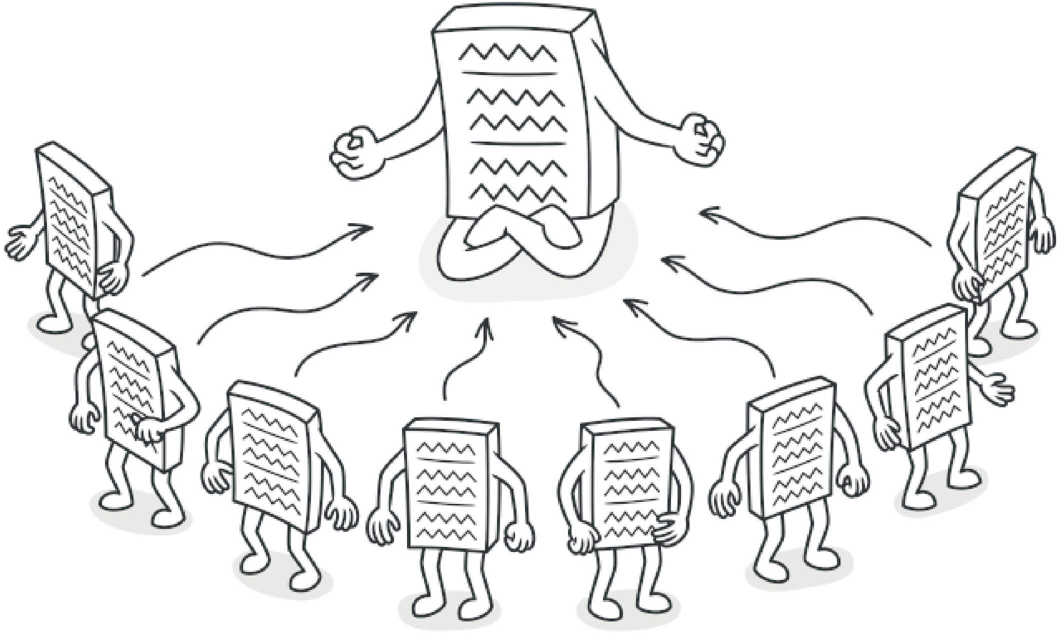
```
public class Client {
    public static void main(String[] args) {
        Prototype original = new ConcreteClass("field1", "field2");

        // Clone the original ConcreteClass
        Prototype cloned = original.clone();

        // Let's change the color of the cloned ConcreteClass
        cloned.setField2("field3");

        System.out.println("Original ConcreteClass: "
        + original.getField1() + ", Color: " + original.getField2());
        System.out.println("Cloned ConcreteClass: "
        + cloned.getField1() + ", Color: " + cloned.getField2());
    }
}
```

Prototype – Pros and Cons

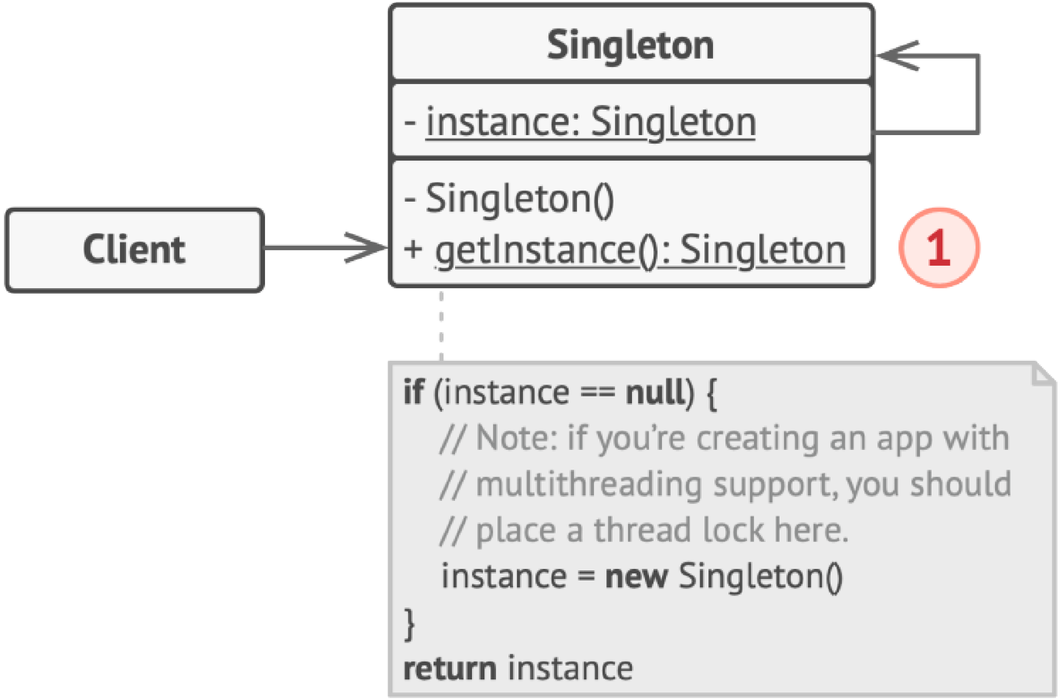
- You can clone objects without coupling to their concrete classes.
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- You can produce complex objects more conveniently.
- You get an alternative to inheritance when dealing with configuration presets for complex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

Creational Pattern – Singleton

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.



Singleton – UML



```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Singleton – Participants

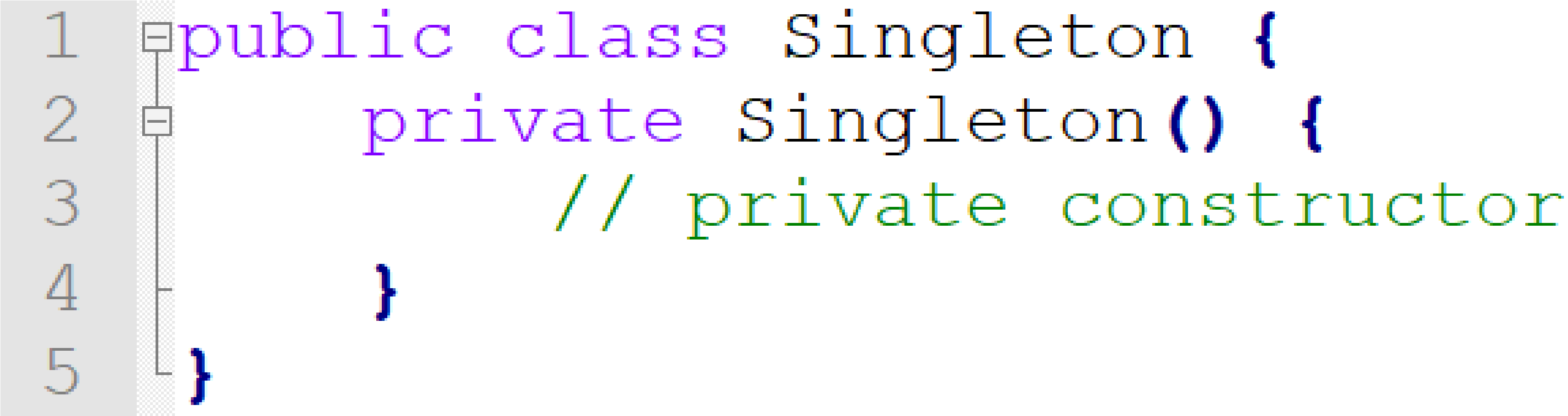
- **Singleton**

  - Defines an Instance operation that lets clients access its unique instance. Instance is a class operation

  - May be responsible for crating its own unique instance

Singleton – Implementation step 1

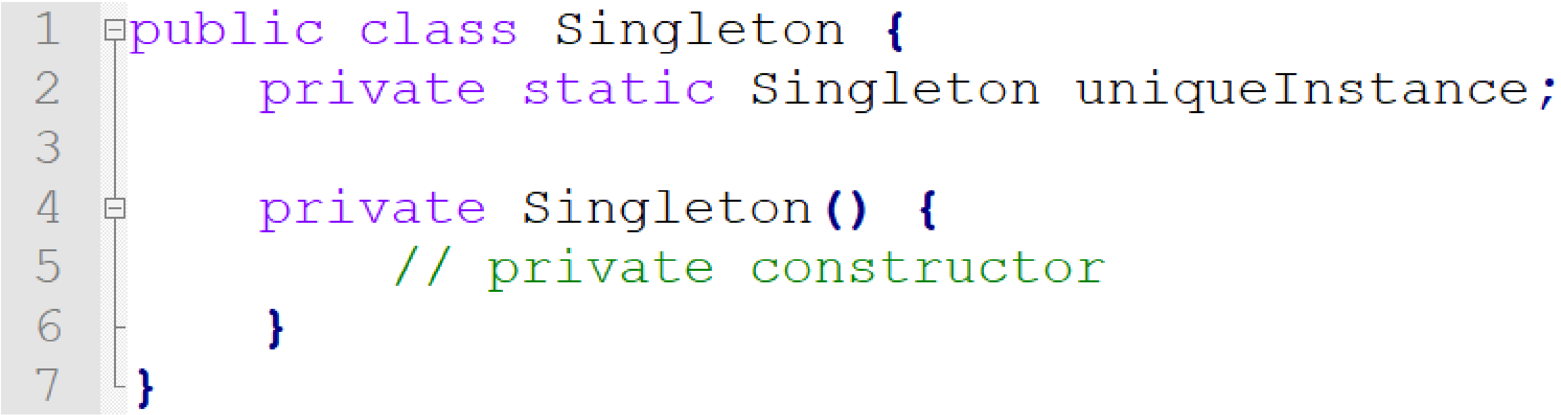
Private Constructor: Make the default constructor private, to prevent other objects from using the **new** operator with the Singleton class.



```
public class Singleton {
        private Singleton() {
                // private constructor
        }
}
```

Singleton – Implementation step 2

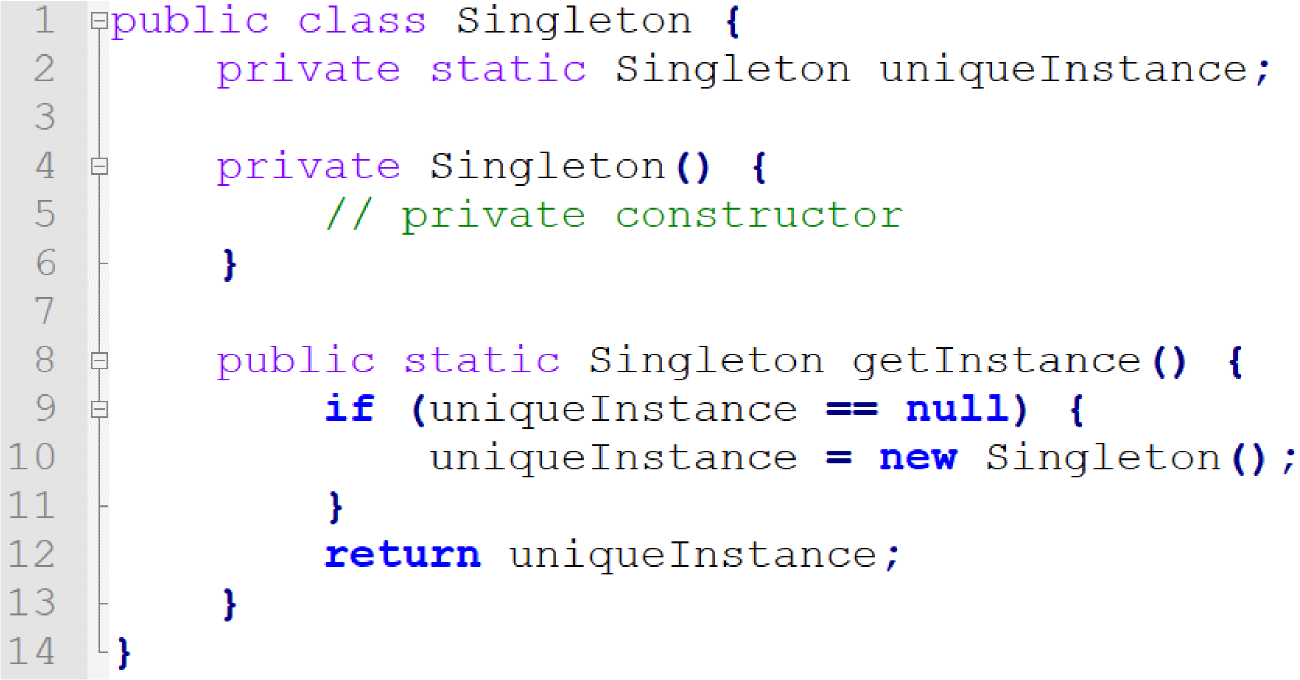
Private Static Variable: Declare a private static variable of the same class that is the only instance of the class.



```
public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {
        // private constructor
    }
}
```

Singleton – Implementation step 3

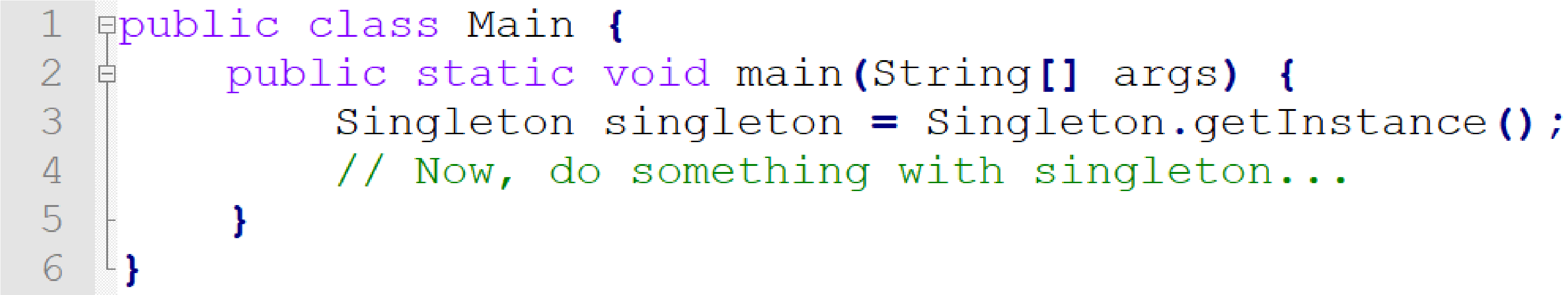
Public Static Method: Create a static method that controls the access to the singleton instance.



```java
public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {
        // private constructor
    }

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

Singleton – Implementation step 4

Client Code: Now, you can get the Singleton object in your client code.



```java
public class Main {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        // Now, do something with singleton...
    }
}
```

Singleton – 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.

Singleton – Use case in Java

- java.lang.Runtime#getRuntime()
- java.awt.Desktop#getDesktop()
- java.lang.System#getSecurityManager()