Getting to Know the Creational Design Patterns

Esteban Herrera
JAVA ARCHITECT

@eh3rrera   www.eherrera.net

Creational Patterns

Abstract Factory

Builder

Factory Method

Singleton

Prototype

Class Creational Patterns

Factory Method

Object Creational Patterns

Abstract Factory

Builder

Singleton

Prototype

Creational Patterns You Should Know

Singleton

Factory and Abstract Factory

Builder

The Singleton Pattern

A Simple Class

```java
public class Calculator {
    public Calculator() { }

    public int add(int n1, int n2) {
        return n1 + n2;
    }

    // ...
}
```

Creating Only One Instance

```
Calculator calculator = new Calculator();
~~Calculator calculator2 = new Calculator();~~
```

Creating Only One Instance

```
Calculator calculator = ~~new~~ Calculator();
```

A Simple Class

```java
public class Calculator {
    public Calculator() { }

    public int add(int n1, int n2) {
        return n1 + n2;
    }

    // ...
}
```

Private Constructor

```java
public class Calculator {
   private Calculator() { }

   public int add(int n1, int n2) {
      return n1 + n2;
   }

   // ...
}
```

Class Method to Get an Instance

```java
public class Calculator {
    private Calculator() { }

    public static Calculator getInstance() { }

    public int add(int n1, int n2) {
        return n1 + n2;
    }

    // ...
}
```

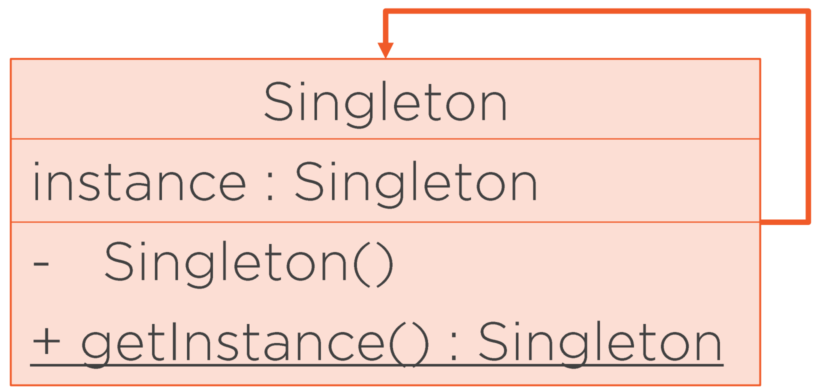
Creating Only One Instance

```
Calculator calculator = Calculator.getInstance();
```

A Singleton Implementation

```
public class Calculator {
   private static Calculator instance;

   private Calculator() { }

   public static Calculator getInstance() {
      if (instance == null)
         instance = new Calculator();
      return instance;
   }

   // ...
}
```

The Singleton Pattern



Multithreading Problems

```
public static Calculator getInstance() {
    if (instance == null)
        instance = new Calculator();
    return instance;
}
```

One instance per thread

Another Singleton Implementation

```java
public class Calculator {
   private static Calculator instance =
                                    new Calculator();

   private Calculator() { }

   public static Calculator getInstance() {
      return instance;
   }

   // ...
}
```

Yet Another Singleton Implementation

```java
public class Calculator {
   private static Calculator instance;

   private Calculator() { }

   public synchronized static Calculator getInstance() {
      if (instance == null)
         instance = new Calculator();
      return instance;
   }

   // ...
}
```

Problems to Be Solved by the Implementation

Reflection

Serialization

Class loaders

And many others...

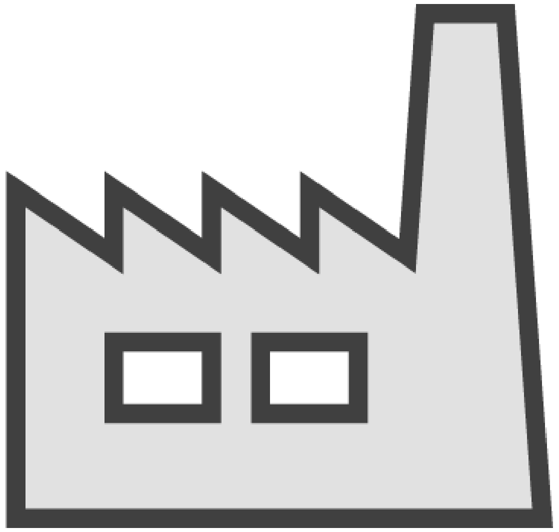
Acceptable Singleton Implementation

```
public enum Calculator {
    INSTANCE;

    public int add(int n1, int n2) {
        return n1 + n2;
    }
    // ...
}
```

As long as you don't need to extend from another class

The Factory and Abstract Factory Patterns

Types of Factories



Simple factory

Factory method

Abstract factory

Creating an Object

```
Vehicle v = new Vehicle();
```

Creating an Object

```
Vehicle v = new Car();
```

Creating an Object

```
Vehicle v = ~~new~~ Motorcycle();
```

Simple Factory

```java
public class VehicleFactory {
   public static Vehicle create(String type) {
      Vehicle v = null;
      if (type.equals("car")) {
         v = new Car();
      } else if (type.equals("motorcycle")) {
         v = new Motorcycle();
      }
      return v;
   }
}
```

Using Simple Factory

```
Vehicle v = VehicleFactory.create("car");
```

Using Simple Factory

```
Vehicle v = VehicleFactory.create("motorcycle");
```

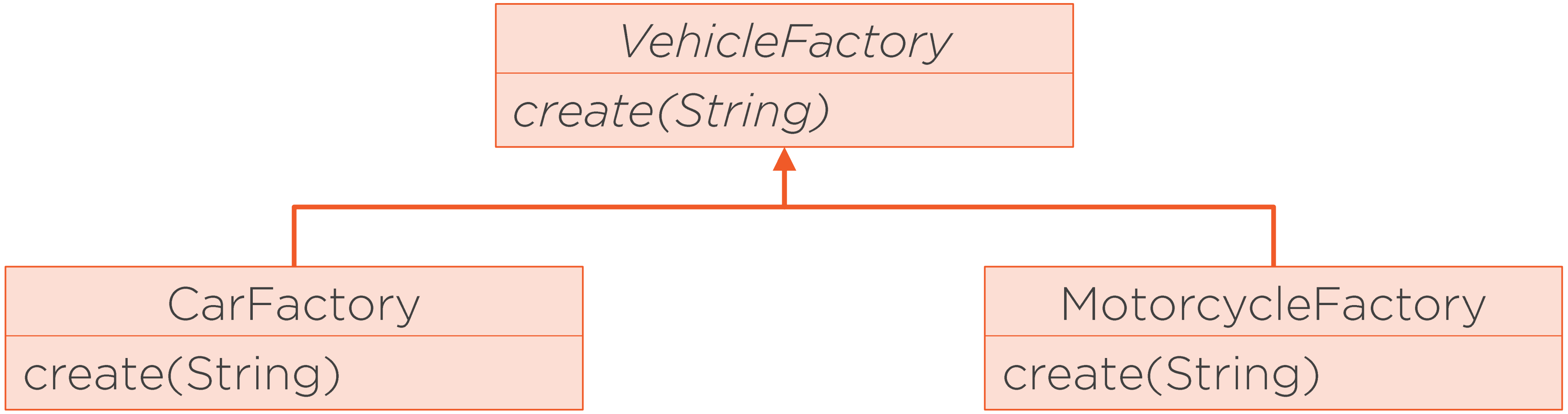
Simple Factory

```java
public class VehicleFactory {
   public static Vehicle create(String type) {
      Vehicle v = null;
      if (type.equals("car")) {
         v = new Car();
      } else if (type.equals("motorcycle")) {
         v = new Motorcycle();
      }
      return v;
   }
}
```

Simple Factory

```java
public class VehicleFactory {
    public static Vehicle create(String type) {
        Vehicle v = null;
        if (type.equals("car")) {
            v = new Car();
        } else if (type.equals("motorcycle")) {
            v = new Motorcycle();
        } else if (/* */) {
            v = /* */;
        } // ...
        return v;
    }
}
```

Creating a Hierarchy of Factories



```
public Vehicle create(String type) {
   Vehicle v = null;
   if (type.equals("car")) {
      v = new CompactCar();
   } else if (type.equals("sedan")) {
      v = new SedanCar();
   }
   return v;
}
```

```
public Vehicle create(String type) {
   Vehicle v = null;
   if (type.equals("scooter")) {
      v = new Scooter();
   } else if (type.equals("sportbike")) {
      v = new SportBike();
   }
   return v;
}
```

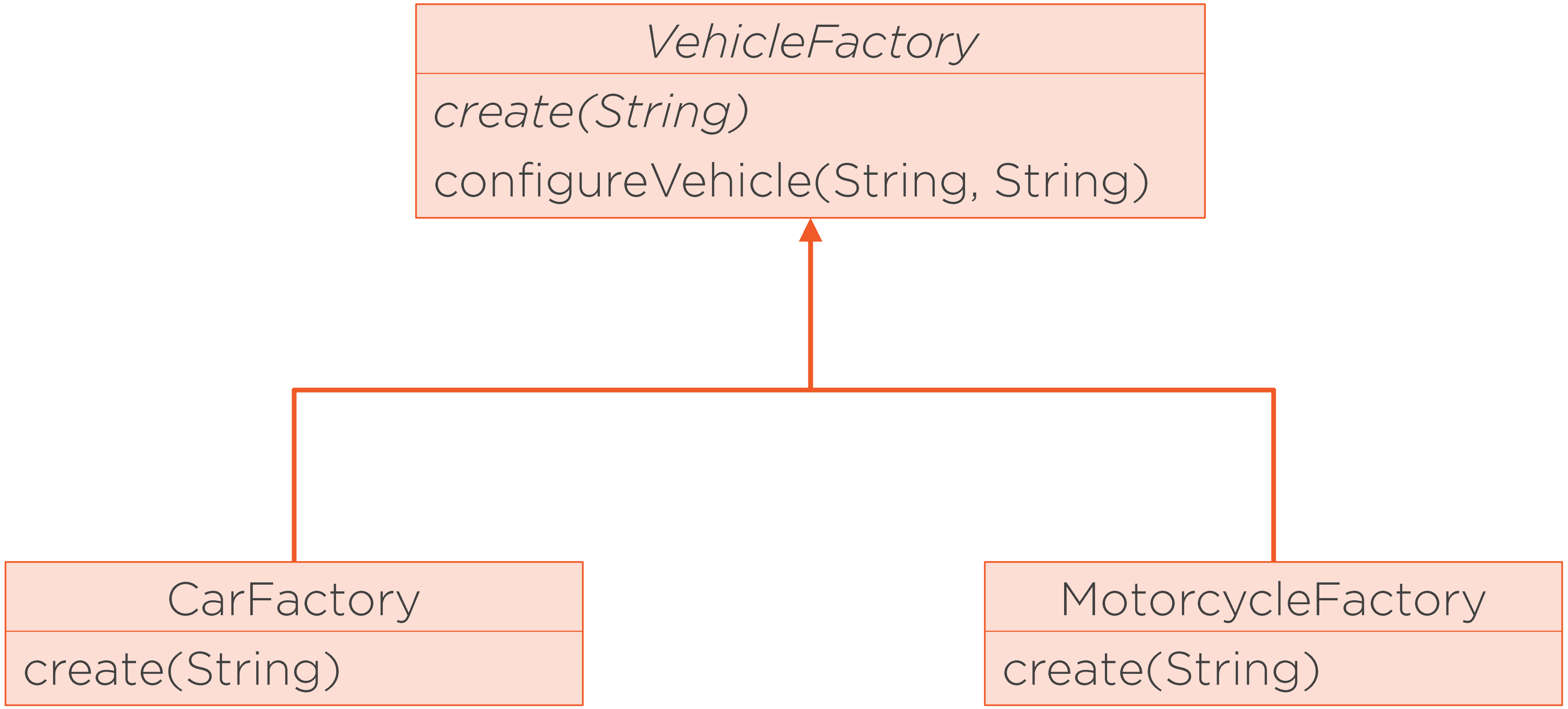
Factory Method

```java
public abstract class VehicleFactory {
    public Vehicle configureVehicle(String type, String color) {
        Vehicle v = create(type);

        v.setColor(color);
        v.build();

        return v;
    }

    public abstract Vehicle create(String type);
}
```

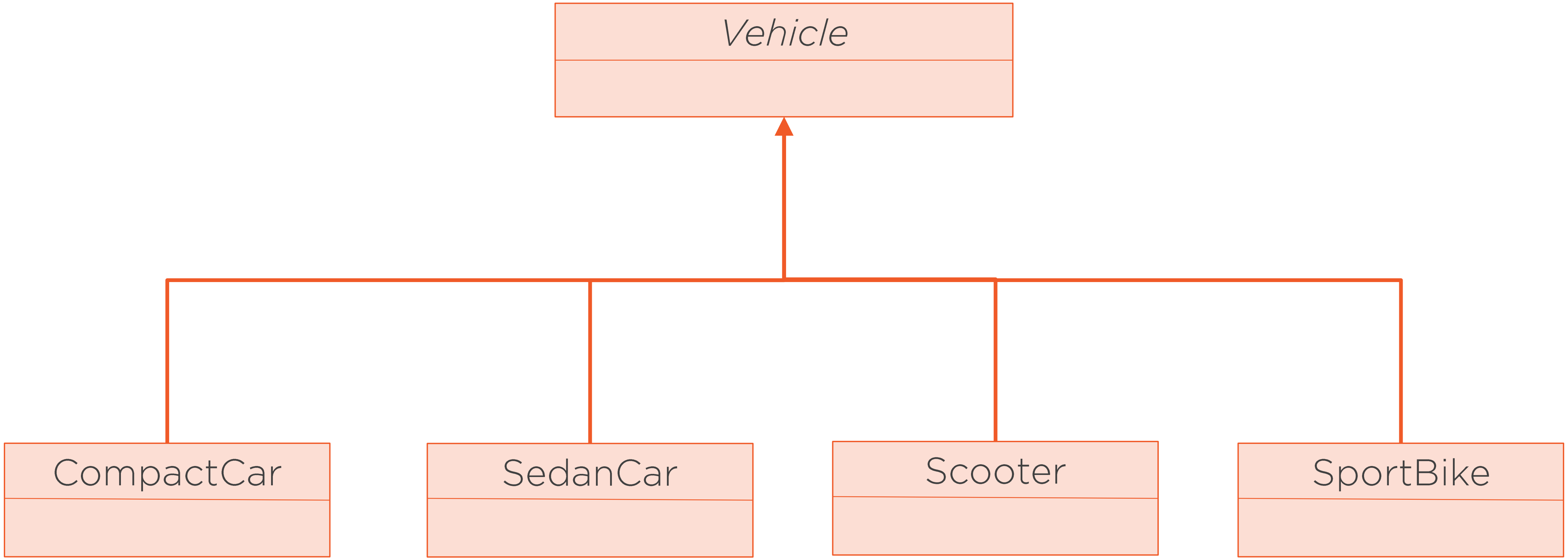
Factory (Template) Method

```java
public abstract class VehicleFactory {
   public Vehicle configureVehicle(String type, String color) {
      Vehicle v = create(type);

      v.setColor(color);
      v.build();

      return v;
   }

   public abstract Vehicle create(String type);
}
```

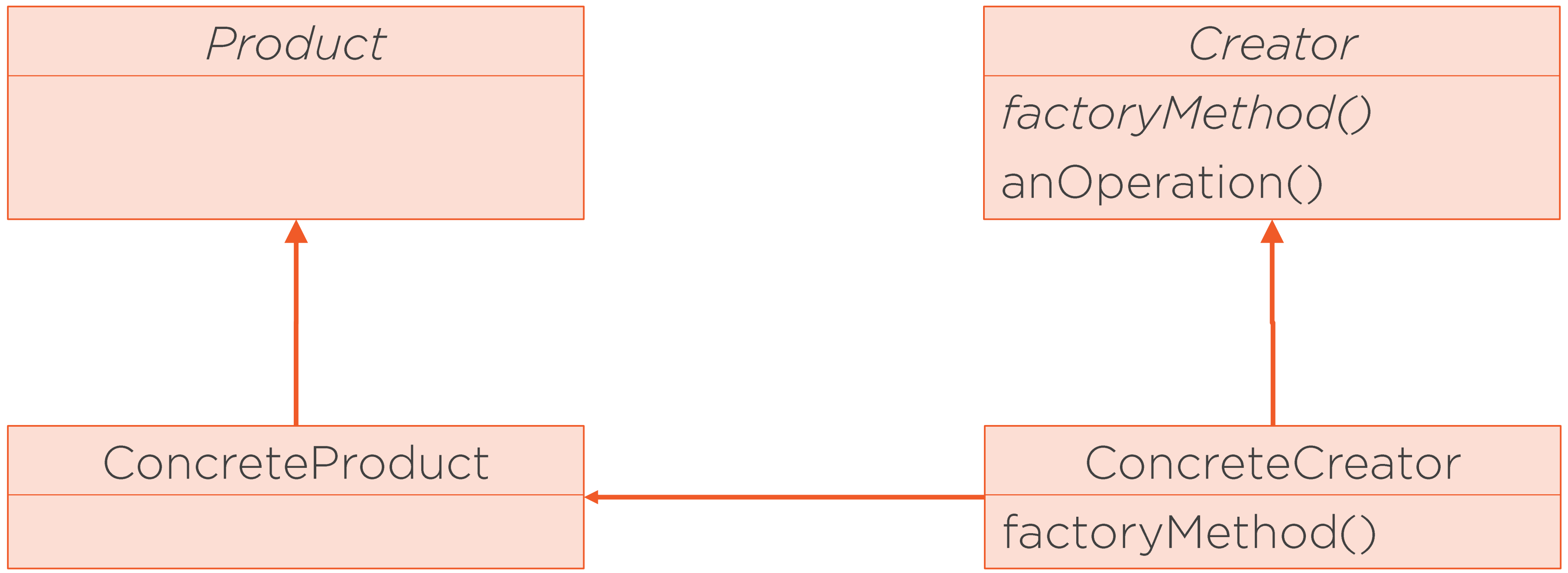
Creator Classes



Product Classes



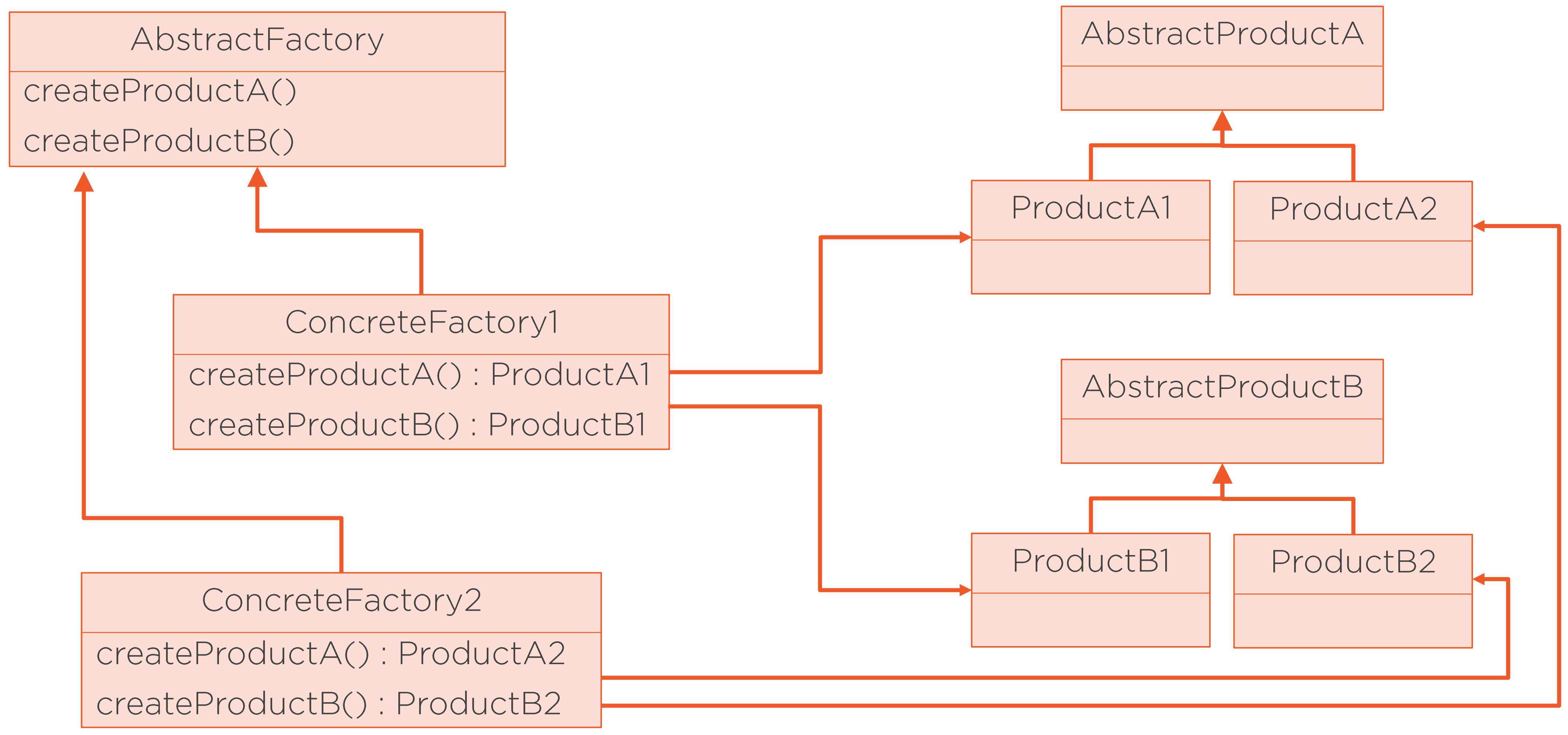
The Factory Method Pattern



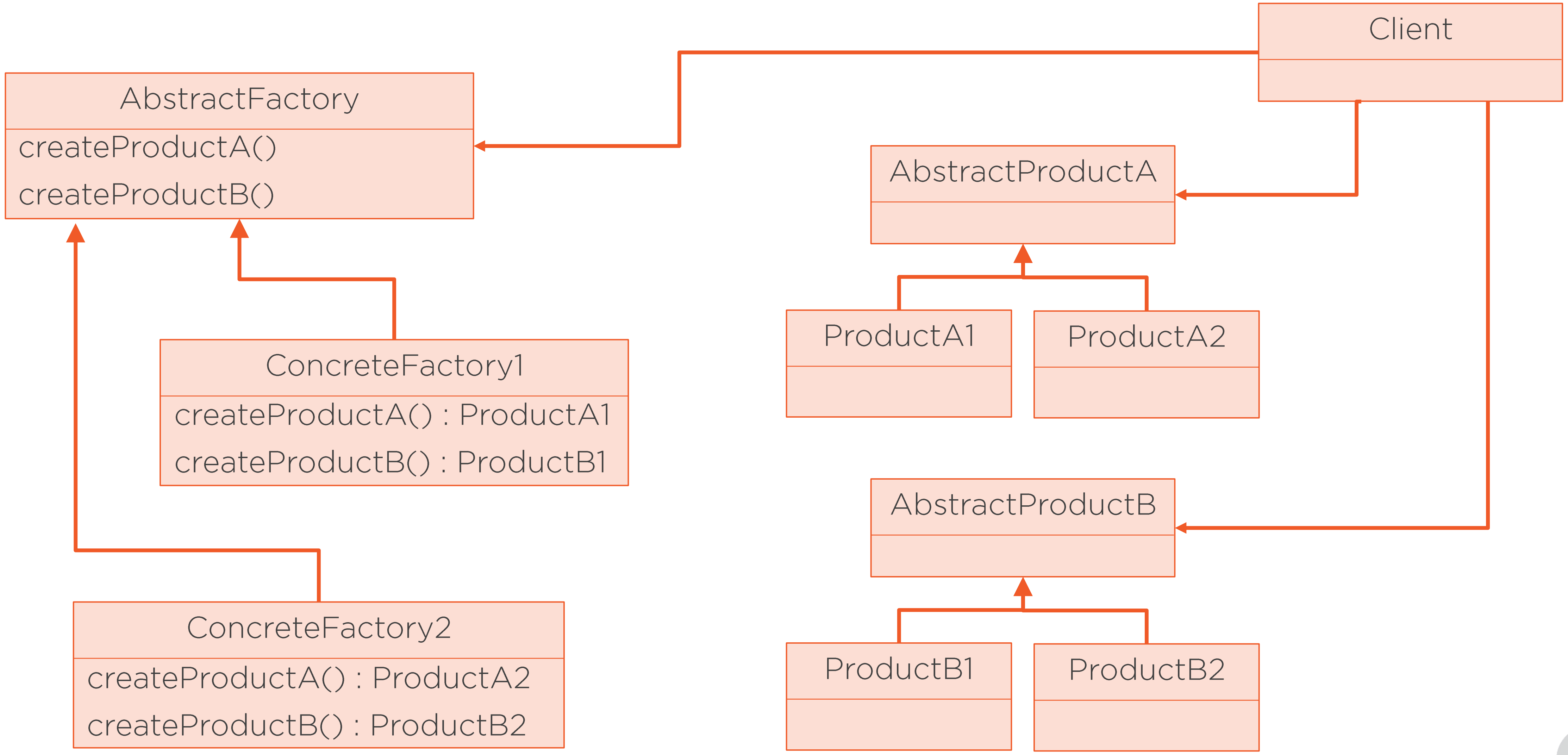
Without Factory Method

```
Vehicle v = null;
if (type.equals("car")) {
    switch(subtype) {
        case "compact": v = new CompactCar(); break;
        case "sedan": v = new SedanCar(); break;
    }
} else if (type.equals("motorcycle")) {
    switch(subtype) {
        case "scooter": v = new Scooter(); break;
        case "sportbike": v = new SportBike(); break;
    }
}
```

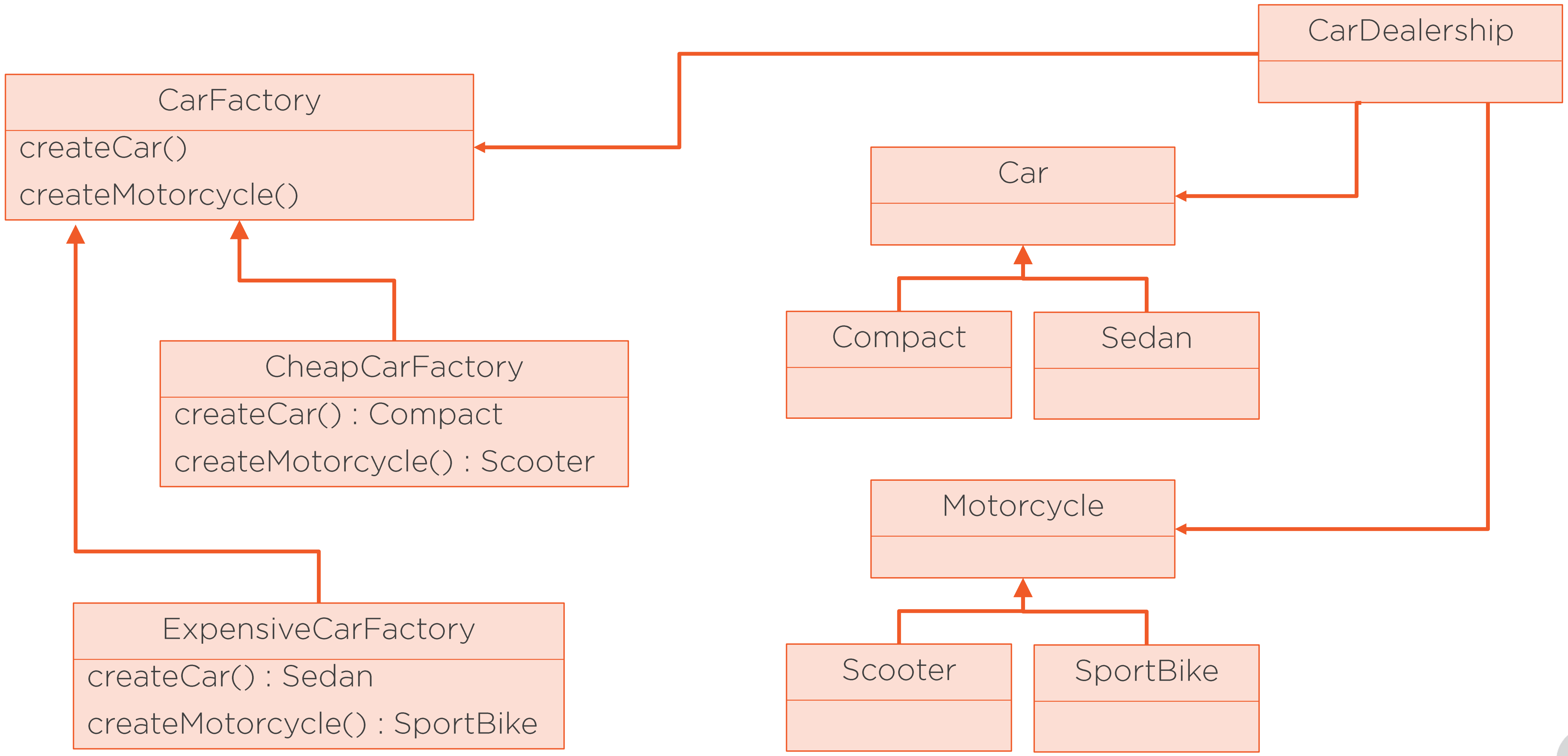
Abstract Factory Pattern



Programming to Interfaces (Abstractions)



Abstract Factory Example



The Builder Pattern

A Simple Class

```java
public class Book {
   private String author;
   private String title;

   public Book(String author, String title) {
      this.author = author;
      this.title = title;
   }
}
```

A Simple Class

```java
public class Book {
   private String author;
   private String title;
   private int pages;

   public Book(String author, String title) {
      // ...
   }

   public Book(String author, String title, int pages) {
      this.author = author;
      this.title = title;
      this.pages = pages;
   }
}
```

A Simple Class?

```
public class Book {
   private String author;
   private String title;
   private int pages;
   private String publisher;

   // ...

   public Book(String author, String title, int pages, String publisher) {
      this.author = author;
      this.title = title;
      this.pages = pages;
      this.publisher = publisher;
   }
}
```

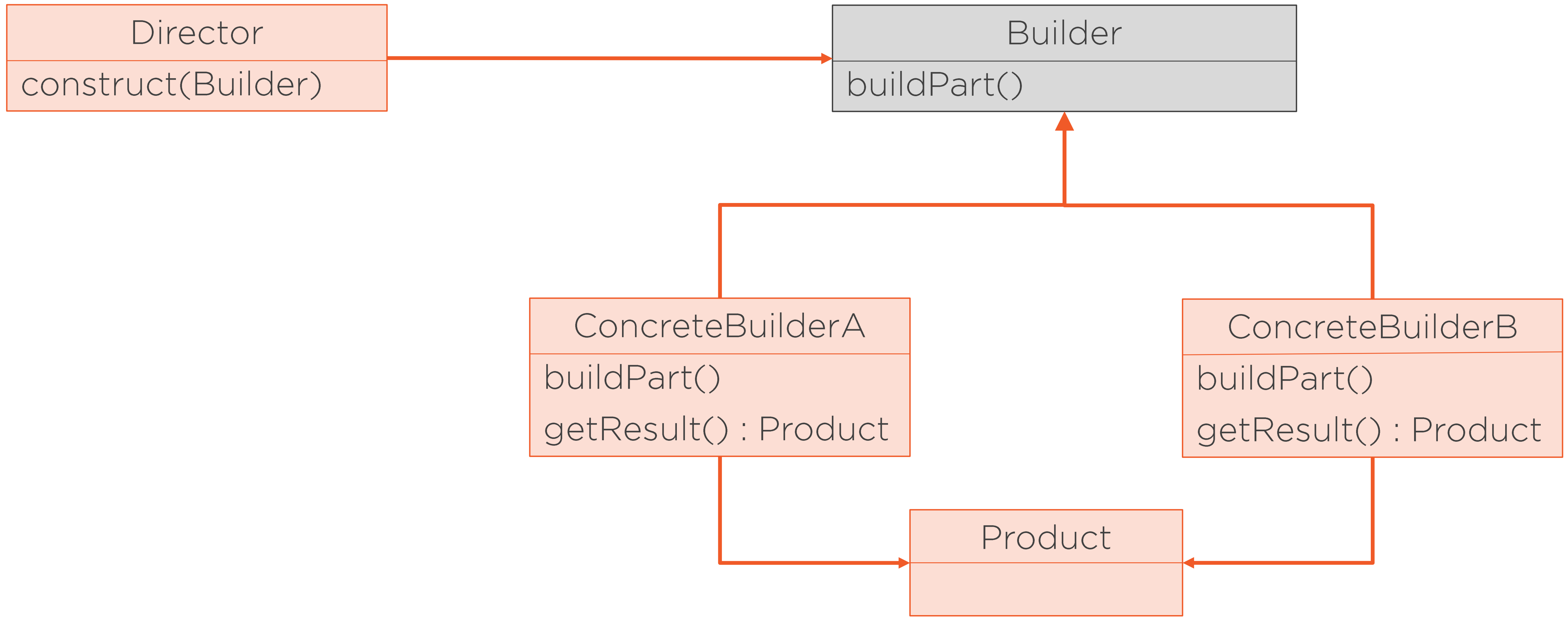
Big Constructor?

```
public class Book {
   private String author;
   private String title;
   private int pages;
   private String publisher;
   private String isbn10;
   private String isbn13;
   private String description;
   private int publicationYear

   public Book(String author, String title, int pages, String publisher,
       String isbn10, String isbn13, String description, int publicationYear) {
      /* ... */
   }
}
```

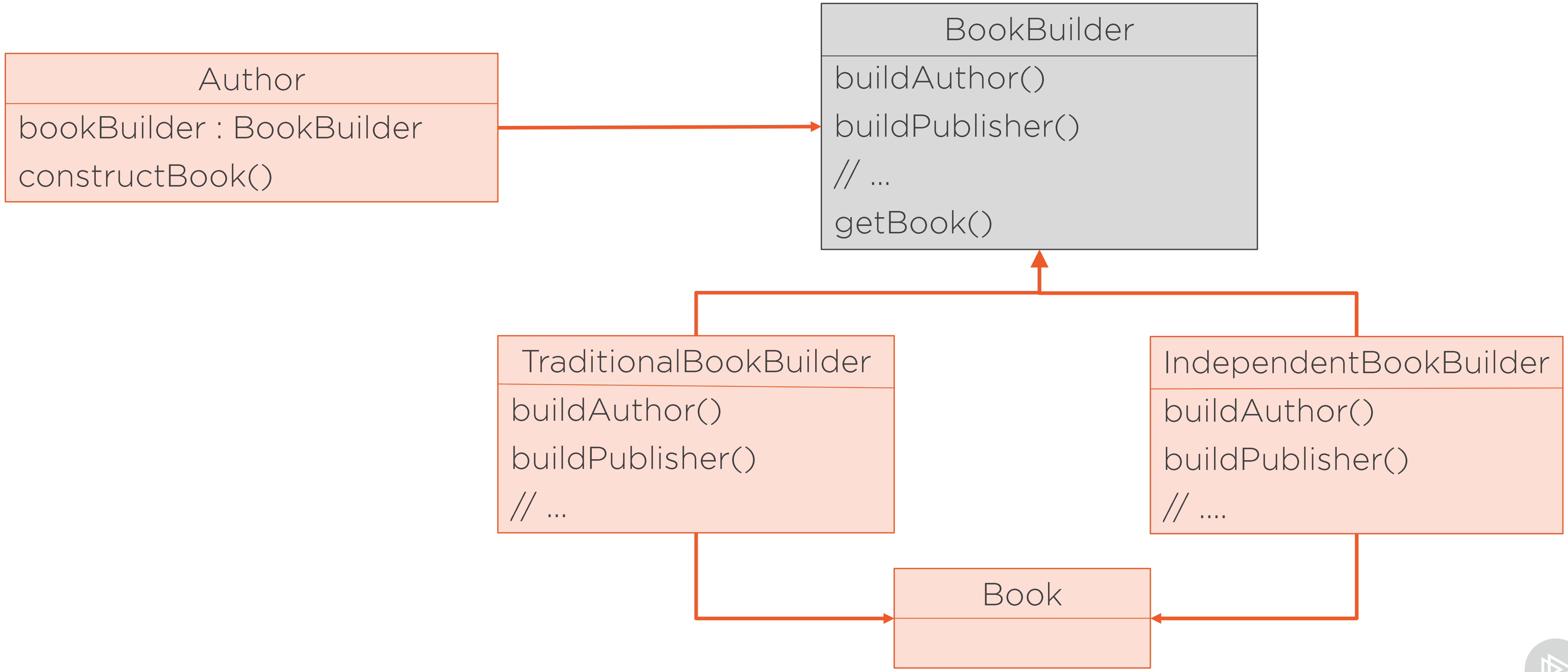
Setter Methods?

```
Book book = new Book();
book.setAuthor("Mary Brown");
book.setTitle("Introduction to Design Patterns");
book.setPages(300);
book.setPublisher("For Programmers");
// ...
```

The Builder Pattern



The Builder Pattern Example



BookBuilder Interface

```
public interface BookBuilder {
   void buildAuthor(String author);
   void buildTitle(String title);
   void buildPages(int pages);
   void buildPublisher();
   void buildIsbn10();
   void buildIsbn13();
   void buildDescription(String description);
   void buildPublicationYear();

   Book getBook();
}
```

IndependentBookBuilder Class

```
public class IndependentBookBuilder implements BookBuilder {
    private Book book;
    public IndependentBookBuilder() {
        this.book = new Book();
    }
   public void buildAuthor(String author) {
      book.setAuthor(author);
   }
   public void String buildPublisher() {
      book.setPublisher(" ");
   }
   // ...
   public Book getBook() {
      return this.book;
   }
}
```

TraditionalBookBuilder Class

```
public class TraditionalBookBuilder implements BookBuilder {
    private Book book;
    public TraditionalBookBuilder() {
        this.book = new Book();
    }
   public void buildAuthor(String author) {
      book.setAuthor(author);
   }
   public void String buildPublisher() {
      book.setPublisher("Traditional Imprint");
   }
   // ...
   public Book getBook() {
      return this.book;
   }
}
```

Author Class

```java
public class Author {
    private BookBuilder bookBuilder;
    private String name;
    // ...
    public void constructBook() {
        this.bookBuilder.buildAuthor(name);
        this.bookBuilder.buildPublisher();
        this.bookBuilder.isbn10();
        // ...
    }
    // ...
    public Book getBook() {
        return this.bookBuilder.getBook();
    }
}
```

Building a Book

```
Author author = new Author();
author.setBookBuilder(new IndependentBookBuilder());

author.constructBook();

Book independentBook = author.getBook();

author.setBookBuilder(new TraditionalBookBuilder());
author.constructBook();
Book traditionalBook = author.getBook();
```

Method Chaining

```
public class IndependentBookBuilder implements BookBuilder {
    // ...
    public BookBuilder buildAuthor(String author) {
        book.setAuthor(author);
        return this;
    }
    public BookBuilder String buildPublisher() {
        book.setPublisher(" ");
        return this;
    }
    // ...
}
```

Author Class

```
public class Author {
   private BookBuilder bookBuilder;
   private String name;
   // ...
   public void constructBook() {
      this.bookBuilder.buildAuthor(name);
               .buildPublisher();
               .isbn10()
      // ...
   }
   // ...
}
```

Simplified Version of the Builder Pattern

```
Book book = Book.Builder.withAuthor(“Bob Anderson")
    .withPublisher("Traditional Imprint")
    .withPages(190)
    .build()
```

Things to Remember

Singleton

- Ensures a class only has one instance, providing a global point of access

Factory Method

- Allows subclasses decide which class to instantiate

Abstract Factory

- Creates hierarchies or families of related objects

Builder

- Separate the construction of an object from its representation