Here is **Item 2: Consider a Builder When Faced with Many Constructor Parameters** from *Effective Java (3rd Edition)*:

**Item 2: Consider a Builder When Faced with Many Constructor Parameters**

Static factories and constructors share a limitation: they do not scale well to large numbers of optional parameters. Consider the case of a class representing the **Nutrition Facts label** that appears on packaged foods. These labels have a few required fields (**serving size, servings per container, calories per serving**) and more than twenty optional fields (**total fat, saturated fat, trans fat, cholesterol, sodium, etc.**). Most products have nonzero values for only a few of these optional fields.

**Problems with Traditional Approaches**

**1. Telescoping Constructor Pattern**

One approach to handle optional parameters is to provide multiple constructors, each with a different combination of required and optional parameters. This is known as the **Telescoping Constructor Pattern**.

Example:

public class NutritionFacts {  
 private final int servingSize; // (mL) required  
 private final int servings; // (per container) required  
 private final int calories; // (per serving) optional  
 private final int fat; // (g/serving) optional  
 private final int sodium; // (mg/serving) optional  
 private final int carbohydrate; // (g/serving) optional  
  
 public NutritionFacts(int servingSize, int servings) {  
 this(servingSize, servings, 0);  
 }  
  
 public NutritionFacts(int servingSize, int servings, int calories) {  
 this(servingSize, servings, calories, 0);  
 }  
  
 public NutritionFacts(int servingSize, int servings, int calories, int fat) {  
 this(servingSize, servings, calories, fat, 0);  
 }  
  
 public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {  
 this(servingSize, servings, calories, fat, sodium, 0);  
 }  
  
 public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium, int carbohydrate) {  
 this.servingSize = servingSize;  
 this.servings = servings;  
 this.calories = calories;  
 this.fat = fat;  
 this.sodium = sodium;  
 this.carbohydrate = carbohydrate;  
 }  
}

While this pattern works, it **does not scale well**. If a class has **many optional parameters**, you end up with a **large number of overloaded constructors**, making code hard to maintain and read.

**2. JavaBeans Pattern**

Another approach is to use a **JavaBeans pattern**, where you create an object using a parameterless constructor and set properties using setter methods.

Example:

public class NutritionFacts {  
 private int servingSize = -1; // Required; no default value  
 private int servings = -1; // Required; no default value  
 private int calories = 0;  
 private int fat = 0;  
 private int sodium = 0;  
 private int carbohydrate = 0;  
  
 public NutritionFacts() { }  
  
 public void setServingSize(int servingSize) { this.servingSize = servingSize; }  
 public void setServings(int servings) { this.servings = servings; }  
 public void setCalories(int calories) { this.calories = calories; }  
 public void setFat(int fat) { this.fat = fat; }  
 public void setSodium(int sodium) { this.sodium = sodium; }  
 public void setCarbohydrate(int carbohydrate) { this.carbohydrate = carbohydrate; }  
}  
Usage:  
NutritionFacts cocaCola = new NutritionFacts();  
cocaCola.setServingSize(240);  
cocaCola.setServings(8);  
cocaCola.setCalories(100);  
cocaCola.setSodium(35);  
cocaCola.setCarbohydrate(27);

**Issues with JavaBeans Pattern:**

* **Mutability:** Objects are no longer immutable, making them **not thread-safe**.

**What Does It Mean?**

* In the **JavaBeans pattern**, an object is created using a **no-argument constructor**, and its fields are set using **setter methods**.
* This means that the **internal state of the object can change** after it is created.
* In contrast, an **immutable object** (like String or Integer) **cannot be changed** after its creation.

**Why is Mutability a Problem?**

* **Thread-Safety Issues**:
  + If multiple threads access a **mutable object**, and at least one thread modifies it, the object is **not thread-safe**.
  + Race conditions can occur when **one thread modifies the object while another reads it**, leading to inconsistent results.
  + Synchronization mechanisms (like synchronized methods or locks) may be needed to ensure correctness, which can introduce performance overhead.

**Example of Thread Safety Issue**

public class NutritionFacts {  
 private int servingSize;  
 private int servings;  
 private int calories;  
  
 public NutritionFacts() {}  
  
 public void setServingSize(int servingSize) { this.servingSize = servingSize; }  
 public void setServings(int servings) { this.servings = servings; }  
 public void setCalories(int calories) { this.calories = calories; }  
}

Now, let’s see how two threads can interfere with each other:

NutritionFacts facts = new NutritionFacts();  
  
// Thread 1: Sets values  
new Thread(() -> {  
 facts.setServingSize(250);  
 facts.setServings(2);  
}).start();  
  
// Thread 2: Reads values at the same time  
new Thread(() -> {  
 System.out.println("Serving Size: " + facts.getServingSize());  
 System.out.println("Servings: " + facts.getServings());  
 }).start();

**Potential Problem**

* If **Thread 2 reads the object before Thread 1 finishes setting all values**, it might print an **incomplete** or **inconsistent state**, such as:

Serving Size: 250  
Servings: 0 // The second thread reads before Thread 1 updates it!

* The object is **not thread-safe**, and its state **can change unexpectedly**.

**Solution: Immutability with Builder**

* If we use a **Builder pattern**, we can ensure that the object is fully initialized before it is used.
* Once built, the object is **immutable**, meaning **its state cannot change** after creation.
* **Thread safety is ensured** because there are **no setters** that can modify the object.
* **Inconsistent state:** Since setters can be called in any order, the object may exist in an **invalid or incomplete state** at any point in time.

### ****What Does It Mean?****

* Since the JavaBeans pattern uses **setters** to initialize an object, a programmer might **forget to set some required fields** or **set them in the wrong order**.
* This means the object can **exist in an invalid or incomplete state**.

### ****Why Is This a Problem?****

* **Objects should always be in a valid state** after creation.
* If an object has **required fields**, it should not be possible to create it without providing values for those fields.
* With setters, there's **no guarantee** that all required fields have been set before the object is used.

### ****Example of Inconsistent State****

### Usage: Forgetting to Set Required Field

NutritionFacts facts = new NutritionFacts();  
facts.setServingSize(250); // Set serving size  
facts.setCalories(100); // Set calories  
// Forgot to set 'servings'  
System.out.println(facts.getServings()); // Prints 0 (invalid state)

### ****The Problem Here****

* The servings field is **left uninitialized** (0 by default), which **might not make sense** for the object.
* There is **no way to enforce that all required fields are set** before using the object.

### ****Solution: Using a Constructor or Builder****

* If we use a **constructor**, we can **require** mandatory fields, ensuring a valid object.
* If we use a **Builder pattern**, we can ensure that the object **cannot be used before all required fields are provided**

1. **Solution: The Builder Pattern**

To address these issues, Joshua Bloch recommends using the **Builder Pattern**. A builder provides a **fluent interface** to construct an object step by step.

Instead of making the desired object directly, the client calls a constructor (or static factory) with all of the required parameters and gets a builder object. Then the client calls setter-like methods on the builder object to set each optional parameter of interest. Finally, the client calls a parameterless build method to generate the object, which is typically immutable.e. The builder is typically a static member class (Item 24) of the class it builds. Here’s how it looks in practice:

**Builder Implementation**

public class NutritionFacts {

private final int servingSize;

private final int servings;

private final int calories;

private final int fat;

private final int sodium;

private final int carbohydrate;

public static class Builder {

// Required parameters

private final int servingSize;

private final int servings;

// Optional parameters - initialized to default values

private int calories = 0;

private int fat = 0;

private int sodium = 0;

private int carbohydrate = 0;

public Builder(int servingSize, int servings) {

this.servingSize = servingSize;

this.servings = servings;

}

public Builder calories(int val) { calories = val; return this; }

public Builder fat(int val) { fat = val; return this; }

public Builder sodium(int val) { sodium = val; return this; }

public Builder carbohydrate(int val) { carbohydrate = val; return this; }

public NutritionFacts build() {

return new NutritionFacts(this);

}

}

private NutritionFacts(Builder builder) {

servingSize = builder.servingSize;

servings = builder.servings;

calories = builder.calories;

fat = builder.fat;

sodium = builder.sodium;

carbohydrate = builder.carbohydrate;

}

}public class NutritionFacts {  
 private final int servingSize;  
 private final int servings;  
 private final int calories;  
 private final int fat;  
 private final int sodium;  
 private final int carbohydrate;  
  
 public static class Builder {  
 // Required parameters  
 private final int servingSize;  
 private final int servings;  
  
 // Optional parameters - initialized to default values  
 private int calories = 0;  
 private int fat = 0;  
 private int sodium = 0;  
 private int carbohydrate = 0;  
  
 public Builder(int servingSize, int servings) {  
 this.servingSize = servingSize;  
 this.servings = servings;  
 }  
  
 public Builder calories(int val) { calories = val; return this; }  
 public Builder fat(int val) { fat = val; return this; }  
 public Builder sodium(int val) { sodium = val; return this; }  
 public Builder carbohydrate(int val) { carbohydrate = val; return this; }  
  
 public NutritionFacts build() {  
 return new NutritionFacts(this);  
 }  
 }  
  
 private NutritionFacts(Builder builder) {  
 servingSize = builder.servingSize;  
 servings = builder.servings;  
 calories = builder.calories;  
 fat = builder.fat;  
 sodium = builder.sodium;  
 carbohydrate = builder.carbohydrate;  
 }  
}

**Using the Builder**

NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)  
 .calories(100)  
 .sodium(35)  
 .carbohydrate(27)  
 .build();

The Builder pattern simulates named optional parameters as found in Python and Scala.

Validity checks were omitted for brevity. To detect invalid parameters as soon as possible, you can check parameter validity in the builder’s constructor and methods.

**The Challenge in Inheritance and subclass:**

Implementing the Builder design pattern in the context of inheritance presents unique challenges, primarily due to the need for method chaining across hierarchical classes. In Chapter 2 of *Effective Java*, Joshua Bloch introduces the Builder pattern as a solution for constructing complex objects. However, when extending this pattern to subclasses, ensuring that each builder method returns the correct type becomes intricate.

In a class hierarchy, such as Vehicle (base class), Car (subclass), and ElectricCar (subclass of Car), each class may have its own builder. The primary issue arises when a method in a subclass's builder calls a method from its superclass's builder, which returns an instance of the superclass's builder, disrupting the fluent interface. This can lead to compilation errors or require casting, both of which are undesirable.

**The Solution:**

To address this, we can employ the *curiously recurring template pattern* (CRTP) using Java generics. This approach ensures that each builder method returns the correct builder type, maintaining the fluent interface across the inheritance hierarchy.

**Example Implementation:**

1. **Base Class and Builder:**
2. public class Vehicle3 {  
    private final String make;  
    private final String model;  
     
    protected Vehicle3(VehicleBuilder3<?, ?> builder) {  
    this.make = builder.make;  
    this.model = builder.model;  
    }  
     
    public static VehicleBuilder3<?, ?> builder() {  
    return new VehicleBuilder3Impl();  
    }  
     
    public static abstract class VehicleBuilder3<C extends Vehicle3, B extends VehicleBuilder3<C, B>> {  
    private String make;  
    private String model;  
     
    protected abstract B self();  
     
    public B make(String make) {  
    this.make = make;  
    return self();  
    }  
     
    public B model(String model) {  
    this.model = model;  
    return self();  
    }  
     
    public abstract C build();  
    }  
     
    private static class VehicleBuilder3Impl extends VehicleBuilder3<Vehicle3, VehicleBuilder3Impl> {  
    @Override  
    protected VehicleBuilder3Impl self() {  
    return this;  
    }  
     
    @Override  
    public Vehicle3 build() {  
    return new Vehicle3(this);  
    }  
    }  
   }

Subclass and Its Builder:

package org.example.vehicle.thirdsolution;  
  
  
public class Car3 extends Vehicle3 {  
 private final int seatingCapacity;  
  
 protected Car3(Car3Builder<?, ?> builder) {  
 super(builder);  
 this.seatingCapacity = builder.seatingCapacity;  
 }  
  
 public static Car3Builder<?, ?> builder() {  
 return new Car3BuilderImpl();  
 }  
  
 public static abstract class Car3Builder<C extends Car3, B extends Car3Builder<C, B>>  
 extends VehicleBuilder3<C, B> {  
 private int seatingCapacity;  
  
 public B seatingCapacity(int seatingCapacity) {  
 this.seatingCapacity = seatingCapacity;  
 return self();  
 }  
 }  
  
 private static class Car3BuilderImpl extends Car3Builder<Car3, Car3BuilderImpl> {  
 @Override  
 protected Car3BuilderImpl self() {  
 return this;  
 }  
  
 @Override  
 public Car3 build() {  
 return new Car3(this);  
 }  
 }  
}

Usage:

Car3 car3 = Car3.*builder*()  
 .seatingCapacity(3)  
 .make("Ford")  
 .model("F")  
 .build();

In this implementation:

* Each builder class is parameterized with its corresponding product type (C) and the builder type itself (B).
* The self() method, abstract in the base builder, is implemented to return this in each concrete builder subclass.
* This pattern ensures that method calls in the builder chain return the correct builder type, preserving the fluent interface across the inheritance hierarchy.

By adopting this approach, as detailed you can effectively manage object construction in complex class hierarchies while maintaining clean and readable code.

**Advantages of the Builder Pattern**

1. **Readable Code**
   * The builder provides a fluent API, making object creation **easy to read**.
   * The parameter names make it **clear what each value represents**.
2. **Immutability**
   * The object fields are **final**, making the class immutable and **thread-safe**.
3. **Scalability**
   * New optional fields can be **added easily** without modifying existing constructors.
4. **Validation**
   * The builder can **validate parameters** before creating the object.

**When to Use the Builder Pattern**

✅ Use the Builder pattern when:

* The class has **many optional parameters**.
* You want **readable** and **maintainable** code.
* You need an **immutable** object.
* The object construction requires **complex validation**.

❌ Avoid the Builder pattern when:

* The class has **few parameters** (constructors or static factories might be simpler).
* Performance is **critical** (builders introduce extra objects).

**Conclusion**

* **Telescoping constructors** lead to unreadable and inflexible code.
* **JavaBeans pattern** causes **mutability issues** and potential **inconsistencies**.
* **The Builder pattern** solves these problems by:
  + Making object creation **clear and readable**.
  + Ensuring **immutability and thread safety**.
  + Allowing **flexibility** when adding new fields.

Joshua Bloch's *Effective Java* strongly recommends the **Builder pattern** as the best practice for handling **large numbers of constructor parameters**.