Optional Parameters in Java: Strategies and Approaches

by Eugen Paraschiv MVB · **May. 04, 18 · Java Zone · Tutorial**

Unlike some languages such as Kotlin and Python, Java doesn't provide built-in support for optional parameter values. Callers of a method must supply all of the variables defined in the method declaration.

In this article, we'll explore some strategies for dealing with optional parameters in Java. We'll look at the strengths and weaknesses of each approach and highlight the trade-offs involved with selecting one strategy over another.

Example Overview

Let's consider a simple *MultiVitamin* class for our use here:

```
public class MultiVitamin {

    private String name;    // required
    private int vitaminA;   // in mcg
    private int vitaminC;   // in mg
    private int calcium;    // in mg
    private int iron;       // in mg

    // constructor(s)
}
```

The logic responsible for creating new instances of a *MultiVitamin* for men may, for example, need to supply a larger value for iron. Instances of a *MultiVitamin* for women might require more calcium. Essentially, each variant supported by the system might require values for some parameters but would prefer to supply known default values for the optional ones.

Constraining how instances can be created can generally lead to APIs that are easier to read and use as intended.

Method Overloading/Telescoping Constructors

When working with optional parameters, method overloading is one of the more obvious and common

approaches available.

The idea here is that we start with a method that only takes the required parameters. We provide an additional method which takes a single optional parameter. We then provide yet another method which takes two of these parameters, and so on.

The methods which take fewer parameters supply default values for the more verbose signatures:

```
static final int DEFAULT_IRON_AMOUNT = 20;

// instance fields

public MultiVitaminOverloading(
  String name) {
    this(name, 0);
}

public MultiVitaminOverloading(
  String name, int vitaminA) {
    this(name, vitaminA, 0);
}

public MultiVitaminOverloading(
  String name, int vitaminA, int vitaminC) {
    this(name, vitaminA, vitaminC, 0);
}

public MultiVitaminOverloading(
  String name, int vitaminA, int vitaminC, int calcium) {
    this(name, vitaminA, vitaminC, calcium, DEFAULT_IRON_AMOUNT);
}

public MultiVitaminOverloading (
  String name,
  int vitaminA,
  int vitaminC,
  int calcium,
  int iron) {
    this.name = name;
    this.vitaminA = vitaminA;
    this.vitaminC = vitaminC;
    this.calcium = calcium;
    this.iron = iron;
}

// getters
```

We can observe the *telescoping* property of these signatures in this example; they flow to the right as we're adding more parameters.

The simplicity and familiarity of the method overloading approach make it **a good choice for use cases with a small number of optional parameters**. We can extract default values for any optional parameters to a named constant to improve readability as we've done here with *DEFAULT_IRON_AMOUNT*.

Also, note that using this approach does not prevent us from making the class immutable. We can ensure that instances of the class are thread-safe and always in a consistent state by declaring the instance fields as final and only providing getters.

The main downside of using this approach is that it does not scale well – as the number of parameters increases.*MultiVitaminOverloading* is already difficult to read and maintain with only four optional parameters.

This only gets worse with the fact that our optional parameters are of the same type. Clients could easily order the parameters wrongly – such a mistake would not be noticed by the compiler and would likely result in a subtle bug at runtime.

Consider using this if the number of optional parameters is small and if the risk of callers supplying parameters in the wrong order is minimal.

Static Factory Methods

Joshua Bloch, in his book – Effective Java, recommends in Item 1, to "...consider static factory methods instead of constructors." With this approach, **static methods with particular names can be used instead of public constructors to clarify the API** used for instance creation:

```
1  // constants
2
3  // instance fields
4
5  public static MultiVitaminStaticFactoryMethods forMen(String name) {
6      return new MultiVitaminStaticFactoryMethods(
7        name, 5000, 60, CALCIUM_AMT_DEF, IRON_AMT_MEN);
8  }
9
10 public static MultiVitaminStaticFactoryMethods forWomen(String name) {
11     return new MultiVitaminStaticFactoryMethods(
12       name, 5000, 60, CALCIUM_AMT_WOMEN, IRON_AMT_DEF);
13 }
14
15 private MultiVitaminStaticFactoryMethods(
16   String name,
17   int vitaminA,
18   int vitaminC,
19   int calcium,
20   int iron) {
21     this.name = name;
```

```
22      this.vitaminA = vitaminA;
23      this.vitaminC = vitaminC;
24      this.calcium = calcium;
25      this.iron = iron;
26  }
27
28  // getters
```

The idea here is to **carefully pair method names with signatures so that the intention is obvious**. We define one or more private constructors, and call them only by the named factory methods.

By making our constructors private, the caller must make an explicit choice of signature based on the desired parameters. The author then has complete control over which methods to provide, how to name them, and what defaults will the parameters, that are not supplied by the caller, have.

While simple to implement and understand, **this approach also does not scale well** with a large number of optional parameters.

This strategy is often the best choice if the number of optional parameters is small and if we can choose descriptive names for each variant.

The Builder Pattern Approach

The Builder pattern is another way of handling optional parameters but takes a little bit of work to set up.

We start by defining our class with a private constructor but then introduce a static nested class to function as a builder. The builder class exposes methods for setting parameters and for building the instance.

Creating instances of the class involves making use of the builder's fluent API – passing in the mandatory parameters, setting any optional parameters, and calling the *build()* method:

```
1  MultiVitaminWithBuilder vitamin
2    = new MultiVitaminWithBuilder.MultiVitaminBuilder("Maximum Strength")
3      .withCalcium(100)
4      .withIron(200)
5      .withVitaminA(50)
6      .withVitaminC(1000)
7      .build();
```

We can now define our *MultiVitaminBuilder* as a static nested class of the enclosing type.

This allows us to keep the constructor of the enclosing type private and forces callers to use the builder:

```
1  public static class MultiVitaminBuilder {
2      private static final int ZERO = 0;
3      private final String name; // required
4      private final int vitaminA = ZERO;
5      // other params
```

```
6
7       public MultiVitaminBuilder(String name) {
8           this.name = name;
9       }
10
11      public MultiVitaminBuilder withVitaminA(int vitaminA) {
12          this.vitaminA = vitaminA;
13          return this;
14      }
15
16      // other fluent api methods
17
18      public MultiVitaminWithBuilder build() {
19          return new MultiVitaminWithBuilder(this);
20      }
21  }
```

One of the main advantages of the builder pattern is that **it scales well with large numbers of optional and mandatory parameters**.

In our example here, we require the mandatory parameter in the constructor of the builder. We expose all of the optional parameters in the rest of the builder's API.

Another advantage is that it's **much more difficult to make a mistake** when setting values for optional parameters. We have explicit methods for each optional parameter, and we don't expose callers to bugs that can arise due to calling methods with parameters that are in the wrong order.

Lastly, the builder approach cleanly provides us with a finely grained level of control over validation. With our builder, we know the instance we create is in a valid state and we won't be able to alter it.

The most obvious downside to using a builder is that **it's way more complicated to set up**. The purpose of the construct might not be immediately apparent to a novice developer.

The builder pattern should be considered for use cases involving a large number of mandatory and optional parameters. Additionally, consider this strategy when supplied values are well-served by fine-grained validation or other constraints.

For detailed sample code and a more thorough walkthrough of this strategy, check out this article on creational patterns.

Mutability With Accessors

Using standard getters and setters is a simple way to work with an object that has optional instance parameters.

We're using a default constructor with mandatory parameters to create the object.

We're then invoking the setter methods to set the value of each optional parameter as needed. We can set the default values for optional parameters within a constructor, if necessary:

```
public class MultiVitamin {

    private String name;    // required
    private int vitaminA;   // in mcg

    // other instance params

    public MultiVitamin(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int getVitaminA() {
        return vitaminA;
    }

    public void setVitaminA(int vitaminA) {
        this.vitaminA = vitaminA;
    }

    // other getters and setters
}
```

This approach is the ubiquitous *JavaBeans* pattern and is likely the simplest strategy available for working with optional parameters. There are two key strengths this approach has over alternatives.

The pattern is arguably the most familiar of them all. Nearly all modern IDE's can automatically generate the necessary code given the class definition.

There are, unfortunately, serious drawbacks to using this approach, especially if thread safety is a concern. The use of this pattern requires that the object is mutable since we can change it after its creation.

Since the creation of the instance and setting of its state are decoupled and do not occur atomically, it's possible that the instance could be used before it's in a valid state. In a sense, we're splitting the construction of the object over multiple calls.

You can consider this pattern when thread safety and creating a robust API isn't a primary concern.

Allowing Nulls

It's typically a bad idea to allow method callers to supply null values and this widely considered an anti-pattern.

For the sake of demonstration, let's see what this looks like in practice:

```
MultiVitaminAllowingNulls vitamin
    = new MultiVitaminAllowingNulls("Unsafe Vitamin", null, null, null, null);
```

```
2     = new MultiVitaminAllowingNulls( "unsafe vitamin", null, null, null, null);
```

The strategy of allowing nulls for optional parameters offers nothing when compared to alternatives. In order to be sure that nulls are allowed, the caller needs to know the implementation details of the class. This fact alone makes this strategy a poor choice.

Also, the code itself does not read well. **Simply put, you should avoid this pattern whenever possible.**

Varargs

Java 5 added variable-length arguments to provide a way of to declare that a method accepts 0 or more arguments of a specified type. There are certain restrictions on the usage of varags that are in place to avoid ambiguity:

- there can be only one variable argument parameter
- the variable argument parameter must be the last in the method signature

The restrictions placed on varargs make it a viable solution in only a small set of use cases.

The following block shows a well-formed, but a contrived example:

```
1  public void processVarargIntegers(String label, Integer... others) {
2      System.out.println(
3        String.format("processing %s arguments for %s", others.length, label));
4      Arrays.asList(others)
5        .forEach(System.out::println);
6  }
```

Given that usage of varargs requires only one variable argument parameter, it may be tempting to declare *Object* as the type and then perform custom logic within the method to check each parameter and cast as necessary.

This is not ideal, because it requires the caller to have intimate knowledge of the method implementation to use it safely. Also, the logic required within the method implementation can be messy and hard to maintain.

You can try to use varargs for any method signature that contains an optional parameter – which cleanly maps to 0 or more values of the same type.

And you can read this writeup for a more thorough walkthrough of varargs.

Conclusion

In this article, we've looked at a variety of strategies for working with optional parameters in Java, such as method overloading, the builder pattern, and the ill-advised strategy of allowing callers to supply null values.

We highlighted the relative strengths and weaknesses of each strategy and provided usage for each. Also, we took a quick look at the varargs construct as an additional means of supporting optional parameters in more generalized method signatures.

As always, all source code used in this article can be found over on GitHub.