Chapter 2

Item 1: Consider static factory methods instead of constructors

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique a class can provide a public *static factory method*, that returns an instance of the class. Here’s a simple example from Boolean.

public static Boolean valueOf(boolean b) {

return b ? Boolean.TRUE : Boolean.FALSE;

}

Note that a static factory method is not the same as the *Factory Method* pattern from *Design Patterns.*

**Advantages**

1. **One advantage of static factory methods is that, unlike constructors, they have names.**

A class can have only a single constructor with a given signature. programmers have been known to get around this restriction by providing two constructors whose parameter lists differ only in the order of their parameter types. This is a really bad idea. The user of such an API will never be able to remember which constructor is which and will end up calling the wrong one by mistake. People reading code that uses these constructors will not know what the code does without referring to the class documentation.

1. **A second advantage of static factory methods is that, unlike constructors, they are not required to create a new object each time they’re invoked.**

allows immutable classes to use preconstructed instances, or to cache instances as they’re constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects. The Boolean.valueOf(boolean) method illustrates this technique: it *never* creates an object. This technique is similar to the *Flyweight* pattern. It can greatly improve performance if equivalent objects are requested often, especially if they are expensive to create.

1. **A third advantage of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type.**
2. **A fourth advantage of static factories is that the class of the returned object can vary from call to call as a function of the input parameters.**
3. **A fifth advantage of static factories is that the class of the returned object need not exist when the class containing the method is written.**

**Disadvantages**

1. **The main limitation of providing only static factory methods is that classes without public or protected constructors cannot be sub classed.**

They do not stand out in API documentation in the way that constructors do, so it can be difficult to figure out how to instantiate a class that provides static factory methods instead of constructors. The Javadoc tool may someday draw attention to static factory methods. In the meantime, you can reduce this problem by drawing attention to static factories in class or interface documentation and by adhering to common naming conventions. Here are some common names for static factory methods. This list is far from exhaustive:

1. **A second shortcoming of static factory methods is that they are hard for programmers to find.**

They do not stand out in API documentation in the way that constructors do, so it can be difficult to figure out how to instantiate a class that provides static factory methods instead of constructors. The Javadoc tool may someday draw attention to static factory methods. In the meantime, you can reduce this problem by drawing attention to static factories in class or interface documentation and by adhering to common naming conventions. Here are some common names for static factory methods. This list is far from exhaustive:

• **from**—A *type-conversion method* that takes a single parameter and returns a

corresponding instance of this type, for example:

Date d = Date.from(instant);

• **of**—An *aggregation method* that takes multiple parameters and returns an instance of this type that incorporates them, for example:

Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);

• **valueOf**—A more verbose alternative to from and of, for example:

BigInteger prime = BigInteger.valueOf(Integer.MAX\_VALUE);

• **instance** or **getInstance**—Returns an instance that is described by its parameters (if any) but cannot be said to have the same value, for example:

StackWalker luke = StackWalker.getInstance(options);

• **create** or **newInstance**—Like instance or getInstance, except that the method guarantees that each call returns a new instance, for example:

Object newArray = Array.newInstance(classObject, arrayLen);

• **get*Type***—Like getInstance, but used if the factory method is in a different class. *Type* is the type of object returned by the factory method, for example:

FileStore fs = Files.getFileStore(path);

• **new*Type***—Like newInstance, but used if the factory method is in a different class. *Type* is the type of object returned by the factory method, for example:

BufferedReader br = Files.newBufferedReader(path);

• ***type***—A concise alternative to get*Type* and new*Type*, for example:

List<Complaint> litany = Collections.list(legacyLitany);

In summary, static factory methods and public constructors both have their uses, and it pays to understand their relative merits. Often static factories are preferable, so avoid the reflex to provide public constructors without first considering static factories.

Item 2: Consider a builder when faced with many constructor parameters

1. **the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.**

public class NutritionFacts {

private final int servingSize;

private final int servings;

private final int calories;

private int fat = 0;

private int sodium = 0;

private int carbohydrate = 0;

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);

}

}

NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);

six parameters this may not seem so bad, but it quickly gets out of hand as the number of parameters increases.

The reader is left wondering what all those values mean and must carefully count parameters to find out. Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won’t complain, but the program will misbehave at runtime.

**A second alternative** when you’re faced with many optional parameters in a constructor is the *JavaBeans* pattern, in which you call a parameter less constructor to create the object and then call setter methods to set each required parameter

NutritionFacts cocaCola = new NutritionFacts();

cocaCola.setServingSize(240);

cocaCola.setServings(8);

cocaCola.setCalories(100);

cocaCola.setSodium(35);

cocaCola.setCarbohydrate(27);

Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls.

1. **a JavaBean may be in an inconsistent state partway through its construction. with JavaBeans**

**pattern it’s not possible to make a class immutable**

Attempting to use an object when it’s in an inconsistent state may cause failures. It is possible to reduce these disadvantages by manually “freezing” the object when its construction is complete and not allowing it to be used until frozen, but this variant is unwieldy and rarely used in practice. Moreover, it can cause errors at runtime because the compiler cannot ensure that the programmer calls the freeze method on an object before using it.

**Luckily, there is a third alternative** that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern.

**// Builder Pattern**

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;

private NutritionFacts(Builder builder) {

servingSize = builder.servingSize;

servings = builder.servings;

calories = builder.calories;

fat = builder.fat;

sodium = builder.sodium;

carbohydrate = builder.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() {

NutritionFacts nutritionFacts = new NutritionFacts(this)

validate(user);

return;

}

private void validateUserObject(User user) {

//Do some basic validations to check

// throw an IllegalArgumentException

}

}

}

The NutritionFacts class is immutable, and all parameter default values are in one place. The builder’s setter methods return the builder itself so that invocations can be chained, resulting in a *fluent API*. Here’s how the client code looks:

NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)

.calories(100).sodium(35).carbohydrate(27).build();

This client code is easy to write and, more importantly, easy to read. **The Builder pattern simulates named optional parameters** as found in Python andScala.

The Builder pattern has disadvantages as well. In order to create an object, you must first create its builder. While the cost of creating this builder is unlikely to be noticeable in practice, it could be a problem in performance-critical situations. Also, the Builder pattern is more verbose than the telescoping constructor pattern, so it should be used only if there are enough parameters to make it worthwhile, say four or more. But keep in mind that you may want to add more parameters in the future.

In summary, **the Builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters**, especially if many of the parameters are optional or of identicaltype. Client code is much easier to read and write with builders than withtelescoping constructors, and builders are much safer than JavaBeans.

Item 3: Enforce the singleton property with a private constructor or an enum type

A *singleton* is simply a class that is instantiated exactly once. Singletons typically represent a stateless object. **Making a class a** **singleton can make it difficult to test its clients** because it’s impossible to substitute a mock implementation for a singleton unless it implements an interface that serves as its type.

**// Singleton with public final field**

public class Elvis {

**private static final Elvis INSTANCE = new Elvis();**

private Elvis() {}

public getInstance(){

return INSTANCE;

}

}

To make a singleton class that uses either of these approaches *serializable*, it is not sufficient merely to add implements Serializable to its declaration. To maintain the singleton guarantee, declare all instance fields transient and provide a readResolve method. Otherwise, each time a serialized instance is deserialized, a new instance will be created.

private Object readResolve() {

return INSTANCE;

}

**a single-element enum type is often the best way to implement a singleton.**

public enum Elvis {

INSTANCE;

}

Item 4: Enforce non insatiability with a private constructor

class can be made non instantiable by including a private constructor. As a side effect, this idiom also prevents the class from being sub classed.

**Item 5: Prefer dependency injection to hardwiring resources**

**Item 6: Avoid creating unnecessary objects**

You can often avoid creating unnecessary objects by using *static factory methods* in preference to constructors on immutable classes that provide both. For example, the factory method Boolean.valueOf(String) is preferable to the constructor Boolean(String).

String s = new String("bikini"); **// DON'T DO THIS!**

The argument to the String constructor ("bikini") is itself a String instance.

The improved version is simply the following:

String s = "bikini";

In addition to reusing immutable objects, you can also reuse mutable objects if you know they won’t be modified.

**prefer primitives to boxed primitives, and watch out for unintentional autoboxing.**

Long sum = 0L; to long sum = 0L;

**Item 7: Eliminate obsolete object references**

Consider the following simple stack implementation:

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size];

}

The program has a “memory leak,” which can silently manifest itself as reduced performance due to increased garbage collector activity or increased memory footprint. In extreme cases, such memory leaks can cause disk paging and even program failure with an OutOfMemoryError, but such failures are relatively rare.

So where is the memory leak? If a stack grows and then shrinks, the objects that were popped off the stack will not be garbage collected, even if the program using the stack has no more references to them. This is because the stack maintains *obsolete references* to these objects. An obsolete reference is simply a reference that will never be dereferenced again.

The fix for this sort of problem is simple: null out references once they become obsolete.

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

**elements[size] = null; // Eliminate obsolete reference**

return result;

}

**Another common source of memory leaks is caches.** Once you put an object reference into a cache, it’s easy to forget that it’s there and leave it in the cache long after it becomes irrelevant. There are several solutions to this problem. If you’re lucky enough to implement a cache for which an entry is relevant exactly so long as there are references to its key outside of the cache, represent the cache as a WeakHashMap; entries will be removed automatically after they become obsolete.

**Item 8: Avoid finalizers and cleaners**

**Finalizers are unpredictable, often dangerous, and generally unnecessary.** The Java 9 replacement for finalizers is *cleaners*. **Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.**

One shortcoming of finalizers and cleaners is that there is no guarantee they will be executed promptly. It can take arbitrarily long between the time that an object becomes unreachable and the time its finalizer or cleaner runs. This means that you should **never do anything time-critical in a finalizer or** **cleaner.**

Don’t be seduced by the methods System.gc and System.runFinalization. They may increase the odds of finalizers or cleaners getting executed, but they don’t guarantee it. Two methods once claimed to make this guarantee: System.runFinalizersOnExit and its evil twin, Runtime.runFinalizersOnExit. These methods are fatally flawed and have been deprecated for decades.

Another problem with finalizers is that an uncaught exception thrown during finalization is ignored, and finalization of that object terminates [JLS, 12.6]. Uncaught exceptions can leave other objects in a corrupt state. If another thread attempts to use such a corrupted object, arbitrary nondeterministic behaviour may result. Normally, an uncaught exception will terminate the thread and print a stack trace, but not if it occurs in a finalizer—it won’t even print a warning. Cleaners do not have this problem because a library using a cleaner has control over its thread.

**Item 9: Prefer try-with-resources to try-finally**

try (BufferedReader br = new BufferedReader(

new FileReader(path))) {

return br.readLine();

}

}

**A**LTHOUGH Object is a concrete class, it is designed primarily for extension. All of its nonfinal methods (equals, hashCode, toString, clone, and finalize) have explicit *general contracts* because they are designed to be overridden.

**Chapter 3: Methods Common to All Objects**

**Item 10: Obey the general contract when overriding equals**

**Item 11: Always override hashCode when you override equals**

If two objects are equal according to the equals(Object) method, then calling hashCode on the two objects must produce the same integer result. If two objects are unequal according to the equals(Object) method, it is *not* required that calling hashCode on each of the objects must produce distinct results.

Compute an int hash code c for the field:

i. If the field is of a primitive type, compute *Type*.hashCode(f), where *Type* is the boxed primitive class corresponding to f’s type.

ii. If the field is an object reference and this class’s equals method compares the field by recursively invoking equals, recursively invoke hashCode on the field.

iii. Combine the hash code c computed into result.

@Override public int hashCode() {

int result = hashCode;

if (result == 0) {

result = Short.hashCode(areaCode);

result = 31 \* result + Short.hashCode(prefix);

result = 31 \* result + Short.hashCode(lineNum);

hashCode = result;

}

return result;

}

**Item 12: Always override toString**

**Item 13: Override clone judiciously**

So what *does* Cloneable do, given that it contains no methods? It determines the behavior of Object’s protected clone implementation: if a class implements Cloneable, Object’s clone method returns a field-by field copy of the object; otherwise it throws CloneNotSupportedException. **in practice, a class implementing Cloneable is expected to provide a properly functioning public clone method.**

**Chapter 4. Classes and Interfaces**

**Item 15: Minimize the accessibility of classes and members**

**Item 16: In public classes, use accessor methods, not public fields**

// Encapsulation of data by accessor methods and mutators

class Point {

private double x;

private double y;

public double getX() { return x; }

public double getY() { return y; }

public void setX(double x) { this.x = x; }

public void setY(double y) { this.y = y; }

}

**Item 17: Minimize mutability**

1. **Don’t provide methods that modify the object’s state**
2. **Ensure that the class can’t be extended.**
3. **Make all fields private.**

**The major disadvantage of immutable classes is that they require a separate object for each distinct value. Classes should be immutable unless there’s a very good reason to make them mutable. Declare every field private final unless there’s a good reason to do otherwise.**

**Item 34: Use enums instead of int constants**

public static final int APPLE\_FUJI = 0;

public static final int APPLE\_PIPPIN = 1;

public static final int APPLE\_GRANNY\_SMITH = 2;

public enum Fruit

{

    APPLE\_FUJI("<https://prod.domain.com:1088/>"),

    APPLE\_PIPPIN("<https://sit.domain.com:2019/>"),

    APPLE\_GRANNY\_SMITH("<https://cit.domain.com:8080/>");

    private int val;

    Environment(int val) {

        this.val = val;

    }

    public int getValue() {

        return val;

    }

}

public enum Environment

{

    PROD("<https://prod.domain.com:1088/>"),

    SIT("<https://sit.domain.com:2019/>"),

    CIT("<https://cit.domain.com:8080/>"),

    DEV("<https://dev.domain.com:21323/>");

    private String url;

    Environment(String envUrl) {

        this.url = envUrl;

    }

    public String getUrl() {

        return url;

    }

}

<https://howtodoinjava.com/java/enum/java-enum-string-example/>

<http://tutorials.jenkov.com/java/enums.html>

<https://www.baeldung.com/java-enum-map>

**Chapter 7. Lambdas and Streams**

**Item 47: Prefer Collection to Stream as a return type**