Creating & Destroying Objects

# Consider Static Factory Methods instead of Constructors

A class can **instead of, or in addition to**, public constructors, provide a public *static factory method*, which is simply a static method that returns an instance of the class . An example in the Boolean class:

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

### Static factory methods have names so you can convey information

One advantage of static factory methods is that, **unlike constructors**, they have names. If the parameters to a constructor do not, in and of themselves, describe the object being returned, a static factory **with a well-chosen name** is easier to use and the resulting client code easier to read.

For example, the constructor BigInteger(int, int, Random), which returns a BigInteger that is probably prime, would have been better expressed as a static factory method named BigInteger.probablePrime. (This method was added in Java 4.)

### They have names, so you can have same set of parameters, but different variations of returned objects

A class can have **only a single constructor with a given signature**.

You can work around this by providing two constructors whose parameter lists differ only in the order of their parameter types. This is a really bad idea.(without docs, users are in trouble).

**Because they have names, static factory methods don’t share this restriction**. cases where a class seems to require multiple constructors with the same signature, **replace the constructors with static factory methods and carefully chosen names to highlight their differences.**

### They are not required to return a “NEW” object each time they’re called

unlike constructors, they are not required to create a new object each time they’re invoked.

This allows

* immutable classes (Item 17) to use preconstructed instances.

Why is he emphasizing on Immutable classes?

*It probably refers to immutable value classes where you guarantee that no two equal objects exist at the same time( if a.equals(b) -> a==b), or Singlton 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**.(Did you learn what the flyweight pattern is?)

* **It can greatly improve performance if equivalent objects are requested often**, **especially if they are expensive to create**.
* The ability of static factory methods to return the same object from repeated invocations **allows classes to maintain strict control over what instances exist at any time.** Classes that do this are said to be instance-controlled.

There are several reasons to **write instance-controlled classes**.

* Instance control allows a class to guarantee that it is a singleton (Item 3) or non-instantiable (Item 4).
* Also, it allows an immutable value class (Item 17) to make the guarantee that no two equal instances exist: a.equals(b) if and only if a == b. **This is the basis of the Flyweight pattern [Gamma95]. Enum types (Item 34) provide this guarantee.**

**READ more about immutable value classes, Enum types, and the Flyweight pattern.**

### They can return a subtype of their return type

A third advantage of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type. **This gives you great flexibility in choosing the class of the returned object.**

One application of this flexibility is that an API can **return objects without making their classes public.** Hiding implementation classes in this fashion leads to a very compact API.

This technique lends itself to interface-based frameworks (Item 20), where interfaces provide natural return types for static factory methods.

Prior to Java 8, interfaces couldn’t have static methods. By convention, static factory methods for an interface named Type were put in a non-instantiable companion class (Item 4) named Types.

For example, **the Java Collections Framework has forty-five utility implementations of its interfaces, providing unmodifiable collections, synchronized collections, and the like**. Nearly all of these implementations are exported via static factory methods in one non-instantiable class (java.util.Collections). **The classes of the**

**returned objects are all non-public.**

* The Collections Framework API is much smaller than it would have been had it exported forty-five separate **public classes**, one for each convenience implementation.
* It is not just the *bulk* of the API that is reduced but the *conceptual weight:* the number and difficulty of the concepts that programmers must master in order to use the API. The programmer knows that the returned object has precisely the API specified by its interface, **so there is no need to read additional class documentation for the implementation class.**
* Furthermore, using such a static factory method requires the client to refer to the returned object by interface rather than implementation class, which is generally good practice (*I add: it means that you can’t use a reference that has the type of the implementation class and the type of the reference has to be the interface because the impl classes are not even public*)

As of Java 8, the restriction that interfaces cannot contain static methods was eliminated, so there is typically little reason to provide a non-instantiable companion class for an interface. Many public static members that would have been at home in such a class should instead be put in the interface itself. *(I add: It means that for example a static method in the java.utils.Collections class that provides an immutable Set can be moved to the Set interface itself)*

Note, however, that it may still be necessary to put the bulk of the implementation code behind these static methods in a separate **package-private class**.

**This is because Java 8 requires all static members of an interface to be public**. **Java 9 allows private static methods**, but static fields and static member classes are still required to be public.

### The class of the returned object can vary from call to call as a function of the input parameters, and even from release to release

Any subtype of the declared return type is permissible. The class of the returned object can **also** **vary from release to release**. The EnumSet class **has no public constructors**, only static factories.

In the OpenJDK implementation, they return an instance of one of two subclasses, depending on the size of the underlying enum type:

if it has sixtyfour or fewer elements, as most enum types do, the static factories return a RegularEnumSet instance, which is backed by a single long;

if the enum type has sixty-five or more elements, the factories return a JumboEnumSet

instance, backed by a long array.

The existence of these two implementation classes is invisible to clients. If RegularEnumSet ceased to offer performance advantages for small enum types, it could be eliminated from a future release with no ill effects. Similarly, a future release could add a third or fourth implementation of EnumSet if it proved beneficial for performance. Clients neither know nor care about the class of the object they get back from the factory; they care only that it is some subclass of EnumSet.

### the class of the returned object need not exist when the class containing the method is written.

Such flexible static factory methods form the basis of service provider *frameworks*, like the Java Database Connectivity API (JDBC).

A service provider framework is a system in which providers implement a service, and the system makes the implementations available to clients, decoupling the clients from the implementations.

There are three essential components in a service provider framework:

* a ***service interface***, which represents an implementation;
* **a *provider registration* *API***, which providers use to register implementations, *I add: a JDBC driver implementation for example, registers itself in a static initialization block.*
* and a ***service access API***, which clients use to obtain instances of the service.

The service access API may allow clients to specify criteria for choosing an implementation. In the absence of such criteria, the API returns an instance of a default implementation, or

allows the client to cycle through all available implementations. **The service access API is the flexible static factory that forms the basis of the service provider framework.**

* An optional **fourth** component of a service provider framework is a ***service provider interface*.**

it describes a factory **object** that produce instances of theservice interface. In the absence of a service provider interface, implementationsmust be instantiated reflectively (Item 65).

*I add: I think the service interface is the actual implementation that a service provider must provide, but with a service provider interface the provider implement it and the framework calls this method to get an instance of the implementation of the service interface. In JDBC the Driver interface plays this role which has a method* Connection connect(url) *which I assume establishes a connection and returns the implementation of the Connection that this Driver provides.*

*I think it’s basically and interface that b implementing it you’re saying that I’m providing that service that you want, call my method and get an instance and then you can give it to the client through the service access API*

In the case of JDBC:

* **Connection**plays the part of the service interface
* **DriverManager.registerDriver**is the provider registration API
* **DriverManager.getConnection** is theservice access API
* **Driver** is the service provider interface.

There are many variants of the service provider framework pattern. For example, the service access API can return a richer service interface to clients than the one furnished by providers. **This is the Bridge pattern** [Gamma95].

Dependency injection frameworks (Item 5) can be viewed as powerful service providers. **HOW?**

Since Java 6, the platform includes a general-purpose service provider framework, java.util.ServiceLoader, so you needn’t, and generally shouldn’t, write your own (Item 59). JDBC doesn’t use ServiceLoader, as the former predates the latter.

(Could we ever need this service loader class?)

## Limitations of public static factory methods

### Inheritance is not possible without protected/public constructors

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

For example, it is impossible to subclass any of the convenience implementation classes in the Collections Framework. **Arguably this can be a blessing in disguise because it encourages programmers to use composition instead of inheritance** (Item 18), **and is required for immutable types** (Item 17). **(And why exactly should an immutable type be final and not be subclassed? I guess we will find out when in item 17)**

### They are hard to find

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 methodthat **takes a single parameter and returns a corresponding instance of this type**, for example: Date d = Date.from(instant);

**• of**

An aggregation methodthat **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);

**•** **getType**

**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);

**• newType**

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.**