챕터2 객체의 생성과 소멸

C H A P T E R 2 Creating and Destroying Objects

이 챕터는 객체의 생성과 소멸에 관한 것이다. 객체를 생성하는 시기와 방법, 생성을 피하는 시기와 방법, 적시에 소멸시키는 방법, 파괴전에 필요한 정리작업을 관리하는 방법이다.

precede 미국∙영국 [prɪˈsiːd] 듣기 영국식 듣기 어휘등급

1. 동사 …에 앞서다[선행하다]

2. 동사 … 앞에 가다

THIS chapter concerns creating and destroying objects: when and how to create

them, when and how to avoid creating them, how to ensure they are destroyed in a

timely manner, and how to manage any cleanup actions that must precede their

destruction.

Item 1: Consider static factory methods instead of constructors

항목 1 : 생성자 대신 정적 팩토리 메서드 고려

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique that should be a part of every programmer’s toolkit. A class can provide a public static factory method, which is simply a static method that returns an instance of the class. Here’s a simple example from Boolean (the boxed primitive class for boolean). This method translates a boolean primitive value into a Boolean object reference:

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 [Gamma95]. The static factory method described in this

item has no direct equivalent in Design Patterns.

A class can provide its clients with static factory methods instead of, or in

addition to, public constructors. Providing a static factory method instead of a

public constructor has both advantages and disadvantages.

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, theconstructor 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.)

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.

Because they have names, static factory methods don’t share the restriction

discussed in the previous paragraph. In 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.

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. This

allows immutable classes (Item 17) 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 [Gamma95]. 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 noninstantiable (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.

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 noninstantiable 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 noninstantiable class

(java.util.Collections). The classes of the returned objects are all nonpublic.

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 (Item 64).

As of Java 8, the restriction that interfaces cannot contain static methods was

eliminated, so there is typically little reason to provide a noninstantiable 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. 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.

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. 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 (Item 36) 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 sixty-four 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 beneficialfor 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.

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. 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; 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, which describes a factory object that produce instances of the

service interface. In the absence of a service provider interface, implementations

must be instantiated reflectively (Item 65). In the case of JDBC, Connection

plays the part of the service interface, DriverManager.registerDriver is the

provider registration API, DriverManager.getConnection is the service access

API, and 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. 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.

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).

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 wayfor 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.

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. 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; 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, which describes a factory object that produce instances of the

service interface. In the absence of a service provider interface, implementations

must be instantiated reflectively (Item 65). In the case of JDBC, Connection

plays the part of the service interface, DriverManager.registerDriver is the

provider registration API, DriverManager.getConnection is the service access

API, and 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. 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.

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).

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