1. **Static Factory Method.**

Factory Method: "Define an interface for creating an object, but let the classes which implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses" (c) GoF.

"Static factory method is simply a static method that returns an instance of a class." (c) Effective Java, Joshua Bloch. Usually this method is inside a particular class.

The difference:

The key idea of static factory method is to gain control over object creation and delegate it from constructor to static method. The decision of object to be created is like in Abstract Factory made outside the method (in common case, but not always). While the key (!) idea of Factory Method is to delegate decision of what instance of class to create inside Factory Method. E.g. classic Singleton implementation is a special case of static factory method. Example of commonly used static factory methods:

valueOf , getInstance , newInstance

Example : A screenshot of a computer code

Description automatically generated with medium confidence

Advantages:

* **One advantage of static factory methods is that, unlike constructors, they have names.**
* **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.**
* **A third advantage of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type**
  + One application of this flexibility is that an API can return objects without making their classes 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.
* 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:

* The main limitation of providing only static factory methods is that classes without public or protected constructors cannot be subclassed.
* A second shortcoming of static factory methods is that they are hard for programmers to find.

By convention, static factory methods for an interface named Type were put in a noninstantiable companion class ([Item 4](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch2.xhtml#lev4)) 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.

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.

1. **Consider a builder when faced with many constructor parameters.**

Traditionally, programmers have used the *telescoping constructor* pattern, in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters. Example : A screenshot of a computer program

Description automatically generated with low confidence

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

It is a form of the *Builder* pattern [[Gamma95](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ref.xhtml#rGamma95)]. 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. The builder is typically a static member class ([Item 24](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch4.xhtml#lev24)) of the class it builds. Here’s how it looks in practice:

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with low confidence

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 identical type. Client code is much easier to read and write with builders than with telescoping constructors, and builders are much safer than JavaBeans.

1. **Enforce the singleton property with a private constructor or and enum type.**

A *singleton* is simply a class that is instantiated exactly once [[Gamma95](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ref.xhtml#rGamma95)]. Singletons typically represent either a stateless object such as a function ([Item 24](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch4.xhtml#lev24)) or a system component that is intrinsically unique.

There are two common ways to implement singletons. Both are based on keeping the constructor private and exporting a public static member to provide access to the sole instance. In one approach, the member is a final field:

A picture containing text, screenshot, font, algebra

Description automatically generated

In the second approach to implementing singletons, the public member is a static factory method:

A picture containing text, screenshot, font, algebra

Description automatically generated

To make a singleton class that uses either of these approaches *serializable* ([Chapter 12](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch12.xhtml#ch12)), 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 ([Item 89](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch12.xhtml#lev89)). Otherwise, each time a serialized instance is deserialized, a new instance will be created, leading, in the case of our example, to spurious Elvis sightings. To prevent this from happening, add this readResolve method to the Elvis class:

A picture containing text, screenshot, font, algebra

Description automatically generated

1. **Enforce non-instantiability with a private constructor.**

Occasionally you’ll want to write a class that is just a grouping of static methods and static fields. Such *utility classes* were not designed to be instantiated: an instance would be nonsensical. In the absence of explicit constructors, however, the compiler provides a public, parameterless *default constructor.* A default constructor is generated only if a class contains no explicit constructors, so a class can be made noninstantiable by including a private constructor:

As a side effect, this idiom also prevents the class from being subclassed. All constructors must invoke a superclass constructor, explicitly or implicitly, and a subclass would have no accessible superclass constructor to invoke.A screen shot of a computer program

Description automatically generated with low confidence

1. **Prefer Dependency injection to hardwiring resources.**

What is required is the ability to support multiple instances of the class (in our example, SpellChecker), each of which uses the resource desired by the client (in our example, the dictionary). A simple pattern that satisfies this requirement is to **pass the resource into the constructor when creating a new instance**. This is one form of *dependency injection*: the dictionary is a *dependency* of the spell checker and is *injected* into the spell checker when it is created. The dependency injection pattern is so simple that many programmers use it for years without knowing it has a name. It preserves immutability ([Item 17](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch4.xhtml#lev17)), so multiple clients can share dependent objects.

A screenshot of a computer code

Description automatically generated with low confidence

In summary, do not use a singleton or static utility class to implement a class that depends on one or more underlying resources whose behavior affects that of the class, and do not have the class create these resources directly. Instead, pass the resources, or factories to create them, into the constructor (or static factory or builder). This practice, known as dependency injection, will greatly enhance the flexibility, reusability, and testability of a class.

1. **Avoid creating unnecessary objects.**

It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed. Reuse can be both faster and more stylish. An object can always be reused if it is immutable

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

The statement creates a new String instance each time it is executed, and none of those object creations is necessary. The argument to the String constructor ("String") is itself a String instance, functionally identical to all of the objects created by the constructor. If this usage occurs in a loop or in a frequently invoked method, millions of String instances can be created needlessly. You can often avoid creating unnecessary objects by using *static factory methods*

**// Hideously slow! Can you spot the object creation?**

private static long sum() {

     Long sum = 0L;

     for (long i = 0; i <= Integer.MAX\_VALUE; i++)

         sum += i;

    return sum;

}

This program gets the right answer, but it is *much* slower than it should be, due to a one-character typographical error. The variable sum is declared as a Long instead of a long, which means that the program constructs about 231 unnecessary Long instances (roughly one for each time the long i is added to the Long sum). Changing the declaration of sum from Long to long reduces the runtime from 6.3 seconds to 0.59 seconds on my machine. The lesson is clear: **prefer primitives to boxed primitives, and watch out for unintentional autoboxing.**

1. **Eliminate obsolete object references.**

If you switched from a language with manual memory management, such as C or C++, to a garbage-collected language such as Java, your job as a programmer was made much easier by the fact that your objects are automatically reclaimed when you’re through with them. It seems almost like magic when you first experience it. It can easily lead to the impression that you don’t have to think about memory management, but this isn’t quite true. Memory leaks in garbage-collected languages (more properly known as *unintentional object retentions*) are insidious. If an object reference is unintentionally retained, not only is that object excluded from garbage collection, but so too are any objects referenced by that object, and so on. Even if only a few object references are unintentionally retained, many, many objects may be prevented from being garbage collected, with potentially large effects on performance.

**Nulling out object references should be the exception rather than the norm.** The best way to eliminate an obsolete reference is to let the variable that contained the reference fall out of scope. This occurs naturally if you define each variable in the narrowest possible scope ([Item 57](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch9.xhtml#lev57)).

**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. Remember that WeakHashMap is useful only if the desired lifetime of cache entries is determined by external references to the key, not the value.

**A third common source of memory leaks is listeners and other callbacks.** If you implement an API where clients register callbacks but don’t deregister them explicitly, they will accumulate unless you take some action. One way to ensure that callbacks are garbage collected promptly is to store only *weak references* to them, for instance, by storing them only as keys in a WeakHashMap.

1. **Avoid finalizers and cleaners.**

**Finalizers are unpredictable, often dangerous, and generally unnecessary.** Their use can cause erratic behavior, poor performance, and portability problems. Finalizers have a few valid uses, which we’ll cover later in this item, but as a rule, you should avoid them. As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries. 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’ll be executed promptly [JLS, 12.6]. 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.**

The promptness with which finalizers and cleaners are executed is primarily a function of the garbage collection algorithm, which varies widely across implementations.

They have perhaps two legitimate uses. One is to act as a safety net in case the owner of a resource neglects to call its close method. While there’s no guarantee that the cleaner or finalizer will run promptly (or at all), it is better to free the resource late than never if the client fails to do so.

A second legitimate use of finalizers and cleaners concerns objects with *native peers*. A native peer is a native (non-Java) object to which a normal object delegates via native methods.

1. **Prefer try-with-resource instead of try-finally.**

Historically, a try-finally statement was the best way to guarantee that a resource would be closed properly, even in the face of an exception or return:

Even the correct code for closing resources with try-finally statements, as illustrated in the previous two code examples, has a subtle deficiency. The code in both the try block and the finally block is capable of throwing exceptions. For example, in the firstLineOfFile method, the call to readLine could throw an exception due to a failure in the underlying physical device, and the call to close could then fail for the same reason. Under these circumstances, the second exception completely obliterates the first one. There is no record of the first exception in the exception stack trace, which can greatly complicate debugging in real systems—usually it’s the first exception that you want to see in order to diagnose the problem.

All of these problems were solved in one fell swoop when Java 7 introduced the try-with-resources statement [JLS, 14.20.3]. To be usable with this construct, a resource must implement the AutoCloseable interface, which consists of a single void-returning close method.

A screenshot of a computer code

Description automatically generated with medium confidence

A picture containing text, screenshot, font

Description automatically generated

Not only are the try-with-resources versions shorter and more readable than the originals, but they provide far better diagnostics. Consider the firstLineOfFile method. If exceptions are thrown by both the readLine call and the (invisible) close, the latter exception is *suppressed* in favor of the former. In fact, multiple exceptions may be suppressed in order to preserve the exception that you actually want to see. These suppressed exceptions are not merely discarded; they are printed in the stack trace with a notation saying that they were suppressed. You can also access them programmatically with the getSuppressed method, which was added to Throwable in Java 7.