Java theory and practice: Is that your final answer?

Guidelines for the effective use of the final keyword

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

01 October 2002

The `final` keyword is often misused -- it is overused when declaring classes and methods, and underused when declaring instance fields. This month, Java practitioner Brian Goetz explores some guidelines for the effective use of `final`.

View more content in this series

Like its cousin, the `const` keyword in C, `final` means several different things depending on the context. The `final` keyword can be applied to classes, methods, or fields. When applied to a class, it means that the class cannot be subclassed. When applied to a method, it means that the method cannot be overridden by a subclass. When applied to a field, it means that the field's value must be assigned *exactly* once in each constructor and can never change after that.

Most Java texts properly describe the usage and consequences of using the `final` keyword, but offer little in the way of guidance as to when, and how often, to use `final`. In my experience, `final` is vastly overused for classes and methods (generally because developers mistakenly believe it will enhance performance), and underused where it will do the most good -- in declaring class instance variables.

Why is this class final?

It is very common, especially in open source projects, for a developer to declare a class as `final`, but not give any indication as to why this decision was made. Some time later, especially if the original developer is no longer involved in the maintenance of the code, other developers will invariably start asking "Why was class X declared `final`?" Often, nobody knows, and when someone does know, or is willing to guess, the answer is almost always "because it makes it faster." The common perception is that declaring classes or methods `final` makes it easier for the compiler to inline method calls, but this perception is incorrect (or at the very least, greatly overstated).

`final` classes and methods can be a significant inconvenience when programming -- they limit your options for reusing existing code and extending the functionality of existing classes. While

sometimes a class is made `final` for a good reason, such as to enforce immutability, the benefits of using `final` should outweigh the inconvenience. Performance enhancement is almost always a bad reason to compromise good object-oriented design principles, and when the performance enhancement is small or nonexistent, this is a bad trade-off indeed.

Premature optimization

Declaring methods or classes as `final` in the early stages of a project for performance reasons is a bad idea for several reasons. First, early stage design is the wrong time to think about cycle-counting performance optimizations, especially when such decisions can constrain your design the way using `final` can. Second, the performance benefit gained by declaring a method or class as `final` is usually zero. And declaring complicated, stateful classes as `final` discourages object-oriented design and leads to bloated, kitchen-sink classes because they cannot be easily refactored into smaller, more coherent classes.

Like many myths about Java performance, the erroneous belief that declaring classes or methods as `final` results in better performance is widely held but rarely examined. The argument goes that declaring a method or class as `final` means that the compiler can inline method calls more aggressively, because it knows that at run time this is definitely the version of the method that's going to be called. But this is simply not true. Just because class X is compiled against `final` class Y doesn't mean that the same version of class Y will be loaded at run time. So the compiler cannot inline such cross-class method calls safely, `final` or not. Only if a method is `private` can the compiler inline it freely, and in that case, the `final` keyword would be redundant.

On the other hand, the run-time environment and JIT compiler have more information about what classes are actually loaded, and can make much better optimization decisions than the compiler can. If the run-time environment knows that no classes are loaded that extend Y, then it can safely inline calls to methods of Y, regardless of whether Y is `final` (as long as it can invalidate such JIT-compiled code if a subclass of Y is later loaded). So the reality is that while `final` might be a useful hint to a dumb run-time optimizer that doesn't perform any global dependency analysis, its use doesn't actually enable very many compile-time optimizations, and is not needed by a smart JIT to perform run-time optimizations.

Deja vu -- The register keyword all over again

The use of `final` for optimization decisions is very similar to the deprecated `register` keyword in C. The `register` keyword was motivated by the desire to let the programmer help the optimizer, but in reality this turned out not to be very helpful. As much as we'd like to believe otherwise, compilers are generally better than humans at making code optimization decisions, especially on modern RISC processors. In fact, most C compilers ignore the `register` keyword entirely. Early C compilers ignored it because they did no optimization at all; present-day compilers ignore it because they can make better optimization decisions without it. In either case, the `register` keyword adds little performance benefit, much like the `final` keyword when applied to Java classes or methods. If you want to optimize your code, stick to optimizations that will make a big difference, like using efficient algorithms and not performing redundant calculations -- and leave the cycle-counting optimizations to the compiler and JVM.

Use final to preserve immutability

While performance is not a good reason to declare a class or method as `final`, there are still good reasons to sometimes write `final` classes. The most common is that `final` guarantees that classes intended to be immutable *stay* immutable. Immutable classes are very useful for simplifying the design of object-oriented programs -- immutable objects require less defensive coding and offer relaxed synchronization requirements. You wouldn't want to build the assumption into your code that a class is immutable and then have someone extend it in a way that makes it mutable. Declaring immutable classes as `final` guarantees that errors of this type don't creep into your program.

Another reason to use `final` for classes or methods is to prevent linkages between methods from being broken. For example, suppose that the implementation of some method of class X assumes that method M will behave in a certain way. Declaring X or M as `final` will prevent derived classes from redefining M in such a way as to cause X to behave incorrectly. While it might be better to implement X without these internal dependencies, it's not always practical, and using `final` prevents such incompatible modifications in the future.

If you must use final classes or methods, document why

In any event, when you do choose to declare a method or class `final`, document the reasons why. Otherwise, future maintainers will likely be confused about whether there was a good reason (since there often isn't) and will be constrained by your decision without the benefit of your motivation. In many cases, it makes sense to defer the decision to declare a class or method as `final` until later in the development process, when you have better information about how your classes interact and might be extended. You may find you don't need to make the class `final` at all, or you might be able to refactor your classes so as to apply `final` to a smaller, simpler class.

Final fields

`final` fields are so different from `final` classes or methods that it's almost unfair to make them share the same keyword. A `final` field is a read-only field, whose value is guaranteed to be set exactly once at construction time (or at class initialization time for `static final` fields.) As discussed earlier, with `final` classes and methods you should always ask yourself if you really *need* to use `final`. With `final` fields, you should ask yourself the opposite question -- does this field really *need* to be mutable? You might be surprised at how often the answer is no.

Documentation value

`final` fields have several benefits. Declaring fields as `final` has valuable documentation benefits for developers who want to use or extend your class -- not only does it help explain how the class works, but it enlists the compiler's help in enforcing your design decisions. Unlike with `final` methods, declaring a `final` field helps the optimizer make better optimization decisions, because if the compiler knows the field's value will not change, it can safely cache the value in a register. `final` fields also provide an extra level of safety by having the compiler enforce that a field is read-only.

In the extreme case, a class whose fields are all `final` primitives or `final` references to immutable objects, the class becomes immutable itself -- a very convenient situation indeed. Even if the class is not wholly immutable, making certain portions of its state immutable can greatly simplify development -- you don't have to synchronize to guarantee that you are seeing the current value of a `final` field or to ensure that no one else is changing that portion of the object's state.

So why are `final` fields so underused? One reason is because they can be a bit cumbersome to use correctly, especially for object references whose constructors can throw exceptions. Because a `final` field must be initialized exactly once in every constructor, if the construction of a `final` object reference may throw an exception, the compiler may complain that the field might not be initialized. The compiler is generally smart enough to realize that initialization in each of two exclusive code branches, such as in an `if...else` block, constitutes exactly one initialization, but is often less forgiving with `try...catch` blocks. For example, most Java compilers won't accept the code in Listing 1:

```
public class Foo {
  private final Thingie thingie;

  public Foo() {
    try {
      thingie = new Thingie();
    }
    catch (ThingieConstructionException e) {
      thingie = Thingie.getDefaultThingie();
    }
  }
}
```

But they would accept the code in Listing 2, which is equivalent:

```
public class Foo {
  private final Thingie thingie;

  public Foo() {
    Thingie tempThingie;
    try {
      tempThingie = new Thingie();
    }
    catch (ThingieConstructionException e) {
      tempThingie = Thingie.getDefaultThingie();
    }
    thingie = tempThingie;
  }
}
```

Limitations of final fields

`final` fields still have some serious limitations. While an array reference can be declared as `final`, the elements of the array cannot. This means that classes that expose `public final` array fields or return references to those fields through their methods such as the `DangerousStates` class shown in Listing 3, are not immutable. Similarly, while an object reference may be declared as a `final` field, the object to which it refers may still be mutable. If you wish to create immutable objects using `final` fields, you must prevent references to arrays or mutable objects from escaping

from your class. One easy way to do this without cloning the array repeatedly is to turn arrays into `List`s, such as in the `SafeStates` class shown in Listing 3.

```
// Not immutable -- the states array could be modified by a malicious
caller
public class DangerousStates {
  private final String[] states = new String[] { "Alabama", "Alaska", ... };

  public String[] getStates() {
    return states;
  }
}


// Immutable -- returns an unmodifiable List instead
public class SafeStates {
  private final String[] states = new String[] { "Alabama", "Alaska", ... };
  private final List statesAsList
    = new AbstractList() {
        public Object get(int n) {
          return states[n];
        }

        public int size() {
          return states.length;
        }
      };

  public List getStates() {
    return statesAsList;
  }
}
```

Why was `final` not extended to apply to arrays and referenced objects, similar to the use of `const` in C and C++? The semantics and use of `const` in C++ are quite confusing, meaning different things depending on where in the expression it appears. The Java architects were trying to save us from this confusion, but unfortunately they created some new confusion in the process.

Some final words

There are a few basic guidelines you can follow to use `final` effectively with classes, methods, and fields. In particular, don't try to use `final` as a performance management tool; there are much better, less constraining ways to enhance your program's performance. Use `final` where it reflects the fundamental semantics of your program: to indicate that classes are intended to be immutable or that fields are intended to be read-only. If you choose to create `final` classes or methods, make sure you clearly document why you did -- your colleagues will thank you.

Resources

- Josh Bloch's *Effective Java Programming Language Guide* contains a wealth of useful hints and guidelines for writing better code.
- Jack Shirazi's Java Performance Tuning Web site has a wealth of performance tuning tips, including tips related to the use of `final`. *Caveat emptor*: Many of the older tips give incorrect advice about the performance implications of `final`.
- Renaud Waldura offers some situations in which `final` is useful in "The Final Word on the final keyword."
- The EarthWeb code guru offers some meditations on `final`.
- The semantics of `final` fields will change somewhat (to provide stronger initialization safety guarantees) when JSR 133 is implemented.
- Find hundreds of Java technology-related resources at the *developerWorks*Java technology zone.

About the author

Brian Goetz

Brian Goetz is a software consultant and has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California. See Brian's published and upcoming articles in popular industry publications.