The “final” Keyword in Java

Last updated: January 8, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Reviewed by: Kevin Gilmore (https://www.baeldung.com/editor/kevin-author)

Core Java (https://www.baeldung.com/category/java/core-java)

Definition (https://www.baeldung.com/tag/definition)

Java Keyword (https://www.baeldung.com/tag/java-keyword)

1. Overview

While inheritance enables us to reuse existing code, sometimes we do need to **set limitations on extensibility** for various reasons; the *final* keyword allows us to do exactly that.

In this tutorial, we’ll take a look at what the *final* keyword means for classes, methods, and variables.

2. *Final* Classes (/)

Classes marked as *final* can't be extended. If we look at the code of Java core libraries, we'll find many *final* classes there. One example is the *String* class.

Consider the situation if we can extend the *String* class, override any of its methods, and substitute all the *String* instances with the instances of our specific *String* subclass.

The result of the operations over *String* objects will then become unpredictable. And given that the *String* class is used everywhere, it's unacceptable. That's why the *String* class is marked as *final*.

Any attempt to inherit from a *final* class will cause a compiler error. To demonstrate this, let's create the *final* class *Cat*:

```
public final class Cat {

    private int weight;

    // standard getter and setter
}
```

And let's try to extend it:

```
public class BlackCat extends Cat {
}
```

We'll see the compiler error:

```
The type BlackCat cannot subclass the final class Cat
```

Note that **the *final* keyword in a class declaration doesn't mean that the objects of this class are immutable**. We can change the fields of *Cat* object freely:

```
Cat cat = new Cat();
cat.setWeight(1);

assertEquals(1, cat.getWeight());
```

We just can't extend it.

(/)

If we follow the rules of good design strictly, we should create and document a class carefully or declare it *final* for safety reasons. However, we should use caution when creating *final* classes.

Notice that making a class *final* means that no other programmer can improve it. Imagine that we're using a class and don't have the source code for it, and there's a problem with one method.

If the class is *final,* we can't extend it to override the method and fix the problem. In other words, we lose extensibility, one of the benefits of object-oriented programming.

3. *Final* Methods

Methods marked as *final* cannot be overridden. When we design a class and feel that a method shouldn't be overridden, we can make this method *final*. We can also find many *final* methods in Java core libraries.

Sometimes we don't need to prohibit a class extension entirely, but only prevent overriding of some methods. A good example of this is the *Thread* class. It's legal to extend it and thus create a custom thread class. But its *isAlive()* methods is *final*.

This method checks if a thread is alive. It's impossible to override the *isAlive()* method correctly for many reasons. One of them is that this method is native. Native code is implemented in another programming language and is often specific to the operating system and hardware it's running on.

Let's create a *Dog* class and make its *sound()* method *final*:

```
public class Dog {
    public final void sound() {
        // ...
    }
}
```

Now let's extend the *Dog* class and try to override its *sound()* method:

```
public class BlackDog extends Dog {
    public void sound() {
    }
}
```

We'll see the compiler error:

```
- overrides
com.baeldung.finalkeyword.Dog.sound
- Cannot override the final method from Dog
sound() method is final and can't be overridden
```

If some methods of our class are called by other methods, we should consider making the called methods *final*. Otherwise, overriding them can affect the work of callers and cause surprising results.

If our constructor calls other methods, we should generally declare these methods *final* for the above reason.

What's the difference between making all methods of the class *final* and marking the class itself *final*? In the first case, we can extend the class and add new methods to it.

In the second case, we can't do this.

4. *Final* Variables

Variables marked as *final* can't be reassigned. Once a *final* variable is initialized, it can't be altered.

4.1. *Final* Primitive Variables

Let's declare a primitive *final* variable *i*, then assign 1 to it.

And let's try to assign a value of 2 to it:

```
public void whenFinalVariableAssign_thenOnlyOnce() {
    final int i = 1;
    //...
    i=2;
}
```

The compiler says:

```
The final local variable i may already have been assigned
```

4.2. *Final* Reference Variables

If we have a *final* reference variable, we can't reassign it either. But **this doesn't mean that the object it refers to is immutable**. We can change the properties of this object freely.

To demonstrate this, let's declare the *final* reference variable *cat* and initialize it:

```
final Cat cat = new Cat();
```

If we try to reassign it we'll see a compiler error:

```
The final local variable cat cannot be assigned. It must be blank and not
using a compound assignment
```

But we can change the properties of *Cat* instance:

```
cat.setWeight(5);

assertEquals(5, cat.getWeight());
```

4.3. *Final* Fields

***Final* fields can be either constants or write-once fields.** To distinguish them, we should ask a question – would we include this field if we were to serialize the object? If no, then it's not part of the object, but a constant.

Note that according to naming conventions, class constants should be uppercase, with components separated by underscore ("_") characters:

```
static final int MAX_WIDTH = 999;
```

Note that **any *final* field must be initialized before the constructor completes**.

For *static final* fields, this means that we can initialize them:

- upon declaration as shown in the above example
- in the static initializer block

For instance *final* fields, this means that we can initialize them:

- upon declaration
- in the instance initializer block
- in the constructor

Otherwise, the compiler will give us an error.

4.4. *Final* Parameters

The *final* keyword is also legal to put before method parameters. **A *final* parameter can't be changed inside a method**:

```
public void methodWithFinalArguments(final int x) {
    x=1;
}
```

The above assignment causes the compiler error:

```
The final local variable x cannot be assigned. It must be blank and not
using a compound assignment
```

5. Conclusion

(/)

In this article, we learned what the *final* keyword means for classes, methods, and variables. Although we may not use the *final* keyword often in our internal code, it may be a good design solution.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.