final (Java)

From Wikipedia, the free encyclopedia

In the Java programming language, the **`final`** keyword is used in several different contexts to define an entity that can only be assigned once.

Once a **`final`** variable has been assigned, it always contains the same value. If a **`final`** variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object (this property of **`final`** is called *non-transitivity*[1]). This applies also to arrays, because arrays are objects; if a **`final`** variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.[2]

Contents

Final classes

A **final class** cannot be subclassed. Doing this can confer security and efficiency benefits, so many of the Java standard library classes are final, such as `java.lang.System (https://docs.oracle.com/javase/8/docs/api/java/lang/System.html)` and `java.lang.String (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)`.

Example:

```
public final class MyFinalClass {...}

public class ThisIsWrong extends MyFinalClass {...} // forbidden
```

Restricted subclasses are often referred to as "soft final" classes.[3]

Final methods

A final method cannot be overridden or hidden by subclasses.[4] This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class.[5]

Example:

```
public class Base
{
    public       void m1() {...}
    public final void m2() {...}

    public static       void m3() {...}
    public static final void m4() {...}
}

public class Derived extends Base
{
    public void m1() {...}  // OK, overriding Base#m1()
    public void m2() {...}  // forbidden

    public static void m3() {...}  // OK, hiding Base#m3()
    public static void m4() {...}  // forbidden
}
```

A common misconception is that declaring a method as `final` improves efficiency by allowing the compiler to directly insert the method wherever it is called (see inline expansion). Because the method is loaded at runtime, compilers are unable to do this. Only the runtime environment and JIT compiler know exactly which classes have been loaded, and so only they are able to make decisions about when to inline, whether or not the method is final.[6]

Machine code compilers which generate directly executable, platform-specific machine code, are an exception. When using static linking, the compiler can safely assume that methods and variables computable at compile-time may be inlined.

Final variables

A **final variable** can only be initialized once, either via an initializer or an assignment statement. It does not need to be initialized at the point of declaration: this is called a "blank final" variable. A blank final instance variable of a class must be definitely assigned in every constructor of the class in which it is declared; similarly, a blank final static variable must be definitely assigned in a static initializer of the class in which it is declared; otherwise, a compile-time error occurs in both cases.[7] (Note: If the variable is a reference, this means that the variable cannot be re-bound to reference another object. But the object that it references is still mutable, if it was originally mutable.)

Unlike the value of a constant, the value of a final variable is not necessarily known at compile time. It is considered good practice to represent final constants in all uppercase, using underscore to separate words.[8]

Example:

```
public class Sphere {

    // pi is a universal constant, about as constant as anything can be.
    public static final double PI = 3.141592653589793;

    public final double radius;
    public final double xPos;
    public final double yPos;
    public final double zPos;

    Sphere(double x, double y, double z, double r) {
         radius = r;
         xPos = x;
         yPos = y;
         zPos = z;
    }

    [...]
}
```

Any attempt to reassign `radius`, `xPos`, `yPos`, or `zPos` will result in a compile error. In fact, even if the constructor doesn't set a final variable, attempting to set it outside the constructor will result in a compilation error.

To illustrate that finality doesn't guarantee immutability: suppose we replace the three position variables with a single one:

```
    public final Position pos;
```

where `pos` is an object with three properties `pos.x`, `pos.y` and `pos.z`. Then `pos` cannot be assigned to, but the three properties can, unless they are final themselves.

Like full immutability, the use of final variables has great advantages, especially in optimization. For instance, `Sphere` will probably have a function returning its volume; knowing that its radius is constant allows us to memoize the computed volume. If we have relatively few `Spheres` and we need their volumes very often, the performance gain might be substantial. Making the radius of a `Sphere` `final` informs developers and compilers that this sort of optimization is possible in all code that uses `Spheres`.

Though it appears to violate the `final` principle, the following is a legal statement:

```
for (final SomeObject obj : someList) {
   // do something with obj
}
```

Since the obj variable goes out of scope with each iteration of the loop, it is actually redeclared each iteration, allowing the same token (i.e. `obj`) to be used to represent multiple variables.[9]