Difference between final and effectively final

Ask Question

I'm playing with lambdas in Java 8 and I came across warning `local variables referenced from a lambda expression must be final or effectively final` . I know that when I use variables inside anonymous class they must be final in outer class, but still - what is the difference between *final* and *effectively final*?

java lambda inner-classes final java-8

asked Jan 5 '14 at 19:30

alex
3,978 12 48 76

2 Lots of answers, yet all essentially amount to "no difference." But is that really true? Unfortunately, I can't seem to find a Language Specification for Java 8. – Aleksandr Dubinsky Jan 8 '14 at 23:56

3 @AleksandrDubinsky docs.oracle.com/javase/specs – eis Jun 10 '15 at 8:07

@AleksandrDubinsky not "really" true. I found one exception to this rule. A local variable initialized with a constant is not a constant expression to the compiler. You cannot use such a variable for a case in a switch/case until you explicitly add the final keyword. E.g. "int k = 1; switch(someInt) { case k: ...". – Henno Vermeulen Oct 16 '15 at 15:19

11 Answers

> ... starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively final. **A variable or parameter whose value is never changed after it is initialized is effectively final.**

For example, suppose that the variable numberLength is not declared final, and you add the marked assignment statement in the PhoneNumber constructor:

```
PhoneNumber(String phoneNumber) {
    numberLength = 7;   // <== assignment to numberLength
    String currentNumber = phoneNumber.replaceAll(
        regularExpression, "");
    if (currentNumber.length() == numberLength)
        formattedPhoneNumber = currentNumber;
    else
        formattedPhoneNumber = null;
}
```

Because of this assignment statement, the variable numberLength is not effectively final anymore. **As a result, the Java compiler generates an error message similar to "local variables referenced from an inner class must be final or effectively final"** where the inner class PhoneNumber tries to access the numberLength variable:

http://codeinventions.blogspot.in/2014/07/difference-between-final-and.html

http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

edited Nov 9 '15 at 12:49

ryvantage
6,689 9 42 76

answered Jan 5 '14 at 19:32

SURESH ATTA
101k 23 126 219

52 +1 Note: if a reference is not changed it is effectively final even if the object referenced is changed. – Peter Lawrey Jan 5 '14 at 20:20

@SURESH ATTA Why cannot a local class access a enclosing block's effectively final variable? Is there any way to change the variable (which is a reference or a primitive variable) in local class, if so it is no longer effectively final variable. – stanleyerror Feb 6 '15 at 5:36

1 @stanleyerror This might be of some help: stackoverflow.com/questions/4732544/… – Amit Oct 6 '15 at 9:06

1 I think more useful than an example of *not* effectively final, is an example of when something *is* effectively final. Though the description does make it clear. Var need not be declared final if no code changes it's value. – Skychan Oct 16 '15 at 15:31

+1 for great answer. My favorite work-around is to use the java.util.concurrent.atomic.AtomicXxxx classes when I get this error: github.com/GlenKPeterson/UncleJim/wiki/… – GlenPeterson Nov 23 '15 at 13:31

I find the simplest way to explain "effectively final" is to imagine adding the `final` modifier to a variable declaration. If, with this change, the program continues to behave in the same way, both at compile time and at run time, then that variable is effectively final.

[Edit: Henno Vermeulen, in a comment below, points out a minor exception to this rule. The rule is still very simple and useful, but syntax rules mean it isn't universally applicable.]

edited Oct 18 '15 at 5:22

answered Jan 6 '14 at 0:07

Maurice Naftalin
7,805 2 15 15

4 This is true, as long as the understanding of java 8's "final" is well understood. Otherwise I'd look at a variable not declared final that you made an assignment to later, and erroneously think it wasn't final. You might say "of course"... but not everybody pays as much attention to the latest language version changes as they ought. – fool4jesus Jun 8 '15 at 14:09

7 One exception to this rule is that a local variable initialized with a constant is not a constant expression to the compiler. You cannot use such a variable for a case in a switch/case until you explicitly add the final keyword. E.g. "int k = 1; switch(someInt) { case k: ...". – Henno Vermeulen Oct 16 '15 at 15:17

1 @HennoVermeulen switch-case is not an exception to the rule in this answer. The language specifies that `case k` requires a *constant expression* which could be a *constant variable* ("A constant variable is a final variable of primitive type or type String that is initialized with a constant expression" JLS 4.12.4) which is a special case of a final variable. – Colin D Bennett Apr 3 at 19:00

According to the docs:

> A variable or parameter whose value is never changed after it is initialized is effectively final.

Basically, if the compiler finds a variable does not appear in assignments outside of its initialization, then the variable is considered *effectively final*.

For example, consider some class:

```
public class Foo {

    public void baz(int bar) {
        // While the next line is commented, bar is effectively final
        // and while it is uncommented, the assignment means it is not
        // effectively final.

        // bar = 2;
    }
}
```

edited Feb 11 '16 at 22:39

5gon12eder
17.2k 2 24 64

answered Jan 5 '14 at 19:37

Mark Elliot
54.3k 14 119 147

The docs talk about local variables. `bar` in your example is not a local variaable, but a field. "Effectively final" in the error message as above does not apply to fields at all. – Antti Haapala Jan 1 '15 at 18:47

3 @AnttiHaapala `bar` is a parameter here, not a field. – peter.petrov Feb 11 '16 at 22:42

From a article by 'Brian Goetz',

> 'Effectively final' is a variable which would not give compiler error if it were to be appended by 'final'

lambda-state-final- Brian Goetz

answered Nov 15 '15 at 23:14

Ajeet Ganga
4,138 7 41 59

this answer is shown as a quote, however there is no such exact text in Brian's article, for sure not the word *appended*. This is a quote instead: *Informally, a local variable is effectively final if its initial value is never changed -- in other words, declaring it final would not cause a compilation failure.* – lcfd Apr 11 at 7:26

From the article verbatim copy: Informally, a local variable is effectively final if its initial value is never changed -- in other words, declaring it final would not cause a compilation failure. – Ajeet Ganga Apr 17 at 2:00

This variable below is **final**, so we can't change it's value once initialised. If we try to we'll get a compilation error...

```
final int variable = 123;
```

But if we create a variable like this, we can change it's value...

```
int variable = 123;
variable = 456;
```

But in **Java 8**, all variables are **final** by default. But the existence of the 2nd line in the code makes it **non-final**. So if we remove the 2nd line from the above code, our variable is now **"effectively final"**...

```
int variable = 123;
```

So.. **Any variable that is assigned once and only once, is "effectively final"**.

edited Jun 3 '17 at 23:38

answered Dec 23 '16 at 17:07

Eurig Jones

4,057 6 37 64

When a lambda expression uses an assigned local variable from its enclosing space there is an important restriction. A lambda expression may only use local variable whose value doesn't change. That restriction is referred as "**variable capture**" which is described as; *lambda expression capture values, not variables*.
The local variables that a lambda expression may use are known as "**effectively final**".
An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as final, although doing so would not be an error.
Let's see it with an example, we have a local variable i which is initialized with the value 7, with in the lambda expression we are trying to change that value by assigning a new value to i. This will result in compiler error - "*Local variable i defined in an enclosing scope must be final or effectively final*"

```
@FunctionalInterface
interface IFuncInt {
    int func(int num1, int num2);
    public String toString();
}

public class LambdaVarDemo {

    public static void main(String[] args){
        int i = 7;
        IFuncInt funcInt = (num1, num2) -> {
            i = num1 + num2;
            return i;
        };
    }
}
```

edited Jan 29 '17 at 15:21

Brad Larson ♦

160k 40 359 534

answered Aug 20 '15 at 16:27

infoj

421 5 3

A variable is **final** or **effectively final** when **it's initialized once** and it's never **mutated** in its owner class. And we **can't initialize** it in **loops** or **inner classes**.

Final:

```
final int number;
number = 23;
```

Effectively Final:

```
int number;
number = 34;
```

edited Dec 7 '15 at 16:01

answered Dec 7 '15 at 15:56

samadadi

536 7 18

```
public class LambdaScopeTest {
    public int x = 0;
    class FirstLevel {
        public int x = 1;
        void methodInFirstLevel(int x) {

            // The following statement causes the compiler to generate
            // the error "local variables referenced from a lambda expression
            // must be final or effectively final" in statement A:
            //
```

```
            // x = 99;

        }
    }
}
```

As others have said, a variable or parameter whose value is never changed after it is initialized is effectively final. In the above code, if you change the value of `x` in inner class `FirstLevel` then the compiler will give you the error message:

> Local variables referenced from a lambda expression must be final or effectively final.

edited Aug 26 '14 at 6:44
dimo414
28.9k 12 96 155

answered Jan 5 '14 at 19:53
Tenacious
1,325 1 11 21

> If you could add the `final` modifier to a local variable, it was *effectively final.*

Lambda expressions can access

- static variables,
- instance variables,
- effectively final method parameters, and
- effectively final local variables.

Source: OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide, Jeanne Boyarsky, Scott Selikoff

answered Jun 10 '17 at 15:32
snr
2,904 12 29

Effective final topic is described in JLS 4.12.4 and the last paragraph consists a clear explanation:

> If a variable is effectively final, adding the final modifier to its declaration will not introduce any compile-time errors. Conversely, a local variable or parameter that is declared final in a valid program becomes effectively final if the final modifier is removed.

answered Apr 25 at 4:24
Dmitry N.
1 2

> However, starting in Java SE 8, a local class can access local variables and parameters of the >enclosing block that are final or effectively final.

This didn't start on Java 8, I use this since long time. This code used (before java 8) to be legal:

```
String str = ""; //<-- not accesible from anonymous classes implementation
final String strFin = ""; //<-- accesible
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
         String ann = str; // <---- error, must be final (IDE's gives the hint);
         String ann = strFin; // <---- legal;
         String str = "legal statement on java 7,"
               +"Java 8 doesn't allow this, it thinks that I'm trying to use the
str declared before the anonymous impl.";
         //we are forced to use another name than str
    }
);
```

edited Oct 30 '15 at 10:54

answered Sep 3 '14 at 17:35
FiruzzZ
101 1 11

2 The statement refers to the fact that in <Java 8, *only* `final` variables can be accessed, but in Java 8 *also* those that are *effectively* final. – Antti Haapala Jan 1 '15 at 18:33

I only see code which doesn't work, regardless of whether you are using Java 7 or Java 8. – Holger Aug 21 '15 at 9:13