Compile Time Constants in Java

In this tutorial we’ll get a basic understanding of Compile Time Constants. This is not an exhaustive tutorial about compile time constants, but this should be enough for most people. We’ll understand what compile time constants are and how are they special. First let’s go through the general rules regarding compile time constants.

* They must be declared final
* They are of primitive data types or String
* They must be initialized with their declaration.
* Their value must be constant expression.

If you don’t understand all of this, don’t worry, we’ll go through these rules now.

**Final variables of primitive data types and Strings can be compile time constants.** No other type of variables are compile time constants, not even the wrapper classes. They must also be initialized with their declaration, otherwise they’ll not be compile time constants. Lets take a few examples. All of the following are compile time constants,

|  |  |
| --- | --- |
| 1 2 3 4 | final int i = 10; final int j = 20, k = 30; final [String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) s = "Hello"; final float f = 10.5f; |

The following are not compile time constants,

|  |  |
| --- | --- |
| 1 2 3 4 5 | final [Integer](http://www.google.com/search?hl=en&q=allinurl%3Ainteger+java.sun.com&btnI=I%27m%20Feeling%20Lucky) i = 10;    *//not a primitive or String* int j = 10;              *//not final* final int k;             *//not initialized with declaration* k = 10; final int l, m = l = 20; *//both l and m are not compile time constants* |

The last example might not be intuitive. We are initializing l and m in the same statement as their declaration, but they will not be compile time constants. Compile time constants have to be initialized right with their declaration and you can’t use a different variable in the middle of their declaration.

The last rule about compile time constants is that the compiler must be able to deduce their value. So their value can be any expression which contains literals and other compile time constants. Lets have a look at some valid values for compile time constants,

|  |  |
| --- | --- |
| 1 2 3 | final int i = 10 \* 20;  *//uses only literals* final int j = i;        *//uses another compile time constant* final int k = i \* 20;   *//mixing compile time constant and literal* |

If you use a variable in the expression to assign value to a final variable, then the compiler won’t be able to figure out the value of the final variable, so it will not be a compile time constant. The final variables given below are not compile time constants, because their value is not a constant expression,

|  |  |
| --- | --- |
| 1 2 3 | int i = 10; final int j = i;                 *//using a variable* final int k = [Math](http://www.google.com/search?hl=en&q=allinurl%3Amath+java.sun.com&btnI=I%27m%20Feeling%20Lucky).round(10.2);  *//method call* |

## What’s So special About Compile Time Constants

Now that we know what compile time constants are, we’ll now see why are they special. As the name implies, compile time constants get special treatment by the compiler. The first example of the special treatment is that they are **implicit down casted** (just like literals). If you assign an int variable with value 10 to a short variable, you’ll get an error, but if you assign a int which is a compile time constant with the value 10 to a short variable, that will compile fine. Here is an example,

|  |  |
| --- | --- |
| 1 2 3 4 | int a = 10; short s1 = a;   *//error* final int b = 10; short s2 = b;   *//fine* |

Since the compiler knows the value of compile time constants, the compiler knows that the value of b is 10, so the compiler knows that the value of b is within the range of short data type. You can also use compile time constants as **case values in switch-case statements** as shown in the following example,

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | final int a = 10; int b = 10; switch(b) {     case a:         *//some code*     case 20:         *//some code* } |

But remember, case values cannot be String or floating point numbers, so you can’t use compile time constants of float, double or String data types as case values.

Both the behaviors that we observed above i.e. implicit downcast and case labels are because of a common reason. Wherever you use a compile time constant, the compiler replaces their use with their actual value. So if I write this code,

|  |  |
| --- | --- |
| 1 2 3 | final int i = 10; short j = i; [System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println(i); |

After compilation it will become,

|  |  |
| --- | --- |
| 1 2 3 | final int i = 10; short j = 10; [System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println(10); |

As you can see, at both the places where we used i, the compiler replaced it with its value 10. This is the reason why they can be used as case values and that is why they are implicitly downcasted. This replacing of value has serious implications in some cases. Suppose I have two classes in two different files like this,

|  |  |
| --- | --- |
| 1 2 3 | public class ConstantClass {     public static final int MY\_CONST = 10; } |
| 1 2 3 4 5 | public class Main {     public static void main([String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky)[] args) {         [System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println(ConstantClass.MY\_CONST);     } } |

Now if I compile both classes and run Main class, the output will be 10. Since MY\_CONST is a compile time constant, so the compiler replaces it with its value in the main method. Now I go and change the value of MY\_CONST to 20 and compile only ConstantClass class. Then I run the Main class (without compiling it again), the output will still be 10 i.e. the value of MY\_CONST when Main class was compiled. We’ll have to recompile Main class to see the new value of MY\_CONST as the output.

One more special treatment that compile time constants receive is that you can declare static compile time constants inside non-static inner classes. As you might know, non-static inner classes cannot have static members. But you are allowed to add static compile time constants to non-static inner classes.