What is the best way to implement constants in Java? [closed]

Asked 16 years, 3 months ago Modified 6 years, 11 months ago Viewed 920k times

369

votes

Closed. This question is opinion-based. It is not currently accepting answers.

Closed 6 years ago.

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

I've seen examples like this:

```
public class MaxSeconds {
   public static final int MAX_SECONDS = 25;
}
```

and supposed that I could have a Constants class to wrap constants in, declaring them static final. I know practically no Java at all and am wondering if this is the best way to create constants.

java constants

Share

edited Sep 16, 2008 at 6:27

jjnguy

139k ● 53 ● 297 ● 326

asked Sep 15, 2008 at 19:39

mk.

26.2k ● 13 ● 39 ● 41

1 just to add java constants : public/private – ms_27 May 31, 2016 at 6:54

Similar: Sharing constant strings in Java across many classes? – Basil Bourque Jun 2, 2017 at 23:46

Comments disabled on deleted / locked posts / reviews

28 Answers

Sorted by: Highest score (default)

401 That is perfectly acceptable, probably even the standard.

votes

```
(public/private) static final TYPE NAME = VALUE;
```

where `TYPE` is the type, `NAME` is the name in all caps with underscores for spaces, and `VALUE` is the constant value;

I highly recommend NOT putting your constants in their own classes or interfaces.

As a side note: Variables that are declared final and are mutable can still be changed; however, the variable can never point at a different object.

For example:

```
public static final Point ORIGIN = new Point(0,0);

public static void main(String[] args){

    ORIGIN.x = 3;

}
```

That is legal and `ORIGIN` would then be a point at (3, 0).

Share

edited Oct 22, 2012 at 17:31

answered Sep 15, 2008 at 19:39

jjnguy

139k ● 53 ● 297 ● 326

19 You can even 'import static MaxSeconds.MAX_SECONDS;' so that you don't have to spell it MaxSeconds.MAX_SECONDS – Aaron Maenpaa Sep 15, 2008 at 19:42

23 The previous comment about import static only applies to Java 5+. Some feel the shorthand isn't worth the possible confusion as to where the constant came from, when reading long code, MaxSeconds.MAX_SECONDS may be easier to follow then going up and looking at the imports. – MetroidFan2002 Sep 15, 2008 at 20:35

5 Because you can change the objetc like jjnguy have showed, is best if your constatns are immutable objects or just plain primitive/strings. – marcospereira Sep 16, 2008 at 1:38

14 If you're reading this question please read the next two answers below before you take this answer too seriously, despite being accepted it's controversial if not wrong. – orbfish Oct 22, 2012 at 17:28

3 @jjnguy you are not wrong, but if I read only the question and the first couple lines of your answer, I would be under the impression that "a Constants Class" is "perfectly acceptable, probably even the standard". *That* notion *is* wrong. – Zach Lysobey Oct 16, 2013 at 21:47

234

votes

I would highly advise against having a single constants class. It may seem a good idea at the time, but when developers refuse to document constants and the class

grows to encompass upwards of 500 constants which are all not related to each other at all (being related to entirely different aspects of the application), this generally turns into the constants file being completely unreadable. Instead:

- If you have access to Java 5+, use enums to define your specific constants for an application area. All parts of the application area should refer to enums, not constant values, for these constants. You may declare an enum similar to how you declare a class. Enums are perhaps the most (and, arguably, only) useful feature of Java 5+.
- If you have constants that are only valid to a particular class or one of its subclasses, declare them as either protected or public and place them on the top class in the hierarchy. This way, the subclasses can access these constant values (and if other classes access them via public, the constants aren't only valid to a particular class...which means that the external classes using this constant may be too tightly coupled to the class containing the constant)
- If you have an interface with behavior defined, but returned values or argument values should be particular, it is perfectly acceptible to define constants on that interface so that other implementors will have access to them. However, avoid creating an interface just to hold constants: it can become just as bad as a class created just to hold constants.

Share

answered Sep 15, 2008 at 19:51

MetroidFan2002

29.9k ● 16 ● 65 ● 81

12 totally agree... this can be documented as a classical anti-pattern for large projects. – Das Sep 24, 2008 at 11:14

30 Off topic, but... Generics certainly aren't perfect, but I think you could make a pretty good case for them being a useful feature of Java 5 :) – Martin McNulty Jun 27, 2011 at 10:08

3 @ŁukaszL. The question itself is really opinion based (whenever "best" arises, it typically is a matter of opinion), so the answer is a valid answer to this question. I've given an answer as to what (yes, I *believe* to be, so yes it is an opinion, because again "best" changes with time and is generally opinion based) the best way to implement constants in Java is. – MetroidFan2002 Jan 10, 2014 at 21:10

1 None of the options above give you a true Global variable. What if I have something that is used everywhere but is not an enum? Do I make an enum of just one thing? – markthegrea Jun 12, 2014 at 13:06

1 If you need a global constant that spans all modules, there is probably something wrong with your design strategy. If you *really* need a global constant, make a public final class for it at the top level package and stick it there. Then delete it as soon as you realize not *all* of your classes actually need that constant, and move it to the package that references it the most. You can share a constant across packages, but it is a code smell to require a global constant

that is not an enumerated type, as enumerated types can have behavior, but a string is a string, an int is an int, etc. – MetroidFan2002 Jun 21, 2014 at 3:59

120

votes

It is a **BAD PRACTICE** to use interfaces just to hold constants (named *constant interface pattern* by Josh Bloch). Here's what Josh advises:

> If the constants are strongly tied to an existing class or interface, you should add them to the class or interface. For example, all of the boxed numerical primitive classes, such as Integer and Double, export MIN_VALUE and MAX_VALUE constants. If the constants are best viewed as members of an enumerated type, you should export them with an **enum** type. Otherwise, you should export the constants with a noninstantiable utility class.

Example:

```
// Constant utility class
package com.effectivejava.science;
public class PhysicalConstants {
    private PhysicalConstants() { }  // Prevents instantiation

    public static final double AVOGADROS_NUMBER   = 6.02214199e23;
    public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    public static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

About the naming convention:

> By convention, such fields have names consisting of capital letters, with words separated by underscores. It is critical that these fields contain either primitive values or references to immutable objects.

Share

edited Feb 2, 2016 at 0:20

shmosel

50.6k ● 7 ● 77 ● 145

answered Sep 15, 2008 at 19:58

Marcio Aguiar

14.5k ● 6 ● 40 ● 42

23 If you're going to call something bad practice, perhaps you should explain why you think it is? – GlennS Dec 20, 2012 at 17:43

14 This is an old post, but better to use `abstract` notation for the class instead of private constructor. – Aritz Apr 26, 2013 at 10:10

5 Matt, while biker's recommendation is not false, I would advocate for final classes, rather than abstract classes. biker's point in saying that was that you want to ensure that your constants class is not alterable. So, by marking it as final, you're not allowing it to be subclassed or instantiated. This also helps to encapsulate its static functionality and doesn't allow other

developers to subclass it and make it do things it wasn't designed to do. – liltitus27 Dec 10, 2013 at 15:56

7 @XtremeBiker Marking the class `abstract` instead of a private constructor does not completely prevent instantiation, because one could subclass it and instantiate the subclass (not that it's a good idea to do so, but it's possible). @ToolmakerSteve You can't subclass a class with a private constructor (at least not without a big bad hack), because the constructor of the subclass needs to call the (now private) constructor of its superclass. So, marking it `final` is unnecessary (but perhaps more explicit). – import this Nov 4, 2014 at 16:56

3 It is even worse to use class just to hold constants. If you never want to create an object of that class why you use regular **class** in the first place? – MaxZoom May 5, 2015 at 21:31

---

36 votes

In Effective Java (2nd edition), it's recommended that you use enums instead of static ints for constants.

There's a good writeup on enums in Java here:
http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html

Note that at the end of that article the question posed is:

> So when should you use enums?

With an answer of:

> Any time you need a fixed set of constants

Share

edited Jul 29, 2016 at 7:16

Grzegorz Rożniecki
28k ● 11 ● 93 ● 116

answered Sep 15, 2008 at 20:00

shelfoo
1,557 ● 13 ● 15

---

21 votes

Just avoid using an interface:

```
public interface MyConstants {
    String CONSTANT_ONE = "foo";
}

public class NeddsConstant implements MyConstants {

}
```

It is tempting, but violates encapsulation and blurs the distinction of class definitions.

Share

answered Sep 15, 2008 at 19:45

stimpy

1 Indeed static imports are far preferable. – Aaron Maenpaa Sep 15, 2008 at 19:55

1 This practice is a strange use of interfaces, which are intended to state the services provided by a class. – Rafa Romero Oct 15, 2014 at 9:49

1 I'm not disagreeing with avoiding use of interface for constants, but your example is misleading I think. You don't need to implement the interface to get access to the constant because its implicitly static, public, and final. So, its simpler than you have described here. – djangofan Jan 7, 2017 at 20:11

That's true...it's violating encapsulation, I prefer to use **final Constant class** which look more object oriently and **preferable over interface OR enum**. **Note:** Enum can be **avoid** in case of Android is not needed. – CoDe Sep 3, 2017 at 10:20

18 votes

I use following approach:

```
public final class Constants {
  public final class File {
    public static final int MIN_ROWS = 1;
    public static final int MAX_ROWS = 1000;

    private File() {}
  }

  public final class DB {
    public static final String name = "oups";

    public final class Connection {
      public static final String URL = "jdbc:tra-ta-ta";
      public static final String USER = "testUser";
      public static final String PASSWORD = "testPassword";

      private Connection() {}
    }

    private DB() {}
  }

  private Constants() {}
}
```

Than, for example, I use `Constants.DB.Connection.URL` to get constant. It looks more "object oriently" as for me.

Share

edited Jul 12, 2012 at 16:35

Sharjeel Aziz

8,535 ● 5 ● 40 ● 37

answered Jan 10, 2012 at 19:47

albus.ua

261 ● 2 ● 7

4 Interesting, but cumbersome. Why wouldn't you instead create constants in the class most closely associated with them, as recommended by others? For example, in your DB code elsewhere do you have a base class for your connections? E.g. "ConnectionBase". Then you can put the constants there. Any code that is working with connections will likely already have an import such that can simply say "ConnectionBase.URL" rather than "Constants.DB.Connection.URL". – ToolmakerSteve Jul 22, 2014 at 0:41

3 @ToolmakerSteve But what about general constants that multiple classes can use? for example, styles, web services urls, etc...? – Daniel Gomez Rico Jan 27, 2015 at 19:19

Putting all the constants in one class has one advantage - maintenance. You always know exactly where to find the constants. I'm not saying this makes this technique better, I'm just providing one advantage. And what @albus.ua has done is categorized his constants, which is a pretty good idea especially if the constants class contains many many constant values. This technique will help keep the class manageable and helps to better describe the purpose of the constant. – Nelda.techspiress May 10, 2017 at 14:50

17 votes

Creating static final constants in a separate class can get you into trouble. The Java compiler will actually optimize this and place the actual value of the constant into any class that references it.

If you later change the 'Constants' class and you don't do a hard re-compile on other classes that reference that class, you will wind up with a combination of old and new values being used.

Instead of thinking of these as constants, think of them as configuration parameters and create a class to manage them. Have the values be non-final, and even consider using getters. In the future, as you determine that some of these parameters actually should be configurable by the user or administrator, it will be much easier to do.

Share

answered Sep 16, 2008 at 3:36

Kevin Day
16.4k ● 8 ● 48 ● 71

+1 for this excellent caveat. (If you can't guarantee it's a permanent constant, consider a getter instead, as big_peanut_horse mentions.) The same applies with const in C#, by the way: msdn.microsoft.com/en-us/library/e6w8fe1b.aspx – Jon Coombs Jun 20, 2014 at 16:20

13 votes

The number one mistake you can make is creating a globally accessible class called with a generic name, like Constants. This simply gets littered with garbage and you lose all ability to figure out what portion of your system uses these constants.

Instead, constants should go into the class which "owns" them. Do you have a constant called TIMEOUT? It should probably go into your Communications() or

Connection() class. MAX_BAD_LOGINS_PER_HOUR? Goes into User(). And so on and so forth.

The other possible use is Java .properties files when "constants" can be defined at run-time, but not easily user changeable. You can package these up in your .jars and reference them with the Class resourceLoader.

Share

answered Sep 15, 2008 at 20:26

Yann Ramin

33.1k ● 3 ● 62 ● 83

> And of course, you would never want to access a constant from more than one class, or avoid clutter at the top of a class. – orbfish Nov 12, 2014 at 18:35

---

6 votes

That's the right way to go.

Generally constants are *not* kept in separate "Constants" classes because they're not discoverable. If the constant is relevant to the current class, keeping them there helps the next developer.

Share

answered Sep 15, 2008 at 19:44

Jason Cohen

83k ● 26 ● 110 ● 114

---

5 votes

What about an enumeration?

Share

answered Sep 15, 2008 at 19:50

Sébastien D.

171 ● 3

---

5 votes

I prefer to use getters rather than constants. Those getters might return constant values, e.g. `public int getMaxConnections() {return 10;}`, but anything that needs the constant will go through a getter.

One benefit is that if your program outgrows the constant--you find that it needs to be configurable--you can just change how the getter returns the constant.

The other benefit is that in order to modify the constant you don't have to recompile everything that uses it. When you reference a static final field, the value of that constant is compiled into any bytecode that references it.

Share

answered Sep 16, 2008 at 1:30

big_peanut_horse
71 ● 1

5 Well recompiling referencing classes is hardly a burden in the 21st century. And you should never use the accessor/mutator (getter/setter) model for things *other* than accessing and mutating member variables. Constants are conceptually meant to be immediates in nature, while getter/setters are (both) meant to manage **state**. Besides, you're only asking for confusion: people won't expect a getter to yield a mere constant. – user4229245 Jan 4, 2015 at 20:39

5
votes

I agree that using an interface is not the way to go. Avoiding this pattern even has its own item (#18) in Bloch's Effective Java.

An argument Bloch makes against the constant interface pattern is that use of constants is an implementation detail, but implementing an interface to use them exposes that implementation detail in your exported API.

The `public|private static final TYPE NAME = VALUE;` pattern is a good way of declaring a constant. Personally, I think it's better to avoid making a separate class to house all of your constants, but I've never seen a reason not to do this, other than personal preference and style.

If your constants can be well-modeled as an enumeration, consider the enum structure available in 1.5 or later.

If you're using a version earlier than 1.5, you can still pull off typesafe enumerations by using normal Java classes. (See this site for more on that).

Share

edited Jul 29, 2016 at 7:22

Grzegorz Rożniecki
28k ● 11 ● 93 ● 116

answered Sep 15, 2008 at 20:00

Rob Dickerson
1,469 ● 9 ● 6

Some constants may be used to call an API. For example, see interface org.springframework.transaction.TransactionDefinition. It has a list of constants like int PROPAGATION_REQUIRED = 0; – borjab Aug 31, 2015 at 8:12

I know this is old, but the link to Bloch's Effective Java is broken. Could you provide another or another reference supporting that "using an interface is not the way to go" please? – Nelda.techspiress May 10, 2017 at 14:41

4
votes

Based on the comments above I think this is a good approach to change the old-fashioned global constant class (having public static final variables) to its enum-like equivalent in a way like this:

```
public class Constants {

    private Constants() {
        throw new AssertionError();
    }

    public interface ConstantType {}

    public enum StringConstant implements ConstantType {
        DB_HOST("localhost");
        // other String constants come here

        private String value;
        private StringConstant(String value) {
            this.value = value;
        }
        public String value() {
            return value;
        }
    }

    public enum IntConstant implements ConstantType {
        DB_PORT(3128),
        MAX_PAGE_SIZE(100);
        // other int constants come here

        private int value;
        private IntConstant(int value) {
            this.value = value;
        }
        public int value() {
            return value;
        }
    }

    public enum SimpleConstant implements ConstantType {
        STATE_INIT,
        STATE_START,
        STATE_END;
    }

}
```

So then I can refer them to like:

```
Constants.StringConstant.DB_HOST
```

Share

edited Jun 27, 2011 at 9:39

DarthJDG

16.6k ● 11 ● 51 ● 56

answered Jun 26, 2011 at 21:16

Lorand Bendig

10.7k ● 1 ● 39 ● 46

4 Why? How is that an improvement? Every reference is now cumbersome (Constants.StringConstant.whatever). IMHO, you are going down a bumpy road here. – ToolmakerSteve Jul 22, 2014 at 0:49

3 votes

A good object oriented design should not need many publicly available constants. Most constants should be encapsulated in the class that needs them to do its job.

Share

edited Jan 28, 2009 at 21:12

John Topley
115k ● 47 ● 199 ● 240

answered Sep 15, 2008 at 23:58

Bradley Harris
932 ● 1 ● 6 ● 12

1 Or injected into that class via the constructor. – WW. Jun 25, 2010 at 1:37

2 votes

There is a certain amount of opinion to answer this. To start with, constants in java are generally declared to be public, static and final. Below are the reasons:

```
public, so that they are accessible from everywhere
static, so that they can be accessed without any instance. Since they are
constants it
  makes little sense to duplicate them for every object.
final, since they should not be allowed to change
```

I would never use an interface for a CONSTANTS accessor/object simply because interfaces are generally expected to be implemented. Wouldn't this look funny:

```
String myConstant = IMyInterface.CONSTANTX;
```

Instead I would choose between a few different ways, based on some small trade-offs, and so it depends on what you need:

```
1.  Use a regular enum with a default/private constructor. Most people would
define
     constants this way, IMHO.
  - drawback: cannot effectively Javadoc each constant member
  - advantage: var members are implicitly public, static, and final
  - advantage: type-safe
  - provides "a limited constructor" in a special way that only takes args
which match
     predefined 'public static final' keys, thus limiting what you can pass to
the
     constructor

2.  Use a altered enum WITHOUT a constructor, having all variables defined with
     prefixed 'public static final' .
  - looks funny just having a floating semi-colon in the code
```

```
  - advantage: you can JavaDoc each variable with an explanation
  - drawback: you still have to put explicit 'public static final' before each
variable
  - drawback: not type-safe
  - no 'limited constructor'

3.  Use a Class with a private constructor:
  - advantage: you can JavaDoc each variable with an explanation
  - drawback: you have to put explicit 'public static final' before each
variable
  - you have the option of having a constructor to create an instance
     of the class if you want to provide additional functions related
     to your constants
     (or just keep the constructor private)
  - drawback: not type-safe

4. Using interface:
  - advantage: you can JavaDoc each variable with an explanation
  - advantage: var members are implicitly 'public static final'
  - you are able to define default interface methods if you want to provide
additional
     functions related to your constants (only if you implement the interface)
  - drawback: not type-safe
```

Share

edited Jan 8, 2017 at 22:06

answered Jan 7, 2017 at 20:24

djangofan

29.6k ● 61 ● 207 ● 301

2

votes

> What is the best way to implement constants in Java?

One approach that we should really avoid : using interfaces to define constants.

Creating a interface specifically to declare constants is really the worst thing : it defeats the reason why interfaces were designed : defining method(s) contract.

Even if an interface already exists to address a specific need, declaring the constants in them make really not sense as constants should not make part of the API and the contract provided to client classes.

To simplify, we have broadly 4 valid approaches.

With `static final String/Integer` field :

- 1) using a class that declares constants inside but not only.
- 1 variant) creating a class dedicated to only declare constants.

With `Java 5 enum` :

- 2) declaring the enum in a related purpose class (so as a nested class).
- 2 variant) creating the enum as a standalone class (so defined in its own class file).

TLDR : Which is the best way and where locate the constants ?

In most of cases, the enum way is probably finer than the `static final String/Integer` way and personally I think that the `static final String/Integer` way should be used only if we have good reasons to not use enums.
And about where we should declare the constant values, **the idea is to search whether there is a single existing class that owns a specific and strong functional cohesion with constant values. If we find such a class, we should use it as the constants holder**. Otherwise, the constant should be associated to no one particular class.

`static final String` / `static final Integer` versus `enum`

Enums usage is really a way to strongly considered.
Enums have a great advantage over `String` or `Integer` constant field.
They set a stronger compilation constraint. If you define a method that takes the enum as parameter, you can only pass a enum value defined in the enum class(or null).
With String and Integer you can substitute them with any values of compatible type and the compilation will be fine even if the value is not a defined constant in the `static final String` / `static final Integer` fields.

For example, below two constants defined in a class as `static final String` fields :

```
public class MyClass{

   public static final String ONE_CONSTANT = "value";
   public static final String ANOTHER_CONSTANT = "other value";
   . . .
}
```

Here a method that expects to have one of these constants as parameter :

```
public void process(String constantExpected){
    ...
}
```

You can invoke it in this way :

```
process(MyClass.ONE_CONSTANT);
```

or

```
process(MyClass.ANOTHER_CONSTANT);
```

But no compilation constraint prevents you from invoking it in this way :

```
process("a not defined constant value");
```

You would have the error only at runtime and only if you do at a time a check on the transmitted value.

With enum, checks are not required as the client could only pass a enum value in a enum parameter.

For example, here two values defined in a enum class (so constant out of the box):

```
public enum MyEnum {

    ONE_CONSTANT("value"), ANOTHER_CONSTANT(" another value");

    private String value;

    MyEnum(String value) {
       this.value = value;
    }
         ...
}
```

Here a method that expects to have one of these enum values as parameter :

```
public void process(MyEnum myEnum){
    ...
}
```

You can invoke it in this way :

```
process(MyEnum.ONE_CONSTANT);
```

or

```
process(MyEnum.ANOTHER_CONSTANT);
```

But the compilation will never allow you from invoking it in this way :

```
process("a not defined constant value");
```

Where should we declare the constants ?

If your application contains a single existing class that owns a specific and strong functional cohesion with the constant values, the 1) and the 2) appear more intuitive.
Generally, it eases the use of the constants if these are declared in the main class that manipulates them or that has a name very natural to guess that we will find it inside.

For example in the JDK library, the exponential and pi constant values are declared in a class that declare not only constant declarations (`java.lang.Math`).

```
   public final class Math {
          ...
       public static final double E = 2.7182818284590452354;
       public static final double PI = 3.14159265358979323846;
         ...
   }
```

The clients using mathematics functions rely often on the `Math` class. So, they may find constants easily enough and can also remember where `E` and `PI` are defined in a very natural way.

If your application doesn't contain an existing class that has a very specific and strong functional cohesion with the constant values, the 1 variant) and the 2 variant) ways appear more intuitive.
Generally, it doesn't ease the use of the constants if these are declared in one class that manipulates them while we have also 3 or 4 other classes that manipulate them as much as and no one of these classes seems be more natural than others to host constant values.
Here, defining a custom class to hold only constant values makes sense.
For example in the JDK library, the `java.util.concurrent.TimeUnit` enum is not declared in a specific class as there is not really one and only one JDK specific class that appear as the most intuitive to hold it :

```
public enum TimeUnit {
    NANOSECONDS {
      .....
    },
    MICROSECONDS {
      .....
    },
    MILLISECONDS {
      .....
    },
    SECONDS {
```

```
      .....
    },
      .....
}
```

Many classes declared in `java.util.concurrent` use them : `BlockingQueue`, `ArrayBlockingQueue<E>`, `CompletableFuture`, `ExecutorService` , ... and really no one of them seems more appropriate to hold the enum.

Share

edited Dec 29, 2017 at 12:41

answered Jun 9, 2017 at 8:18

davidxxx
131k • 23 • 228 • 225

1

vote

A Constant, of any type, can be declared by creating an immutable property that within a class (that is a member variable with the `final` modifier). Typically the `static` and `public` modifiers are also provided.

```
public class OfficePrinter {
    public static final String STATE = "Ready";
}
```

There are numerous applications where a constant's value indicates a selection from an n-tuple (e.g. *enumeration*) of choices. In our example, we can choose to define an Enumerated Type that will restrict the possible assigned values (i.e. improved *type-safety*):

```
public class OfficePrinter {
    public enum PrinterState { Ready, PCLoadLetter, OutOfToner, Offline };
    public static final PrinterState STATE = PrinterState.Ready;
}
```

Share

answered Sep 15, 2008 at 21:17

Ryan Delucchi
7,748 • 13 • 51 • 60

1

vote

A single, generic constants class is a bad idea. Constants should be grouped together with the class they're most logically related to.

Rather than using variables of any kind (especially enums), I would suggest that you use methods. Create a method with the same name as the variable and have it return the value you assigned to the variable. Now delete the variable and replace all references to it with calls to the method you just created. If you feel that the constant is

generic enough that you shouldn't have to create an instance of the class just to use it, then make the constant method a class method.

Share

answered Sep 15, 2008 at 22:17

ab

---

1 vote

FWIW, a timeout in seconds value should probably be a configuration setting (read in from a properties file or through injection as in Spring) and not a constant.

Share

answered Sep 16, 2008 at 1:33

Tim Howland

7,970 ● 4 ● 29 ● 46

---

1 vote

What is the difference

1.

```
public interface MyGlobalConstants {
    public static final int TIMEOUT_IN_SECS = 25;
}
```

2.

```
public class MyGlobalConstants {
    private MyGlobalConstants () {} // Prevents instantiation
    public static final int TIMEOUT_IN_SECS = 25;
}
```

and using `MyGlobalConstants.TIMEOUT_IN_SECS` wherever we need this constant. I think both are same.

Share

edited Dec 5, 2012 at 6:15

Matthieu

16.4k ● 10 ● 61 ● 86

answered Dec 9, 2010 at 21:11

chandrayya

11 ● 1

---

1 This appears to be essentially a comment in response to the answer bincob gave. I think they do behave very similarly, but bincob's point was that they do not in any way define an interface. And the suggestion was to add constants to real classes, not to create a MyGlobalConstants skeleton class. (Although this occasionally makes sense; use a "static class" by having static constants and a private constructor to prevent instantiation; see java.lang.math.) Consider using enum. – Jon Coombs Jun 20, 2014 at 16:14

Also putting "final" in the class declaration will prevent subclassing. (In C# you could do "static", which means "final abstract", so no need for explicit private constructor.) – Jon Coombs Jun 20,

2014 at 16:34

Yes, @JonCoombs but "final" does not prevent it's direct instantiation. And Java disallows both final and abstract to appear together for classes, hence the endlessly appearing private constructor to prevent both instantiation AND subclassing. I have no idea why "final abstract" is disallowed, other than it at first glance seems contradictory in how it reads: "you can't subclass but this class is meant to be subclassed". – user4229245 Jan 4, 2015 at 20:45

0 votes

I wouldn't call the class the same (aside from casing) as the constant ... I would have at a minimum one class of "Settings", or "Values", or "Constants", where all the constants would live. If I have a large number of them, I'd group them up in logical constant classes (UserSettings, AppSettings, etc.)

Share

answered Sep 15, 2008 at 19:41

Joel Martinez

47.7k ● 26 ● 134 ● 185

Having a class called Constants was what I planned on doing, that was just a small example I found. – mk. Sep 15, 2008 at 19:47

0 votes

To take it a step further, you can place globally used constants in an interface so they can be used system wide. E.g.

```
public interface MyGlobalConstants {
    public static final int TIMEOUT_IN_SECS = 25;
}
```

But don't then implement it. Just refer to them directly in code via the fully qualified classname.

Share

answered Sep 15, 2008 at 19:45

Andrew Harmel-Law

7,879 ● 12 ● 45 ● 56

The point about declaring them in an interface (and not implementing it) is that you can miss out the "public static final". – Tom Hawtin - tackline Sep 16, 2008 at 18:44

9 Interfaces are for defining a behavioural contract, not a convenience mechanism for holding constants. – John Topley Jan 28, 2009 at 21:16

@JohnTopley Yeah, but it works. ;) – trusktr Oct 4, 2013 at 1:14

0 For Constants, Enum is a better choice IMHO. Here is an example

votes

public class myClass {

```
public enum myEnum {
    Option1("String1", 2),
    Option2("String2", 2)
    ;
    String str;
            int i;

            myEnum(String str1, int i1) { this.str = str1 ; this.i1 = i }


}
```

Share

answered Sep 15, 2008 at 20:45

mmansoor

620 ● 6 ● 11

0
votes

One of the way I do it is by creating a 'Global' class with the constant values and do a static import in the classes that need access to the constant.

Share

answered Sep 15, 2008 at 21:44

Javamann

2,902 ● 2 ● 25 ● 22

0
votes

`static final` is my preference, I'd only use an `enum` if the item was indeed enumerable.

Share

answered Aug 31, 2012 at 1:49

wulfgarpro

6,906 ● 12 ● 74 ● 112

0
votes

I use `static final` to declare constants and go with the ALL_CAPS naming notation. I have seen quite a few real life instances where all constants are bunched together into an interface. A few posts have rightly called that a bad practice, primarily because that's not what an interface is for. An interface should enforce a contract and should not be a place to put unrelated constants in. Putting it together into a class that cannot be instantiated (through a private constructor) too is fine if the constant semantics don't belong to a specific class(es). I always put a constant in the class that it's most related to, because that makes sense and is also easily maintainable.

Enums are a good choice to represent a range of values, but if you are storing standalone constants with an emphasis on the absolute value (eg. TIMEOUT = 100

ms) you can just go for the `static final` approach.

Share

answered May 15, 2013 at 12:45

bincob

919 ● 1 ● 9 ● 13

0

votes

I agree with what most are saying, it is best to use enums when dealing with a collection of constants. However, if you are programming in Android there is a better solution: IntDef Annotation.

```
@Retention(SOURCE)
@IntDef({NAVIGATION_MODE_STANDARD, NAVIGATION_MODE_LIST,NAVIGATION_MODE_TABS})
public @interface NavigationMode {}
public static final int NAVIGATION_MODE_STANDARD = 0;
public static final int NAVIGATION_MODE_LIST = 1;
public static final int NAVIGATION_MODE_TABS = 2;
...
public abstract void setNavigationMode(@NavigationMode int mode);
@NavigationMode
public abstract int getNavigationMode();
```

IntDef annotation is superior to enums in one simple way, it takes significantly less space as it is simply a compile-time marker. It is not a class, nor does it have the automatic string-conversion property.

Share

edited Sep 20, 2016 at 19:37

answered Jul 28, 2016 at 14:40

Quinn Turner

242 ● 1 ● 2 ● 11

> While I agree with the best practice of not using interface in Java alone and avoiding enum in Android, this replacement in Android only works when you are using a small number of fields. It is a significant memory savings but can lead to bloat as you are interface per field in a mixed enum. E.g. if I have a mixed enum that defines what an object takes in its constructor, I cannot save anything with this approach, and am back to non type safe constants because in Android, you don't want too many classes/interfaces period. – Droid Teahouse Aug 26, 2017 at 22:36

0

votes

It is **BAD habit and terribly ANNOYING practice** to quote Joshua Bloch without understanding the basic ground-zero fundamentalism.

I have not read anything Joshua Bloch, so either

- he is a terrible programmer
- or the people so far whom I find quoting him (Joshua is the name of a boy I presume) are simply using his material as religious scripts to justify their software

religious indulgences.

As in Bible fundamentalism all the biblical laws can be summed up by

- Love the Fundamental Identity with all your heart and all your mind
- Love your neighbour as yourself

and so similarly software engineering fundamentalism can be summed up by

- devote yourself to the ground-zero fundamentals with all your programming might and mind
- and devote towards the excellence of your fellow-programmers as you would for yourself.

Also, among biblical fundamentalist circles a strong and reasonable corollary is drawn

- First love yourself. Because if you don't love yourself much, then the concept "love your neighbour as yourself" doesn't carry much weight, since "how much you love yourself" is the datum line above which you would love others.

Similarly, if you do not respect yourself as a programmer and just accept the pronouncements and prophecies of some programming guru-nath WITHOUT questioning the fundamentals, your quotations and reliance on Joshua Bloch (and the like) is meaningless. And therefore, you would actually have no respect for your fellow-programmers.

The fundamental laws of software programming

- laziness is the virtue of a good programmer
- you are to make your programming life as easy, as lazy and therefore as effective as possible
- you are to make the consequences and entrails of your programming as easy, as lazy and therefore as effective as possible for your neigbour-programmers who work with you and pick up your programming entrails.

Interface-pattern constants is a bad habit ???

Under what laws of fundamentally effective and responsible programming does this religious edict fall into ?

Just read the wikipedia article on interface-pattern constants (https://en.wikipedia.org/wiki/Constant_interface), and the silly excuses it states against interface-pattern constants.

- Whatif-No IDE? Who on earth as a software programmer would not use an IDE? Most of us are programmers who prefer not to have to prove having macho aescetic survivalisticism thro avoiding the use of an IDE.
    - Also - wait a second proponents of micro-functional programming as a means of not needing an IDE. Wait till you read my explanation on data-model normalization.
- Pollutes the namespace with variables not used within the current scope? It could be proponents of this opinion
    - are not aware of, and the need for, data-model normalization
- Using interfaces for enforcing constants is an abuse of interfaces. Proponents of such have a bad habit of
    - not seeing that "constants" must be treated as contract. And interfaces are used for enforcing or projecting compliance to a contract.
- It is difficult if not impossible to convert interfaces into implemented classes in the future. Hah .... hmmm ... ???
    - Why would you want to engage in such pattern of programming as your persistent livelihood? IOW, why devote yourself to such an AMBIVALENT and bad programming habit ?

Whatever the excuses, there is NO VALID EXCUSE when it comes to FUNDAMENTALLY EFFECTIVE software engineering to delegitimize or generally discourage the use of interface constants.

It doesn't matter what the original intents and mental states of the founding fathers who crafted the United States Constitution were. We could debate the original intents of the founding fathers but all I care is the written statements of the US Constitution. And it is the responsibility of every US citizen to exploit the written literary-fundamentalism, not the unwritten founding-intents, of the US Constitution.

Similarly, I do not care what the "original" intents of the founders of the Java platform and programming language had for the interface. What I care are the effective features the Java specification provides, and I intend to exploit those features to the fullest to help me fulfill the fundamental laws of responsible software programming. I don't care if I am perceived to "violate the intention for interfaces". I don't care what Gosling or someone Bloch says about the "proper way to use Java", unless what they say does not violate my need to EFFECTIVE fulfilling fundamentals.

The Fundamental is Data-Model Normalization

It doesn't matter how your data-model is hosted or transmitted. Whether you use interfaces or enums or whatevernots, relational or no-SQL, if you don't understand the

need and process of data-model normalization.

We must first define and normalize the data-model of a set of processes. And when we have a coherent data-model, ONLY then can we use the process flow of its components to define the functional behaviour and process blocks a field or realm of applications. And only then can we define the API of each functional process.

Even the facets of data normalization as proposed by EF Codd is now severely challenged and severely-challenged. e.g. his statement on 1NF has been criticized as ambiguous, misaligned and over-simplified, as is the rest of his statements especially in the advent of modern data services, repo-technology and transmission. IMO, the EF Codd statements should be completely ditched and new set of more mathematically plausible statements be designed.

A glaring flaw of EF Codd's and the cause of its misalignment to effective human comprehension is his belief that humanly perceivable multi-dimensional, mutable-dimension data can be efficiently perceived thro a set of piecemeal 2-dimensional mappings.

The Fundamentals of Data Normalization

What EF Codd failed to express.

Within each coherent data-model, these are the sequential graduated order of data-model coherence to achieve.

1. The Unity and Identity of data instances.
    - design the granularity of each data component, whereby their granularity is at a level where each instance of a component can be uniquely identified and retrieved.
    - absence of instance aliasing. i.e., no means exist whereby an identification produces more than one instance of a component.
2. Absence of instance crosstalk. There does not exist the necessity to use one or more other instances of a component to contribute to identifying an instance of a component.
3. The unity and identity of data components/dimensions.
    - Presence of component de-aliasing. There must exist one definition whereby a component/dimension can be uniquely identified. Which is the primary definition of a component;
    - where the primary definition will not result in exposing sub-dimensions or member-components that are not part of an intended component;

4. Unique means of component dealiasing. There must exist one, and only one, such component de-aliasing definition for a component.
5. There exists one, and only one, definition interface or contract to identify a parent component in a hierarchical relationship of components.
6. Absence of component crosstalk. There does not exist the necessity to use a member of another component to contribute to the definitive identification of a component.
    - In such a parent-child relationship, the identifying definition of a parent must not depend on part of the set of member components of a child. A member component of a parent's identity must be the complete child identity without resorting to referencing any or all of the children of a child.
7. Preempt bi-modal or multi-modal appearances of a data-model.
    - When there exists two candidate definitions of a component, it is an obvious sign that there exists two different data-models being mixed up as one. That means there is incoherence at the data-model level, or the field level.
    - A field of applications must use one and only one data-model, coherently.
8. Detect and identify component mutation. Unless you have performed statistical component analysis of huge data, you probably do not see, or see the need to treat, component mutation.
    - A data-model may have its some of its components mutate cyclically or gradually.
    - The mode may be member-rotation or transposition-rotation.
    - Member-rotation mutation could be distinct swapping of child components between components. Or where completely new components would have to be defined.
    - Transpositional mutation would manifest as a dimensional-member mutating into an attribute, vice versa.
    - Each mutation cycle must be identified as a distinct data-modal.
9. Versionize each mutation. Such that you can pull out a previous version of the data model, when perhaps the need arise to treat an 8 year old mutation of the data model.

In a field or grid of inter-servicing component-applications, there must be one and only one coherent data-model or exists a means for a data-model/version to identify itself.

Are we still asking if we could use Interface Constants? Really ?

There are data-normalization issues at stake more consequential than this mundane question. IF you don't solve those issues, the confusion that you think interface constants cause is comparatively nothing. Zilch.

From the data-model normalization then you determine the components as variables, as properties, as contract interface constants.

Then you determine which goes into value injection, property configuration placeholding, interfaces, final strings, etc.

If you have to use the excuse of needing to locate a component easier to dictate against interface constants, it means you are in the bad habit of not practicing data-model normalization.

Perhaps you wish to compile the data-model into a vcs release. That you can pull out a distinctly identifiable version of a data-model.

Values defined in interfaces are completely assured to be non-mutable. And shareable. Why load a set of final strings into your class from another class when all you need is that set of constants ??

So why not this to publish a data-model contract? I mean if you can manage and normalize it coherently, why not? ...

```
public interface CustomerService {
  public interface Label{
    char AssignmentCharacter = ':';
    public interface Address{
      String Street = "Street";
      String Unit= "Unit/Suite";
      String Municipal = "City";
      String County = "County";
      String Provincial = "State";
      String PostalCode = "Zip"
    }

    public interface Person {
      public interface NameParts{
        String Given = "First/Given name"
        String Auxiliary = "Middle initial"
        String Family = "Last name"
      }
    }
  }
}
```

Now I can reference my apps' contracted labels in a way such as

```
CustomerService.Label.Address.Street
CustomerService.Label.Person.NameParts.Family
```

This confuses the contents of the jar file? As a Java programmer I don't care about the structure of the jar.

This presents complexity to osgi-motivated runtime swapping ? Osgi is an extremely efficient means to allow programmers to continue in their bad habits. There are better alternatives than osgi.

Or why not this? There is no leakage of of the private Constants into published contract. All private constants should be grouped into a private interface named "Constants", because I don't want to have to search for constants and I am too lazy to repeatedly type "private final String".

```
public class PurchaseRequest {
  private interface Constants{
    String INTERESTINGName = "Interesting Name";
    String OFFICIALLanguage = "Official Language"
    int MAXNames = 9;
  }
}
```

Perhaps even this:

```
public interface PurchaseOrderConstants {
  public interface Properties{
    default String InterestingName(){
       return something();
    }
    String OFFICIALLanguage = "Official Language"
    int MAXNames = 9;
  }
}
```

The only issue with interface constants worth considering is when the interface is implemented.

This is not the "original intention" of interfaces? Like I would care about the "original intention" of the founding fathers in crafting the US Constitution, rather than how the Supreme Court would interpret the written letters of the US Constitution ???

After all, I live in the land of the free, the wild and home of the brave. Be brave, be free, be wild - use the interface. If my fellow-programmers refuse to use efficient and lazy means of programming, am I obliged by the golden rule to lessen my programming efficiency to align with theirs? Perhaps I should, but that is not an ideal situation.

Share

edited Jan 10, 2018 at 12:18

answered Jan 10, 2018 at 11:41

Blessed Geek

21.6k ● 25 ● 108 ● 179