Does reflection breaks the idea of private methods, because private methods can be access outside of the class?

Asked 14 years, 5 months ago Modified 1 year, 10 months ago

Viewed 7k times Part of PHP Collective

30

Does reflection break the idea of private methods? Because private methods can be accessed from outside of the class? (Maybe I don't understand the meaning of reflection or miss something else, please tell me) http://en.wikipedia.org/wiki/Reflection_%28computer_science%29

Edit: If relection breaks the idea of private methods - do we use private methods only for program logic and not for program security?

Thanks

PHP c# java php reflection junit

Share

Improve this question

Follow

edited Feb 19, 2014 at 15:26

RobIII

8,811 ● 4 ● 45 ● 97

asked Jul 21, 2010 at 15:00

Ben

25.8k ● 35 ● 112 ● 166

19 Reflection doesnt break encapsulation. Developers do ;)
– Gordon Jul 21, 2010 at 15:20

2 Likewise, casting breaks the idea of static typing, right?
– Ken Jul 21, 2010 at 15:28

@Ken: Only if it's C-style casting, without limits.
– Steven Sudit Jul 21, 2010 at 15:32

What about downcasting? It forces the runtime to do a type check, which defeats the purpose of static typing, i.e., compile-time type checking. – Ken Jul 21, 2010 at 16:12

@Steven Sudit, how can it? since then C-style casting converts object to another type in some reasonable way?
– Valentin Golev Jul 21, 2010 at 17:04

14 Answers

Sorted by: Highest score (default)

28

> do we use private methods only for program logic and not for program security?

It is not clear what you mean by "program security". Security cannot be discussed in a vacuum; what resources are you thinking of protecting against what threats?

The CLR code access security system is intended to protect resources of *user data* from the threat of *hostile*

partially trusted code running on the user's machine.

The relationship between reflection, access control and security in the CLR is therefore complicated. Briefly and not entirely accurately, the rules are these:

- full trust means full trust. **Fully trusted code can access every single bit of memory in the process.** That includes private fields.
- The ability to reflect on privates in partial trust is controlled by a permission; if it is not granted then partial trust code may not do reflection on privates.

See Link for details.

- The desktop CLR supports a mode called "restricted skip visibility" in which the rules for how reflection and the security system interact are slightly different. Basically, partially trusted code that has the right to use private reflection may access a private field via reflection if the partially trusted code is accessing a private field from a type that comes from an assembly with *equal* or *less* trust.

See

Link

for details

The executive summary is: you *can* lock partially trusted code down sufficiently that it is not able to use reflection

to look at private stuff. You *cannot* lock down full trust code; that's why it's called "full trust". If you want to restrict it then *don't trust it*.

So: does making a field private protect it from the threat of low trust code attempting to read it, and thereby steal user's data? *Yes*. Does it protect it from the threat of *high trust* code reading it? *No*. If the code is both trusted by the user and hostile to the user *then the user has a big problem*. They should not have trusted that code.

Note that for example, making a field private does not protect *a secret in your code* from *a user who has your code and is hostile to you*. The security system protects *good users* from *evil code*. It doesn't protect *good code* from *evil users*. If you want to make something private to keep it from a *user* then you are on a fool's errand. If you want to make it private to keep a secret from *evil hackers who have lured the user into running hostile low-trust code* then that is a good technique.

Share Improve this answer Follow

edited Feb 24, 2023 at 12:08

Glorfindel

22.6k ● 13 ● 89 ● 116

answered Jul 21, 2010 at 22:19

Eric Lippert

659k ● 183 ● 1.3k ● 2.1k

Reflection does provide a way to circumvent Java's Access Protection Modifiers and therefore **violates strict**

19

encapsulation as it realised in C++ and Java. However this does not matter as much as you might think.

Access Protection Modifiers are intended to assist programmers to develop modular well factored systems, not to be uncompromising gate keepers. There are sometimes very good reasons to break strict encapsulation such as **Unit Testing and framework development**.

While it may initially be difficult to stomach the idea that Access Protection Modifiers are easily circumventable, try to remember that there are many languages (Python, Ruby etc.) that do not have them at all. These languages are used to build large and complex systems just like languages which do provide access protection.

There is some debate on whether Access Protection Modifiers are a help or a hindrance. Even if you do value access protection treat it like a helping hand, not the making or breaking of your project.

Share Improve this answer Follow

edited Jul 21, 2010 at 21:46

answered Jul 21, 2010 at 15:08

Tendayi Mawushe

26.1k ● 7 ● 52 ● 57

1 The granddaddy of object-oriented programming languages, Smalltalk, doesn't have them, either. In fact, the first language I remember seeing Access Protection Modifiers in

was C++. While I might agree that "encapsulation" is an OO concept, I don't think that the C++-style "public/private/protected" system is. You can do encapsulation without that. – Ken Jul 21, 2010 at 15:26

2 Reflection does provide a way to circumvent Java's Access Protection Modifiers, and therefore rapes encapsulation – Enriquev Jul 21, 2010 at 16:50

@Ken: Technically, encapsulation just requires combining the data and the functions that act upon it (now called methods) into a single entity (called a class). Having said that, being able to remove some of the contents of that entity from public view aids encapsulation so much that it's generally accepted as a basic requirement. – Steven Sudit Jul 21, 2010 at 17:22

1 rubi+python+php have private methods – Ben Jul 21, 2010 at 21:09

Steven: Granted. I'm just pointing out that the C++ approach isn't the only way to hide internal values. SICP (see section 3.1.1), for example, implements local state using environments (`let`). Scheme doesn't have `private` but that doesn't mean you can't have private local state. – Ken Jul 22, 2010 at 5:13

15

Yes, but it is not a problem.

Encapsulation is not about security or secrets, just about organizing things.

Reflection is not part of 'normal' programming. If you want to use it to break encapsulation, you accept the risks (versioning problems etc)

Reflection should only be used when there are no better (less invasive) ways to accomplish something.

Reflection is for system-level 'tooling' like persistence mapping and should be tucked away in well tested libraries. I would find any use of reflection in normal application code suspect.

I started with "it is not a problem". I meant: as long as you use reflection as intended. And careful.

Share Improve this answer Follow

edited Aug 6, 2013 at 21:06

answered Jul 21, 2010 at 15:21

Henk Holterman

273k ● 32 ● 348 ● 533

1 Encapsulation is about hiding away details of implementation so that you can maintain state in such a way as to avoid having it become invalid. Reflection, whether you consider it "normal", does violate this in a *potentially* problematic way, as it can bypass all checks on internal consistency. – Steven Sudit Jul 21, 2010 at 15:30

@Steven: Yes, but that is why you reserve Reflection for special problems. – Henk Holterman Jul 21, 2010 at 15:37

Hmm. Does deserialization qualify as a special problem? – Steven Sudit Jul 21, 2010 at 15:40

Interesting. I suspect that, by your standards, I do a great deal of "special" programming. – Steven Sudit Jul 21, 2010 at 16:13

What about compilation of dynamic languages (a la IronPython)? – RCIX Jul 26, 2010 at 1:48

10

It's like your house. Locks only keep out honest people, or people who aren't willing to pick your lock.

Data is data, if someone is determined enough, they can do anything with your code. Literally anything.

So yes, reflection will allow people to do things you don't want them to do with your code, for example access private fields and methods. However, the important thing is that people will not accidentally do this. If they're using reflection, they know they're doing something they probably aren't intended to do, just like no one accidentally picks the lock on your front door.

Share Improve this answer

Follow

answered Jul 21, 2010 at 15:09

corsiKa

82.5k ● 26 ● 159 ● 207

8

No, reflection doesn't break the idea of private methods. At least not per se. There is nothing that says that reflection can't obey access restrictions.

Badly designed reflection breaks the idea of private methods, but that doesn't have anything to do with reflection per se: *anything* which is badly designed can break the idea of private methods. In particular, a bad

design of private methods can also obviously break the idea of private methods.

What do I mean by *badly designed*? Well, as I said above, there is nothing stopping you from having a language in which reflection obeys access restrictions. The problem with this is that e.g. debuggers, profilers, coverage tools, IntelliSense, IDEs, tools in general *need* to be able to violate access restrictions. Since there is no way to present different different versions of reflection to different clients, most languages opt for tools over safety. (E is the counterexample, which has absolutely no reflective capabilities whatsoever, as a conscious design choice.)

But, who says that you cannot present different versions of reflection to different clients? Well, the problem is simply that in the classical implementation of reflection, all objects are reponsible for reflecting on themselves, and since there is only one of every object, there can be only version of reflection.

So, where does the idea of *bad design* come in? Well, note the word "responsible" in the above paragraph. Every object is responsible for reflecting on itself. Also, every object is responsible for whatever it is that it was written for in the first place. In other words: every object has at least *two* responsibilities. This violates one of the basic principles of object-oriented design: the Single Responsibility Principle.

The solution is rather simple: break up the object. The original object is simply responsible for whatever it was originally written for. And there is another object (called a *Mirror* because it is an object that reflects other objects) which is responsible for reflection. And now that the responsibility for reflection is broken out into a separate object, what's stopping us from having not one, but two, three, *many* Mirror Objects? One that respects access restrictions, one that only allows an object to reflect upon itself but not any other objects, one that only allows introspection (i.e. is read-only), one that only allows to reflect on read-only callsite information (i.e. for a profiler), one that gives full access to the entire system including violating access restrictions (for a debugger), one that only gives read-only access to the method names and signatures and respects access restrictions (for IntelliSense) and so on …

As a nice bonus, this means that Mirrors are essentially Capabilities (in the capability-security sense of the word) for reflection. IOW: Mirrors are the Holy Grail on the decade-long quest to reconcile security and runtime dynamic metaprogramming.

The concept of Mirrors was originally invented in Self from where it carried over into Animorphic Smalltalk/Strongtalk and then Newspeak. Interestingly, the Java Debugging Interface is based on Mirrors, so the designers of Java (or rather the JVM) clearly knew about them, but Java's reflection is broken.

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 • 1

answered Jul 21, 2010 at 17:55

Jörg W Mittag

369k • 79 • 453 • 661

5

Yes, reflection breaks this idea. Native languages also have some tricks to break OOP rules, for example, in C++ it is possible to change private class members using pointer tricks. However, by using these tricks, we get the code which can be incompatible with future class versions - this is the price we pay for breaking OOP rules.

Share Improve this answer

Follow

answered Jul 21, 2010 at 15:06

Alex F

43.3k • 42 • 150 • 217

Arguably, even if we didn't have reflection, we could use an `unsafe` block with a raw pointer to mess with the internals of a class. – Steven Sudit Jul 21, 2010 at 15:08

It may be worth noting that Code Access Security can be used to mitigate such issues. Also, reflection is permissions based by the CLR, so in a partial trust environment reflection might throw a security exception. – Josh Jul 21, 2010 at 15:12

@Henk: I missed the memo. When did CAS get deprecated? – Steven Sudit Jul 21, 2010 at 16:13

As of Fx 4 it is turned off by default. I'm not sure if the new claims system can control reflection. Here's a link: simple-

talk.com/dotnet/.net-framework/… – Henk Holterman Jul 21, 2010 at 16:22

Actually in C++ you can access private members by putting the following line in the header file `#define private public` . – JeremyP Jul 21, 2010 at 17:15

5

It does, as other already stated.

However, I remember that in Java there can be a security manager active, that could prevent you from accessing any private methods, even with reflection, if you don't have the rights to do so. If you run your local JVM, such a manager is usually not active.

Share Improve this answer

Follow

answered Jul 21, 2010 at 15:15

xor_eq

4,063 ● 1 ● 30 ● 34

4

Yes, Reflection *could* be used to violate encapsulation and even cause incorrect behavior. Keep in mind that the assembly needs to be trusted to perform reflection, so there are still some protections in place.

Share Improve this answer

Follow

answered Jul 21, 2010 at 15:07

Steven Sudit

19.6k ● 1 ● 52 ● 54

Yes it breaks the encapsulation, if you want it to. However, it can be put to good use - like writing unit tests

4

for private methods, or sometimes - as I have learned from my own experience - getting around bugs in third party APIs :)

Note that encapsulation != security. Encapsulation is an object oriented design concept and is only meant for improving the design. For security, there is SecurityManager in java.

Share Improve this answer

Follow

edited Jul 21, 2010 at 18:28

answered Jul 21, 2010 at 15:13

samitgaur

5,661 ● 2 ● 30 ● 33

3

Yes. Reflection breaks encapsulation principle. That's not only to get access to private members but rather expose whole structure of a class.

Share Improve this answer

Follow

edited Jul 21, 2010 at 15:13

answered Jul 21, 2010 at 15:06

Arseny

7,331 ● 4 ● 38 ● 52

I think this is a matter of opinion, but if you are using reflection to get around the encapsulation put in place by

2

a developer on a class, then you are defeating the purpose.

So, to answer your question, it breaks the idea of encapsulation (or information hiding), which simply states that private properties/methods are private so they cant be mucked with outside the class.

Share Improve this answer

Follow

answered Jul 21, 2010 at 15:04

hvgotcodes

120k ● 33 ● 207 ● 237

2

Reflection makes it possible for any CLR class to examine and manipulate properties and fields of other CLR classes, but not necessarily to do so sensibly. It's possible for a class to obscure the meaning of properties and fields or protect them against tampering by having them depend in non-obvious fashion upon each other, static fields, underlying OS info, etc.

For example, a class could keep in some variable an encrypted version of the OS handle for its main window. Using reflection, another class could see that variable, but without knowing the encryption method it could not identify the window to which it belonged or make the variable refer to another window.

I've seen classes that claim to act as "universal serializers"; they can be very useful if applied to something like a data-storage-container class which is missing a "serializable" attribute but is otherwise entirely

straightforward. They will produce gobbledygook if applied to any class whose creator has endeavored to obscure things.

Share Improve this answer

Follow

answered Jul 21, 2010 at 15:16

supercat

80.8k ● 9 ● 174 ● 220

2

Yes, it does break encapsulation. But there are many good reasons to use it in some situations.

For example:

I use MSCaptcha in some websites, but it renders a < div> around the < img > tag that messes with my HTML. Then i can use a standard < img> tag and use reflection to get the value of the captcha's image id to construct a URL.

The image id is a private Property but using reflection i can get that value.

Share Improve this answer

Follow

answered Jul 21, 2010 at 15:22

Carlos Muñoz

17.8k ● 8 ● 57 ● 81

you bring good example not to use reflaction, because in your case reflaction breaks the encapsulation.Only when she not breaks like unit test etc.. its proper to use it. – Ben Jul 21, 2010 at 21:14

1

access control through private/protected/package/public is not primarily meant for security.

it helps good guys to do the right thing, but doesn't prevent bad guys from doing wrong things.

generally we assume others are good guys, and we include their code into our application without though.

if you can't trust the guy of a library you are including, you are screwed.

Share Improve this answer

Follow

answered Jul 21, 2010 at 20:12

irreputable

45.4k ● 9 ● 68 ● 93