Struct like objects in Java

Asked 16 years, 3 months ago   Modified 5 years, 11 months ago   Viewed 361k times

199

Is it completely against the Java way to create struct like objects?

```
class SomeData1 {
    public int x;
    public int y;
}
```

I can see a class with accessors and mutators being more Java like.

```
class SomeData2 {
    int getX();
    void setX(int x);

    int getY();
    void setY(int y);

    private int x;
    private int y;
}
```

The class from the first example is notationally convenient.

```
// a function in a class
public int f(SomeData1 d) {
    return (3 * d.x) / d.y;
}
```

This is not as convenient.

```
// a function in a class
public int f(SomeData2 d) {
    return (3 * d.getX()) / d.getY();
}
```

java oop struct

Share
Improve this question
Follow

edited Nov 2, 2018 at 7:32
BSMP
4,786 ● 8 ● 35 ● 45

asked Aug 31, 2008 at 9:17
Chris de Vries
57.6k ● 6 ● 33 ● 27

9 Instead of public mutable fields, consider either public immutable fields or package-local mutable fields. Either would be better IMHO. – Peter Lawrey Apr 25, 2010 at 11:46

Remember that, while the getters and setters are ugly/verbose, that's kind of the heart of Java. It's a non-concise language. On the other side, though, you should NEVER type any of that, since that's what your IDE does for you. In a dynamic language you have to type less, but you have to type (generally, though IDEs may help). – Dan Rosenstark May 1, 2010 at 16:47

Ironically, while OO has it's strengths in terms of encapsulation, this there is a price to be paid CPU and storage-wise. The Garbage collector (almost completely) removes the necessity of worrying about when object references should be cleared. The current trend is going full-circle by employing C-like structs off-heap. This is perfect for caching-type solutions, inter process communications, faster memory-intensive operations, lower GC o/h and can even benefit from lower storage o/h for your data-sets. If you know what you're doing, You wouldn't be asking this question... so think again! – KRK Owner Apr 20, 2014 at 23:19

@user924272: Re "The current trend is going full-circle by employing C-like structs off-heap". Which you would do in Java how??? IMHO, this is an area where Java is showing its age... – ToolmakerSteve Jun 17, 2014 at 23:17

@ToolmakerSteve -I'm seeing a circle. I'm not the only one. Companies like Azul are hot on pause-less garbage collection. Java is old. True. Engineers who spot a weakness and do something about it, rather than moan? They deserve respect! +10 to Azul from me :-) – KRK Owner Jun 20, 2014 at 2:04

20 Answers

Sorted by: Highest score (default)

304

It appears that many Java people are not familiar with the Sun Java Coding Guidelines which say it is quite appropriate to use public instance variable when the class is essentially a "Struct", if Java supported "struct" (when there is no behavior).

People tend to think getters and setters are the Java way, as if they are at the heart of Java. This is not so. If you follow the Sun Java Coding Guidelines, using public instance variables in appropriate situations, you are actually writing better code than cluttering it with needless getters and setters.

Java Code Conventions from 1999 and still unchanged.

10.1 Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten-often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. ***In other words, if you***

would have used a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.

http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-137265.html#177

http://en.wikipedia.org/wiki/Plain_old_data_structure

http://docs.oracle.com/javase/1.3/docs/guide/collections/designfaq.html#28

Share
Improve this answer
Follow

edited Jul 5, 2017 at 6:20
ItamarG3
4,122 ● 6 ● 32 ● 44

answered Dec 19, 2011 at 18:29
developer.g
3,144 ● 2 ● 15 ● 7

93 +1 For actually having an authoritative source. Every other answer is people spinning their opinions like they're facts. – ArtOfWarfare Oct 3, 2012 at 12:20

1 There is a Java Beans specification which is an industry-standard way of accessing properties using get and set methods... see en.wikipedia.org/wiki/JavaBeans for an overview. – KRK Owner Apr 12, 2014 at 23:10

4 @user924272: How is that Java Beans spec relevant to this answer, which discusses when it is appropriate to use "public instance variables"? If the spec were a standard way of automagically turning instance variables into properties, ala C#, it might be relevant. But it isn't, right? It merely specifies the naming of the boilerplate getters and setters that would need to be created, to make such a mapping. – ToolmakerSteve Jun 17, 2014 at 22:42

@ToolmakerSteve. This is a java question. Also, the question eludes to a a common problem where a specification exists. From an old-school perspective, it's far easier to debug field mutations when there's a standard way to do it - set a breakpoint at a setter. This has probably been made obsolete with modern debuggers, yet I'm compelled to frown upon classes that directly "punch" objects... while this is okay for smaller applications, it is a real headache for larger apps and large organisations – KRK Owner Jun 20, 2014 at 1:53

1 Since java 16 (2021), some versions ago, there's the record class modifier now! docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/… also fully released – brunoais Mar 13, 2023 at 10:46

223

Use common sense really. If you have something like:

```
public class ScreenCoord2D{
    public int x;
    public int y;
}
```

Then there's little point in wrapping them up in getters and setters. You're never going to store an x, y coordinate in whole pixels any other way. Getters and setters will only slow you down.

On the other hand, with:

```
public class BankAccount{
    public int balance;
}
```

You might want to change the way a balance is calculated at some point in the future. This should really use getters and setters.

It's always preferable to know *why* you're applying good practice, so that you know when it's ok to bend the rules.

Share

Improve this answer

Follow

edited Aug 21, 2015 at 8:03
Ankush soni
1,449 ● 1 ● 15 ● 30

answered Aug 31, 2008 at 14:44
izb
51.7k ● 39 ● 118 ● 172

3 I go along with this answer and say further that you can create a class with public fields so long as the fields are **bold**independent from each other. i.e. one field doesn't depend on another. This can be quite useful in many cases, for multiple return values from a function, or for polar co-odinates. { angle, length } that go together but do not depend on each other in any intrinsic way. – Spacen Jasset Jun 5, 2012 at 20:05

@SpacenJasset: FYI, I don't see how your examples (multiple return values; polar coordinates) have any bearing on whether to use public fields vs getter/setters. In the case of multiple return values, it might even be counter-productive, because arguably the caller should only be GETTING the returned values, which argues in favor of public getters and private setters (immutable). This might also be true for returning polar coordinates from an (x, y) object - consider ACCUMULATIVE math errors, as changes to individual components of polar coord are converted back to (x, y). – ToolmakerSteve Jun 17, 2014 at 22:29

@SpacenJasset: But I agree with your principle. – ToolmakerSteve Jun 17, 2014 at 22:35

1 You have a valid point, but I feel like the pixels thing is a bad example since making sure the pixel is within the window (just an example) *is* something that someone might do, and besides, letting someone set the pixel to `(-5, -5)` might not be a good idea. :-) – Horsey Mar 29, 2018 at 11:11

@Horsey Being within the window (which one?) isn't the responsibility of the Coordinate and why might be (-5, -5) not a good idea? Depends on where the origin point is. Center maybe? – BlackJack Dec 5, 2022 at 16:46

62

This is a commonly discussed topic. The drawback of creating public fields in objects is that you have no control over the values that are set to it. In group projects where there are many programmers using the same code, it's important to avoid side effects. Besides, sometimes it's better to return a copy of field's object or transform it somehow etc. You can mock such methods in your tests. If you create a new class you might not see all possible actions. It's like defensive programming - someday getters and setters may be helpful, and it doesn't cost a lot to create/use them. So they are sometimes useful.

In practice, most fields have simple getters and setters. A possible solution would look like this:

```
public property String foo;
a->Foo = b->Foo;
```

Update: It's highly unlikely that property support will be added in Java 7 or perhaps ever. Other JVM languages like Groovy, Scala, etc do support this feature now. - Alex Miller

Share

Improve this answer

Follow

edited Jul 10, 2016 at 16:58

Nathan majicvr.com
1,021 ● 2 ● 15 ● 33

answered Aug 31, 2008 at 9:50

Bartosz Bierkowski
2,792 ● 1 ● 19 ● 18

28 That's too bad, I like C#-style properties (which sounds like what you're talking about) – Jon Onstott Apr 25, 2011 at 22:26

2 So use overloading... private int _x; public void x(int value) { _x = value; } public int x() { return _x; } – Gordon Jan 7, 2012 at 20:44

12 I prefer being able to use `=`, which in my opinion makes the code cleaner. – Svish Jan 17, 2012 at 12:40

6 @T-Bull: Just because you CAN have two `x` s that are two different things, doesn't make it a good idea. IMHO, that is a poor suggestion, as it could lead to confusion by human readers. Basic principle: don't make a reader do a double-take; make it obvious what you are saying -- Use different names for different entities. Even if the difference is merely to prepend an underscore. Don't rely on surrounding punctuation to differentiate the entities. – ToolmakerSteve Jun 17, 2014 at 22:14

3 @ToolmakerSteve: That "could lead to confusion" is the most common and still the weakest argument when it comes to defending coding dogma (as opposed to coding style). There's always someone who COULD be confused by the simplest things. You'll always find somebody complaining that he made a mistake after staying awake and coding for half a week or some such and then blaming the misleading coding style. I do not let that count. That style is valid, obvious and keeps the namespace clean. Also, there are not different entities here, there's /one/ entity and some boilerplate code around it. – T-Bull Jun 25, 2014 at 8:47

54

To address mutability concerns you can declare x and y as final. For example:

```
class Data {
  public final int x;
  public final int y;
  public Data( int x, int y){
    this.x = x;
    this.y = y;
    }
}
```

Calling code that attempts to write to these fields will get a compile time error of "field x is declared final; cannot be assigned".

The client code can then have the 'short-hand' convenience you described in your post

```
public class DataTest {
    public DataTest() {
        Data data1 = new Data(1, 5);
        Data data2 = new Data(2, 4);
        System.out.println(f(data1));
        System.out.println(f(data2));
    }

    public int f(Data d) {
        return (3 * d.x) / d.y;
    }

    public static void main(String[] args) {
        DataTest dataTest = new DataTest();
    }
}
```

Share

Improve this answer

Follow

edited Oct 15, 2008 at 3:58

answered Sep 2, 2008 at 1:51

Brian
13.6k ● 11 ● 59 ● 83

3 Thanks - an answer that is useful and concise. Shows how to get the benefit of field syntax, when don't want mutability. – ToolmakerSteve Jun 17, 2014 at 22:44

I attempted to use final instances of a final class with final fields as struct "instances", but in a `case` expression referring such instance field, I got `Case expressions must be constant expressions`. what is the way around this? what is the idiomatic concept here? – n611x007 Jul 8, 2014 at 15:28

1 final fields are still references to objects, which are not constants, as they will be initialized when the class is first used. The compiler cannot know the "value" of the object references. Constants must be known when compiling. – Johan Tidén Oct 10, 2014 at 13:11

1 **+1** Really important answer. The usefulness of immutable classes cannot be underestimated in my opinion. Their "fire-and-forget" semantics make reasoning about code often simpler, especially in multithreaded environments, where they can be shared arbitrarily between threads, without synchronization. – TheOperator Mar 7, 2016 at 15:23

Do not use `public` fields

12

Don't use `public` fields when you really want to wrap the internal behavior of a class. Take `java.io.BufferedReader` for example. It has the following field:

```
private boolean skipLF = false; // If the next character is a line feed, skip
it
```

`skipLF` is read and written in all read methods. What if an external class running in a separate thread maliciously modified the state of `skipLF` in the middle of a read? `BufferedReader` will definitely go haywire.

Do use `public` fields

Take this `Point` class for example:

```
class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return this.x;
    }

    public double getY() {
        return this.y;
    }

    public void setX(double x) {
        this.x = x;
    }

    public void setY(double y) {
        this.y = y;
    }
}
```

This would make calculating the distance between two points very painful to write.

```
Point a = new Point(5.0, 4.0);
Point b = new Point(4.0, 9.0);
double distance = Math.sqrt(Math.pow(b.getX() - a.getX(), 2) +
Math.pow(b.getY() - a.getY(), 2));
```

The class does not have any behavior other than plain getters and setters. It is acceptable to use public fields when the class represents just a data structure, and does not have, *and* never will have behavior (thin getters and setters is *not* considered behavior here). It can be written better this way:

```
class Point {
    public double x;
    public double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

Point a = new Point(5.0, 4.0);
Point b = new Point(4.0, 9.0);
double distance = Math.sqrt(Math.pow(b.x - a.x, 2) + Math.pow(b.y - a.y, 2));
```

Clean!

But remember: Not only your class must be absent of behavior, but it should also have *no* reason to have behavior in the future as well.

(This is exactly what this answer describes. To quote "Code Conventions for the Java Programming Language: 10. Programming Practices":

> One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a `struct` instead of a class (if Java supported `struct`), then it's appropriate to make the class's instance variables public.

So the official documentation also accepts this practice.)

Also, if you're extra sure that members of above `Point` class should be immutable, then you could add `final` keyword to enforce it:

```
public final double x;
public final double y;
```

Share
Improve this answer
Follow

edited Jan 11, 2019 at 4:31

answered Sep 25, 2014 at 6:48
sampathsris
22.3k ● 12 ● 72 ● 101

Since java 16 (2021), some versions ago, there's the record class modifier now! docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/… also fully released – brunoais Mar 13, 2023 at 10:46

8

By the way, the structure you're giving as an example already exist in the Java base class library as `java.awt.Point`. It has x and y as public fields, check it out for yourself.

If you know what you're doing, and others in your team know about it, then it is okay to have public fields. But you shouldn't rely on it because they can cause headaches as in bugs related to developers using objects as if they were stack allocated structs (java objects are always sent to methods as references and not as copies).

Share Improve this answer Follow

answered Aug 31, 2008 at 14:24

Spoike
122k ● 45 ● 142 ● 158

+1 Good mention of an issue -- a way in which the result is still NOT like a C struct. However, the issue you raise about java objects always being by reference, is not improved by creating a setter instead of having a public writable field (which is the essence of OP's question -- which representation to use). Rather, it is an argument in favor of doing NEITHER. It is an argument for IMMUTABILITY. Which can either be done as a `public final` field, as in Brian's answer, or by having a public getter, but no public setter. That is, whether one uses fields or accessors is immaterial. – ToolmakerSteve Jun 17, 2014 at 23:09

8

Re: aku, izb, John Topley...

Watch out for mutability issues...

It may seem sensible to omit getters/setters. It actually may be ok in some cases. The real problem with the proposed pattern shown here is mutability.

The problem is once you pass an object reference out containing non-final, public fields. Anything else with that reference is free to modify those fields. You no longer have any control over the state of that object. (Think what would happen if Strings were mutable.)

It gets bad when that object is an important part of the internal state of another, you've just exposed internal implementation. To prevent this, a copy of the object must be returned instead. This works, but can cause massive GC pressure from tons of single-use copies created.

If you have public fields, consider making the class read-only. Add the fields as parameters to the constructor, and mark the fields final. Otherwise make sure you're not exposing internal state, and if you need to construct new instances for a return value, make sure it won't be called excessively.

See: "Effective Java" by Joshua Bloch -- Item #13: Favor Immutability.

PS: Also keep in mind, all JVMs these days will optimize away the getMethod if possible, resulting in just a single field-read instruction.

Share Improve this answer Follow

answered Aug 31, 2008 at 16:27
Mark Renouf
31k ● 19 ● 96 ● 125

12 How does a getter/setter resolve this issue. You still have a reference, you don't have any synchronization with the operations. Getters/setters don't provide protection in and of themselves. – he_the_great Nov 23, 2008 at 18:39

1 Getters and setters can provide synchronization if required. You're also expecting getters and setter to do much more than they were specified to do. Regardless of this, the problem of synchronisation still remamins. – KRK Owner Apr 12, 2014 at 23:13

7

I have tried this in a few projects, on the theory that getters and setters clutter up the code with semantically meaningless cruft, and that other languages seem to do just fine with convention-based data-hiding or partitioning of responsibilities (e.g. python).

As others have noted above, there are 2 problems that you run into, and they're not really fixable:

- Just about any automated tool in the java world relies on the getter/setter convention. Ditto for, as noted by others, jsp tags, spring configuration, eclipse tools, etc. etc... Fighting against what your tools expect to see is a recipe for long sessions trolling through google trying to find that non-standard way of initiating spring beans. Really not worth the trouble.
- Once you have your elegantly coded application with hundreds of public variables you will likely find at least one situation where they're insufficient- where you absolutely need immutability, or you need to trigger some event when the variable gets set, or you want to throw an exception on a variable change because it sets an object state to something unpleasant. You're then stuck with the unenviable choices between cluttering up your code with some special method everywhere the variable is directly referenced, having some special access form for 3 out of the 1000 variables in your application.

And this is in the best case scenario of working entirely in a self-contained private project. Once you export the whole thing to a publicly accessible library these problems will become even larger.

Java is very verbose, and this is a tempting thing to do. Don't do it.

Share Improve this answer Follow

answered Dec 1, 2008 at 20:50

Steve B.

57.2k ● 12 ● 97 ● 134

1 Excellent discussion of the problems of going down the road of public fields. A definite shortcoming of Java, which annoys me when I have to switch back to Java from C# (which learned from Java's inconvenience here). – ToolmakerSteve Jun 17, 2014 at 23:02

4

If the Java way is the OO way, then yes, creating a class with public fields breaks the principles around information hiding which say that an object should manage its own internal state. (So as I'm not just spouting jargon at you, a benefit of information hiding is that the internal workings of a class are hidden behind an interface - say you wanted to change the mechanism by which your struct class saved one of its fields, you'll probably need to go back and change any classes that use the class...)

You also can't take advantage of the support for JavaBean naming compliant classes, which will hurt if you decide to, say, use the class in a JavaServer Page which is written using Expression Language.

The JavaWorld article [Why Getter and Setter Methods are Evil] article also might be of interest to you in thinking about when not to implement accessor and mutator methods.

If you're writing a small solution and want to minimise the amount of code involved, the Java way may not be the right way - I guess it always depends on you and the problem you're trying to solve.

Share
Improve this answer
Follow

edited Aug 31, 2008 at 9:39

answered Aug 31, 2008 at 9:33
brabster
43.5k ● 29 ● 149 ● 189

1 +1 for link to article "Why Getter and Setter Methods are Evil". However, your answer would have been clearer if you pointed out that BOTH public fields AND public getter/setters are EQUALLY NOT the Java way: As explained in that article -- when possible, don't do either. Instead, provide clients with methods specific to what they need to do, rather than mirroring the internal representation of the instance. HOWEVER, this doesn't really address the question being asked (which to use, for simple "struct" case), which is better answered by developer.g, izb, and Brian. – ToolmakerSteve Jun 17, 2014 at 22:56

3

There is nothing wrong with that type of code, provided that the author ***knows*** they are structs (or data shuttles) instead of objects. Lots of Java developers can't tell the difference between a well-formed object (not just a subclass of java.lang.Object, but a *true* object in a specific domain) and a pineapple. Ergo,they end up writing structs when they need objects and viceversa.

Share Improve this answer Follow

answered Apr 15, 2010 at 2:59
luis.espinal
10.5k ● 6 ● 43 ● 56

the pineapple make me laugh :) – Guillaume May 14, 2012 at 15:32

However, this answer says nothing about what that difference is. Ragging on unskilled developers doesn't help those developers know when to make a struct, and when to make a class. – ToolmakerSteve Jun 17, 2014 at 23:04

It doesn't say anything about the difference because they author is aware of the difference (he knows what a struct-like entity is). The author is asking if using structs are appropriate in a OO language, and I say yes (depending on the problem domain.) If the question was about what makes an object a real domain object, then you would have a leg to stand in your argument. – luis.espinal Jun 26, 2014 at 13:21

Furthermore, the only type of unskilled developer that should not know the difference would be the one that is still in school. This is extremely fundamental, like knowing the difference between a linked list and a hash table. If a person graduates with a 4-year software degree without knowing what a true object is, then either that person is not cut for this career, or he/she should go back to school and demand a refund. I'm serious. – luis.espinal Jun 26, 2014 at 13:23

But to satisfy your complain I'll provide an answer (which has little to anything to do with the original question and which deserves its own thread). Objects have behavior and encapsulate state. Structs do not. Objects are state machines. Structs are just for aggregating data. If people want a more elaborate answer, they are free to create a new question where we can elaborate on it to our hearts' content. – luis.espinal Jun 26, 2014 at 13:25

3

A very-very old question, but let me make another short contribution. Java 8 introduced lambda expressions and method references. Lambda expressions can be simple method references and not declare a "true" body. But you cannot "convert" a field into a method reference. Thus

```
stream.mapToInt(SomeData1::x)
```

isn't legal, but

```
stream.mapToInt(SomeData2::getX)
```

is.

Share Improve this answer Follow

answered Jul 10, 2014 at 15:03

Lyubomyr Shaydariv
21.1k ● 12 ● 69 ● 107

1 In this case you can use `data -> data.x`, which is still reasonable – James Kraus Oct 8, 2018 at 21:22

The problem with using public field access is the same problem as using new instead of a factory method - if you change your mind later, all existing callers are broken. So,

2

from an API evolution point of view, it's usually a good idea to bite the bullet and use getters/setters.

One place where I go the other way is when you strongly control access to the class, for example in an inner static class used as an internal data structure. In this case, it might be much clearer to use field access.

By the way, on e-bartek's assertion, it is highly unlikely IMO that property support will be added in Java 7.

Share Improve this answer Follow

answered Sep 15, 2008 at 16:21
Alex Miller
70.1k ● 25 ● 124 ● 168

1 This is true, but if you're using Java it's not a concern, since you can refactor an entire code-base in one click (Eclipse, Netbeans, probably VIM and Emacs too). If you've opted for one of its dynamic friends, like Groovy, even a simple encapsulation could have you fixing for hours or days. Luckily you'd have your test cases which would inform you... how much more code you have to fix. – Dan Rosenstark May 1, 2010 at 16:28

2 You assume of course that all of the users are *in your code base* but of course, that's often not true. Often it's not even possible for various non-technical reasons to change code that might be in your own code base. – Alex Miller May 5, 2010 at 1:27

2

I frequently use this pattern when building private inner classes to simplify my code, but I would not recommend exposing such objects in a public API. In general, the more frequently you can make objects in your public API immutable the better, and it is not possible to construct your 'struct-like' object in an immutable fashion.

As an aside, even if I were writing this object as a private inner class I would still provide a constructor to simplify the code to initialize the object. Having to have 3 lines of code to get a usable object when one will do is just messy.

Share Improve this answer Follow

answered Oct 15, 2008 at 4:28
Kris Nuttycombe
4,580 ● 1 ● 28 ● 30

1

I don't see the harm if you know that it's always going to be a simple struct and that you're never going to want to attach behaviour to it.

Share Improve this answer Follow

answered Aug 31, 2008 at 9:20
John Topley
115k ● 47 ● 199 ● 240

1

This is a question on Object Oriented Design, not Java the language. It's generally good practice to hide data types within the class and expose only the methods that are part of the class API. If you expose internal data types, you can never change them in the future. If you hide them, your only obligation to the user is the method's return and argument types.

Share Improve this answer Follow

answered Aug 31, 2008 at 10:13
Jonathan
7,401 ● 5 ● 32 ● 36

1

Sometime I use such class, when I need to return multiple values from a method. Of course, such object is short lived and with very limited visibility, so it should be OK.

Share Improve this answer Follow

answered Nov 23, 2008 at 18:43
PhiLho
41.1k ● 6 ● 99 ● 136

0

You can make a simple class with public fields and no methods in Java, but it is still a class and is still handled syntactically and in terms of memory allocation just like a class. There is no way to genuinely reproduce structs in Java.

Share Improve this answer Follow

answered Oct 15, 2008 at 4:35
cblades
436 ● 3 ● 12

0

As with most things, there's the general rule and then there are specific circumstances. If you are doing a closed, captured application so that you know how a given object is going to be used, then you can exercise more freedom to favor visibility and/or efficiency. If you're developing a class which is going to be used publicly by others beyond your control, then lean towards the getter/setter model. As with all things, just use common sense. It's often ok to do an initial round with publics and then change them to getter/setters later.

Share Improve this answer Follow

answered Apr 25, 2010 at 4:13

Evvo
499 ● 5 ● 8

0

Aspect-oriented programming lets you trap assignments or fetches and attach intercepting logic to them, which I propose is the right way to solve the problem. (The issue of whether they should be public or protected or package-protected is orthogonal.)

Thus you start out with unintercepted fields with the right access qualifier. As your program requirements grow you attach logic to perhaps validate, make a copy of the object being returned, etc.

The getter/setter philosophy imposes costs on a large number of simple cases where they are not needed.

Whether aspect-style is cleaner or not is somewhat qualitative. I would find it easy to see just the variables in a class and view the logic separately. In fact, the raison d'etre for Apect-oriented programming is that many concerns are cross-cutting and compartmentalizing them in the class body itself is not ideal (logging being an example -- if you want to log all gets Java wants you to write a whole bunch of getters and keeping them in sync but AspectJ allows you a one-liner).

The issue of IDE is a red-herring. It is not so much the typing as it is the reading and visual pollution that arises from get/sets.

Annotations seem similar to aspect-oriented programming at first sight however they require you to exhaustively enumerate pointcuts by attaching annotations, as opposed to a concise wild-card-like pointcut specification in AspectJ.

I hope awareness of AspectJ prevents people from prematurely settling on dynamic languages.

Share Improve this answer Follow

answered Dec 19, 2011 at 19:02
necromancer
24.6k ● 22 ● 70 ● 117

AOJ has downside of opaque processing and putting weight into a precompiler stage. It can be very difficult for someone new to a project to understand what the code is doing and also how to make changes. I speak from experience on this from a 5K SLOC project that felt like several X larger due to the complexity of determining what was actually happening.
– WestCoastProjects Feb 6, 2021 at 14:37

Here I create a program to input Name and Age of 5 different persons and perform a selection sort (age wise). I used an class which act as a structure (like C programming

0

language) and a main class to perform the complete operation. Hereunder I'm furnishing the code...

```java
import java.io.*;

class NameList {
    String name;
    int age;
}

class StructNameAge {
    public static void main(String [] args) throws IOException {

        NameList nl[]=new NameList[5]; // Create new radix of the structure
NameList into 'nl' object
        NameList temp=new NameList(); // Create a temporary object of the
structure

        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

        /* Enter data into each radix of 'nl' object */

        for(int i=0; i<5; i++) {
            nl[i]=new NameList(); // Assign the structure into each radix

            System.out.print("Name: ");
            nl[i].name=br.readLine();

            System.out.print("Age: ");
            nl[i].age=Integer.parseInt(br.readLine());

            System.out.println();
        }

        /* Perform the sort (Selection Sort Method) */

        for(int i=0; i<4; i++) {
            for(int j=i+1; j<5; j++) {
                if(nl[i].age>nl[j].age) {
                    temp=nl[i];
                    nl[i]=nl[j];
                    nl[j]=temp;
                }
            }
        }

        /* Print each radix stored in 'nl' object */

        for(int i=0; i<5; i++)
            System.out.println(nl[i].name+" ("+nl[i].age+")");
    }
}
```

The above code is Error Free and Tested... Just copy and paste it into your IDE and ... You know and what??? :)

Share   edited Feb 19, 2013 at 9:36   answered Feb 19, 2013 at 5:50

Improve this answer

Follow

Avik Kumar Goswami
15 ● 3