Class with single method -- best approach?

Asked 16 years, 2 months ago Modified 4 years, 7 months ago Viewed 100k times

189

Say I have a class that's meant to perform a single function. After performing the function, it can be destroyed. Is there any reason to prefer one of these approaches?

```
// Initialize arguments in constructor
MyClass myObject = new MyClass(arg1, arg2, arg3);
myObject.myMethod();

// Pass arguments to method
MyClass myObject = new MyClass();
myObject.myMethod(arg1, arg2, arg3);

// Pass arguments to static method
MyClass.myMethod(arg1, arg2, arg3);
```

I was being intentionally vague about the details, to try to get guidelines for different situations. But I didn't really have in mind simple library functions like Math.random(). I'm thinking more of classes that perform some specific, complex task, but only require one (public) method to do it.

c# java oop static-methods

Share
Improve this question
Follow

edited Mar 9, 2015 at 5:18
Robert Harvey
181k ● 48 ● 346 ● 512

asked Oct 15, 2008 at 17:41
JW.
51.6k ● 39 ● 121 ● 150

15 Answers

Sorted by: Highest score (default)

276

I used to love utility classes filled up with static methods. They made a great consolidation of helper methods that would otherwise lie around causing redundancy and maintenance hell. They're very easy to use, no instantiation, no disposal, just fire'n'forget. I guess this was my first unwitting attempt at creating a service oriented architecture - lots of stateless services that just did their job and nothing else. As a system grows however, dragons be coming.

Polymorphism

Say we have the method UtilityClass.SomeMethod that happily buzzes along. Suddenly we need to change the functionality slightly. Most of the functionality is the same, but we have to change a couple of parts nonetheless. Had it not been a static

method, we could make a derivate class and change the method contents as needed. As it's a static method, we can't. Sure, if we just need to add functionality either before or after the old method, we can create a new class and call the old one inside of it - but that's just gross.

Interface woes
Static methods cannot be defined through interfaces for logic reasons. And since we can't override static methods, static classes are useless when we need to pass them around by their interface. This renders us unable to use static classes as part of a strategy pattern. We might patch some issues up by passing delegates instead of interfaces.

Testing
This basically goes hand in hand with the interface woes mentioned above. As our ability of interchanging implementations is very limited, we'll also have trouble replacing production code with test code. Again, we can wrap them up but it'll require us to change large parts of our code just to be able to accept wrappers instead of the actual objects.

Fosters blobs
As static methods are usually used as utility methods and utility methods usually will have different purposes, we'll quickly end up with a large class filled up with non-coherent functionality - ideally, each class should have a single purpose within the system. I'd much rather have a five times the classes as long as their purposes are well defined.

Parameter creep
To begin with, that little cute and innocent static method might take a single parameter. As functionality grows, a couple of new parameters are added. Soon further parameters are added that are optional, so we create overloads of the method (or just add default values, in languages that support them). Before long, we have a method that takes 10 parameters. Only the first three are really required, parameters 4-7 are optional. But if parameter 6 is specified, 7-9 are required to be filled in as well... Had we created a class with the single purpose of doing what this static method did, we could solve this by taking in the required parameters in the constructor, and allowing the user to set optional values through properties, or methods to set multiple interdependent values at the same time. Also, if a method has grown to this amount of complexity, it most likely needs to be in its own class anyways.

Demanding consumers to create an instance of classes for no reason
One of the most common arguments is, why demand that consumers of our class create an instance for invoking this single method, while having no use for the instance afterwards? Creating an instance of a class is a very very cheap operation in most languages, so speed is not an issue. Adding an extra line of code to the

consumer is a low cost for laying the foundation of a much more maintainable solution in the future. And finally, if you want to avoid creating instances, simply create a singleton wrapper of your class that allows for easy reuse - although this does make the requirement that your class is stateless. If it's not stateless, you can still create static wrapper methods that handle everything, while still giving you all the benefits in the long run. Finally, you could also make a class that hides the instantiation as if it was a singleton: MyWrapper.Instance is a property that just returns new MyClass();

Only a Sith deals in absolutes

Of course, there are exceptions to my dislike of static methods. True utility classes that do not pose any risk to bloat are excellent cases for static methods - System.Convert as an example. If your project is a one-off with no requirements for future maintenance, the overall architecture really isn't very important - static or non static, doesn't really matter - development speed does, however.

Standards, standards, standards!

Using instance methods does not inhibit you from also using static methods, and vice versa. As long as there's reasoning behind the differentiation and it's standardised. There's nothing worse than looking over a business layer sprawling with different implementation methods.

Share

Improve this answer

Follow

edited Feb 23, 2020 at 17:09

Callum Watkins

2,981 ● 4 ● 32 ● 53

answered Oct 15, 2008 at 21:03

Mark S. Rasmussen

35.4k ● 4 ● 42 ● 58

11 "As a system grows..." you refactor? – Rodney Gitzel Feb 21, 2014 at 17:51

1 @Mark S. Rasmussen Great post. Would you pass arguments to method or initialize arguments in constructor? – user3667089 Dec 15, 2015 at 3:11

3 @user3667089 General arguments that are needed for the class to exist logically, those I would pass in the constructor. These would typically be used in most/all methods. Method specific arguments I'd pass in that specific method. – Mark S. Rasmussen Dec 16, 2015 at 14:36

1 @MarkS.Rasmussen Just to clarify, in the case of a class with a single method, you will pass it in as a constructor because it is used for all (that single one) method? – user3667089 Dec 16, 2015 at 16:46

3 Most of these reasons have to do with bad code management, not from using static methods. In F# and other functional languages we use what are basically static methods (functions with no instance-state) everywhere and don't have these problems. That said, F# provides a better paradigm for using functions (functions are first class, functions can be curried and partially applied) which makes them more feasible than in C#. A bigger reason to use classes is because that's what C# is built for. All of the Dependency Injection constructs in .NET Core revolve around instance classes with deps. – Justin J Stark Jul 30, 2019 at 20:55

95

I prefer the static way. Since the Class is not representing an object it doesn't make sense to make an instance of it.

Classes that only exist for their methods should be left static.

Share
Improve this answer
Follow

edited Nov 15, 2010 at 16:30

answered Oct 15, 2008 at 17:44

jjnguy
139k ● 53 ● 297 ● 326

21 -1 "Classes that only exist for their methods should be left static." – Rookian Aug 13, 2011 at 16:00

8 @Rookian why do you disagree with that? – jjnguy Aug 14, 2011 at 15:27

6 an example would be the repository pattern. The class contains only methods. Following your approach does not allow to use interfaces. => increase coupling – Rookian Aug 14, 2011 at 16:23

2 @Rook, in general people shouldn't be creating classes that are used just for methods. In the example you gave static methods aren't a good idea. But for simple utility methods the static way is best. – jjnguy Aug 14, 2011 at 20:52

11 Absolutely correct :) With your conditon "for simple utility methods" I agree with you completely, but you did make a general rule in your answer which is wrong imo. – Rookian Aug 15, 2011 at 9:14

19

If there is no reason to have an instance of the class created in order to execute the function then use the static implementation. Why make the consumers of this class create an instance when one is not needed.

Share Improve this answer Follow

answered Oct 15, 2008 at 17:46

Ray Jezek
6,665 ● 1 ● 19 ● 7

17

If you don't need to save the *state* of the object, then there's no need to instantiate it in the first place. I'd go with the single static method that you pass parameters to.

I'd also warn against a giant Utils class that has dozens of unrelated static methods. This can get disorganized and unwieldy in a hurry. It's better to have many classes, each with few, related methods.

Share Improve this answer Follow

answered Oct 15, 2008 at 18:24

Bill the Lizard
405k ● 211 ● 572 ● 889

6

I would say the Static Method format would be the better option. And I would make the class static as well, that way you wouldn't have to worry about accidentally creating an instance of the class.

Share Improve this answer Follow

answered Oct 15, 2008 at 17:46

Nathen Silver
439 ● 4 ● 7

5

I really don't know what the situation is here, but I would look at putting it as a method in one of the classes that arg1,arg2 or arg3 belong to -- If you can semantically say that one of those classes would own the method.

Share Improve this answer Follow

answered Oct 15, 2008 at 17:46

Chris Cudmore
30.1k ● 12 ● 59 ● 95

5

I'd suggest that its hard to answer based on the information provided.

My gut is that if you are just going to have one method, and that you are going to throw the class away immediately, then make it a static class that takes all the parameters.

Of course, its hard to tell exactly why you need to create a single class just for this one method. Is it the typical "Utilities class" situation as most are assuming? Or are you implementing some sort of rule class, of which there might be more in the future.

For instance, have that class be plugable. Then you'd want to create an Interface for your one method, and then you'd want to have all the parameters passed into the interface, rather than into the constructor, but you wouldn't want it to be static.

Share Improve this answer Follow

answered Oct 15, 2008 at 21:12

Nick
5,935 ● 1 ● 30 ● 38

Can your class be made static?

3

If so, then I'd make it a 'Utilities' class that I would put all my one-function classes in.

Share Improve this answer Follow

answered Oct 15, 2008 at 17:45

Robert Deml

12.5k ● 21 ● 69 ● 92

2 I somewhat disagree. In projects I'm involved with, your Utilities class could have hundreds of unrelated methods in it. – Paul Tomblin Oct 15, 2008 at 17:46

3 @Paul: Generally agreed. I hate to see classes with dozens (or yes, hundreds) of totally unrelated methods. If one must take this approach, at least split these into smaller, related sets of utilities (EG, FooUtilities, BarUtilities, etc). – John Rudy Oct 15, 2008 at 18:01

9 one class- one resposibility. – Andreas Petersson Oct 15, 2008 at 18:45

3

If this method is stateless and you don't need to pass it around, then it makes the most sense to define it as static. If you DO need to pass the method around, you might consider using a delegate rather than one of your other proposed approaches.

Share Improve this answer Follow

answered Oct 15, 2008 at 18:58

Joel Wietelmann

1,457 ● 2 ● 9 ● 16

3

For simple applications and `internal` helpers, I would use a static method. For applications with components, I'm loving the Managed Extensibility Framework. Here's an excerpt from a document I'm writing to describe the patterns you'll find across my APIs.

- Services
  - Defined by an `I[ServiceName]Service` interface.
  - Exported and imported by the interface type.
  - The single implementation is provided by the host application and consumed internally and/or by extensions.
  - The methods on the service interface are thread-safe.

As a contrived example:

```
public interface ISettingsService
{
    string ReadSetting(string name);
```

```
    void WriteSetting(string name, string value);
}

[Export]
public class ObjectRequiringSettings
{
    [Import]
    private ISettingsService SettingsService
    {
        get;
        set;
    }

    private void Foo()
    {
        if (SettingsService.ReadSetting("PerformFooAction") == bool.TrueString)
        {
            // whatever
        }
    }
}
```

Share Improve this answer Follow

answered Jan 15, 2010 at 18:05

Sam Harwell

99.7k ● 22 ● 214 ● 282

2

I would just do everything in the constructor. like so:

```
new MyClass(arg1, arg2, arg3);// the constructor does everything.
```

or

```
MyClass my_object(arg1, arg2, arg3);
```

Share

Improve this answer

Follow

edited May 6, 2014 at 10:19

pravprab

2,293 ● 3 ● 28 ● 43

answered Oct 15, 2008 at 18:28

yigal

What if you need to return anything besides an object of type MyClass? – Bill the Lizard Oct 15, 2008 at 18:37

14 I consider it bad practice to put execution logic in a constructor. Good development is about semantics. Semantically, a constructor exists to construct an object, not to perform other tasks. – mstrobl Oct 15, 2008 at 18:41

bill, if you need return value, pass reference to the ret vvalue: MyClass myObject(arg1, arg2, arg3,retvalue); mstrobl,if the object is not needed, why create it? and this trick can actualy help you in some instances. – yigal Oct 15, 2008 at 18:51

If an object has no state, i.e. no vars, then instancing it will *not* produce any allocation - neither on heap nor stack. You can think of objects in C++ as just C function calls with a hidden first parameter pointing to a struct. – mstrobl Oct 15, 2008 at 19:01

mstrobl, it like a c function on steroids because you can define private functions that the ctor can use. you can also use the class member variables to help pass data to the private functions. – yigal Oct 15, 2008 at 19:06

0

One more important issue to consider is whether the system would be running at a multithreaded environment, and whether it would be thread-safe to have a static method or variables...

You should pay attention to the system state.

Share Improve this answer Follow

answered Jul 27, 2012 at 7:49

NZal

879 ● 3 ● 11 ● 19

0

You might be able to avoid the situation all together. Try to refactor so that you get `arg1.myMethod1(arg2, arg3)`. Swap arg1 with either arg2 or arg3 if it makes more sense.

If you do not have control over the class of arg1, then decorate it :

```
class Arg1Decorator
    private final T1 arg1;
    public Arg1Decorator(T1 arg1) {
        this.arg1 = arg1;
    }
    public T myMethod(T2 arg2, T3 arg3) {
        ...
    }
 }

 arg1d = new Arg1Decorator(arg1)
 arg1d.myMethod(arg2, arg3)
```

The reasoning is that, in OOP, data and methods processing that data belong together. Plus you get all the advantages that Mark mentionned.

Share

Improve this answer

Follow

edited Jan 31, 2013 at 14:36

answered Jan 31, 2013 at 11:09

Hugo Wood

2,260 ● 17 ● 19

0

i think, if properties of your class or the instance of class will not be used in constructors or in your methods, methods are not suggested to be designed like 'static' pattern. static method should be always thinked in 'help' way.

Share Improve this answer Follow

answered Feb 10, 2017 at 9:39

user7545070

1

0

Depending on whether you want to only do something or do and return something you could do this:

```
public abstract class DoSomethingClass<T>
{
    protected abstract void doSomething(T arg1, T arg2, T arg3);
}

public abstract class ReturnSomethingClass<T, V>
{
    public T value;
    protected abstract void returnSomething(V arg1, V arg2, V arg3);
}

public class DoSomethingInt extends DoSomethingClass<Integer>
{
    public DoSomethingInt(int arg1, int arg2, int arg3)
    {
        doSomething(arg1, arg2, arg3);
    }

    @Override
    protected void doSomething(Integer arg1, Integer arg2, Integer arg3)
    {
        // ...
    }
}

public class ReturnSomethingString extends ReturnSomethingClass<String,
Integer>
{
    public ReturnSomethingString(int arg1, int arg2, int arg3)
    {
        returnSomething(arg1, arg2, arg3);
    }

    @Override
    protected void returnSomething(Integer arg1, Integer arg2, Integer arg3)
    {
        String retValue;
        // ...
        value = retValue;
    }
}
```

```
public class MainClass
{
    static void main(String[] args)
    {
        int a = 3, b = 4, c = 5;

        Object dummy = new DoSomethingInt(a,b,c);  // doSomething was called,
dummy is still around though
        String myReturn = (new ReturnSomethingString(a,b,c)).value; //
returnSomething was called and immediately destroyed
    }
}
```

Share Improve this answer Follow

answered May 14, 2020 at 20:29

Mark Walsh
1,035 ● 1 ● 8 ● 12