# How can I find all objects of a particular type?

devblogs.microsoft.com/oldnewthing/20100812-00

August 12, 2010

Raymond Chen

More than one customer has asked a question like this:

> I'm looking for a way to search for all instances of a particular type at runtime. My goal is to invoke a particular method on each of those instances. Note that I did not create these object myself or have any other access to them. Is this possible?

Imagine what the world would be like if it were possible.

For starters, just imagine the fun you could have if you could call `typeof(SecureString).GetInstances()`. Vegas road trip!

More generally, it breaks the semantics of AppDomain boundaries, since grabbing all instances of a type lets you get objects from another AppDomain, which fundamentally violates the point of AppDomains. (Okay, you could repair this by saying that the `GetInstances` method only returns objects from the current AppDomain.)

This imaginary `GetInstances` method might return objects which are awaiting finalization, which violates one of the fundamental assumptions of a finalizer, namely that there are no references to the object: If there were, then it wouldn't be finalized! (Okay, you could repair this by saying that the `GetInstances` method does not return objects which are awaiting finalization.)

On top of that, you break the syncRoot pattern.

```
class Sample {
 private object syncRoot = new object();
 public void Method() {
  lock(syncRoot) { ... };
 }
}
```

If it were possible to get all objects of a particular class, then anybody could just reach in and grab your private `syncRoot` and call `Monitor.Enter()` on it. Congratuations, the private synchronization object you created is now a public one that anybody can screw with, defeating the whole purpose of having a private syncRoot. You can no longer reason about your syncRoot because you are no longer in full control of it. (Yes, this can already be done with reflection, but at least when reflecting, you know that you're grabbing somebody's private field called `syncRoot`, so you already recognize that you're doing something dubious. Whereas with `GetInstances`, you don't know what each of the returned objects is being used for. Heck, you don't even know if it's being used! It might just be garbage lying around waiting to be collected.)

More generally, code is often written on the expectation that an object that you never give out a reference to is not accessible to others. Consider the following code fragment:

```
using (StreamWriter sr = new StreamWriter(fileName)) {
 sr.WriteLine("Hello");
}
```

If it were possible to get all objects of a particular class, you may find that your customers report that they are getting an `ObjectDisposedException` on the call to `WriteLine`. How is that possible? The disposal doesn't happen until the close-brace, right? Is there a bug in the CLR where it's disposing an object too soon?

Nope, what happened is that some other thread did exactly what the customer was asking for a way to do: It grabbed all existing `StreamWriter` instances and invoked `StreamWriter.Close` on them. It did this immediately after you constructed the `StreamWriter` and before you did your `sr.WriteLine()`. Result: When your `sr.WriteLine()` executes, it finds that the stream was already closed, and therefore the write fails.

More generally, consider the graffiti you could inject into all output files by doing

```
foreach (StreamWriter sr in typeof(StreamWriter).GetInstances()) {
 sr.Write("Kilroy was here!");
}
```

or even crazier

```
foreach (StringBuilder rb in typeof(StringBuilder).GetInstances()) {
 sb.Insert(0, "DROP TABLE users; --");
}
```

Now no `StringBuilder` is safe—the contents of any `StringBuilder` can be corrupted at any time!

If you could obtain all instances of a type, the fundamental logic behind computer programming breaks down. It effectively becomes impossible to reason about code because anything could happen to your objects *at any time*.

If you need to be able to get all instances of a class, you need to add that functionality to the class itself. ( `GCHandle` or `WeakReference` will come in handy here.) Of course, if you do this, then you clearly opted into the “anything can happen to your object at any time outside your control” model and presumably your code operates accordingly. You made your bed; now you get to lie in it.

(And I haven’t even touched on thread safety.)

**Bonus reading**: Questionable value of SyncRoot on Collections.