Type Checking: typeof, GetType, or is?

Ask Question

1332

456

I've seen many people use the following code:

```
Type t = typeof(obj1);
if (t == typeof(int))
    // Some code here
```

But I know you could also do this:

```
if (obj1.GetType() == typeof(int))
    // Some code here
```

Or this:

```
if (obj1 is int)
    // Some code here
```

Personally, I feel the last one is the cleanest, but is there something I'm missing? Which one is the best to use, or is it personal preference?

c#

asked Jun 11 '09 at 19:10

jasonh

14.8k 11 53 57

23 Don't forget `as` ! – RCIX Oct 7 '09 at 23:52

77 `as` isn't really type checking though... – jasonh Oct 8 '09 at 0:48

45 `as` is certainly a form of type-checking, every bit as much as `is` is! It effectively uses `is` behind the scenes, and is used all over the place in MSDN in places where it improves code cleanliness versus `is`. Instead of checking for `is` first, a call to `as` establishes a typed variable that's ready for use: If it's null, respond appropriately; otherwise, proceed. Certainly something I've seen and used quite a bit. – Zaccone Aug 20 '14 at 16:07

14 There is significant performance difference in favor of `as` / `is` (covered in stackoverflow.com/a/27813381/477420) assuming its semantic works for your case. – Alexei Levenkov Jan 7 '15 at 6:46

@samusarin it doesn't "use" reflection. The `GetType` method you are linking to is in `System.Reflection.Assembly` -- a completely different method and irrelevant here. – Kirk Woll Aug 11 '17 at 20:59

14 Answers

1642

All are different.

- `typeof` takes a type name (which you specify at compile time).
- `GetType` gets the runtime type of an instance.
- `is` returns true if an instance is in the inheritance tree.

Example

```
class Animal { }
class Dog : Animal { }

void PrintTypes(Animal a) {
    Console.WriteLine(a.GetType() == typeof(Animal)); // false
    Console.WriteLine(a is Animal);                   // true
    Console.WriteLine(a.GetType() == typeof(Dog));    // true
    Console.WriteLine(a is Dog);                      // true
}
```

```
Dog spot = new Dog();
PrintTypes(spot);
```

> What about `typeof(T)` ? Is it also resolved at compile time?

Yes. T is always what the type of the expression is. Remember, a generic method is basically a whole bunch of methods with the appropriate type. Example:

```
string Foo<T>(T parameter) { return typeof(T).Name; }

Animal probably_a_dog = new Dog();
Dog    definitely_a_dog = new Dog();

Foo(probably_a_dog); // this calls Foo<Animal> and returns "Animal'
Foo<Animal>(probably_a_dog); // this is exactly the same as above
Foo<Dog>(probably_a_dog); // !!! This will not compile. The paramet
cannot pass in an Animal.

Foo(definitely_a_dog); // this calls Foo<Dog> and returns "Dog"
Foo<Dog>(definitely_a_dog); // this is exactly the same as above.
Foo<Animal>(definitely_a_dog); // this calls Foo<Animal> and returr
Foo((Animal)definitely_a_dog); // this does the same as above, retu
```

edited Aug 7 '18 at 3:22

themefield
608 7 15

answered Jun 11 '09 at 19:15

Jimmy
67.2k 13 107 132

25 Ah, so if I have a Ford class that derives from Car and an instance of Ford, checking "is Car" on that instance will be true. Makes sense! – jasonh Jun 11 '09 at 19:19

1 To clarify, I was aware of that, but I commented before you added a

code sample. I wanted to try to add some plain English clarity to your

code sample. I wanted to try to add some plain English clarity to your already excellent answer. – jasonh Jun 11 '09 at 19:25

2 @Jimmy So what about performance? `typeof` vs. `GetType()` ? – Shimmy Jan 26 '12 at 4:59

11 @Shimmy if typeof is evaluated at compile time and GetType() is evaluated at runtime, then it makes sense that GetType() incurs a slight performance hit – Cedric Mamo Dec 10 '12 at 13:35

6 @PrerakK `new Dog().GetType() is Animal` returns false (and your other version as well) since `.GetType()` returns an object of type `Type` , and `Type` is not an `Animal` . – Maarten Nov 7 '14 at 9:45

11

If you're using C# 7, then it is time for an update to Andrew Hare's great answer. Pattern matching has introduced a nice shortcut that gives us a typed variable within the context of the if statement, without requiring a separate declaration/cast and check:

```
if (obj1 is int integerValue)
{
    integerValue++;
}
```

This looks pretty underwhelming for a single cast like this, but really shines when you have many possible types coming into your routine. The below is the old way to avoid casting twice:

```
Button button = obj1 as Button;
if (button != null)
{
    // do stuff...
    return;
}
TextBox text = obj1 as TextBox;
if (text != null)
{
    // do stuff...
    return;
```

```
}
Label label = obj1 as Label;
if (label != null)
{
    // do stuff...
    return;
}
// ... and so on
```

Working around shrinking this code as much as possible, as well as avoiding duplicate casts of the same object has always bothered me. The above is nicely compressed with pattern matching to the following:

```
switch (obj1)
{
    case Button button:
        // do stuff...
        break;
    case TextBox text:
        // do stuff...
        break;
    case Label label:
        // do stuff...
        break;
    // and so on...
}
```

EDIT: Updated the longer new method to use a switch as per Palec's comment.

edited Jun 26 '18 at 19:56

answered Feb 1 '18 at 15:47

JoelC

2,473 8 31 33

1 Using `switch` statement with pattern matching is advisable in this case. – Palec Jun 10 '18 at 17:41

How would you cope with a is not ? In this particular code block? `if (obj1 is int integerValue) { integerValue++; }` – Ben Vertonghen Feb 25 at 22:28

Ben, if I understand your question, I would just have an else statement to handle other cases since you can't put a not integer into an integer variable. :) – JoelC Feb 28 at 13:53

46

```
Type t = typeof(obj1);
if (t == typeof(int))
    // Some code here
```

This is an error. The typeof operator in C# can only take type names, not objects.

```
if (obj1.GetType() == typeof(int))
    // Some code here
```

This will work, but maybe not as you would expect. For value types, as you've shown here, it's acceptable, but for reference types, it would only return true if the type was the *exact same* type, not something else in the inheritance hierarchy. For instance:

```
class Animal{}
class Dog : Animal{}

static void Foo(){
    object o = new Dog();

    if(o.GetType() == typeof(Animal))
        Console.WriteLine("o is an animal");
    Console.WriteLine("o is something else");
}
```

This would print `"o is something else"`, because the type of `o` is `Dog`, not `Animal`. You can make this work, however, if you use the `IsAssignableFrom` method of the `Type` class.

```
if(typeof(Animal).IsAssignableFrom(o.GetType())) // note use of test
    Console.WriteLine("o is an animal");
```

This technique still leaves a major problem, though. If your variable is null, the call to `GetType()` will throw a NullReferenceException. So to make it work correctly, you'd do:

```
if(o != null && typeof(Animal).IsAssignableFrom(o.GetType()))
    Console.WriteLine("o is an animal");
```

With this, you have equivalent behavior of the `is` keyword. Hence, if this is the behavior you want, you should use the `is` keyword, which is more readable and more efficient.

```
if(o is Animal)
    Console.WriteLine("o is an animal");
```

In most cases, though, the `is` keyword still isn't what you really want, because it's usually not enough just to know that an object is of a certain type. Usually, you want to actually *use* that object as an instance of that type, which requires casting it too. And so you may find yourself writing code like this:

```
if(o is Animal)
    ((Animal)o).Speak();
```

But that makes the CLR check the object's type up to two times. It will check it once to satisfy the `is` operator, and if `o` is indeed an `Animal`, we make it check again to validate the cast.

It's more efficient to do this instead:

```
Animal a = o as Animal;
if(a != null)
    a.Speak();
```

The `as` operator is a cast that won't throw an exception if it fails, instead returning `null`. This way, the CLR checks the object's type just once, and after that, we just need to do a null check, which is more efficient.

But beware: many people fall into a trap with `as`. Because it doesn't throw exceptions, some people think of it as a "safe" cast, and they use it exclusively, shunning regular casts. This leads to errors like this:

```
(o as Animal).Speak();
```

In this case, the developer is clearly assuming that `o` will *always* be an `Animal`, and as long as their assumption is correct, everything works fine. But if they're wrong, then what they end up with here is a `NullReferenceException`. With a regular cast, they would have gotten an `InvalidCastException` instead, which would have more correctly identified the problem.

Sometimes, this bug can be hard to find:

```
class Foo{
    readonly Animal animal;

    public Foo(object o){
        animal = o as Animal;
    }

    public void Interact(){
        animal.Speak();
    }
}
```

This is another case where the developer is clearly expecting `o` to be an `Animal` every time, but this isn't obvious in the constructor,

where the `as` cast is used. It's not obvious until you get to the `Interact` method, where the `animal` field is expected to be positively assigned. In this case, not only do you end up with a misleading exception, but it isn't thrown until potentially much later than when the actual error occurred.

In summary:

- If you only need to know whether or not an object is of some type, use `is`.
- If you need to treat an object as an instance of a certain type, but you don't know for sure that the object will be of that type, use `as` and check for `null`.
- If you need to treat an object as an instance of a certain type, and the object is supposed to be of that type, use a regular cast.

edited Jan 11 '17 at 17:02

answered Jun 11 '09 at 19:34

P Daddy
23.7k 7 58 85

what is wrong with this if(o is Animal) ((Animal)o).Speak(); ? can you please give more details? – batmaci Jan 10 '17 at 22:11

2 @batmaci: it's in the answer -- it causes two type checks. The first time is `o is Animal`, which requires the CLR to check if the type of the variable `o` is an `Animal`. The second time it checks is when it casts in the statement `((Animal)o).Speak()`. Rather than check twice, check once using `as`. – siride Jul 28 '18 at 15:15

```
if (c is UserControl) c.Enabled = enable;
```

-3

edited Sep 6 '16 at 13:18

Complexity

3,255 2 22 57

answered Aug 12 '16 at 10:31

Paulos02

124 1 4

4 Please edit with more information. Code-only and "try this" answers are discouraged, because they contain no searchable content, and don't explain why someone should "try this". – abarisone Sep 6 '16 at 14:33

13

I had a `Type` -property to compare to and could not use `is` (like `my_type is _BaseTypetoLookFor`), but I could use these:

```
base_type.IsInstanceOfType(derived_object);
base_type.IsAssignableFrom(derived_type);
derived_type.IsSubClassOf(base_type);
```

Notice that `IsInstanceOfType` and `IsAssignableFrom` return `true` when comparing the same types, where IsSubClassOf will return `false`. And `IsSubclassOf` does not work on interfaces, where the other two do. (See also this question and answer.)

```
public class Animal {}
public interface ITrainable {}
public class Dog : Animal, ITrainable{}

Animal dog = new Dog();

typeof(Animal).IsInstanceOfType(dog);     // true
typeof(Dog).IsInstanceOfType(dog);        // true
typeof(ITrainable).IsInstanceOfType(dog); // true

typeof(Animal).IsAssignableFrom(dog.GetType());      // true
```

```
typeof(Dog).IsAssignableFrom(dog.GetType());          // true
typeof(ITrainable).IsAssignableFrom(dog.GetType()); // true

dog.GetType().IsSubclassOf(typeof(Animal));            // true
dog.GetType().IsSubclassOf(typeof(Dog));               // false
dog.GetType().IsSubclassOf(typeof(ITrainable)); // false
```

edited May 23 '17 at 10:31

Community ♦

1 1

answered May 15 '12 at 10:39

Yahoo Serious

2,370 1 22 27

0

Used to obtain the System.Type object for a type. A typeof expression takes the following form:

```
System.Type type = typeof(int);
```

Example:

```
    public class ExampleClass
    {
       public int sampleMember;
       public void SampleMethod() {}

       static void Main()
       {
          Type t = typeof(ExampleClass);
          // Alternatively, you could use
          // ExampleClass obj = new ExampleClass();
          // Type t = obj.GetType();

          Console.WriteLine("Methods:");
          System.Reflection.MethodInfo[] methodInfo = t.GetMethods()

          foreach (System.Reflection.MethodInfo mInfo in methodInfo)
```

```
            Console.WriteLine(mInfo.ToString());

         Console.WriteLine("Members:");
         System.Reflection.MemberInfo[] memberInfo = t.GetMembers()

         foreach (System.Reflection.MemberInfo mInfo in memberInfo)
            Console.WriteLine(mInfo.ToString());
      }
   }
   /*
    Output:
       Methods:
       Void SampleMethod()
       System.String ToString()
       Boolean Equals(System.Object)
       Int32 GetHashCode()
       System.Type GetType()
       Members:
       Void SampleMethod()
       System.String ToString()
       Boolean Equals(System.Object)
       Int32 GetHashCode()
       System.Type GetType()
       Void .ctor()
       Int32 sampleMember
   */
```

This sample uses the GetType method to determine the type that is used to contain the result of a numeric calculation. This depends on the storage requirements of the resulting number.

```
class GetTypeTest
{
    static void Main()
    {
        int radius = 3;
        Console.WriteLine("Area = {0}", radius * radius * Math.P
        Console.WriteLine("The type is {0}",
                          (radius * radius * Math.PI).GetType()
        );
    }
}
```

```
/*
Output:
Area = 28.2743338823081
The type is System.Double
*/
```

edited Apr 14 '16 at 7:52

answered Apr 14 '16 at 7:17

Muhammad Awais
2,037 1 25 22

-5

Performance test typeof() vs GetType():

```
using System;
namespace ConsoleApplication1
    {
    class Program
    {
        enum TestEnum { E1, E2, E3 }
        static void Main(string[] args)
        {
            {
                var start = DateTime.UtcNow;
                for (var i = 0; i < 1000000000; i++)
                    Test1(TestEnum.E2);
                Console.WriteLine(DateTime.UtcNow - start);
            }
            {
                var start = DateTime.UtcNow;
                for (var i = 0; i < 1000000000; i++)
                    Test2(TestEnum.E2);
                Console.WriteLine(DateTime.UtcNow - start);
            }
            Console.ReadLine();
        }
        static Type Test1<T>(T value) => typeof(T);
```

```
        static Type Test2(object value) => value.GetType();
    }
}
```

Results in debug mode:

```
00:00:08.4096636
00:00:10.8570657
```

Results in release mode:

```
00:00:02.3799048
00:00:07.1797128
```

answered Aug 3 '15 at 13:58

Alexander Vasilyev
1,102 1 13 21

1 One should not use DateTime.UtcNow for performance measures. With your code but with Stopwatch class I got persistently opposite results for Debug mode. UseTypeOf: 00:00:14.5074469 UseGetType: 00:00:10.5799534 . Release mode is the same, as expected – Alexey Shcherbak Nov 25 '15 at 3:46

@AlexeyShcherbak The difference between Stopwatch and DateTime.Now cannot be more than 10-20 ms, check your code again. And I don't care about milliseconds in my test. Also my code will be several lines of code longer with Stopwatch. – Alexander Vasilyev Nov 25 '15 at 9:21

1 it's bad practice in general, not in your particular case. – Alexey Shcherbak Nov 25 '15 at 9:23

3 @AlexanderVasilyev Amount of lines of code should never be used as an argument to do something *documentedly* misleading. As seen in msdn.microsoft.com/en-us/library/system.datetime(v=vs.110).aspx, `DateTime` should not be used if you're concerned about times below *100ms*, since it uses the OS's timeframe. Comparatively with `Stopwatch`, which uses the processors' `Tick`, the resolution used by a `DateTime` in Win7 is a whooping 15ms. – Eric Wu Mar 6 '16 at 5:54

-5

You can use "typeof()" operator in C# but you need to call the namespace using System.IO; You must use "is" keyword if you wish to check for a type.

edited Apr 7 '14 at 7:01

answered Apr 3 '14 at 11:06

androidrill

51 2 10

6 `typeof` is not defined in a namespace, it is a keyword. `System.IO` has nothing to do with this. – Arturo Torres Sánchez Sep 18 '15 at 0:13

8

I prefer **is**

That said, if you're using **is**, you're likely *not* using inheritance properly.

Assume that Person : Entity, and that Animal : Entity. Feed is a virtual method in Entity (to make Neil happy)

```
class Person
{
  // A Person should be able to Feed
  // another Entity, but they way he feeds
  // each is different
  public override void Feed( Entity e )
  {
    if( e is Person )
    {
      // feed me
    }
```

```
    else if( e is Animal )
    {
      // ruff
    }
  }
}
```

Rather

```
class Person
{
  public override void Feed( Person p )
  {
    // feed the person
  }
  public override void Feed( Animal a )
  {
    // feed the animal
  }
}
```

edited Oct 23 '13 at 14:10

JackPoint
2,931 25 38

answered Jun 11 '09 at 19:15

bobobobo
36k 45 214 305

1 True, I would never do the former, knowing that Person derives from Animal. – jasonh Jun 11 '09 at 19:22

3 The latter isn't really using inheritance, either. Foo should be a virtual method of Entity that is overridden in Person and Animal. – Neil Williams Jun 11 '09 at 19:31

1 @bobobobo I think you mean "overloading", not "inheritance". – lc. Jun 11 '09 at 19:42

@lc: No, I mean inheritance. The first example is a kind of incorrect way (using *is*) to get different behavior. The second example uses overloading yes, but avoids use of *is*. – bobobobo Jun 11 '09 at 19:47

1 The problem with the example is that it wouldn't scale. If you added new entities that needed to eat (for example an Insect or a Monster) you would need to add a new method in the Entity class and then override it in subclasses that would feed it. This isn't any more preferable than a list if (entity is X) else if (entity is Y)... This violates the LSP and OCP, inheritance probably isn't the best solution to the problem. Some form of delegation would probably be preferred. – ebrown Jun 11 '09 at 21:12

177

Use `typeof` when you want to get the type at *compilation time*. Use `GetType` when you want to get the type at *execution time*. There are rarely any cases to use `is` as it does a cast and, in most cases, you end up casting the variable anyway.

There is a fourth option that you haven't considered (especially if you are going to cast an object to the type you find as well); that is to use `as`.

```
Foo foo = obj as Foo;

if (foo != null)
    // your code here
```

This only uses **one** cast whereas this approach:

```
if (obj is Foo)
    Foo foo = (Foo)obj;
```

requires **two**.

edited Feb 22 '13 at 10:16

loxdog
710 2 9 27

answered Jun 11 '09 at 19:14

Andrew Hare
281k 54 584 604

4 With the changes in .NET 4 does `is` still perform a cast? – ahsteele Mar 21 '13 at 21:04

6 Is this answer correct? Is it true that you really can pass an instance into typeof()? My experience has been No. But I guess it's generally true that checking an instance might have to happen at runtime, whereas checking a class should be doable at compile time. – Jon Coombs Oct 8 '13 at 2:13

4 @jon (4 yrs after your q.), no, you cannot pass an instance into `typeof()`, and this answer doesn't suggest you can. You pass in the type instead, i.e., `typeof(string)` works, `typeof("foo")` does not. – Abel Sep 20 '17 at 1:30

I don't believe `is` performs cast as such, rather special operation in IL. – abatishchev Oct 22 '18 at 20:20

2

It depends on what I'm doing. If I need a bool value (say, to determine if I'll cast to an int), I'll use `is`. If I actually need the type for some reason (say, to pass to some other method) I'll use `GetType()`.

edited Jun 11 '09 at 19:20

Andrew Hare
281k 54 584 604

answered Jun 11 '09 at 19:14

AllenG
7,485 22 38

1 Good point. I forgot to mention that I got to this question after looking at several answers that used an if statement to check a type. – jasonh Jun 11 '09 at 19:20

66

1.

```
Type t = typeof(obj1);
if (t == typeof(int))
```

This is illegal, because typeof only works on types, not on variables. I assume obj1 is a variable. So, in this way typeof is static, and does its work at compile time instead of runtime.

2.

```
if (obj1.GetType() == typeof(int))
```

This is true if obj1 is exactly of type int. If obj1 derives from int, the if condition will be false.

3.

```
if (obj1 is int)
```

This is true if obj1 is an int, or if it derives from a class called int, or if it implements an interface called int.

answered Jun 11 '09 at 19:17

Scott Langham
39.1k 29 111 178

Thinking about 1, you're right. And yet, I've seen it in several code samples here. It should be Type t = obj1.GetType(); – jasonh Jun 11 '09 at 19:26

4 Yep, I think so. "typeof(obj1)" doesn't compile when I try it. – Scott Langham Jun 11 '09 at 19:39

4 It's impossible to derive from System.Int32 or any other value type in C# – reggaeguitar Dec 4 '14 at 19:34

can you tell what would be typeof(typeof(system.int32)) – Sana Jun 27 '18 at 12:33

@Sana, why don't you try it :) I would imagine though you get back an instance of System.Type that represents the type System.Type! Documentation for typeof is here: docs.microsoft.com/en-us/dotnet/csharp/language-reference/... – Scott Langham Jul 2 '18 at 10:46

0

The last one is cleaner, more obvious, and also checks for subtypes. The others do not check for polymorphism.

answered Jun 11 '09 at 19:16

thecoop
35.7k 11 106 164

5

I believe the last one also looks at inheritance (e.g. Dog is Animal == true), which is better in most cases.

answered Jun 11 '09 at 19:14

StriplingWarrior
109k 20 185 235

protected by CharithJ Nov 9 '16 at 3:24

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?