How is duck typing different from the old 'variant' type and/or interfaces?

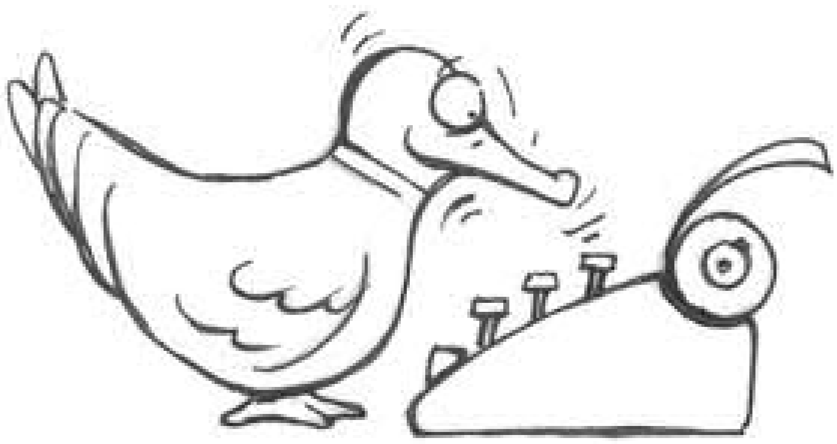
Asked 16 years, 1 month ago Modified 5 years, 2 months ago

Viewed 5k times

47

I keep seeing the phrase "duck typing" bandied about, and even ran across a code example or two. I am way too ~~lazy~~ busy to do my own research, can someone tell me, briefly:

- the difference between a 'duck type' and an old-skool 'variant type', and
- provide an example of where I might prefer duck typing over variant typing, and
- provide an example of something that i would *have* to use duck typing to accomplish?



I don't mean to seem fowl by doubting the power of this 'new' construct, and I'm not ducking the issue by refusing to do the research, but I am quacking up at all the flocking hype i've been seeing about it lately. It looks like *no* typing (aka dynamic typing) to me, so I'm not seeing the advantages right away.

ADDENDUM: Thanks for the examples so far. It seems to me that using something like 'O->can(Blah)' is equivalent to doing a reflection lookup (which is probably not cheap), and/or is about the same as saying (O is IBlah) which the compiler might be able to check for you, but the latter has the advantage of distinguishing my IBlah interface from your IBlah interface while the other two do not. Granted, having a lot of tiny interfaces floating around for every method would get messy, but then again so can checking for a lot of individual methods...

...so again i'm just not getting it. Is it a fantastic time-saver, or the same old thing in a brand new sack? Where is the example that *requires* duck typing?

interface **variant** **duck-typing**

Share

Improve this question

Follow

edited Feb 8, 2017 at 14:09

Community Bot
1 ● 1

asked Nov 14, 2008 at 3:40

Steven A. Lowe
61.1k ● 19 ● 135 ● 204

I think, you mean not "variant", but rather late-binding through IDispatch interface. – wasker Nov 16, 2008 at 3:36

nicely done! Very nicely done. Thanks for making me smile today. – Tim Nov 26, 2008 at 19:05

Takes some work to teach the ducks to type, but the returns in productivity are quite worthwhile ;-) – Rudiger Mar 21, 2010 at 0:55

10 Answers

Sorted by: Highest score (default)

34

In some of the answers here, I've seen some incorrect use of terminology, which has lead people to provide wrong answers.

So, before I give my answer, I'm going to provide a few definitions:

1. Strongly typed

   A language is strongly typed if it enforces the type safety of a program. That means that it guarantees two things: something called progress and something else called preservation. Progress basically means that all "validly typed" programs can in fact be run by the computer, They may crash, or throw an exception, or run for an infinite loop, but they can actually be run. Preservation means that if a program is "validly typed" that it will always be "Validly typed", and that no variable (or memory location) will contain a value that does not conform to its assigned type.

Most languages have the "progress" property. There are many, however, that don't satisfy the "preservation" property. A good example, is C++ (and C too). For example, it is possible in C++ to coerce any memory address to behave as if it was any type. This basically allows programmers to violate the type system any time they want. Here is a simple example:

```
struct foo
{
    int x;
    iny y;
    int z;
}

char * x = new char[100];
foo * pFoo = (foo *)x;
foo aRealFoo;
*pFoo = aRealFoo;
```

This code allows someone to take an array of characters and write a "foo" instance to it. If C++ was strongly typed this would not be possible. Type safe languages, like C#, Java, VB, lisp, ruby, python, and many others, would throw an exception if you tried to cast an array of characters to a "foo" instance.

2. Weakly typed

   Something is weakly typed if it is not strongly typed.

3. Statically typed

   A language is statically typed if its type system is verified at compile time. A statically typed language can be either "weakly typed" like C or strongly typed like C#.

4. Dynamically typed

   A dynamically typed language is a language where types are verified at runtime. Many languages have a mixture, of some sort, between static and dynamic typing. C#, for example, will verify many casts dynamically at runtime because it's not possible to check them at compile time. Other examples are languages like Java, VB, and Objective-C.

   There are also some languages that are "completely" or "mostly" dynamically typed, like "lisp", "ruby", and "small talk"

5. Duck typing

   Duck typing is something that is completely orthogonal to static, dynamic, weak, or strong typing. It is the practice of writing code that will work with an object regardless of its underlying type identity. For example, the following VB.NET code:

   ```
   function Foo(x as object) as object
       return x.Quack()
   end function
   ```

   Will work, regardless of what the type of the object is that is passed into "Foo", provided that is defines a method called "Quack". That is, if the object looks like a duck, walks like a duck, and talks like a duck, then it's a duck. Duck typing comes in many forms. It's possible to have static duck typing, dynamic duck typing, strong duck typing, and weak duck typing. C++ template functions are a good example of "weak static duck typing". The example show in "JaredPar's" post shows an example of "strong static duck typing". Late binding in VB (or code in Ruby or Python) enables "strong dynamic duck typing".

6. Variant

   A variant is a dynamically typed data structure that can hold a range of predefined data types, including strings, integer types, dates, and com objects. It then defines a bunch of operations for assigning, converting, and manipulating data stored in variants. Whether or not a variant is strongly typed depends on the language in which it is used. For example, a variant in a VB 6 program is strongly typed. The VB runtime ensures that operations written in VB code will conform to the typing rules for variants. Tying to add a string to an IUnknown via the variant type in VB will result in a runtime error. In C++, however, variants are weakly typed because all C++ types are weakly typed.

OK.... now that I have gotten the definitions out of the way, I can now answer your question:

A variant, in VB 6, enables one form of doing duck typing. There are better ways of doing duck typing (Jared Par's example is one of the best), than variants, but you can do duck typing with variants. That is, you can write one piece of code that will operate on an object regardless of its underlying type identity.

However, doing it with variants doesn't really give a lot of validation. A statically typed duck type mechanism, like the one JaredPar describes gives the benefits of duck typing, plus some extra validation from the compiler. That can be really helpful.

Share Improve this answer Follow

edited Oct 1, 2019 at 7:33
Tim Rutter
4,679 ● 4 ● 25 ● 50

answered Nov 14, 2008 at 8:47

Scott Wisniewski
25k ● 8 ● 62 ● 90

3 Thanks for the extended info, but I am curious about one statement: "Something is weekly typed if it is not strongly typed." - Wouldn't something be weekly typed if it was only typed once per week? ;-) – Steven A. Lowe Nov 14, 2008 at 15:31

2 Yea... I'm not the best at spell checking my posts. – Scott Wisniewski Nov 14, 2008 at 23:30

20

The simple answer is variant is weakly typed while duck typing is strongly typed.

Duck typing can be summed up nicely as "if it walks like a duck, looks like a duck, acts like a duck, then it's a duck."

It computer science terms consider duck to be the following interface.

```
interface IDuck {
  void Quack();
}
```

Now let's examine Daffy

```
class Daffy {
  void Quack() {
    Console.WriteLine("Thatsssss dispicable!!!!");
  }
}
```

Daffy is not actually an IDuck in this case. Yet it acts just like a Duck. Why make Daffy implement IDuck when it's quite obvious that Daffy is in fact a duck.

This is where Duck typing comes in. It allows a type safe conversion between any type that has all of the behaviors of a IDuck and an IDuck reference.

```
IDuck d = new Daffy();
d.Quack();
```

The Quack method can now be called on "d" with complete type safety. There is no chance of a runtime type error in this assignment or method call.

Share Improve this answer Follow

edited Feb 12, 2010 at 11:30

John Topley
115k ● 47 ● 199 ● 240

answered Nov 14, 2008 at 7:09

JaredPar
753k ● 151 ● 1.3k ● 1.5k

7 What I don't understand about Ducktyping is this: how can you be sure that Daffy's Quack method is equal to IDuck.Quack? Just because it has the same name and the type of the parameters matches, doesn't necessarily mean the method does what you want/need. ... – Otherside Nov 14, 2008 at 13:47

1 ... If Daffy explicitly implements the IDuck interface you are more certain that you are calling the correct method. For example a Refresh method could mean: refresh/repaint the GUI, or reload data from the database and refresh this object. – Otherside Nov 14, 2008 at 13:49

6 That's the danger in duck typing - there is no semantic checking, only syntactic checking. But then, just because you are 'more sure' when dealing with a type that implements an interface, doesn't mean you are 'certain.' – Erik Forbes Nov 14, 2008 at 23:41

10 @Otherside, You have to let go of the idea that interfaces are contracts. The sad truth is they are not behavior contracts. They are simply data passing contracts. Duck typing is no worse in this respect that explicitly implemented interfaces. – JaredPar Nov 15, 2008 at 20:15

5

Duck typing is just another term for dynamic typing or late-binding. A variant object that parses/compiles with any member access (e.g., obj.Anything) that may or not actually be defined during runtime is duck typing.

Share Improve this answer Follow

answered Nov 14, 2008 at 3:47

Mark Cidade
99.8k ● 33 ● 229 ● 237

+1 thanks - i added a request for examples if you'd like to take a shot at that too – Steven A. Lowe Nov 14, 2008 at 3:51

4

Probably nothing *requires* duck-typing, but it can be convenient in certain situations. Say you have a method that takes and uses an object of the sealed class Duck from some 3rd party library. And you want to make the method testable. And Duck has an awfully big API (kind of like ServletRequest) of which you only need to care about a small subset. How do you test it?

One way is to make the method take something that quacks. Then you can simply create a quacking mock object.

Share Improve this answer Follow

answered Nov 16, 2008 at 5:00

Germán
4,565 ● 3 ● 34 ● 38

Try reading the very first paragraph of the Wikipedia article on duck typing.
[Duck typing on Wikipedia]

3

I can have an interface (IRunnable) that defines the method Run().
If I have another class with a method like this:
public void RunSomeRunnable(IRunnable rn) { ... }

In a duck type friendly language I could pass in any class that had a Run() method into the RunSomeRunnable() method.
In a statically typed language the class being passed into RunSomeRunnable needs to explicitly implement the IRunnable interface.

"If it Run() like a duck"

variant is more like object in .NET at least.

Share Improve this answer Follow

answered Nov 14, 2008 at 3:57
BuddyJoe
71.1k ● 115 ● 301 ● 473

+1 thanks - i added a request for examples if you'd like to take a shot at that too – Steven A. Lowe Nov 14, 2008 at 4:12

1 I can't believe you missed out on using the method Quack() in that example ;) – korona Nov 14, 2008 at 8:56

2

@Kent Fredric

Your example can most certainly be done without duck typing by using explicit interfaces...uglier yes, but it's not impossible.

And personally, I find having well defined contracts in interfaces much better for enforcing quality code, than relying on duck typing...but that's just my opinion and take it with a grain of salt.

```
public interface ICreature { }
public interface IFly { fly();}
public interface IWalk { walk(); }
public interface IQuack { quack(); }
// ETC

// Animal Class
public class Duck : ICreature, IWalk, IFly, IQuack
{
    fly() {};
    walk() {};
    quack() {};
}

public class Rhino: ICreature, IWalk
{
    walk();
}

// In the method
List<ICreature> creatures = new List<ICreature>();
creatures.Add(new Duck());
creatures.Add(new Rhino());

foreach (ICreature creature in creatures)
{
    if (creature is IFly)
         (creature as IFly).fly();
    if (creature is IWalk)
         (creature as IWalk).walk();
}
// Etc
```

Share Improve this answer Follow

edited Nov 14, 2008 at 4:35

answered Nov 14, 2008 at 4:30

FlySwat
175k ● 75 ● 248 ● 314

aren't you supposed to wait for Jon Skeet to answer every question first? ;-) – Steven A. Lowe Nov 14, 2008 at 4:32

I think he's sleeping (What time is it in the UK?) – FlySwat Nov 14, 2008 at 4:36

it is currently 4:47AM in the UK – Steven A. Lowe Nov 14, 2008 at 4:47

yeah, fine and dandy for a short list, wait till you have an interface for *every* function in the class. So much redundant code. Duck-typing makes better. – Kent Fredric Nov 14, 2008 at 5:56

For what its worth, you can do monkey patching in php too, just you have to implement it yourself, and nobody will want to use that code :) – Kent Fredric Nov 14, 2008 at 6:00

2

In regards to your request for an example of something you'd *need* to use duck typing to accomplish, I don't think such a thing exists. I think of it like I think about whether to use recursion or whether to use iteration. Sometimes one just works better than the other.

In my experience, duck typing makes code more readable and easier to grasp (both for the programmer and the reader). But I find that more traditional static typing eliminates a lot of needless typing errors. There's simply no way to objectively say one is better than another or even to say what situations one is more effective than the other.

I say that if you're comfortable using static typing, then use it. But you should at least try duck typing out (and use it in a nontrivial project if possible).

To answer you more directly:

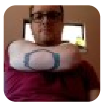
> ...so again i'm just not getting it. Is it a fantastic time-saver, or the same old thing in a brand new sack?

It's both. You're still attacking the same problems. You're just doing it a different way. Sometimes that's really all you need to do to save time (even if for no other reason to force yourself to think about doing something a different way).

Is it a panacea that will save all of mankind from extinction? No. And anyone who tells you otherwise is a zealot.

Share Improve this answer Follow

answered Nov 16, 2008 at 4:04



Jason Baker
198k ● 138 ● 382 ● 520

1

A variant (at least as I've used them in VB6) holds a variable of a single, well-defined, usually static type. E.g., it might hold an int, or a float, or a string, but variant ints are used as ints, variant floats are used as floats, and variant strings are used as strings.

Duck typing instead uses dynamic typing. Under duck typing, a variable might be usable as an int, or a float, or a string, if it happens to support the particular methods that an int or float or string supports in a particular context.

Example of variants versus duck typing:

For a web application, suppose I want my user information to come from LDAP instead of from a database, but I still want my user information to be useable by the rest of the web framework, which is based around a database and an ORM.

Using variants: No luck. I can create a variant that can contain a UserFromDbRecord object or a UserFromLdap object, but UserFromLdap objects won't be usable by routines that expect objects from the FromDbRecord hierarchy.

Using duck typing: I can take my UserFromLdap class and add a couple of methods that make it act like a UserFromDbRecord class. I don't need to replicate the entire FromDbRecord interface, just enough for the routines that I need to use. If I do this right, it's an extremely powerful and flexible technique. If I do it wrong, it produces very confusing and brittle code (subject to breakage if either the DB library or the LDAP library changes).

Share Improve this answer Follow

edited Nov 14, 2008 at 4:12

answered Nov 14, 2008 at 3:52

Josh Kelley
58.3k ● 20 ● 162 ● 256

thanks for the example! so i can just add the methods that i'm going to need rather than adding every method required by an interface (or changing the base class) - that's pretty cool. Doesn't that get confusing if you do it a lot though?
– Steven A. Lowe Nov 14, 2008 at 4:16

1

I think the core point of duck typing is how it is used. One uses method detection and introspection of the entity in order to know what to do with it, instead of declaring in advance what it *will* be (where you know what to do with it).

It's probably more practical in OO languages, where primitives are not primitives, and are instead objects.

I think the best way to sum it up, in variant type, an entity is/can be anything, and what it is is uncertain, as opposed to an entity only *looks* like anything, but you can work out what it is by asking it.

Here's something I don't believe is plausible without ducktyping.

```
sub dance {
     my $creature = shift;
     if( $creature->can("walk") ){
         $creature->walk("left",1);
         $creature->walk("right",1);
         $creature->walk("forward",1);
         $creature->walk("back",1);
     }
     if( $creature->can("fly") ){
          $creature->fly("up");
          $creature->fly("right",1);
          $creature->fly("forward",1);
          $creature->fly("left", 1 );
          $creature->fly("back", 1 );
          $creature->fly("down");
     } else if ( $creature->can("walk") ) {
         $creature->walk("left",1);
         $creature->walk("right",1);
         $creature->walk("forward",1);
         $creature->walk("back",1);
     } else if ( $creature->can("splash") ) {
         $creature->splash( "up" ) for ( 0 .. 4 );
     }
     if( $creature->can("quack") ) {
         $creature->quack();
     }
 }

 my @x = ();
 push @x, new Rhinoceros ;
 push @x, new Flamingo;
 push @x, new Hyena;
 push @x, new Dolphin;
 push @x, new Duck;

 for my $creature (@x){

    new Thread(sub{
```

Any other way would require you to put type restrictions on for functions, which would cut out different species, needing you to create different functions for different species, making the code really hellish to maintain.

And that really sucks in terms of just trying to perform good choreography.

Share Improve this answer Follow

edited Feb 12, 2010 at 11:33
John Topley
115k ● 47 ● 199 ● 240

answered Nov 14, 2008 at 3:48

Kent Fredric
57.3k ● 14 ● 111 ● 151

thanks for the example! could i not achieve the same thing if i had IWalk, IFly, ISplash, and IQuack interfaces? Granted that would be more painful... alternately I could use reflection to see if the methods were available (which is probably what the 'can' operator does under the hood)...? – Steven A. Lowe Nov 14, 2008 at 4:17

1 Well, I guess in essence there is no /direct/ difference, you can virtually do dynamic typing in static languages if you **try** hard enough, C++ has a lamda implementation for instance. Just the language is centered around that style making it easier. – Kent Fredric Nov 14, 2008 at 6:47

0

Everything you can do with duck-typing you can also do with interfaces. Duck-typing is fast and comfortable, but some argue it can lead to errors (if two distinct methods/properties are named alike). Interfaces are safe and explicit, but people might say "why state the obvious?". Rest is a flame. Everyone chooses what suits him and no one is "right".

Share Improve this answer

Follow

answered Feb 12, 2010 at 11:26

zefciu

2,036 ● 2 ● 17 ● 40