

Duck Typing

Duck Typing

Boo is a statically typed language, like Java or C#. This means your boo applications will run about as fast as those coded in other statically typed languages for .NET or Mono. But using a statically typed language sometimes constrains you to an inflexible and verbose coding style, with the sometimes necessary type declarations (like "x as int", but this is not often necessary due to boo's [Type Inference](#)) and sometimes necessary type casts (see [Casting Types](#)). Boo's support for [Type Inference](#) and eventually generics help here, but...

Sometimes it is appropriate to give up the safety net provided by static typing. Maybe you just want to explore an API without worrying too much about method signatures or maybe you're creating code that talks to external components such as COM objects. Either way the choice should be yours not mine.

Along with the normal types like object, int, string...boo has a special type called "duck". The term is inspired by the ruby programming language's duck typing feature ("If it walks like a duck and quacks like a duck, it must be a duck").

If you declare an object as type duck or cast to type duck (or turn on the implicit duck typing option, see below), then boo will not try to resolve any methods or operations you call on that object at compile time. Instead it will convert those operations into methods that do not resolve until runtime. So at compile time it simply trusts that you know what you are doing ("it walks like a duck so it must be a duck"). This is easier to understand by looking at some examples.

Illustrative Example: Static vs Duck Types

This code illustrates some things you can do with duck types that you can't with static types (unless you add in casting, see [Casting Types](#)).

```
static1 as int //type is fixed to be an
integer
dynamic1 as duck //can be anything

static1 = 0
dynamic1 = 0
print static1+1 //-> 1
print dynamic1+1 //-> 1

#static1 = "Some string" //error, cannot
convert string to int
dynamic1 = "Some string" //dynamic1 can be
"any" type of thing
```

```
#print static1.ToUpper() //error, static1 is
an int, not a string
print dynamic1.ToUpper() //-> SOME STRING

//You can convert a static type to a dynamic
duck type:
dynamic2 as duck = static1
print dynamic2 //-> 0
dynamic2 = "Some string"
print dynamic2.ToUpper() //-> SOME STRING

//or convert a dynamic type to a static type
static2 as string = dynamic1
print static2.ToUpper()
```

```
#static3 as int = dynamic1 //error, cannot
cast string to int
#print static3 + 2
```

If it helps, whenever you see "duck", think "any" or "anything". That object can be anything, *and* I can do whatever to it since it is duck typed and not static typed.

A Practical Example: Automating Internet Explorer via COM Interop

```
import System.Threading

def CreateInstance(progid):
    type =
System.Type.GetTypeFromProgID(progid)
    return type()

ie as duck =
CreateInstance("InternetExplorer.Application
")
ie.Visible = true
ie.Navigate2("http://www.go-mono.com/monolog
ue/")

Thread.Sleep(50ms) while ie.Busy

document = ie.Document
print "$(document.title) is
$(document.fileSize) bytes long."
```

See also how the example would look [without duck typing](#).

Implicit Duck Typing Option (-ducky)

You can replace "ie as duck" in the internet explorer example with "ie" and still have it run if you turn on the implicit duck typing option. See also the examples in the [tests/testcases/ducky/](#) folder.

You can set this option in various ways depending on which tool you are using:

- For the booi.exe and booc.exe command line tools, add a "-ducky" option when running those tools on the command line.
- In the Boo [Interactive Interpreter](#), implicit duck typing is turned on by default. You can turn it off by typing "interpreter.Ducky = false".
- In the [Boo AddIn For SharpDevelop](#) its off by default, but you'll see a "duck typing by default" option by selecting from the menu Project -> Project Options, selecting the Compiling tab, and then clicking on the configurations for your debug and release builds.

What this option does is basically make anything that is of type "object" be of type "duck" instead.

Turning on the ducky option allows you to more quickly test out code, and makes coding in boo feel much more like coding in Python. However, it slows down the speed of your application due to not using static typing. So when you are done testing, you may wish to turn the ducky option back off and add in various type declarations and casts. See [Casting Types](#).

Regular [Type Inference](#) is still in effect even when the ducky option is turned on, so if you declare "x = 4", then x is still an int. This allows your code to still be convertible to run without the ducky option by adding in the appropriate type declarations and casts.

Advanced: intercept duck typed method calls using IQuackFu

See

- [Dynamic Inheritance - fun with IQuackFu](#)
- [duck-5.boo](#)
- [XmlObject.boo](#)

The -ducky option now automatically treats any class that implements the IQuackFu interface as a duck type too.

CLS Compliance

Your compiled assembly will still be CLS compliant (usable from other languages) if you use duck types. "Duck" is only a virtual type during the compile process. Duck types are converted into regular, CLS-compliant "object" types in the assembly. Again, see what the IE example looks like [without duck typing](#) to get an idea of how the compiler converts your code. Also, IQuackFu is a regular interface. Not special at all except to the boo compiler.