Boo Documentation

Release 0.9.5

Rodrigo B. de Oliveira

February 09, 2016

Contents

Boo is a new object oriented statically typed programming language for the Common Language Infrastructure with a python inspired syntax and a special focus on language and compiler extensibility.

Boo

CHAPTER 1

FAQ

Frequently asked questions. Have a question? Post it here or to one of our Mailing Lists.

1.1 License

boo is licensed under a MIT/BSD style license. The current license is always available here.

1.2 How complete is boo at the present moment? What's its current status?

Boo is already usable for a large variety of tasks but there still are lots of things in our todo list.

1.3 Performance: since it is statically typed, can I expect a performance equal or close to c# or vb.net?

Yes.

1.4 How different is it from Python?

See Gotchas for Python Users for a summary.

1.5 Is it feasable to use boo for building desktop or asp.net applications?

Yes. Boo can already be used to implement WinForms/GTK# applications. Take a look at the extras/boox folder for an example.

On the asp.net front, thanks to Ian it's already possible to directly embed boo code inside asp.net pages, handlers or webservices. examples/asp.net should give you an idea of how everything works right now.

1.6 (Sharp|Mono)Develop bindings?

Daniel Grunwald has made great progress on the SharpDevelop front. Recent versions of SharpDevelop include solution creation in Boo.

As for MonoDevelop, it includes a Boo binding, written by Peter Johanson, leveraging the parser code written by Daniel Grunwald for the SharpDevelop binding. It includes Boo project creation/editing/compiling, as well as an interactive shell with Gtk# integration. See the monodevelop page for more information on installing it.

1.7 I see references on the site for .NET 1.1 and .NET 2.0, does Boo support .NET 3.0?

.NET 3.0 is actually just an update to the framework and not to the CLR or any of the "official" languages. As such, it should be supported by any .NET-2.0-supporting language, such as Boo.

Also, Boo release .78 is the last version of Boo that will support .NET 1.1.

1.8 What's a good way to get started with Boo (editors/IDEs)?

Fire up a console and check out booish - a built-in editor to check out the basics. Then grab a copy of Sharpdevelop or monodevelop to dive into developing with Boo.

1.9 What do people use for building 'real' Boo applications?

On Windows, Sharpdevelop is the most robust and stable IDE for developing BOO applications. Linux and Mac users develop with their favorite text editor. The monodevelop team are hard at work developing a more professional development environment that will support Boo along with other .NET languages.

1.10 When will version 1.0 be available?

When Boo is written in Boo it will be dubbed version 1.0.

Back to Home

Documentation

CHAPTER 2

Getting Started

- Warm up by reading the boo manifesto
- Download the latest distro or get the latest sources
- Check out some cool language features and the language reference.
- Learn how to edit, how to run and how to compile boo programs
- You might also be interested in building boo yourself
- Consider joining one of the mailing lists or the Boo Newsgroup
- Read and contribute some boo recipes and tutorials.
- boo enthusiasts hang out in the #boo IRC channel.
- Try the boo add-in for the free SharpDevelop IDE.
- Check out some applications coded in boo.
- Experienced programmers can check out the Boo Survival Guide at boo-contrib for quick-access to Boo information.

CHAPTER 3

Boo Primer

by Cameron Kenneth Knight

A version is also available in Portuguese, translated by Cássio Rogério Eskelsen.

A version is also available in Traditional Chinese, translated by Yan-ren Tsai aka Elleryq.

The english version is downloadable in a PDF.

3.1 Starting out

3.1.1 Overview

Boo is an amazing language that combines the syntactic sugar of Python, the features of Ruby, and the speed and safety of C#.

Like C#, Boo is a statically-typed language, which means that types are important. This adds a degree of safety that Python and other dynamically-typed languages do not currently provide.

It fakes being a dynamically-typed language by inference. This makes it seem much like Python's simple and programmer-friendly syntax.

CSharp

```
int i = 0;
MyClass m = new MyClass();
```

Boo

```
i = 0
m = MyClass()
```

3.1.2 Hello, World!

A Hello, World! program is very simple in Boo.

Don't worry if you don't understand it, I'll go through it one step at a time.

helloworld.boo

```
print "Hello, World!"
//    OR
print("Hello, World!")
```

Output

```
Hello, World!
Hello, World!
```

1. First, you must compile the helloworld.boo file to an executable.
2. Open up a new command line
3. `cd` into the directory where you placed the `helloworld.boo` file.
4. `booc helloworld.boo` (this assumes that Boo is installed and in your system path)
5. `helloworld.exe`
6. If you are using Mono, `mono helloworld.exe`
7. Using the print macro, it prints the string "Hello, World!" to the screen. OR
8. Using the print function, it prints the string "Hello, World!" to the screen.

Now these both in the end, do the same thing. They both call `System.Console.WriteLine("Hello, World")` from the .NET Standard Library.

And it's that simple.

Note: Using the macro version `print "Hello, World!"` is recommended.

3.1.3 Comparing code between Boo, C#, and VB.NET

Now you may be wondering how Boo could be as fast as C# or VB.NET.

Using their Hello World programs, I'll show you.

Boo

```
print "Hello World!"

// Output: Hello World!
```

CSharp

```
public class Hello
{
  public static void Main()
  {
    System.Console.WriteLine("Hello World!");
  }
```

```
}

// Output: Hello World!
```

VB.NET

```
Public Class Hello
  Public Shared Sub Main()
    System.Console.WriteLine("Hello World!")
  End Sub
End Class

' Output: Hello World!
```

All three have the same end result and all three are run in the .NET Framework.

All three are first translated into MSIL, then into executable files.

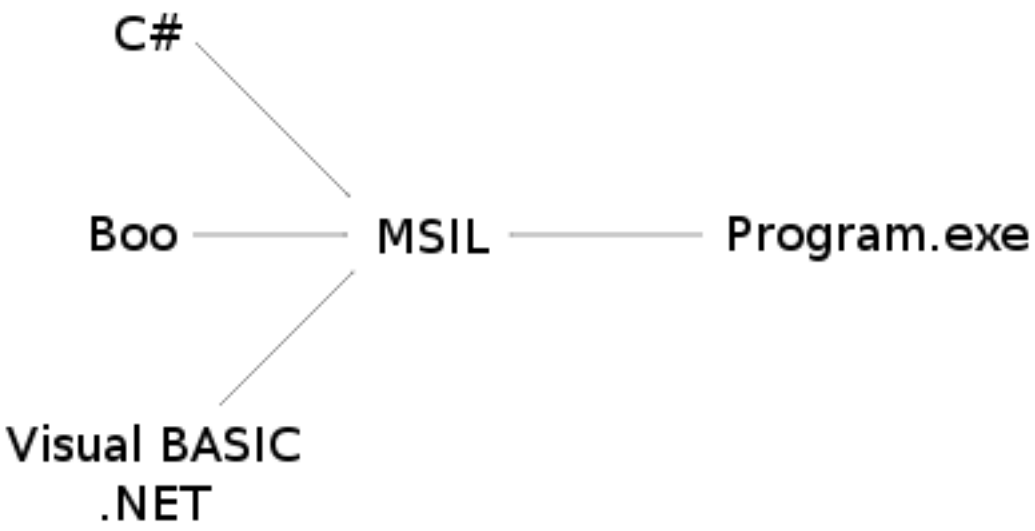

Fig. 3.1: MSIL Diagram

If you were to take the executables created by their compilers, and disassemble them with ildasm.exe, you would see a very similar end result, which means that the executables themselves are very similar, so the speed between C# and Boo is practically the same, it just takes less time to write the Boo code.

3.1.4 Booish

`booish` is a command line utility that provides a realtime environment to code boo in. It is great for testing purposes, and I recommend following along for the next few pages by trying out a few things in `booish`. You can invoke it by loading up a terminal, then typing booish (this assumes that Boo is installed and in your system path), or by

double-clicking the booish icon. In `booish`, you can up arrow to re-enter (with editing) a previously entered line. Here's what `booish` will look like:

Booish

```
>>> print "Hello, World!"
Hello, World!
```

3.1.5 Exercises

1. Write a Boo program that prints `Hello, World!`, then prints `Goodbye, World!`
2. Play around with `booish`
3. Advanced: Compile the Hello, World! programs for Boo (using `booc`) and C# (using `csc` or `mcs`), run `ildasm` on each of them and compare the result.

3.2 Variables

3.2.1 Using Booish as a Calculator

There are four basic mathematical operators: addition +, subtraction -, multiplication *, and division /. There are more than just these, but that's what will be covered now.

```
>>> 2 + 4 // This is a comment
6
>>> 2 * 4 # So is this also a comment
8
>>> 4 / 2 /* This is also a comment */
2
>>> (500/10)*2
100
>>> 7 / 3 // Integer division
2
>>> 7 % 3 // Take the remainder
1
>>> -7 / 3
-2
```

You may have noticed that there are 3 types of comments available, //, #, and /* */. These do not cause any affect whatsoever, but help you when writing your code.

Note: When doing single-line comments, use // instead of #

You may have noticed that 7 / 3 did not give 2.333..., this is because you were dividing two integers together.

Note: **Integer** - Any positive or negative number that does not include a fraction or decimal, including zero.

The way computers handle integer division is by rounding to the floor afterwards.

In order to have decimal places, you must use a floating-point number.

Note: **Floating point Number** - Often referred to in mathematical terms as a "rational" number, this is just a number that can have a fractional part.

```
>>> 7.0 / 3.0  # Floating point division
2.33333333333333
>>> -8.0 / 5.0
-1.6
```

If you give a number with a decimal place, even if it's .0, it become a floating-point number.

3.2.2 Types of Numbers

There are 3 kinds of floating point numbers, `single`, `double`, and `decimal`.

The differences between `single` and `double` is the size they take up. `double` is prefered in most situations.

These two also are based on the number 2, which can cause some problems when working with our base-10 number system.

Ususally this is not the case, but in delecate situations like banking, it would not be wise to lose a cent or two on a multi-trillion dollar contract.

Thus `decimal` was created. It is a base-10 number, which means that we wouldn't lose that precious penny.

In normal situations, `double` is perfectly fine. For a higher precision, a `decimal` should be used. Integers, which we covered earlier, have many more types to them.

They also have the possibility to be "unsigned", which means that they must be non-negative.

The size goes in order as such: `byte`, `short`, `int`, and `long`.

In most cases, you will be using `int`, which is the default.

3.2.3 Characters and Strings

Note: **Character** - A written symbol that is used to represent speech.

All the letters in the alphabet are characters. All the numbers are characters. All the symbols of the Mandarin language are characters. All the mathematical symbols are characters.

In Boo/.NET, characters are internally encoded as UTF-16, or Unicode.

Note: **String** - A linear sequence of characters.

The word "apple" can be represented by a `string`.

```
>>> s = "apple"
'apple'
>>> print s
apple
>>> s += " banana"
'apple banana'
>>> print s
apple banana
```

```
>>> c = char('C')
C
>>> print c
C
```

Now you probably won't be using `chars` much, it is more likely you will be using `strings`.

To declare a `string`, you have one of three ways.

1. using double quotes. "apple"
2. using single quotes. 'apple'
3. using tripled double quotes. """apple"""

The first two can span only one line, but the tribbled double quotes can span multiple lines.

The first two also can have backslash-escaping. The third takes everything literally.

```
>>> print "apple\nbanana"
apple
banana
>>> print 'good\ttimes'
good    times
>>> print """Leeroy\Jenkins"""
Leeroy\Jenkins
```

Common escapes are: * {{ ... }} literal backslash * newline * carriage return * tab

If you are declaring a double-quoted `string`, and you want a double quote inside it, also use a backslash.

Same goes for the single-quoted `string`.

```
>>> print "The man said \"Hello\""
The man said "Hello"
>>> print 'I\'m happy'
I'm happy
```

`strings` are immutable, which means that the characters inside them can never change. You would have to recreate the `string` to change a `character`.

Note: **Immutable** - Not capable of being modified after it is created. It is an error to attempt to modify an immutable object. The opposite of immutable is mutable.

3.2.4 String Interpolation

String interpolation allows you to insert the value of almost any valid boo expression inside a `string` by preceeding a lonesome variable name with $, or quoting an expression with $().

```
>>> name = "Santa Claus"
Santa Claus
>>> print "Hello, $name!"
Hello, Santa Claus!
>>> print "2 + 2 = $(2 + 2)"
2 + 2 = 4
```

String Interpolation is the preferred way of adding `strings` together. It is preferred over simple `string` addition.

String Interpolation can function in double-quoted `strings`, including tripled double-quoted `string`.

It does not work in single-quoted `strings`.

To stop String Interpolation from happening, just escape the dollar sign: ${}

3.2.5 Booleans

Note: **Boolean** - A value of `true` or `false` represented internally in binary notation.

Boolean values can only be `true` or `false`, which is very handy for conditional statements, covered in the next section.

```
>>> b = true
true
>>> print b
True
>>> b = false
false
>>> print b
False
```

3.2.6 Object Type

Note: **Object** - The central concept in the object-oriented programming paradigm.

Everything in Boo/.NET is an object.

Although some are value types, like numbers and characters, these are still objects.

```
>>> o as object
>>> o = 'string'
'string'
>>> print o
string
>>> o = 42
42
>>> print o
42
```

The problem with `objects` is that you can't implicitly expect a `string` or an `int`.

If I were to do that same sequence without declaring `o as object`,

```
>>> o = 'string'
'string'
>>> print o
string
>>> o = 42
--------^
ERROR: Cannot convert 'System.Int32' to 'System.String'.
```

This static typing keeps the code safe and reliable.

3.2.7 Declaring a Type

In the last section, you issued the statement `o as object`.

This can work with any type and goes with the syntax `<variable> as <type>`.

`<type>` can be anything from an `int` to a `string` to a `date` to a `bool` to something which you defined yourself, but those will be discussed later. In most cases, Boo will be smart and implicitly figure out what you wanted.

The code `i = 25` is the same thing as `i as int = 25`, just easier on your wrists.

Note: Unless you are declaring a variable beforehand, or declaring it of a different type, don't explicitly state what kind of variable it is. (ie: use `i = 25` instead of `i as int = 25`)

3.2.8 List of Value Types

Boo type	.Net Framework type	Signed?	Size	Possible Values
sbyte	System.Sbyte	Yes	1	-128 to 127
short	System.Int16	Yes	2	-32768 - 32767
int	System.Int32	Yes	4	-2147483648 - 2147483647
long	System.Int64	Yes	8	-9223372036854775808 - 9223372036854775807
byte	System.Byte	No	1	0 - 255
ushort	System.Uint16	No	2	0 - 65535
uint	System.UInt32	No	4	0 - 4294967295
ulong	System.Uint64	No	8	0 - 18446744073709551615
single	System.Single	Yes	4	Approximately ±1.5 x 10-45 - ±3.4 x 1038 with 7 significant figures
double	System.Double	Yes	8	Approximately ±5.0 x 10-324 - ±1.7 x 10308 with 15 or 16 significant figures
decimal	System.Decimal	Yes	12	Approximately ±1.0 x 10-28 - ±7.9 x 1028 with 28 or 29 significant figures
char	System.Char	N/A	2	Any UTF-16 character
bool	System.Boolean	N/A	1	true or false

Note: Never call a type by its .NET Framework type, use the builtin boo types.

3.2.9 Exercises

1. Declare some variables. Go wild.

3.3 Flow control and Conditionals

3.3.1 If Statement

Note: **if statement** - A control statement that contains one or more Boolean expressions whose results determine whether to execute other statements within the If statement.

An `if` statemnt allows you to travel down multiple logical paths, depending on a condition given. If the condition given is `true`, the block of code associated with it will be run.

if statement

```
i = 5
if i == 5:
    print "i is equal to 5."

// Output: i is equal to 5.
```

Warning: notice the difference between `i = 5` and `i == 5`. The former is an assignment while the latter is a comparison.

If you try an assignment while running a conditional, Boo will emit a warning.

You may have noticed that unlike other languages, there is no then-endif or do-end or braces { }. Blocks of code are determined in Boo by its indentation. By this, your code blocks will always be noticeable and readable.

Note: Always use tabs for indentation. In your editor, set the tab-size to view as 4 spaces.

You can have multiple code blocks within eachother as well.

Multiple if statements

```
i = 5
if i > 0:
    print "i is greater than 0."
    if i < 10:
        print "i is less than 10."
        if i > 5:
            print "i is greater than 5."

// Output: i is greater than 0.
//         i is less than 10.
```

3.3.2 If-Else Statement

With the `if` statement comes the `else` statement. It is called when your `if` statement's condition is `false`.

if-else statement

```
i = 5
if i > 5:
    print "i is greater than 5."
else:
    print "i is less than or equal to 5."

// Output: i is less than or equal to 5.
```

Quite simple.

3.3.3 If-Elif-Else Statement

Now `if` you want to check for a condition after your `if` is `false`, that is easy as well. This is done through the `elif` statement.

if-elif-else statement

```
i = 5
if i > 5:
    print "i is greater than 5."
elif i == 5:
    print "i is equal to 5."
elif i < 5:
    print "i is less than 5."

// Output: i is equal to 5.
```

You can have one `if`, any number of `elif` s, and an optional `else`.

3.3.4 Unless Statement

The `unless` statement is handy if you want a readable way of checking if a condition is **not** true.

unless statement

```
i = 5
unless i == 5:
    print "i is not equal to 5."

// Output:
```

It didn't output because i was equal to 5 in that case.

3.3.5 Statement with Modifier

Like in Ruby and Perl, you can follow a statement with a modifier.

‘‘‘boo i = 0 print i i = 5 if true print i i = 10 unless true print i

// Output: 0 // 5 // 5 ‘‘‘

Note: Don't use Statement with Modifier on a long line. In that case, you should just create a code block.

A good rule of thumb is to not use it if the statement is more than 3 words long.

This will keep your code readable and beautiful.

Some common conditionals:

Operator	Meaning	Example
==	equal	5 == 5
!=	not equal	0 != 5
>	greater than	4 > 2
<	less than	2 < 4
>=	greater than or equal to	7 >= 7 and 7 >= 4
<=	less than or equal to	4 <= 8 and 6 <= 6

3.3.6 Not Condition

To check if a condition is not true, you would use `not`.

not condition

```
i = 0
if not i > 5:
    print "i is not greater than 5"

// Output: i is not greater than 5
```

3.3.7 Combining Conditions

To check more than one condition, you would use `and` or `or`. Use parentheses ( ) to change the order of operations.

```
i = 5
if i > 0 and i < 10:
    print "i is between 0 and 10."
if i < 3 or i > 7:
    print "i is not between 3 and 7."
if (i > 0 and i < 3) or (i > 7 and i < 10):
    print "i is either between 0 and 3 or between 7 and 10."

// Output: i is between 0 and 10.
```

Note that `and` requires that both comparisons are true, while `or` requires that only one is true or both are true.

3.3.8 Exercises

1. Given the numbers x = 4, y = 8, and z = 6, compare them and print the middle one.

3.4 Flow control and Loops

3.4.1 For Loop

Note: **For loop** - A loop whose body gets obeyed once for each item in a sequence.

A `for` loop in Boo is not like the `for` loop in languages like C and C#. It is more similar to a `foreach`.

The most common usage for a `for` loop is in conjunction with the `range` function.

The `range` function creates an enumerator which yields numbers.

The `join` function in this case, will create a string from an enumerator.

join and range example

```
join(range(5))          // Output: 0 1 2 3 4
join(range(3, 7))       // Output: 3 4 5 6
join(range(0, 10, 2))   // Output: 0 2 4 6 8
```

`range` can be called 3 ways:

- `range(end)`
- `range(start, end)`
- `range(start, end, step)`

To be used in a `for` loop is quite easy.

for loop

```
for i in range(5):
    print i

// Output: 0
//         1
//         2
//         3
//         4
```

Note: **Practically as fast as C#'s** - The `range` function does not create an array holding all the values called, instead it is an `IEnumerator`, that will quickly generate the numbers you need.

3.4.2 While Loop

Note: **While loop** - A structure in a computer program that allows a sequence of instructions to be repeated while some condition remains true.

The `while` loop is very similar to an `if` statement, except that it will repeat itself as long as its condition is true.

while loop

```
i = 0
while i < 5:
    print i
    i += 1

// Output: 0
//         1
//         2
```

```
//          3
//          4
```

In case you didn't guess, `i += 1` adds 1 to `i`.

3.4.3 Continue Keyword

Note: **Continue keyword** - A keyword used to resume program execution at the end of the current loop.

The `continue` keyword is used when looping. It will cause the position of the code to return to the start of the loop (as long as the condition still holds).

continue statement

```
for i in range(10):
    continue if i % 2 == 0
    print i

// Output: 1
//          3
//          5
//          7
//          9
```

This skips the `print` part of this loop whenever `i` is even, causing only the odds to be printed out.

The `i % 2` actually takes the remainder of `i` / 2, and checks it against 0.

3.4.4 While-Break-Unless Loop

the `while-break-unless` loop is very similar to other languages `do-while` statement.

while-break-unless loop

```
i = 10
while true:
    print i
    i -= 1
    break unless i < 10 and i > 5

// Output: 10
//          9
//          8
//          7
//          6
//          5
```

Normally, this would be a simple while loop.

This is a good method of doing things if you want to accomplish something at least once or have the loop set itself up.

3.4.5 Pass Keyword

The `pass` keyword is useful if you don't want to accomplish anything when defining a code block.

pass statement

```
while true:
    pass //Wait for keyboard interrupt (ctrl-C) to close program.
```

3.4.6 Exercises

1. print out all the numbers from 10 to 1.
2. print out all the squares from 1 to 100.

3.5 Containers and Casting

3.5.1 Lists

Note: **List** - A linked list that can hold a variable amount of objects.

`Lists` are mutable, which means that the `List` can be changed, as well as its children.

lists

```
print([0, 'alpha', 4.5, char('d')])
print List('abcdefghij')
l = List(range(5))
print l
l[2] = 5
print l
l[3] = 'banana'
print l
l.Add(100.1)
print l
l.Remove(1)
print l
for item in l:
    print item

// Output:
// [0, alpha, 4.5, d]
// [a, b, c, d, e, f, g, h, i, j]
// [0, 1, 2, 3, 4]
// [0, 1, 5, 3, 4]
// [0, 1, 5, 'banana', 4]
// [0, 1, 5, 'banana', 4, 100.1]
// [0, 5, 'banana', 4, 100.1]
// 0
// 5
```

```
// 'banana'
// 4
// 100.1
```

As you can see, `Lists` are very flexible, which is very handy.

`Lists` can be defined two ways: 1. by using brackets [] 2. by creating a new `List` wrapping an `IEnumerator`, or an `array`.

3.5.2 Slicing

Slicing is quite simple, and can be done to `strings`, `Lists`, and `arrays`.

It goes in the form `var[start:end]`. both start and end are optional, and must be integers, even negative integers.

To just get one child, use the form `var[position]`. It will return a `char` for a `string`, an `object` for a `List`, or the specified type for an `array`.

Slicing counts up from the number 0, so 0 would be the 1st value, 1 would be the 2nd, and so on.

slicing

```
list = List(range(10))
print list
print list[:5]    // first 5
print list[2:5]   // starting with 2nd, go up to but not including the 5
print list[5:]    // everything past the 5th
print list[:-2]   // everything up to the 2nd to last
print list[-4:-2] // starting with the 4th to last, go up to 2nd to last
print list[5]     // the 6th
print list[-8]    // the 8th from last
print '---'
str = 'abcdefghij'
print str
print str[:5]     // first 5
print str[2:5]    // starting with 3rd, go up to but not including the 6th
print str[5:]     // everything past the 6th
print str[:-2]    // everything before the 2nd to last
print str[-4:-2]  // starting with the 4th to last, to before the 2nd to last
print str[5]      // the 6th
print str[-8]     // the 8th from last

// Output
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
// [0, 1, 2, 3, 4]
// [2, 3, 4]
// [5, 6, 7, 8, 9]
// [0, 1, 2, 3, 4, 5, 6, 7]
// [6, 7]
// 5
// 2
// ---
// abcdefghij
// abcde
// cde
// fghij
// abcdefgh
```

```
// gh
// f
// c
```

I hope you get the idea. Slicing is very powerful, as it allows you to express what you need in a minimal amount of space, while still being readable.

3.5.3 Arrays

Note: **Array** - Arrays are simple objects that hold equally-sized data elements, generally of the same data type.

`Arrays`, unlike `Lists`, cannot change their size. They can still be sliced, just not added on to. `Arrays` can be defined three ways: 1. by using parentheses () 1. If you have 0 members, it's declared: `(,)` 2. If you have 1 member, it's declared: `(member,)` 3. If you have 2 or more members, it's declared: `(one, two)` 2. by creating a new `array` wrapping an `IEnumerator`, or an `List`. 3. by creating a blank `array` with a specified size: `array(type, size)`

arrays

```
print((0, 'alpha', 4.5, char('d')))
print array('abcdefghij')
l = array(range(5))
print l
l[2] = 5
print l
l[3] = 'banana'

// Output
// (0, alpha, 4.5, d)
// (a, b, c, d, e, f, g, h, i, j)
// (0, 1, 2, 3, 4)
// (0, 1, 5, 3, 4)
// ERROR: Cannot convert 'System.String' to 'System.Int32'.
```

`Arrays`, unlike `Lists`, do not necessarily group `objects`. They can group any type, in the case of `array(range(5))`, it made an `array` of `ints`.

3.5.4 List to Array Conversion

If you create a `List` of `ints` and want to turn it into an `array`, you have to explicitly state that the `List` contains `ints`.

list to array conversion

```
list = []
for i in range(5):
    list.Add(i)
    print list
a = array(int, list)
for a_s in a:
    print a_s
```

```
a[2] += 5
print a
list[2] += 5
print list[2]

// Output
// [0]
// [0, 1]
// [0, 1, 2]
// [0, 1, 2, 3]
// [0, 1, 2, 3, 4]
// (0, 1, 2, 3, 4)
// (0, 1, 7, 3, 4)
// ERROR: Operator '+' cannot be used with a left-hand side of type 'System.Object' and a right-hand
```

This didn't work, because the List still gives out `objects` instead of `ints`, even though it only holds `ints`.

3.5.5 Casting

Note: **Typecast** - The conversion of a variable's data type to another data type to bypass some restrictions imposed on datatypes.

To get around a list storing only `objects`, you can `cast an object` individually to what its type really is, then play with it like it should be.

Granted, if you cast to something that is improper, say a `string` to an `int`, Boo will emit an error. There are two ways to cast an object as another data type. 1. using `var as <type>` 2. using `var cast <type>`

casting example

```
list = List(range(5))
print list
for item in list:
    print ((item cast int) * 5)
print '---'
for item as int in list:
    print item * item

// Output
// [0, 1, 2, 3, 4]
// 0
// 5
// 10
// 15
// 20
// ---
// 0
// 1
// 4
// 9
// 16
```

Note: Try not to cast too much. If you are constantly cast-ing, think if there is a better way to write the code.

Note: **Generics** - Generics, which has been part of the .NET Framework since 2.0, will allow you to create a List with a specified data type as its base. So there is a way to not have to cast a List's items every time.

3.5.6 Hashes

Note: **Hash** - A `List` in which the indices may be `objects`, not just sequential integers in a fixed range.

`Hashes` are also called "dictionaries" in some other languages. `Hashes` are very similar to `Lists`, except that the key in which to set values can be an `object`, though usually an `int` or a `string`. `Hashes` can be defined two common ways: 1. by using braces {} 2. by creating a new `Hash` wrapping an `IEnumerator`, or an `IDictionary`.

hash example

```
hash = {'a': 1, 'b': 2, 'monkey': 3, 42: 'the answer'}
print hash['a']
print hash[42]
print '---'
for item in hash:
    print item.Key, '=>', item.Value

# the same hash can be created from a list like this :
ll = [ ('a',1), ('b',2), ('monkey',3), (42, "the answer") ]
hash = Hash(ll)

// Output
// 1
// the answer
// ---
// a => 1
// b => 2
// monkey => 3
// 42 => the answer
```

3.5.7 Exercises

1. Produce a List containing the fibonacci sequence that has 1000 values in it. (See if you can do it in 4 lines)

For Developers

CHAPTER 4

Contributors

There are many ways you can make boo better.

As an user you can contribute by reporting any issues you find, by adding, commenting on and/or fixing the documentation on this website (see that little edit link down the page?), by helping other users through one of our mailing lists and irc channel.

Feel like hacking today? Get the latest sources from our repository and see what you think.

Browse the list of open issues, see anything you like? Take ownership and try to fix it!

4.1 Getting in touch

subscribe to any of our Mailing Lists stop by the boo irc channel: irc://irc.codehaus.org/boo add comments to the pages in our wiki add comments to and/or vote for the issues in our issue tracker If you find a bug or problem Please raise an issue in our issue tracker.

If you can provide a test case then your issue is more likely to be resolved quicker.

4.2 Sending patches

We gladly accept patches if you can find ways to improve, tune or fix boo in some way.

The basic process goes like this:

check out the code from the git repository or update you working copy optionally write new test cases or change existing ones git diff your changes to a file attach that file to the related issue or create a new issue including the details on what's being improved, tuned or fixed Always try to make sure that all the unit tests that were passing before the code changes are still green after them.

Code eventually committed to the repository must follow our coding conventions.

4.3 Using the issue tracker

Before you can raise an issue in the issue tracker you need to register with it. This is quick & painless.

If you want to have a go at fixing an issue you need to be in the list of boo-developers on the issue tracker. To join the group, please mail dev@boo.codehaus.org with the username you used to register with the issue tracker and we'll add you to the group.

4.4 Helping with the documentation

Before you start adding comments to and/or editing the pages on this website you need to register with confluence. This is also quick & painless.

If you want to fix and/or add new pages you need to be in the list of boo-developers on confluence. To join the group, please mail dev@boo.codehaus.org with the username you used to register with confluence and we'll add you to the group.

- Boo
- FAQ
- Manifesto
- Documentation
- Getting Started
- Boo Primer
- Language Guide
- Tutorials
- Cookbook
- Still Need Help?
- For Developers
- Contributors 1 _build/
- Writing Tests
- Coding Conventions
- Boo Style Checker
- Features
- Type Inference
- Closures
- Functions As Objects
- Interactive Interpreter
- Builtin Literals
- Slicing
- String Interpolation
- Syntactic Macros
- Generators
- Community
- Mailing Lists
- IRC Channel

CHAPTER 5

Indices and tables

- genindex
- modindex
- search