Boo Primer

by Cameron Kenneth Knight

Converted to PDF from the original HTML at boo.codehaus.org on May 27, 2008.

NOTE: Hyperlinks do not work in the PDF version.

Part 01 - Starting Out

Part 01 - Starting Out

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.

C#

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

Boo

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

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

Recommendation

Using the macro version (print "Hello, World!") is recommended.

And it's that simple.

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!"
```

Boo Output

```
Hello, World!
```

C#

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

C# Output

```
Hello, World!
```

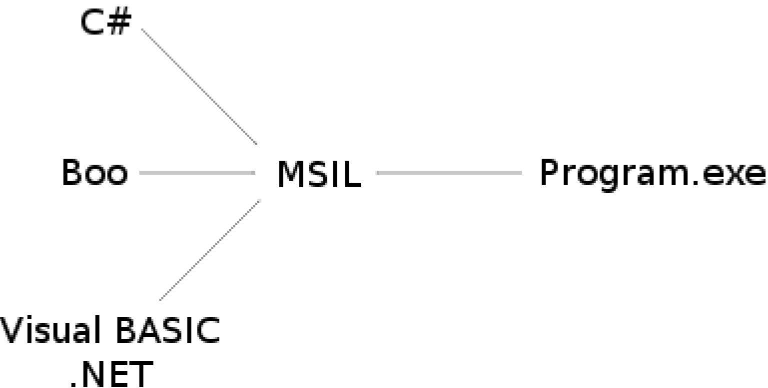
VB.NET

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

VB.NET 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.



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.

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)

Here's what booish will look like:

booish

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

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.

Go on to **Part 02 - Variables**

Part 02 - Variables

Part 02 - Variables

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.

```
$ booish
>>> 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.

Recommendation

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.

Definition: 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.

Definition: Floating=point Number

Often referred to in mathematical terms as a "rational" number, this is just a number that can have a fractional part.

```
$ booish
>>> 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.

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.

Characters and Strings

Definition: 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.

Definition: String

A linear sequence of characters.

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

```
$ booish
>>> 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.

```
$ booish
```

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

Common escapes are:

- {{
  }} literal backslash
- \n newline
- \r carriage return
- \t 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.

  ```
  $ booish
  >>> 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.

Definition: 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.

String Interpolation

String interpolation allows you to insert the value of almost any valid boo expression inside astring by quoting the expression with ${}.

```
$ booish
>>> 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: \${}

Booleans

Definition: 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.

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

Object Type

Definition: Object

The central concept in the object-oriented programming programming paradigm.

Everything in Boo/.NET is an object.
Although some are value types, like numbers and characters, these are still objects.

```
$ booish
>>> 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,

```
$ booish
>>> 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.

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.

Recommendation

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

List of Value Types

Boo type	.NET Framework type	Signed?	Size in bytes	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

Recommendation

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

Exercises

1. Declare some variables. Go wild.

Go on to **Part 03 - Flow Control - Conditionals**

Part 03 - Flow Control - Conditionals

Part 03 - Flow Control - Conditionals

If Statement

Definition: 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.
```

Be careful

notice the difference between i = 5 and i == 5.
i = 5 is an *assignment*
i == 5 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.

Recommendation
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.
```

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.

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 .

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 equal to 5."
```

Output

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

Statement with Modifier

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

Statement with modifier

```
i = 0
print i
i = 5 if true
print i
i = 10 unless true
print i
```

Output

```
0
5
5
```

Recommendation

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

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
```

Combining Conditions

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

combining conditions

```
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."
```

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

Output

```
i is between 0 and 10.
```

Exercises

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

Go on to **Part 04 - Flow Control - Loops**

Part 04 - Flow Control - Loops

Part 04 - Flow Control - Loops

For Loop

Definition: 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 conjuction 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))
join(range(3, 7))
join(range(0, 10, 2))
```

Output

```
0 1 2 3 4
3 4 5 6
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
```

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.

While Loop

Definition: 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.

Continue Keyword

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.

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.

Pass Keyword

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

while-break-unless loop

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

Exercises

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

Go on to **Part 05 - Containers and Casting**

Part 05 - Containers and Casting

Part 05 - Containers and Casting

Lists

Definition: 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.

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 2nd, go up to but not including the 5
print str[5:] // everything past the 5th
print str[:-2] // everything up to the 2nd to last
print str[-4:-2] // starting with the 4th to last, go up to 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.

Arrays

Definition: 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 ()
   - If you have 0 members, it's declared: (,)
   - If you have 1 member, it's declared: (member,)
   - 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.

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

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 side of type 'System.Int32
```

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

Casting

Definition: 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 cast(<type>, var)

casting example

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

Output

```
0
5
10
15
20
---
0
1
4
9
16
```

Recommendation

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

Upcoming feature: Generics

Generics, which will be part of the .NET Framework 2.0, will allow you to create a List with a specified data type as its base.
So soon enough, there will be a way to not have to cast a List's items every time.

Hashes

Definition: 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
```

Exercises

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

Go on to **Part 06 - Operators**

Part 06 - Operators

Part 06 - Operators

Mathematical

Name	Syntax example	Comments
Multiplication	a * b	
Division	a / b	
Remainder	a % b	Often called mod or modulus
Addition	a + b	
Subtraction	a - b	
Exponent	a ** b	Do not confuse this with Bitwise Xor ^
Bitshift Right	a >> b	
Bitshift Left	a << b	
Bitwise And	a & b	
Bitwise Or	a | b	
Bitwise Xor	a ^ b	

The mathematical operators can also be used in the syntax a <operator>= b, for example, a += b. This is merely a shortcut for a = a <operator> b, or in our example a = a + b.

Relational and Logical

Name	Syntax Example	Comments
Less Than	a < b	
Greater Than	a > b	
Less Than or Equal To	a <= b	
Greater Than or Equal To	a >= b	
Equality	a == b	
Inequality	a != b	
Logical And	a and b	Only use when a and b are boolean values
Logical Or	a or b	Only use when a and b are boolean values
Logical Not	not a	Only use when a is a boolean value

Types

Name	Syntax Example	Comments
Typecast	cast(string, a)	
Typecast	a as string	
Type Equality/Compatibility	a isa string	
Type Retrieval	typeof(string)	
Type Retrieval	a.GetType()	

Primary

Name	Syntax Example	Comments
Member	A.B	Classes are described in Part 08 - Classes
Function Call	f(x)	Functions are described in Part 07 - Functions
Post Increment	i++	See Difference between Pre and Post Increment/Decrement
Post Decrement	i--	See Difference between Pre and Post Increment/Decrement
Constructor Call	o = MyClass()	Classes are described in Part 08 - Classes

Unary

Name	Syntax Example	Comments
Negative Value	-5	

Pre Increment	++i	See Difference between Pre and Post Increment/Decrement
Pre Decrement	--i	See Difference between Pre and Post Increment/Decrement
Grouping	(a + b)	

Difference between Pre and Post Increment/Decrement

When writing inline code, Pre Increment/Decrement (++i/--i) commit the action, then return its new value, whereas Post Increment/Decrement (i++/i--) return the current value, then commit the change.

preincrement vs. postincrement

```
num = 0
for i in range(5):
    print num++
print '---'
num = 0
for i in range(5):
    print ++num
```

Output

```
0
1
2
3
4
---
1
2
3
4
5
```

Recommendation

To make your code more readable, avoid using the incrementors and decrementors.
Instead, use i += 1 and i -= 1.

Exercises

1. Put your hands on a wall, move your left foot back about 3 feet, move the right foot back 2 feet.

Go on to **Part 07 - Functions**

Part 07 - Functions

Functions

Definition: Function

A sequence of code which performs a specific task, as part of a larger program, and is grouped as one, or more, statement blocks

Builtin Functions

You have already seen a few functions. range(), print(), and join().
These are functions built into Boo.
Here's a list of all the builtin functions that Boo offers:

Name	Description	Syntax example
print	Prints an object to Standard Out. The equivilent of System.Console.WriteLine	print("hey")
gets	Returns a string of input that originates from System.Console.ReadLine() - Standard Input	input = gets()
prompt	Prompts the user for information.	input = prompt("How are you? ")
join	Walk through an IEnumerable object and put all of those elements into one string.	join([1, 2, 3, 4, 5]) == "1 2 3 4 5"
map	Returns an IEnumerable object that applies a specific function to each element in another IEnumerable object.	map([1, 2, 3, 4, 5], func)
array	Used to create an empty array or convert IEnumerable and ICollection objects to an array	array(int, [1, 2, 3, 4, 5]) == (1, 2, 3, 4, 5)
matrix	Creates a multidimensional array. See Multidimensional Arrays for more info.	matrix(int, 2, 2)
iterator	Creates an IEnumerable from an object	List(iterator('abcde')) == ['a', 'b', 'c', 'd', 'e']
shellp	Start a Process. Returns a Process object	process = shellp("MyProgram.exe", "")
shell	Invoke an application. Returns a string containing the program's output to Standard Out	input = shell("echo hi there", "")
shellm	Execute the specified managed application in a new AppDomain. Returns a string containing the program's output to Standard Out	input = shellm("MyProgram.exe", (,))
enumerate	Creates an IEnumerator from another, but gives it a pairing of (*index*, *value*)	List(enumerate(range(5, 8))) == [(0, 5), (1, 6), (2, 7)]
range	Returns an IEnumerable containing a list of ints	List(range(5)) == [0, 1, 2, 3, 4]
reversed	Returns an IEnumerable with its members in reverse order	List(reverse(range(5))) == [4, 3, 2, 1, 0]
zip	Returns an IEnumerable that is a "mesh" of two or more IEnumerables.	array(List([1, 2, 3], [4, 5, 6])) == [(1, 4), (2, 5), (3, 6)]
cat	Concatenate two or more IEnumerators head-to-tail	List(cat(range(3), range(3, 6)) == [0, 1, 2, 3, 4, 5]

These are all very handy to know. Not required, but it makes programming all that much easier.

Defining Your Own Functions

It's very simple to define your own functions as well.

declaring a function

```
def Hello():
    return "Hello, World!"

print Hello()
```

Output

```
Hello, World!
```

Now it's ok if you don't understand any of that, I'll go through it step-by-step.

1. def Hello():
   - def declares that you are starting to declare a function. def is short for "define".
   - Hello is the name of the function. You could call it almost anything you wanted, as long as it doesn't have any spaces and doesn't start with a number.
   - () this means what kind of arguments the function will take. Since we don't accept any arguments, it is left blank.
   1. return "Hello, World!"
      - return is a keyword that lets the function know what to emit to its invoker.
      - "Hello, World!" is the string that the return statement will send.
2. print Hello()
   - print is the happy little print macro that we covered before.
   - Hello() calls the Hello function with no () arguments.

Like variables, function types are inferred.

```
def Hello():
    return "Hello, World!"
```

will always return a string, so Boo will will infer that string is its return type. You could have done this to achieve the same result:

```
def Hello() as string:
    return "Hello, World!"
```

Recommendation

If it is not obvious, specify the return type for a function.

If Boo cannot infer a return type, it will assume object. If there is no return type then the return type is called 'void', which basically means nothing. To have no return type you can leave off the return, or have a return with no expression. If there are multiple return}}s with different {{return types, it will return the closest common ancestor, often times object but not always.

Arguments

Definition: Argument

A way of allowing the same sequence of commands to operate on different data without re-specifying the instructions.

Arguments are very handy, as they can allow a function to do different things based on the input.

arguments example

```
def Hello(name as string):
    return "Hello, ${name}!"

print Hello("Monkey")
```

Output

```
Hello, Monkey!
```

Here it is again, step-by-step.

1. def Hello(name as string):
    - def declares that you are starting to declare a function.
    - Hello is the name of the function. You could call it almost anything you wanted, as long as it doesn't have any spaces and doesn't start with a number.
    - (name as string) this means what kind of arguments the function will take. This function will take one argument: name. When you call the function, the name must be a string, otherwise you will get a compiler error - "The best overload for the method Hello is not compatible with the argument list '(The,Types, of, The, Parameters, Entered)'."
    1. return "Hello, ${name}!"
        - return is a keyword that exits the function, and optionally return a value to the caller.
        - "Hello, ${name}!" uses String Interpolation to place the value of name directly into the string.
2. print Hello("Monkey")
    - print is the happy little print macro that we covered before.
    - Hello("Monkey") calls the Hello function with the ("Monkey") argument.

Function Overloading

Definition: Overloading

Giving multiple meanings to the same name, but making them distinguishable by context. For example, two procedures with the same name are overloading that name as long as the compiler can determine which one you mean from contextual information such as the type and number of parameters that you supply when you call it.

Function overloading takes place when a function is declared multiple times with different arguments.

overloading example

```
def Hello():
    return "Hello, World!"

def Hello(name as string):
    return "Hello, ${name}!"

def Hello(num as int):
    return "Hello, Number ${num}!"

def Hello(name as string, other as string):
    return "Hello, ${name} and ${other}!"

print Hello()
print Hello("Monkey")
print Hello(2)
print Hello("Cat", "Dog")
```

Output

```
Hello, World!
Hello, Monkey!
Hello, Number 2!
Hello, Cat and Dog!
```

Variable Arguments

There is a way to pass an arbitrary number of arguments.

variable arguments example

```
def Test(*args as (object)):
    return args.Length

print Test("hey", "there")
print Test(1, 2, 3, 4, 5)
print Test("test")

a = (5, 8, 1)
```

```
print Test(*a)
```

Output

```
2
5
1
3
```

The star * lets it known that everything past that is arbitrary.
It is also used to explode parameters, as in print Test(*a) causes 3 arguments to be passed.
You can have required parameters before the *args, just like in any other function, but not after, as after all the required parameters are supplied the rest are past into your argument array.

Exercises

1. Write a function that prints something nice if it is fed an even number and prints something mean if it is fed an odd number.

Go on to **Part 08 - Classes**

Part 08 - Classes

Part 08 - Classes

Definition: Class

A cohesive package that consists of a particular kind of compile-time metadata. A class describes the rules by which objects behave. A class specifies the structure of data which each instance contains as well as the methods (functions) which manipulate the data of the object.

Definition: Object

An instance of a class

Defining a Class

Classes are important because they allow you to split up your code into simpler, logical parts. They also allow for better organization and data manipulation.

declaring a function

```
class Cat:
    pass

fluffy = Cat()
```

This declares a blank class called "Cat". It can't do anything at all, because there's nothing to do with it. fluffy

Recommendation

Name all your classes using PascalCase.
That is, Capitalize every word and don't use spaces.
If it includes an acronym, like "URL", call it "Url".

Fields and Properties

Definition: Field

An element in a class that contains a specific term of information.

Definition: Property

A syntax nicety to use instead of getter/setter functions.

Simply, fields hold information and properies are accessors to that information.

property example

```
class Cat:
    [Property(Name)]
    _name as string

fluffy = Cat()
fluffy.Name = 'Fluffy'
```

1. class Cat: declares the start of a class.
    1. [Property(Name)] declares a property around _name. You named the property "Name".
    2. _name as string declares a field of Cat that is a string called _name.
2. fluffy = Cat() declares an instance of Cat.
3. fluffy.Name = 'Fluffy' accesses the property Name of Cat and sets its value to 'Fluffy'. This will cause Name to set _name to 'Fluffy'.

Fields are not set directly because of security.

Recommendation

Name all your properties using PascalCase, just like classes.
Name all your fields using _underscoredCamelCase, which is similar to PascalCase, only it is prefixed with an underscore and the first letter is lowercase.

There are two other types of properties, a getter and a setter. Technically, a regular property is just the combination of the two.

getter/setter example

```
class Cat:
    [Getter(Name)]
    _name = 'Meowster'

    [Setter(FavoriteFood)]
    _favoriteFood as string

fluffy = Cat()
print fluffy.Name
fluffy.FavoriteFood = 'Broccoli'
```

Output

```
Meowster
```

If you were to try to assign a value to fluffy.Name or retrieve a value from fluffy.FavoriteFood, an error would have occurred, because the code just does not exist for you to do that.

Using the attributes Property, Getter, and Setter are very handy, but it's actually Boo's shortened version of what is really happening. Here's an example of the full code.

explicit property example

```
class Cat:
    Name as string:
        get:
            return _name
        set:
            _name = value

    _name as string

fluffy = Cat()
fluffy.Name = 'Fluffy'
```

Because fields are visible inside their own class, you can see that Name is just a wrapper around _name. Using this expanded syntax is handy if you want to do extra verification or not have it wrap exactly around its field, maybe by trimming whitespace or something like that first.
value is a special keyword for the setter statement, that contains the value to be assigned.

Property Pre-condition

It is also possible to define a precondition that must be met before setting a value directly through the Property shorthand.

property example

```
class Cat:
    [Property(Name, Name is not null)]
    _name as string

fluffy = Cat()
fluffy.Name = null # will raise an ArgumentException
```

Class Modifiers

Modifier	Description
public	Creates a normal, public class, fully accessible to all other types.
protected	Creates a class that is only accessible by its containing class (the class this was declared in) and any inheriting classes.
internal	A class only accessible by the assembly it was declared in.
protected internal	Combination of protected and internal.
private	Creates a class that is only accessible by its containing class (the class this was declared in.)
abstract	Creates a class that cannot be instanced. This is designed to be a base class for others.
final	Creates a class that cannot be inherited from.

Recommendation

Never use the public Class Modifier. It is assumed to be public if you specify no modifier.

class modifier example

```
abstract class Cat:
    [Property(Name)]
    _name as string
```

The abstract keyword is the Class Modifier.

Inheritance

Definition: Inheritance

A way to form new classes (instances of which will be objects) using pre-defined objects or classes where new ones simply take over old ones's implemetions and characterstics. It is intended to help reuse of existing code with little or no modification.

Inheritance is very simple in Boo.

inheritance example

```
class Cat(Feline):
    [Property(Name)]
    _name as string

class Feline:
    [Property(Weight)]
    _weight as single //In Kilograms
```

This causes Cat to inherit from Feline. This gives the members Weight and _weight to Cat, even though they were not declared in Cat itself.
You can also have more than one class inherit from the same class, which promotes code reuse.

More about inheritance is covered in Part 10 - Polymorphism, or Inherited Methods

Classes can inherit from one or zero other classes and any number of interfaces.

To inherit from more than one interface, you would use the notation class Child(IBaseOne, IBaseTwo, IBaseThree):

Interfaces

Definition: Interface

An interface defines a list of methods that enables a class to implement the interface itself.

Interfaces allow you to set up an API (Application Programming Interface) for classes to base themselves off of.

No implementation of code is put inside interfaces, that is up to the classes.

Interfaces can inherit from any number of other interfaces. They cannot inherit from any classes.

interface example

```
interface IFeline:
    def Roar()

    Name:
        get
        set
```

This defines IFeline having one method, Roar, and one property, Name. Properties must be explicitly declared in interfaces. Methods are explained in Part 09 - Methods.

Recommendation

Name your interfaces using PascalCase prefixed with the letter I, such as IFeline.

Difference between Value and Reference Types

There are two types in the Boo/.NET world: Value and Reference types. All classes form Reference types. Numbers and such as was discussed in Part 02 - Variables#List of Value Types are value types.

Definition: null

A keyword used to specify an undefined value for reference variables.

Value types can never be set to null, they will always have a default value. Numbers default value will generally be 0.

Exercises

1. Create a class that inherits from more than one interface.
2. See what happens if you try to inherit from more than one class.

Go on to **Part 09 - Methods**

Part 09 - Methods

Part 09 - Methods

Definition: Method

A function exclusively associated with a class

Defining a Method

Methods must be defined in classes. They are declared just like functions are.

arguments example

```
class Cat:
    def Roar():
        pring "Meow!"

cat = Cat()
cat.Roar()
```

Output

```
Meow!
```

An object of Cat must be instanced, then its methods can be called.

Recommendation

Names of methods should always be verbs.
They should also be declared in PascalCase.

Class Constructor and Destructor

Constructors and Destructors are special methods that are called on when a class is being instanced or destroyed, respectively.

Both are optional.

arguments example

```
class Cat:
    def constructor():
        _name = 'Whiskers'

    def destructor():
        print "${_name} is no more... RIP"

    [Getter(Name)]
    _name as string

cat = Cat()
print cat.Name
```

Output

```
Whiskers
Whiskers is no more... RIP
```

If a constructor has arguments, then they must be supplied when instancing. Destructors cannot have arguments.

arguments example

```
class Cat:
```

```
    def constructor(name as string):
        _name = name

    [Getter(Name)]
    _name as string

cat = Cat("Buttons")
print cat.Name
```

Output

```
Buttons
```

Be Careful

Do not depend on the destructor to always be called.

Method Modifiers

Modifier	Description
abstract	an abstract method has no implementation, which requires that an inheriting class implements it.
static	a static method is common to the entire class, which means that it can be called without ownership of a single instance of the class
virtual	See Part 10 - Polymorphism, or Inherited Methods
override	See Part 10 - Polymorphism, or Inherited Methods

All these modifiers also apply to properties (If they are explicitly declared).
static can also apply to fields.

static example

```
class Animal:
    def constructor():
        _currentId += 1
        _id = currentId

    [Getter(Id)]
    _id as int

    static _currentId = 0
```

This will cause the Id to increase whenever an Animal is instanced, giving each Animal their own, unique Id.

All the methods defined in an interface are automatically declared abstract.
Abstract methods in a class must have a blank code block in its declaration.

abstract example

```
class Feline:
    abstract def Eat():
        pass

interface IFeline:
    def Eat()
```

Both declare roughly the same thing.

Member Visibility

Visibility Level	Description
public	Member is fully accessible to all types.
protected	Member is only visible to this class and inheriting classes.

private	Member is only visible to this class.

Important Information

All fields are by default protected. All methods, properties, and events are by default public.

Recommendation

Fields are typically either protected or private. Usually instead of making a public field, you might make a public property that wraps access to the field instead. This allows subclasses to possibly override behavior.
Methods can have any visibility.
Properties can have any visibility, and typically have both a getter and a setter, or only a getter. Instead of a set only property, consider using a method instead (like "SetSomeValue(val as int)").

Recommendation

It is recommended you prefix field names with an underscore if it is a private field.

Declaring Properties in the Constructor

One very nice feature that boo offers is being able to declare the values of properties while they are being instanced.

abstract example

```
class Box:
    def constructor():
        pass

    [Property(Value)]
    _value as object

box = Box(Value: 42)
print box.Value
```

Output

```
42
```

The constructor didn't take any arguments, yet the Value: 42 bit declared Value to be 42, all in a tighly compact, but highly readable space.

Exercises

1. Create two classes, Predator and Prey. To the Predator class, add an Eat method that eats the Prey. Do not let the Prey be eaten twice.

Go on to **Part 10 - Polymorphism, or Inherited Methods**

Part 10 - Polymorphism, or Inherited Methods

Definition: Polymorphism

The ability for a new object to implement the base functionality of a parent object in a new way.

Two keywords are used to make Polymorphism happen: virtual and override.

You need to describe a method as virtual if you want the ability to override its capabilities.

Polymorphism with Rectangle and Square

```
class Rectangle:
    def constructor(width as single, height as single):
        _width = width
        _height = height

    virtual def GetArea():
        return _width * _height

    _width as single
    _height as single

class Square(Rectangle):
    def constructor(width as single):
        _width = width

    override def GetArea():
        return _width * _width

r = Rectangle(4.0, 6.0)
s = Square(5.0)

print r.GetArea()
print s.GetArea()
print cast(Rectangle, s).GetArea()
```

Output

```
24.0
25.0
25.0
```

Even when casted to a Rectangle, s's .GetArea() functioned like if it were a Square.

An easier example to see is this:

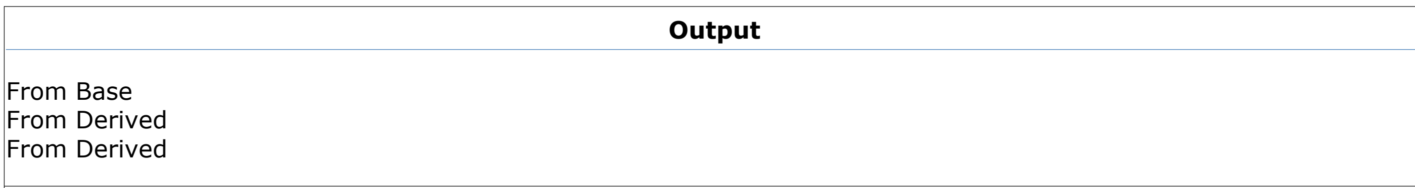
Simplified Polymorphism Example

```
class Base:
    virtual def Execute():
        print 'From Base'

class Derived(Thing):
    override def Execute():
        print 'From Derived'

b = Base()
d = Derived()

print b.Execute()
print d.Execute()
print cast(Base, d).Execute()
```



Output

```
From Base
From Derived
From Derived
```

If I were to leave out the virtual and {{override} keywords,

Output w/o virtual

```
From Base
From Derived
From Base
```

This happens because unless the base method is virtual or abstract, the derived method cannot be declared as override.

Recommendation

Although you do not have to explicitly declare a method as override when inheriting from a virtual method, you should anyway, in case the signatures of the virtual and overriding methods do not match.

In order to override, the base function must be declared as virtual or abstract, have the same return type, and accept the same arguments.

Polymorphism is very handy when dealing with multiple types derived from the same base.

Another Polymorphism Example

```
interface IAnimal:
    def MakeNoise()

class Dog(IAnimal):
    def MakeNoise():
        print 'Woof'

class Cat(IAnimal):
    def MakeNoise():
        print 'Meow'

class Hippo(IAnimal):
    def MakeNoise():
        print '*Noise of a Hippo*'

list = []
list.Add(Dog())
list.Add(Cat())
list.Add(Hippo())

for animal as IAnimal in list:
    list.MakeNoise()
```

Output w/o virtual

```
Woof
Meow
*Noise of a Hippo*
```

Very handy.

Exercises

1. Figure out an exercise

Go on to **Part 11 - Structs**

Part 11 - Structs

Part 11 - Structs

Definition: Struct

Short for structure, a term meaning a data group made of related variables.

The main difference between structs are different than classes is that it is a value type instead of a reference type. This means that whenever you return this value, or set one equal to another, it is actually copying the data not a reference to the data. This is handy, becaues if it is declared without a value, it will default to something besidesnull. It also cannot be compared to null. This eliminates alot of error checking associated with reference types.

Structs also cannot inherit from classes, nor can classes inherit from structs. Structs can however, inherit from interfaces.

Unlike some other languages, structs can have methods.

Declaring a Struct

Declaring a struct is very similar to declaring a {{class}, except that the name is changed.

declaring a struct

```
struct Coordinate:
    def constructor(x as int, y as int):
        _x = x
        _y = y

    _x as int
    _y as int

c as Coordinate
print c.x, c.y
c = Coordinate(3, 5)
print c,x, c.y
```

Output

```
0 0
3 5
```

Here you can see that the struct was instanced without being called, showing the how astruct is a value.

Exercises

1. Figure out a good exercise for this section.

Go on to **Part 12 - Namespaces**

Part 12 - Namespaces

Part 12 - Namespaces

Definition: Namespace

A name that uniquely identifies a set of objects so there is no ambiguity when objects from different sources are used together.

Namespaces are useful because if you have, for example, a Dog namespace and a Furniture namespace, and they both have a Leg class, you can refer to Dog.Leg and Furniture.Leg and be clear about which class you are mentioning.

Declaring a Namespace

To declare a namespace, all that is required is that you put namespace followed by a name at the top of your file.

declaring a namespace

```
namespace Tutorial

class Thing():
    pass
```

This creates your class Tutorial.Thing. While coding inside your namespace, it will be transparently Thing.

To declare a namespace within a namespace, just place a dot . inbetween each other.

Recommendation

Declare a namespace at the top of all your files.
Use PascalCase for all your namespaces.

Importing Another Namespace

To use classes from another namespace, you would use the import keyword.
The most common namespace you will import is System.

importing from a namespace

```
import System

Console.WriteLine()
```

not importing from a namespace

```
System.Console.WriteLine()
```

Both produce the exact same code, it's just easier and clearer with the import.

Recommendation

Don't be afraid to import, just don't import namespaces that you aren't using.

Recommendation

When importing, import included namespaces first, such as System or Boo.Lang.
Then import your 3rd party namespaces.
Alphabetize the two groups seperately.

If you are importing from another assembly, you would use the phrase import <*target*> from <*assembly*>, for example

importing from an external assembly

```
import System.Data from System.Data
import Gtk from "gtk-sharp"
```

System.Data is part of an external library which can be added, System.Data.dll. Gtk is part of the Gtk# library, which, since it has a special name (with a dash in it), it must be quoted.

Recommendation

Only use the import <*target*> from <*assembly*> if you are using one file and one file only. If you are using more than that, you should be using a build tool, such as NAnt, which is discussed in Part 19 - Using the Boo Compiler.

Exercises

1. Figure out a good exercise for this section.

Go on to **Part 13 - Enumerations**

Part 13 - Enumerations

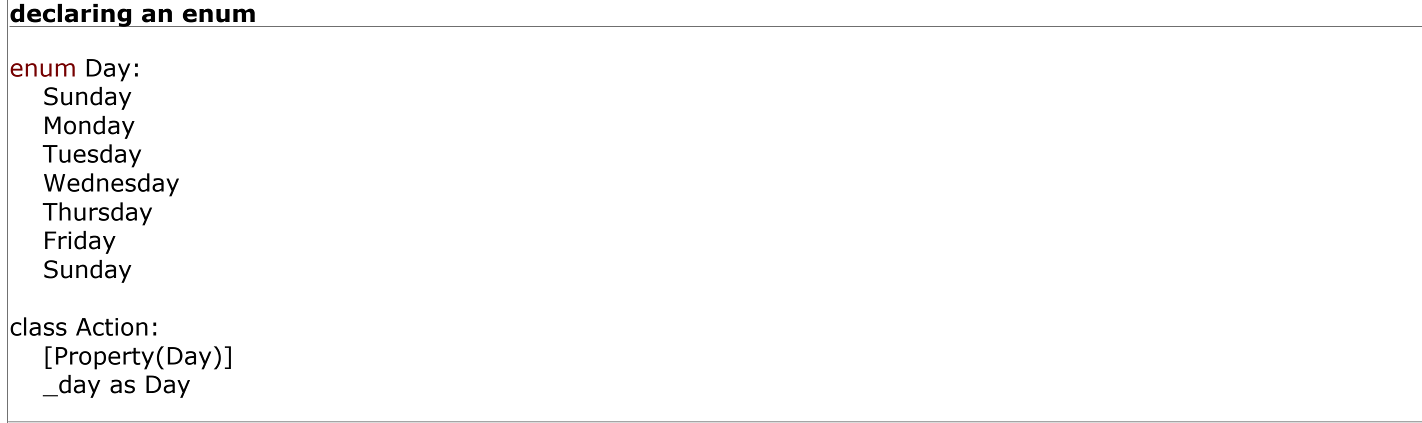
Part 13 - Enumerations

Definition: Enumeration

A set of name to integer value associations.

Declaring an Enumeration

Enumerations are handy to use as fields and properties in classes.



declaring an enum

```
enum Day:
    Sunday
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Sunday

class Action:
    [Property(Day)]
    _day as Day
```

Enumerations are also handy in preventing "magic numbers", which can cause unreadable code.

Definition: Magic Number

Any number outside of -1, 0, 1, or 2.

Enumerations technically assign an integer value to each value, but that should generally be abstracted from view.

declaring an enum

```
enum Test:
    Alpha
    Bravo
    Charlie
```

is the same as

declaring an enum

```
enum Test:
    Alpha = 0
    Bravo = 1
    Charlie = 2
```

Recommendation

Except in special cases, do not assign numbers.

Exercises

1. Think of another good instance of using enums.

Go on to **Part 14 - Exceptions**

Part 14 - Exceptions

Part 14 - Exceptions

Definition: Exception

A mechanism designed to handle runtime errors or other problems (exceptions) inside a computer program.

Exceptions are very important, as they are raised whenever an error occurs in the system. (Or at least they should be.)

Catching Exceptions

An exception stops the program if it is not caught.

Division by Zero

```
print 1 / 0
```

Output

```
System.DivideByZeroException: Attempted to divide by zero.
  at Test.Main(String[] argv)
```

Which stopped the program.

To handle the situation, exceptions must be caught.
Exceptions are either caught in a try-except statement, a try-ensure statement, or a try-except-ensure statement.
Also, all Exceptions are derived from the simple Exception.

try-except example

```
import System

try:
    print 1 / 0
except e as DivideByZeroException:
    print "Whoops"
print "Doing more..."
```

Output

```
Whoops
Doing more...
```

This prevents the code from stopping and lets the program keep running even after it would have normally crashed.

There can be multiple except statements, in case the code can cause multiple Exceptions.

Try-ensure is handy if you are dealing with open streams that need to be closed in case of an error.

try-ensure example

```
import System

try:
    s = MyClass()
    s.SomethingBad()
ensure:
    print "This code will be executed, whether there is an error or not."
```

Output

```
This code will be executed, whether there is an error or not.
System.Exception: Something bad happened.
```

```
  at Test.Main(String[] argv)
```

As you can see, the ensure statement didn't prevent the Exception from bubbling up and causing the program to crash.

A try-except-ensure combines the two.

try-except-ensure example

```
import System

try:
    s = MyClass()
    s.SomethingBad()
except e as Exception:
    print "Problem! ${e.Message}"
ensure:
    print "This code will be executed, whether there is an error or not."
```

Output

```
Problem: Something bad happened.
This code will be executed, whether there is an error or not.
```

Recommendation

If you don't solve the problem in your except statement, use the raise command without any parameters to re-raise the original Exception.

Raising Exceptions

There are times that you want to raise Exceptions of your own.

Raising an Exception

```
import System

def Execute(i as int):
    if i < 10:
        raise Exception("Argument i must be greater than or equal to 10")
    print i
```

If Execute is called with an improper value of i, then the Exception will be raised.

Recommendation

In production environments, you'll want to create your own Exceptions, even if they are just wrappers around Exception with different names.

Recommendation

Never use ApplicationException.

Exercises

1. Think of an exercise

Go on to **Part 15 - Functions as Objects and Multithreading**

Part 15 - Functions as Objects and Multithreading

Part 15 - Functions as Objects and Multithreading

Having Functions act as objects exposes three very useful methods:

1. function.Invoke(<arguments>) as <return type>
2. function.BeginInvoke(<arguments>) as IAsyncResult
3. function.EndInvoke(IAsyncResult) as <return type>

.Invoke just calls the function normally and acts like it was called with just regular parentheses ().
.BeginInvoke starts a seperate thread that does nothing but run the function invoked.
.EndInvoke finishes up the previously invoked function and returns the proper return type.

example of .Invoke

```
def Nothing(x):
    return x

i = 5
assert 5 == Nothing(i)
assert i == Nothing.Invoke(i)
assert i == Nothing.Invoke.Invoke(i)
```

Since .Invoke is a function itself, it has its own .Invoke.

Here's a good example of .BeginInvoke

Multithreading with .BeginInvoke

```
import System
import System.Threading

class FibonacciCalculator:
    def constructor():
        _alpha, _beta = 0, 1
        _stopped = true

    def Calculate():
        _stopped = false
        while not _stopped:
            Thread.Sleep(200)
            _alpha, _beta = _beta, _alpha + _beta
            print _beta

    def Start():
        _result = Calculate.BeginInvoke()

    def Stop():
        _stopped = true
        Calculate.EndInvoke(_result)

    _result as IAsyncResult

    _alpha as ulong
    _beta as ulong

    _stopped as bool

fib = FibonacciCalculator()
fib.Start()
prompt("Press enter to stop...\n")
fib.Stop()
```

The output produces the Fibonacci sequence roughly every 200 milliseconds (because that's what the delay is). This will produce an overflow after it gets up to 2^64.

The important thing is that it stops cleanly if you press Enter.

Exercises

1. Think of an exercise

Go on to **Part 16 - Generators**

Part 16 - Generators

Part 16 - Generators

Generator Expressions

Definition: Generator Expression

An phrase that creates a generator based on the syntax:
<expression> for <declarations> [as <type>] in <iterator> [if|unless <condition>]

Generator Expressions have similar syntax to the for loops that we have covered, and serve a similar purpose.

The best way to learn how to use Generator Expressions is by example, so here we load up a booish prompt.

```
$ booish
>>> List(x for x in range(5)) // simplest Generator Expression
[0, 1, 2, 3, 4]
>>> List(x * 2 for x in range(5)) // get double of values
[0, 2, 4, 6, 8]
>>> List(x**2 for x in range(5)) // get square of values
[0, 1, 4, 9, 16]
>>> List(x for x in range(5) if x % 2 == 0) // check if values are even
[0, 2, 4]
>>> List(x for x in range(10) if x % 2 == 0) // check if values are even
[0, 2, 4]
>>> List(y for y in (x**2 for x in range(10)) if y % 3 != 0) // Generator Expression inside another
[1, 4, 16, 25, 49, 64]
>>> List(cat.Weight for cat in myKitties if cat.Age >= 1.0).Sort()
[6.0, 6.5, 8.0, 8.5, 10.5]
>>> genex = x ** 2 for x in range(5)
generator(System.Int32)
>>> for i in genex:
...     print i
...
0
1
4
9
16
```

The cat-weight example is probably what Generator Expressions are most useful for.

You don't have to create Lists from them either, that's mostly for show.

generators are derived from IEnumerable, so you get all the niceties of the for loop as well.

Recommendation

Don't overdo it with Generator Expressions. If they are causing your code to be less readable, then spread them out a little.

Generator Methods

Definition: Generator Expression

A method that creates a generator based on stating the yield keyword within the method.

A Generator Method is like a regular method that can return multiple times.

Here's a Generator Method that will return exponents of 2.

Generator Method Example

```
def TestGenerator():
    i = 1
    yield i
    for x in range(10):
        i *= 2
        yield i

print List(TestGenerator())
```

Output

```
[1, 2, 4, 8, 16, 32, 64, 128, 512, 1024]
```

Generator Methods are very powerful because they keep all their local variables in memory after a yield. This can allow for certain programming techniques not found in some other languages.

Generators are very powerful and useful.

Exercises

1. Create a Generator that will destroy mankind.

Go on to **Part 17 - Macros**

Part 17 - Macros

Part 17 - Macros

print Macro

The print Macro will display one or more objects to the screen.

There are two ways to call the print macro.

1. With only one argument
2. With two or more arguments

print Example

```
print "Hello there"
print "Hello", "there"
```

Output

```
Hello there
Hello there
```

In the second case, for every case except the last, it will write the string to the screen, write a space, then move on.

In the end, the two will have the same end result.

assert Macro

The assert Macro makes sure that a condition is true, otherwise it raises an AssertionFailedException.

assert can be called with one or two arguments.
The first argument must always be a boolean condition.
The optional second argument is a string that will be sent if the condition fails.

assert Example

```
assert true // this will always pass
assert false, "message" // this will always fail
```

Output

```
Boo.Lang.Runtime.AssertionFailedException: message
  at Tutorial.Main(String[] argv)
```

Recommendation

Never assert a condition that would, in itself, change your code.
e.g. assert iter.MoveNext() would be a bad idea.

using Macro

The using Macro can take any number of arguments, it merely duplicates its behavior each time.

It creates a safety net for objects to be handled during a block, then disposed of as soon as that block is finished.

There are three types of arguments you can declare:

1. <object>
2. <object> = <expression>
3. <expression>

In all three of these, it checks if the underlying object is an IDisposable, which it then disposes of afterward.

using Example

```
import System.IO
```

```
using w = StreamWriter("test.txt"):
    w.WriteLine("Hello there!")
```

This will create the file, write to it, then close it as soon as the using block is finished. Makes it very safe and convenient.

lock Macro

The lock Macro makes sure that, in a multithreaded environment, that a specified object is not being used and prevents another object from using it at the same time.

lock must accept at least one argument, and it will put the lock on all that are given.

lock Example

```
lock database:
    database.Execute("""
        UPDATE messages
        SET
            id = id + 1""")
```

debug Macro

The debug Macro is the exact same as the print Macro, except that it sends its messages to System.Diagnostics.Debug instead of System.Console.

Go on to **Part 18 - Duck Typing**

Part 18 - Duck typing

Part 18 - Duck Typing

Definition: Duck Typing

Duck typing is a humorous way of describing the type non-checking system. Initially coined by Dave Thomas in the Ruby community, its premise is that (referring to a value) "if it walks like a duck, and talks like a duck, then it is a duck".

Even though Boo is a statically typed language, Duck Typing is a way to fake being a dynamic language. Duck typing allows variables to be recognized at runtime, instead of compile time. Though this can add a sense of simplicity, it does remove a large security barrier.

Duck Typing Example

```
d as duck
d = 5 // currently set to an integer.
print d
d += 10 // It can do everything an integer does.
print d
d = "Hi there" // sets it to a string.
print d
d = d.ToUpper() // It can do everything a string does.
print d
```

Output

```
5
15
Hi there
HI THERE
```

Duck typing is very handy if you are loading from a factory or an unpredictable dynamic library.

Recommendation

Do not enable duck typing by default. It should only be used in a few situations.

On a side note, The booish interpreter has duck typing enabled by default. This can be disabled by typing in interpreter.Ducky = false

Here is a practical example of where duck typing is useful.

Practical Duck Typing

```
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/monologue/")

while ie.Busy:
    Thread.Sleep(50ms)

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

Exercises

1. Come up with another good example where duck typing is effective.

Go on to **Part 19 - Using the Boo Compiler**

Part 19 - Using the Boo Compiler

Part 19 - Using the Boo Compiler

The Boo Compiler is typically called in this fashion:
booc <options> <files>

Command-line Options

Option	Description
-v	Verbose
-vv	More Verbose
-vvv	Most Verbose
-r:*<reference_name>*	Add a reference to your project
-t:*<type_name_to_generate>*	Type of file to generate, can be either exe or winexe to make executables (.exe files), or library to make a .dll file
-p:*<pipeline>*	Adds a step *<pipeline>* to the compile.
-c:*<culture>*	Sets which *CultureInfo* to use.
-o:*<output_file>*	Sets the name of the output file
-srcdir:*<source_files>*	Specify where to find the source files.
-debug	Adds debug flags to your code. Good for non-production. (On by default)
-debug-	Does not add debug flags to your code. Good for production environment.
-debug-steps	See AST after each compilation step.
-resource:*<resource_file>*,*<name>*	Add a resource file. *<name>* is optional.
-embedres:*<resource_file>*,*<name>*	Add an embedded resource file. *<name>* is optional.

So, for example, in order to compile your Database code that depends on the library System.Data.dll, you would type:
booc -r:System.Data.dll -o:Database.dll -t:library Database.boo

That would create a fully functional, working compilation of your library: Database.dll

Using NAnt

When working on a large project with multiple files or libraries, it is a lot easier to use NAnt. It is a free .NET build tool.

To do the same command as above, you would create the following build file:

default.build

```
<?xml version="1.0" ?>

<project name="Goomba" default="build">
 <target name="build" depends="database" />
    <target name="database">
         <mkdir dir="bin" />
       <booc output="bin/Database.dll" target="library">
            <references basedir="bin">
               <include name="System.Data.dll" />
           </references>
          <sources>
               <include name="Database.boo" />
         </sources>
       </booc>
 </target>
</project>
```

```
$ nant
NAnt 0.85 (Build 0.85.1869.0; rc2; 2/12/2005)
Copyright (C) 2001-2005 Gerry Shaw
http://nant.sourceforge.net

Buildfile: file:///path/to/default.build
```

```
Target framework: Microsoft .NET Framework 1.1
Target(s) specified: build

build:

database:

    [booc] Compiling 1 file(s) to /path/to/bin/Database.dll.

BUILD SUCCEEDED

Total time: 0.2 seconds.
```

And although that was a long and drawnout version of something so simple, it does make things *a lot* easier when dealing with multiple files.

It also helps that if you make a change to your source files, you don't have to type a long booc phrase over again.

The important part of the build file is the <booc> section. It relays commands to the compiler.
There are four attributes available to use in it:

Attribute	Description
target	Output type, one of library, exe, winexe. Optional. Default: exe.
output	The name of the output assembly. **Required**.
pipeline	AssemblyQualifiedName for the CompilerPipeline type to use. Optional.
tracelevel	Enables compiler tracing, useful for debugging the compiler, one of: Off, Error, Warning, Info, Verbose. Optional. Default: Off.

You are most likely only to use target and output.

For nested elements, you have 3 possibilities:

Nested Element	Description
<sources>	Source files. **Required**.
<references>	Assembly references.
<resources>	Embedded resources.

Inside these you are to put <include /> elements, as in the example.

This is merely a brief overview of NAnt, please go to their website http://nant.sourceforge.net for more information.

Go on to **Part 20 - Structure of a Boo Project**

Part 20 - Structure of a Boo Project

Part 20 - Structure of a Boo Project

On the Project-level

Here I'll use the example of the IRC bot I write: Goomba

```
+ Goomba (Goomba namespace)
  |+ Configuration (Goomba.Configuration namespace)
  |   |- Config.boo
  |      |# class Config
  |+ Data (Goomba.Data namespace)
  |   |- Column.boo
  |   |   |# class Column
  |   |- Database.boo
  |   |   |# enum DatabaseType
  |   |   |# class Database
  |   |- DatabasePreferences.boo
  |   |   |# class DatabasePreferences
  |   |- Result.boo
  |      |# class Result
  |+ Plugins (Goomba.Plugins namespace)
  |   |- DefineCommand.boo
  |   |   |# class DefineCommand
  |   |      |# class Definition
  |   |- Hail.boo
  |   |   |# class Hail
  |   |      |# class HailMessage
  |   |- HelpCommand.boo
  |   |   |# class HelpCommand
  |   |- Logger.boo
  |   |   |# class Logger
  |   |      |# class Message
  |   |      |# class Action
  |   |- Quoter.boo
  |   |   |# class Quoter
  |   |      |# class Quote
  |   |- RawLogger.boo
  |   |   |# class RawLogger
  |   |- UrlGenerator.boo
  |   |   |# class UrlGenerator
  |   |      |# class Engine
  |   |- UserTracker.boo
  |   |   |# class UserTracker
  |   |      |# class User
  |   |- VersionCommand.boo
  |   |   |# class VersionCommand
  |   |- UrlTracker.boo
  |      |# class UrlTracker
  |         |# class Url
  |- ActionEventArgs.boo
  |   |# enum ActionType
  |   |# class ActionEventArgs
  |- DebugLogger.boo
  |   |# enum LogImportance
  |   |# class DebugLogger
  |- Goomba.boo
  |   |# class Goomba
  |   |! Main Body (This will be executed when Goomba.exe is run)
  |- GoombaPreferences.boo
  |   |# class GoombaPreferences
  |- IPlugin.boo
  |   |# interface IPlugin
  |- MessageEventArgs.boo
  |   |# enum MessageType
  |   |# class MessageEventArgs
```

```
|- Sender.boo
   |# enum SenderType
   |# class Sender
```

Which I have set up to create the assemblies Goomba.exe, Goomba.Data.dll, Goomba.Configuration.dll, as well as one assembly per plugin.

You may have noticed a few important things:

- For every directory, it represents a different namespace, with the same name as the directory itself.
- Each .boo file has at most one class in it. That class will have the **same exact** name as the .boo file.
- The "Main Body" section is below the class Goomba definition. Any inline executable code must be at the bottom of a file in the assembly.
- Enums come before classes. This is merely a coding practice that is not required, but recommended. If an enum is larger than 15 values, place it in its own file.

On the File-level

Files must be defined in this order:

1. Module docstring
2. Namespace declaration
3. Import statements
4. Enums/Classes/Structs/Interfaces
5. Functions
6. Main code executed when script is run
7. Assembly attributes

Recommendation

One class per file. If you have more than one class per file, split it up.
If you have a class inside another class, this is acceptable, as it still has one flat class per file.

Go on to **Part 21 - Documentation**

Part 21 - Documentation

Part 21 - Documentation

A communicable material used to explain some attributes of an object, system or procedure.

I've saved the most important for last, as documentation is itself, just as important as the code which it describes.

When documenting your code, be sure to remember:

1. All your documents should be in English.
2. Use full sentences.
3. Avoid spelling/grammar mistakes.
4. Use present tense.

Documentation is placed in tripled double-quoted strings right below what you are documenting.

Documentation with Summary

```
def Hello():
"""Says "hello" to the world."""
    print "Hello, World!"

Hello()
```

That "docstring" is the least you can do to document your code. It gave a simple summary.
If your docstring spans more than one line, then the quotes should go on their own lines.
You may have noticed that 'Says "hello" to the world.' is not a full sentence. For the first sentence in a summary, you can imply "This member".

Parameters are also supposed to documented.

Parameters

```
def Hello(name as string):
"""
Say "hello" to the given name.
Param name: The name to say hello to.
"""
    print "Hello, ${name}!"

Hello()
```

To read it to yourself, it goes as such: 'Say "hello" to the given name. Parameter name is defined as the name to say hello to.'
This keeps in line with using full sentences.

If describing the parameter takes more than one line, you should move it all to a new line and indent.

Long Parameter

```
def Hello(name as string):
"""
Say "hello" to the given name.
Param name:
    The name to say hello to.
    It might do other things as well.
"""
    print "Hello, ${name}!"
```

The same goes with any block.

Here is a list of all the tags that can be used

Tag	Description
No tag	A summary of the member.
Param <*name*>: <*description*>	This specifies the parameter <*name*> of the method.
Returns: <*description*>	This describes what the method returns.
Remarks: <*text*>	This provides descriptive text about the member.
Raises <*exception*>: <*description*>	Gives a reason why an Exception is raised.
Example: <*short_description*>: <*code_block*>	Provides an example.
Include <*filename*>: <*tagpath*>[@<*name*>="<*id*>"]	Includes an excerpt from another file.
Permission <*permission*>: <*description*>	Describe a required Permission.
See Also: <*reference*>	Lets you specify the reference that you might want to appear in a See Also section.

And a list of inline tags

Tag	Description
* <*item*> * <*item*> * <*item*>	Bullet list
# <*item*> # <*item*> # <*item*>	Numbered List
<<*reference*>>	Provides an inline link to a reference. e.g. <int> or <string> would link.
[<*param_reference*>]	References to a parameter of the method.

Here's some examples of proper documentation:

Documentation example

```
import System

class MyClass:
"""Performs specific duties."""
    def constructor():
    """Initializes an instance of <MyClass>"""
        _rand = Random()

    def Commit():
    """Commits an action."""
        pass

    def CalculateDouble(i as int) as int:
    """
    Returns double the value of [i].
    Parameter i: An <int> to be doubled.
    Returns: Double the value of [i].
    """
        return i * 2

    def CauseError():
    """
    Causes an error.
    Remarks: This method has not been implemented.
    Raises NotImplementedException: This has not been implemented yet.
    """
        return NotImplementedException("CauseError() is not implemented")

    def DoSomething() as int:
    """
    Returns a number.
    Example: Here is a short example:
        print DoSomething()
```

```
Returns: An <int>.
See Also: MakeChaos()
"""
    return 0

def MakeChaos():
"""
Creates Chaos.
Include file.xml: Foo/Bar[@id="entropy"]
Permission Security.PermissionSet: Everyone can access this method.
"""
    print "I am making chaos: ${_rand.Next(100)}"

def Execute():
"""
Executes the protocol.
Does one of two things,
# Gets a sunbath.
# Doesn't get a sunbath.
"""
    if _rand.Next(2) == 0:
        print "I sunbathe."
    else:
        print "I decide not to sunbathe."

def Calculate():
"""
Does these things, in no particular order,
* Says "Hello"
* Looks at you
* Says "Goodbye"
"""
    thingsToDo = ["I look at you.", 'I say "Hello."', 'I say "Goodbye."']
    while thingsToDo.Length > 0:
        num = _rand.Next(thingsToDo.Length)
        print thingsToDo[num]
        thingsToDo.RemoveAt(num)

[Property(Name)]
_name as string
"""A name""" // documents the property, not the field

Age as int:
"""An age"""
    get:
        return _rand.Next(8) + 18

_age as int

_rand as Random
```

This should give you a good view on how to document your code.

I think Dick Brandon said it best:

> **Quote: Dick Brandon**
> Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.

Go on to **Part 22 - Useful Links**

Part 22 - Useful Links

Part 22 - Useful Links

- Boo
- Download Boo
- MSDN Search - Very handy if you want to look up something in the standard library.
- Mono Docs - Nice source of information as an alternative to MSDN, or if you work on Mono or Gtk specific projects.
- Google - Has a lot of information within its reach. Searching prefixed with ".NET Framework" or "C#" usually will turn up what you need.
- NAnt - A free build tool for .NET