Pie Programming Language Specification
Jason Bell 3078931
July 7, 2015

Comments

As per C languages: // for single line comments, and /**/ for multiple line comments.

Namespaces

Behave exactly as they do in other .NET languages. Declared as:

namespace<namespace name>:

For example:

```
namespace foo:
	namespace bar:

namespace foo.bar:
```

Imports

“Using” statements in C#. Must be at the namespace level. Each import statement can list multiple names to import:

```
namespace foo:
	import System
	import System.CodeDom, System.Net
	import System.Windows.Forms,
		System.Drawing,
		System.IO
```

Modules

Container for classless functions and variables. One module can be spread across multiple source files. From the perspective of the Pie language, modules are essentially namespaces that have some class-like features in order for them to be accessible from other .NET languages. However, while their declarations are similar to those of classes, they can not be instantiated.

Modules are declared as:

<modifiers>module<module name>{generic type names}

Modules can have either public or internal as their modifier.

A simple module declaration:

```
module Functions:
```

A module declaration with modifiers:

```
internal module Functions:
```

A module declaration with generic type names:

```
internal module Functions{T,K}:
```

Module methods and properties default to public, while module variables default to private. All members of a module receive the “shared” (static) modifier.

Types

Types can be classes, structs, interfaces, enums, or delegates.

All types can have one of two access modifiers: public or internal. All types default to public.

Classes

Classes can also be declared as final (sealed in c#), partial, or abstract.

Methods and properties default to public, variables default to private.

Classes are declared as:

<modifiers>class<class name>{generic type names}(base types):

The simplest class declaration:

```
class foo:
```

With modifiers:

```
public partial class foo:
```

With one or more base types:

```
class foo(bar, cow):
```

With one or more generic type names:

```
class foo{T, K}:
```

With both generic type names and base types:

```
class foo{T, K}
       (bar, cow):
```

Structs

All members default to public.

Structs are declared as:

<modifiers>struct<struct name>{generic type names}(interface types):

In .NET, structs can only inherit interfaces.

```
struct foo{T, K}
       (IDisposable):
```

Enums

Simply a list of constants, declared as:

<modifiers>enum<enum name>:

If constants are not assigned values, they start from zero and increment with each subsequent constant. For example, in the following example Sally will equal 0, Bill will equal 1, Bob will equal 2, and Joe will equal 3.

```
enum People:
       Sally
       Bill
       Bob
       Joe
```

Assigning a value to any of the constants will cause all following constants to increment from that one. For example, in the following example Sally equals 0, Bill equals 33, Bob equals 34, and Joe equals 35.

```
enum People:
       Sally
       Bill = 33
       Bob
       Joe
```

Delegates

Declared as:

<modifier>delegate<return type name><delegate name>(parameter list)

Modifier can be public or internal.

For example:

```
delegate void foo(int i)
```

Interfaces

<modifier>interface<interface name>{generic type names}(base interfaces):

```
interface foo:
       int bar()

       int foo
```

Defines an interface with method bar and property foo.

Member Variables

Declared as:

<modifier><type name>{generic type names}<variable name>

Modifiers can be public, protected, private, internal, shared, or final.

For example:

```
module Variables:
	int foo
	internal float bar
	private List{String} strings
```

Member Methods

Declared as:

<modifier><return type name><method name>{generic type names}(parameter list):

Modifiers can be public, protected, private, internal, or shared.

For example:

```
module Variables:
	void foo(int a):
	internal int pi():
	private int bar{T}(T i):
```

Properties

Declared as:

```
<modifier><type name><property name>:
        get:
        set:
```

Modifiers can be public, protected, private, internal, or shared.

For example:

```
module Variables:
       int a

       int A:
              get:
                     return a
              set:
                     a = value
```

Constructors

<modifier>new(parameter list):

```
class foo:
       new(int i):
```

Explicit Variable Declaration

```
module foo:
       void bar():
              int i
```

Implicit Variable Declaration

```
module foo:
       void bar():
              var i = 0
```

I decided to use the “var” keyword from C# and other languages rather than Python's “i = 0” method of implicit variable declaration. Python's method makes sense in a dynamically typed language where a variable can be reassigned a new type at any time. In a statically typed language the point of type assignment needs to be clearer.

Method Invocation

<method name>{generic type names}(parameter list)

For example:

```
module foo:
	void bar(int i):
		DoStuff(i)

module foo:
	void bar():
		DoStuffGeneric{String}()
```

Indexers

As in C#:

```
module foo:
	void bar(int i):
		var list = new List<String>()
		list.Add("test")
		Console.WriteLine(list[0])
```

Operators

In decreasing order of precedence:

Operator	Associativity	Symbol
Multiplicative	Left	* / %
Additive	Left	+ -
Bitwise shift	Left	<< >>
Relational	Left	< > <= >=
Equality	Left	!= ==
Bitwise AND	Left	&
Bitwise XOR	Left	^
Bitwise OR	Left	|
Logical AND	Left	&&
Logical OR	Left	||
Assignment	Right	*= = /= |= &= %=+= -=

If Blocks

if <condition>:

or

if<condition><expression>

For example:

```
module foo:
	void bar(int i):
		if i > 0 Console.WriteLine("positive")

module foo:
	void bar(int i):
		if i > 0:
			Console.WriteLine("positive")
```

Else Blocks

else:

or

else<expression>

For example:

```
module foo:
	void bar(int i):
		if i > 0 Console.WriteLine("positive")
		else Console.WriteLine("not negative")

module foo:
	void bar(int i):
		if i > 0 Console.WriteLine("positive")
		else:
			Console.WriteLine("not negative")
```

While Loops

while<condition>:

```
module foo:
	void bar(int i):
		while i < 10:
			i += 1
```

For Loops

for<variable>in<value>:

```
module foo:
       void bar():
              for var i in 10:
                     Console.WriteLine(i)
```

Note that the 10 is inclusive: this will count from 0 through 10, NOT 0 through 9.

for<variable>in<value>to<value>:

```
module foo:
       void bar(int [] array):
              for var i in 0 to array.Length - 1:
                     Console.WriteLine(array[i])
```

The next iteration will allow "foreach" type expressions. This will require a more sophisticated Validator than the current one:

```
module foo:
       void bar(String [] array):
              for var s in array:
                     Console.WriteLine(s)
```

Switch Blocks

switch<expression>:
        case <expression>:

```
class foo:
       int bar(int i):
              switch i:
                     case 0:
                            Console.WriteLine(0)
                     case 1:
                            Console.WriteLine(1)
                     default:
                            Console.WriteLine(i)
```

Try/Catch/Finally Blocks

```
class foo:
	int bar(int i):
		try:
			File = new File("test")
		catch FileNotFoundException e:
			Console.WriteLine(e)
		finally:
			File.Close()
```

The Requisite Hello World

```
import System

module Program:
	void Main():
		Console.WriteLine("Hello world!")
```