# `dylan.NET` Programmer's Manual

This is the manual documenting the syntax of the `dylan.NET` programming language.

**Dylan Borg**

# Table of Contents


**1 Introduction** ..... **1**

**2 Syntax** ..... **2**

2.1 General Structure ..... 2
2.2 Referencing Libraries ..... 2
2.3 Importing Namespaces ..... 2
2.4 Debug Mode ..... 3
2.5 Declaring an Assembly ..... 3
2.6 Specifying the Assembly Version ..... 3
2.7 Declaring a Class ..... 4
2.8 Declaring a Field ..... 4
2.9 Declaring a Method ..... 4
2.10 Declaring a Constructor ..... 4
2.11 Conditional Compilation ..... 4

**3 Practical Examples** ..... **5**

# 1 Introduction

`dylan.NET` is a new programming language for the CLR (Common Language Runtime) which produces bytecode in MSIL (Microsoft Intermediate Language) format that is suitable for input to the Xamarin Mono and/or Microsoft .NET Framework virtual machines. The compiler itself i.e. `dylandotnet` runs on the latter virtual machines as well and is in fact *self-hosting, compiles itself.* The intent of this literature is not to explain the command-line invocation of the compiler but rather to explain the syntax of the language (as of v. 11.3.1.4) and to provide some examples that may be used in practice.

**Note:** *Some of the example code in this manual may be an extract from the sample programs accompanying the compiler source code or may be extracts from the compiler source code itself. In such case the source file for the particular extract will be quoted.*

To whet your appetite for the next chapter, below is the standard '`Hello, World!`' program written in `dylan.NET`:

```
#refstdasm "mscorlib.dll"

import System

#debug on

assembly helloworld exe
ver 1.2.0.0

class public auto ansi static Program

        method public static void main()
                Console::WriteLine("Hello, World!")
        end method

end class
```

*Adapted from the '*`helloworld.dyl`*' sample program.*

# 2 Syntax

The aim of this chapter is to explain the structure of of a `dylan.NET` program/library. This will be followed by explanations of the various statement types available in the `dylan.NET` programming language.

## 2.1 General Structure

With reference to the sample program presented in Chapter 1 [Introduction], page 1, the structure of a `dylan.NET` program/library is as follows:

- `#refstdasm` and/or `#refasm` statements which are used to reference class libraries.
- `import` statements which import namespaces from the libraries.
- An optional turning on of debugging as a `#debug` statement.
- The declarations of the assembly name and its version as `assembly` and `ver` statements.
- The actual code for the program/library as a set of classes and their members. *The name '`main`' when used with a method indicates that it is the entry point of the program.*

## 2.2 Referencing Libraries

Sometimes one may require the use of already defined functions and classes that are not part of the current program's/library's sources. The `#refasm` and `#refstdasm` statements allow the referencing of a .NET library inside `dylan.NET` source code. `#refstdasm` allows the refercning of the standard libraries which ship with the .NET runtime. Because of the more organised way used by Mono in installing its libraries, the `#refstdasm` statement can allow referencing any library in the runtime directory (paths are in relation to it) of the .NET profile being used. In the case of `#refasm`, the paths specified are relative to the current working directory.

**Note:** *In both statements, enclosing the path in double-quotes is recommended so as to prevent the compiler's lexer from breaking the path into many tokens, as would happen with '`UNIX`' style paths which include the '`/`' and or '`-`' characters which for* `dylan.NET` *are the division and subtraction operators repectively.*

Some examples of referencing some standard libraries and one non-standard library:

```
//#refstdasm standardlib.dll
#refstdasm "mscorlib.dll"
#refstdasm "System.dll"
#refstdasm "System.Xml.Linq.dll"
#refstdasm "System.Core.dll"
//#refasm path/to/lib.dll
#refasm "lib/mylibrary.dll"
```

## 2.3 Importing Namespaces

The `import` statement allows the import of namespaces. This implies that one does not need to use the full name of classes when writing a class' name since the compiler always tries to prepend imported namespaces with a supplied class name while searching inside the imported libraries and in the generated code for the sources being compiled. The '`Console`' class being used in the code sample in Chapter 1 [Introduction], page 1 is in fact named '`System.Console`' and is found in the '`mscorlib.dll`' library. The `import System` line in the sources allowed the omission of the namespace part of the class name while calling the '`WriteLine`' method.

**Note:** *Enclosing the namespace in double-quotes is needed in certain cases, so as to prevent the compiler's lexer from breaking the namespace spcified into many tokens, as would happen with 'UNIX' style paths which include the '/' and or '-' characters which for* `dylan.NET` *are the division and subtraction operators repectively.*

Some examples of importing some standard namespaces:

```
//import namespace
import System
import System.Xml.Linq
import System.Collections.Generic
import System.Linq
```

**Note:** *In the rare cases when using two or more classes that are* **named the same**, *but are* **from different namsepaces**, *the* `dylan.NET` *compiler will select the class from the namespace that is imported first.*

To make sure the actually needed class gets selected in the latter case, one would need to prepend the namespace to the class' name everytime the class is used. Since the latter methodology is inefficient and requires extra typing, '`aliased namespaces`' can be used instead to be able to assign a short alias to the namepace. As an example, to set the alias '`SCG`' to the '`System.Coillections.Generic`' namespace one would use the following code:

```
//import alias = namespace
import SCG = System.Collections.Generic
```

With the above alias, specifiying the alias as the namespace part of a class name will make the compiler swap it by the namespace assigned to the particular alias.

## 2.4 Debug Mode

The `#debug` statement is used to turn on/off the production of debugging symbols in '`.mdb`' (Mono) or '`.pdb`' (.NET) format. By default debug mode is `off`. Turning the debug symbol making `on` will also imply `#define DEBUG`. For more info on the latter, see Section 2.11 [Conditional Compilation], page 4. To turn debug symbol making on, use `#debug on` or use `#debug off` to make it clearer that no debug symbols will b made. **This statement should occur before the `assembly` and `ver` statements.**

## 2.5 Declaring an Assembly

The `assembly` statement is used to declare an assembly given its name which has to be a single identifier token and mode which can be `exe`, *executable program* or `dll`, *dynamically linked library*. **This statement should occur before the `ver` statement.** See the examples below:

```
//assembly assembly_name exe|dll
assembly testprog exe
assembly testlib dll
```

## 2.6 Specifying the Assembly Version

The `ver` statement defines the version for the assembly declared using a preceding `assembly` statement and actually triggers the creation of a new '`assembly`' by the compiler. The version number is made up of 4 integers, all of which must be specified. For more information on .NET version numbers look up '`System.Version`' on *MSDN*. **The `ver` statement should occur immediately after the `assembly` statement.** An example usage would be:

```
ver 1.1.0.0
```

## 2.7 Declaring a Class

The `class` statement is used when declaring a new class and has the following format:

```
class attributes class_name [extends parent_class] [implements interfaces]
    class_body
end class
```

The *class_name* has to be a valid identifier (current namespace is prepended to the class name in the format *namespace.classs_name*) while the *parentclass* and every member of the comma delimeted list *interfaces* must be valid type names. A valid type name should be similar in form to the following examples:

```
//non-generic
Console
Version
//generic
IEnumerable<of string>
IDictionary<of string, object>
```

The *attributes* list may contain one of the following type attributes:

`public` Makes the class visible to all classes no matter where they are declared

`private` Makes the class be visible only to the assembly in which it is declared

`auto` Lets the runtime lay the class members in its own order

`autochar`

`ansi`

`sequential`
Forces the runtime to lay the class members in the order they were declared

`abstract` Marks the class as being '`abstract`'

`interface`
Marks the class as being '`interface`' (should also use `abstract`)

`beforefieldinit`
Runs the static constructor exactly once, before the first use of the class

`sealed` Disallows inheritance of the class by other classes

`static` Same as specifying all of `sealed`, `beforefieldinit` and `abstract`

## 2.8 Declaring a Field

## 2.9 Declaring a Method

## 2.10 Declaring a Constructor

## 2.11 Conditional Compilation

# 3 Practical Examples