

**FOR PUBLIC
RELEASE**

N I N E

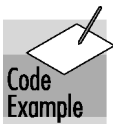
First Steps in PerlNET

*T*his chapter opens the gate into the exciting world of programming Perl within the .NET environment. We start our journey with simple program examples. You will learn how to compile and run PerlNET programs. The new statements that make Perl to the .NET Framework interaction easy to use are introduced. After discussing PerlNET program structure and the use of namespaces, we demonstrate how to incorporate the input and output .NET classes into our programs. Finally, we present the full example program, which involves user interaction and shows how to bring into use Perl-specific constructions inside the .NET environment.

Your First PerlNET Program

As a first step in PerlNET programming, we write a simple program to introduce you to the basics of the new language. Our program outputs a single line of text. Here is the code for the first sample.

```
#  
# Hello.pl  
#  
use namespace "System";  
use PerlNET qw(AUTOCALL);  
  
Console->WriteLine("Hello from Perl!");
```



The above code is saved in the **Hello** folder for this chapter. Optionally, you can just type the program in your favorite editor. If you are using Visual Perl

(see Appendix A), you can open the solution **Hello.sln**. If you are not using Visual Perl, just ignore the solution and project files.

It is commonly known that Perl is a script language and as such is processed by Perl interpreter. So, the first reaction (“Perl instinct”) is to type the following line:

```
perl Hello.pl
```

and to get a “Hello from Perl!” line as an output. If you decided to try it, you got the following probably familiar but unpleasant response:

```
can't locate namespace.pm in @INC (@INC contains: . . .)
at Hello.pl line 4
BEGIN failed - compilation aborted at Hello.pl line 4.
```

Well, this is the moment to remind ourselves that from now on we will use Perl language (or more precisely, its extended version, PerlNET) to target our programs to the .NET environment. Therefore, we should be able to map any Perl program into MSIL (Microsoft Intermediate Language) assembly, which in turn can be executed by the .NET CLR (Common Language Runtime).

The work of compiling and building an assembly is done by **plc.exe** (PerlNET compiler), which comes with the PerlNET distribution. Simply run the following command from your command prompt in the **Hello** directory:

```
plc Hello.pl
```

As a result, **Hello.exe** will be created. Now, you can test your first PerlNET program by executing **Hello.exe**. You should get the following output:

```
Hello from Perl!
```

Congratulations! You’ve just written, built, and executed your first fully functional PerlNET program. Reward yourself with a cup of coffee, and let’s move on to the program discussion.

Sample Overview

The first two lines after the starter comment in our sample are **pragmas**. These are instructions that tell the Perl interpreter how to treat the code that follows the pragma. Usually, you define pragmas at the beginning (header) of your Perl program, and then you write the code. Let us look at the first pragma.

```
use namespace "System";
```

This pragma tells Perl to look up types in the **System** namespace.¹ As a result, we can use the unqualified type **Console** throughout our program. This means

1. All classes in .NET are divided into namespaces. Namespaces are discussed later in this chapter.

that we can write **Console** whenever referring to the **System.Console** class. This class encapsulates a rich functionality of the input/output operations.

Now let us look more closely at the second pragma.

```
use PerlNET qw(AUTOCALL);
```

In short, this line allows us to use the standard Perl call-method syntax when invoking **static** methods (**class** methods) of .NET classes. If we do not import **AUTOCALL**, then we must use the **call** function of the PerlNET module (we discuss this module shortly), as follows:

```
PerlNET::call("System.Console.WriteLine",  
             "Hello from Perl!");
```

The first argument to the **call** function is the **static** method name to call. Starting from the second argument, you should specify the arguments list that you pass to the **static** method. If you specified the **System** namespace with the **use namespace** pragma, then you may omit it and write just **Console.WriteLine** when specifying the **static** method to the **call** function:

```
PerlNET::call("Console.WriteLine", "Hello from Perl!");
```

Combining the two pragmas described above allows us to easily access .NET classes.

```
Console->WriteLine("Hello from Perl!");
```

This statement calls the **static** method **WriteLine** of the **Console** class, which is located in the **System** namespace. The **WriteLine** method is passed a string to output as argument.

PerlNET Module

In the previous section, we introduced the PerlNET module. Throughout this book, we will make wide usage of this module by importing useful functions that help the Perl language tap into the .NET environment.

Whenever you decide to use one of the functions provided by the PerlNET module, you may choose from two forms of syntax:

```
PerlNET::call("System.Console.WriteLine", "Hello");
```

or you may import the function from PerlNET and write as follows:

```
use PerlNET qw(call);  
.  
.  
call("System.Console.WriteLine", "Hello");
```

In most cases, we prefer to use the second form of syntax in this book—importing all the function from the PerlNET module at the header of our Perl

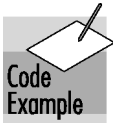
file. If you should use several PerlNET module functions, then you may enumerate them in the single **use** statement instead of importing each separately:

```
use PerlNET qw(AUTOCALL call enum);
```

PLC Limitations

As we saw in the previous sections, **plc.exe**, the PerlNET compiler, is used to build our programs and create assemblies. During compilation, **plc** checks a Perl file for syntactic accuracy. However, this check does not verify the correct spelling of .NET type names or the correct number of arguments passed to .NET methods. This means that you may misspell some .NET class or type name, but the PerlNET compiler will not let you know about this and will create an assembly. As a result, you will be informed about the error only at runtime.

Consider the following code (**HelloErr**), where we intentionally misspelled **Console** and wrote **Consol**:



```
#
# HelloErr.pl
#
use strict;
use namespace "System";
use PerlNET qw(AUTOCALL);
```

```
Consol->WriteLine("Hello, World.\n");
```

If we compile this program, we get no errors and **HelloErr.exe** is created. However, if we run **HelloErr.exe**, then the following error is displayed:

```
System.ApplicationException: Can't locate type Consol
. . .
```

The PerlNET compiler creates the .NET assembly, but internally our code is still being interpreted by the Perl Runtime Interpreter component, which passes our commands to .NET. It serves as a mediator between PerlNET programs and the .NET environment. Therefore, if we write two statements, the first correct and the second with error, then the Perl interpreter will execute the first statement, and on the second, we will get an error message:

```
Console->WriteLine("Hello from Perl");
Consol->WriteLine("Hello, World.\n");
```

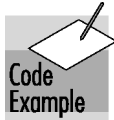
The output will be

```
Hello from Perl
System.ApplicationException: Can't locate type Consol
. . .
```

PerlNET programs, like any Core Perl scripts, should pass through extensive runtime testing before being released.

Main Function

Let's look at a simple program written in C# that performs the same action as our first sample: It displays the single line "Hello from C#." (If you don't know C#, you can refer to Appendix B, "C# Survival Guide for PerlNET Programmers.") We saved the program in the **HelloCs** folder.



```
// Hello.cs
using System;

class Hello
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello from C#");
    }
}
```

As you can see, we defined a class with a single static method **Main**, and this is the minimum requirement of each .NET program.

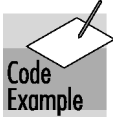
Unlike C# or other .NET-compliant languages, in PerlNET you are not required explicitly to define the **Main** function and wrap your program by the class, which means that our PerlNET programs may be written just as a sequence of statements (like script) and saved in a **.pl** file. In Chapter 12, we will see that, after compiling, our programs are implicitly wrapped by the class that has the static **Main** method. This method is an entry point to our program, and it encapsulates all the statements that we wrote in the **.pl** file in the script-like manner. However, you may want to define the **Main** method explicitly. We will see that it is useful when we learn how to create a graphical user interface in Chapter 14.

Here is how we may rewrite our first sample with the **Main** function definition.

```
#
# HelloMain.pl
#
use namespace "System";
use PerlNET qw(AUTOCALL);

=for interface
    public static void Main(str[] args);
=cut
```

```
sub Main
{
    Console->WriteLine("Hello from Main function");
}
```



The code for the program resides in **HelloMain**. This program introduces additional syntactic structures that are not present in Core Perl. The first thing you may notice is the following POD **=for interface** block:

```
=for interface
    public static void Main(str[] args);
=cut
```

We will make wide use of these blocks when learning about creating .NET components later in the book. Perl is a dynamic type language and recognizes neither method modifiers (**public**, **static**) nor types. As we want to preserve Perl features and still tap into the .NET environment, we are required to find a compromise that is acceptable to both sides. The solution is to use a special **=for interface** block, where all .NET-specific definitions go on. As you can see, our sample defines the **Main** method with the same modifiers, return type, and signature as the C# program.

Namespaces

Earlier in this chapter, we introduced the term **namespace**. In general, all classes in .NET are divided into namespaces to prevent name conflicts. Thus, classes with the same name may reside under the different namespaces. For example, **Samples.Car** and **Examples.Car** are different classes with the same name. We may distinguish them by their namespace.

When writing a program, you may refer to classes or types with a fully qualified name, a name in which the namespace precedes the class name. You can specify the fully qualified name either by delimiting the namespace and class name with two colons or by placing the whole name (both namespace and class) in double quotes and delimiting with a dot.

```
System::Console->WriteLine("Hello");    # Correct
System.Console->WriteLine("Hello");    # Error
"System.Console"->WriteLine("Hello");  # Correct
q(System.Console)->WriteLine("Hello");  # Correct
```

Throughout the book we specify which namespaces to use in the header of our PerlNET programs with the **use namespace** pragma.

Expressions

Internally, PerlNET uses Perl interpreter. This allows us to incorporate all the standard Perl features and expressions in our PerlNET programs. This means that all samples that we introduced in Part 1 may be compiled and executed in the .NET environment without any changes. Let us demonstrate this by a simple Core Perl program, the standard Perl script **freq.pl** for counting word occurrences, which we presented in Chapter 4. Here is the code for the script.

```
#
# freq.pl
#
while(<STDIN>)
{
    @words = split;
    foreach $word (@words)
    {
        $words{$word}++;
    }
}
foreach $word (sort(keys(%words)))
{
    print "$word occurred $words{$word}\n";
}
```

Now, we may run the script either using Perl interpreter,

```
perl freq.pl
```

or by compiling into a .NET assembly and running the assembly:

```
plc freq.pl
freq.exe < text.txt
```

In any case, we get the same result: All word occurrences in text from standard input are calculated and printed. Here is the output for both cases when the input was originated from the following file:

```
Almost everybody can learn Perl programming
Not everybody likes to learn
```

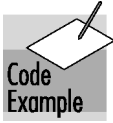
Output:

```
Almost occurred 1
Not occurred 1
Perl occurred 1
can occurred 1
everybody occurred 2
learn occurred 2
likes occurred 1
programming occurred 1
to occurred 1
```

Marshalling Types

Until this chapter, we wrote Core Perl programs without worrying about typing, as Perl has a dynamic typing system. .NET is a strong-typed environment: There is one-to-one correspondence between variables and types. In other words, every variable has one and only one type associated with it.

We do not have to change our attitude with respect to types if our PerlNET program contains Core Perl statements only. However, the problem may occur when working with .NET components and classes.² As we saw in our first sample program, working with .NET classes involves calling methods.³ Many of the methods expect arguments, and we have to supply arguments of the correct type. Consider the following code example from the **Marshalling\Types** folder.



```
#
# Types.pl
#
use strict;
use namespace "System";
use PerlNET qw(AUTOCALL);

my $x = "0";
print Math->Cos($x);
```

This program introduces the static method **Cos** of the **Math** class located in the **System** namespace. This class provides constants and static methods for trigonometric, logarithmic, and other common mathematical functions.

We expect our program to print the numeral **1**, because we rely on Perl for converting the **\$x** variable from string scalar into floating-point number scalar, as we use **\$x** in the numeric context. Unfortunately, building and running the program ruins our expectations, and instead of printing **1** in output, the program exits with the following error message:

```
System.ApplicationException: Can't locate public static
method Cos(System.String) for System.Math.
```

The error means that we supplied an argument with an incorrect type to the **Cos** function, which expects a floating-point number. Perl context-based conversion did not work here because PerlNET runtime modules that serve as a bridge between Perl and .NET environments convert scalars into specific .NET

2. We discuss working with .NET components in detail in Chapter 10. We mention it here for problem-illustrating purposes.
3. In this chapter we work with static .NET methods only, but the discussion of marshalling types is relevant for all .NET methods.

types. PerlNET defines different rules than Core Perl defines for type conversions when we call .NET methods and pass to them Perl scalars.

Internally, Perl caches multiple representations of a value for each variable:

- String
- Integer
- Numeric (float)

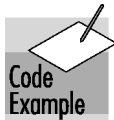
In other words, Perl “remembers” the last representation of a value and sometimes it interprets the type not in the way we want it to.

Back to our example. We assigned 0 to our `$x` variable. The type of value is **String**, and this is how it is remembered by `$x`. To fix it, we should clear the part of `$x`'s memory that remembers the type as **String**. There are several ways to do it. We may perform mathematical operations on `$x` that will not change it (e.g., add zero or multiply by 1) and store the result back to `$x` (the first way in the example below) or pass the mathematical expression to the method without assigning to the variable (the second way shown). Optionally, we may use the **Convert** class from the **System** namespace. Its **ToDouble** function will return a numeric value (the third way). Here are some possible ways to rewrite the **Cos** statement in our **Types.pl** program.

```
$x = $x + 0;
print Math->Cos($x);           # 1-st way
. . .
print Math->Cos($x*1);         # 2-nd way
. . .
print Math->Cos(Convert->ToDouble($x)); # 3-rd way
```

After you pick up the way you like it, rebuild and run the program. You should get a clean execution. We saved the correct version of the program in **Marshalling\TypesFix**.

However, such solutions will work only for float (double) numbers. To avoid any ambiguity in interpreting your Perl scalar variables, PerlNET offers a set of special casting functions for value types. We explain these functions and marshalling types in Chapter 10 when we discuss calling methods.



Input/Output

As we stated in the previous section, all Core Perl constructions may be brought into use in PerlNET. Input/output issues are not an exception.

Performing Output

Instead of using a long call to the **WriteLine** method of the **Console** class, we can write

```
print "Hello from Perl!\n";
```

and we will get the same result as if had written

```
Console->WriteLine("Hello from Perl!");
```

The reason we use the **Console** class rather than Core Perl **print** is to demonstrate the .NET concepts on simple examples and to show the .NET way to do things. You may combine Core Perl and .NET statements to perform output. However, you may run into buffering problems when running the program in some environments, so that your program output would not be in the original order that you meant when designing the program. This may occur when you test your Managed Exe Perl project in Visual Studio running it with Ctrl-F5 (we present the Visual Studio description in Appendix A). The reason for this is that the .NET methods and Core Perl functions use different buffers for output and you do not control the order in which these buffers are flushed. To prevent this, you should add the following statement, which instructs Perl not to buffer the output, at the beginning of your program:

```
$| = 1;
```

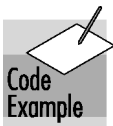
Examine the **IO\ Buffering** sample.

```
#  
# Buffering.pl  
#  
use strict;  
use namespace "System";  
use PerlNET qw(AUTOCALL);  
  
$| = 1;  
Console->WriteLine("1");  
print "2\n";  
Console->WriteLine("3");
```

If you comment the `$| = 1` statement, then the output in the Visual Studio Run window will look like what follows.

```
1  
3  
2
```

This is definitely not the order of printing we meant. If you uncomment the assignment, this will restore the original output order.



Getting Input

Most applications involve user interaction. Core Perl offers a simple way to treat user input:

```
<STDIN>;
```

As you probably guessed, the .NET SDK introduces several methods to treat user input. We will show how to take benefit of these methods.

You may implement a simple text-based user interface with the **ReadLine** method of the **Console** class. This method does not take any parameters and returns the string that it reads from the standard input device.

Before we present a sample program, we should explain another feature of the **WriteLine** method: placeholders. When performing output, often you need to include variables or expressions whose values are known only at runtime. The **WriteLine** method reserves a place for the expression in the output string. For each expression, there are placeholders {0}, {1}, and so on. The expressions should be placed after closing quotes and should be delimited by commas. Here is the simple example.

```
my $x = 3;
Console->WriteLine("x={0};x+1={1};x^2={2}", $x, $x+1, $x**2);
```

The output is

```
x=3;x+1=4;x^2=9
```

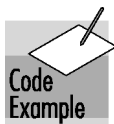
The following sample takes as input float number **x** and calculates its exponent. We use the static method **Exp** of the **Math** class. Our program demonstrates two ways of calculating the exponent.

```
#
# Exp.pl
#
use namespace "System";
use PerlNET qw(AUTOCALL);

Console->WriteLine("Please enter a number:");

# Get input from user
my $x = Console->ReadLine();

Console->WriteLine("1)exp({0}) = {1}", $x, Math->Exp($x));
Console->WriteLine("2)exp({0}) = {1}", $x, Math->E ** $x);
```



You may find the above program in the **IO\Exp** folder. To compile the program type

```
plc Exp.pl
```

If you did not make any syntax mistakes, the compilation should accomplish successfully. Now run **Exp.exe**. Enter some number when the program asks you to, and examine the response. You should get an error message:

```
System.ApplicationException: Can't locate public static
method Exp(System.String) for System.Math.
```

The problem is caused by the **Math->Exp(\$x)** statement. If we comment the whole line, compile, and run again, there should not be any error messages. The output may look as shown below.

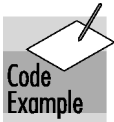
```
Please enter a number:
1
2) exp(1) = 2.71828182845905
```

By now, you should know how to fix the first **WriteLine** statement to get a clean execution. The **ReadLine** method of the **Console** class returns **System.String** that PerlNET runtime module delivers to our program as a Perl string scalar. The solution is to clear the string representation. We do it by passing the **\$x+0** value to the **Exp** method.

```
Console->WriteLine("1)exp({0}) = {1}", $x, Math->Exp($x+0));
```

We saved the correct version of the program in **IO\ExpFix**. Now, after rebuilding and running the exponent program again, you should get no error messages.

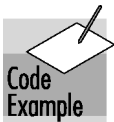
```
Please enter a number:
2.5
1) exp(2.5) = 12.1824939607035
2) exp(2.5) = 12.1824939607035
```



Main Sample

In this section, we combine the knowledge that we provided throughout this chapter. We present the full example program, which you should easily understand. We use the **Console** class, since we want you to feel comfortable when combining .NET types with Core Perl statements and expressions.

Our sample performs calculations of binary operations (+, -, *, /). It gets as an input an expression (**x op y**) where **x** and **y** are integer positive numbers and **op** is one of the four operators mentioned above. Our .NET program validates the input using Perl regular expressions (!) and then evaluates the expression. To test the following code, refer to the **MainSample\BinOper** folder.



```
#
# BinOper.pl
```

```
#
use namespace "System";
use PerlNET qw(AUTOCALL);
use strict;

# Full qualified type reference
q(System.Console)->WriteLine("Please enter expression to
                             evaluate:");

# Get an expression from the user - Another way to refer to
# Console class
my $expr = System::Console->ReadLine();

#remove whitespaces from $expr
$expr =~ s/\s*/g;

# Check for valid format
if ($expr =~ /^(^d+)([\+, \-, \*, \\/]{1})(\d+$)/)
{
    # Evaluate user expression
    my $result = eval $expr;

    # Check if the result is OK
    if (!defined $result)
    {
        die "Illegal operation: $expr";
    }

    # Output - Refer to Console without namespace
    # specification
    Console->WriteLine("Result: {0}\n", $result);
}

# Handle wrong input
else
{
    die "Incorrect expression format";
}
```

As you can see, we take advantage of the powerful Perl constructions to validate input. **eval** simply evaluates an expression if it was supplied in correct form. The **die** statement is used to handle errors. Examples of running the program may look like this:

```
Please enter expression to evaluate:
64 + 36
Result: 100
-----
Please enter expression to evaluate:
d-5
```

```
Incorrect expression format at binoper.pl line 38.
-----
Please enter expression to evaluate:
10 / 0
Illegal operation: 10/0 at binoper.pl line 27.
```

Take some time to play with the program. Supply different inputs and examine the response. Enjoy!

Summary

In this chapter, we touched on the basics of Perl programming within the .NET environment. The combination of Perl and the .NET extensions is called PerlNET. Pragmas were introduced to simplify program code. All classes in .NET are divided into namespaces. **System** namespace exposes **Console** class (among many others), which provides static methods for performing input/output operations and can be used in any PerlNET programs. These methods may be replaced by Core Perl statements (**print** and **<STDIN>**). All other standard Perl constructions and features are supported in PerlNET as well.

There are several ways to write an application. You may choose a script-like form or an object-oriented approach by supplying interface definitions and explicitly defining the **Main** function.

.NET has a wide range of classes that may be incorporated into our programs. We saw how to use classes, which provide static methods: **Console**, **Math**, and **Convert**. In the next chapter, we will learn in detail about the .NET components and review the most popular and useful classes.