

Tutorials

January 21, 2006

Contents

1	Tutorials and examples	1
1.1	Introduction	1
1.2	Short tutorials for writing example applications	1
1.3	Snippets	1
1.4	Larger examples	2
1.4.1	Sokoban solver	2
1.4.2	Nemerle Course programs	2
1.4.3	ICFP'2004 Programming Contest entry	2
1.4.4	The Great Language Shootout Examples	2
1.4.5	ntrace	2
1.4.6	Sioux HTTP server	3
1.5	External projects created using Nemerle	3
1.5.1	RiDL - Compiler tools	3
1.5.2	Speagram	3
1.5.3	Asper	3
2	First Tutorial	3
2.1	Introduction	3
2.1.1	Nemerle, .NET, and Mono	3
2.1.2	Getting Started	4
2.2	Simple Examples	4
2.2.1	Hello World	4
2.2.2	The Adder	5
2.2.3	Counting Lines in a File	6
2.3	Functional examples	7
2.3.1	Rewriting Line Counter without the loop	7
2.3.2	Rewriting line counter without mutable values	8
2.3.3	Type inference	9
2.4	More info	9
3	Second Tutorial	9
3.1	Small app with array and Hashtable	10
3.1.1	Counting unique elements	10
3.1.2	What about functional programming?	11
4	Gtk text editor (tutorial)	11
4.1	Simple text editor in GTK#	11
4.1.1	Compiling examples	11
4.1.2	Part I creating a window	12
4.1.3	Part II adding text input section	12
4.1.4	Part III a simple menu	13
4.1.5	Part IV adding submenus	13
4.1.6	Part V adding tasks to the submenus	14
4.1.7	Result	15

5	System.Windows.Forms tutorial	17
5.1	The first step	18
5.1.1	Buttons, labels and the rest	18
5.1.2	A simple menu	19
5.1.3	Basic events	19
5.1.4	Cleaning up	20
5.2	The second step	21
5.2.1	Painting	21
5.2.2	Overriding event handlers	21
5.2.3	A simple animation with double-buffering	22
5.2.4	Bitmaps and images	23
5.2.5	Adding icons to the menu	24
6	Macros tutorial	24
6.1	What exactly is a macro?	25
6.2	Defining a new macro	25
6.2.1	Compiling a simplest macro	25
6.2.2	Exercise	26
6.3	Operating on syntax trees	26
6.3.1	Quotation operator	26
6.3.2	Matching subexpressions	27
6.3.3	Base elements of grammar	27
6.3.4	Constructs with variable amount of elements	28
6.3.5	Exercise	28
6.4	Adding new syntax to the compiler	28
6.4.1	Exercise	29
6.5	Macros in custom attributes	29
6.5.1	Executing macros on type declarations	29
6.5.2	Manipulating type declarations	30
6.5.3	Execution stages	31
6.6	Reference to more advanced aspects	32
6.6.1	Hygiene and alpha-renaming of identifiers	32
7	ASP.NET (tutorial)	34
7.1	Preparations	34
7.2	Example page	34
8	Remoting (tutorial)	36
8.1	Client	36
8.2	Server	37
8.3	ServerObjects	38
8.4	Compilation	40
8.5	Serializing Variants	40
8.5.1	Using a macro to mark all options with attribute	40
9	PInvoke (tutorial)	41
10	Category:Tutorials	42
10.1	Articles in category "Tutorials"	42
10.1.1	B	42
10.1.2	F	42
10.1.3	G	42
10.1.4	P	42
10.1.5	R	42
10.1.6	S	43
10.1.7	T	43
11	Tutorials and examples	43
11.1	Introduction	43

11.2	Short tutorials for writing example applications	43
11.3	Snippets	43
11.4	Larger examples	44
11.4.1	Sokoban solver	44
11.4.2	Nemerle Course programs	44
11.4.3	ICFP'2004 Programming Contest entry	44
11.4.4	The Great Language Shootout Examples	44
11.4.5	ntrace	44
11.4.6	Sioux HTTP server	44
11.5	External projects created using Nemerle	45
11.5.1	RiDL - Compiler tools	45
11.5.2	Speagram	45
11.5.3	Asper	45

1 Tutorials and examples

1.1 Introduction

This page lists some snippets of Nemerle code you may want to look at. If you have any examples to be submitted here – please edit this page or contact us.

1.2 Short tutorials for writing example applications

The tutorials gathered here are meant to be used as an easy introduction to the the language, by stepping through the process of writing small applications.

- [Don't Panic! - Nemerle Basics Explained](#) – a simple tutorial covering basic concepts of the Nemerle language.
- [Hashtables and foreach](#) – presentation of how type inference makes code cleaner.
- [Building a text editor using GTK#](#).
- [System.Windows.Forms tutorial](#)
- [Macros tutorial](#) – the basics of meta-programming in Nemerle.
- Using [ASP.NET](#) with Nemerle.
- [Remoting](#) – presents a technique of accessing remote objects on the server.
- [PInvoking](#) – gives insight on how to use native (unmanaged) libraries from Nemerle.

There is also [Grokking Nemerle](#) – a ”book” covering most of the language.

1.3 Snippets

You can find various examples of Nemerle code in [snippets](#) directory in Nemerle source tree. You are welcomed to transform them into tutorials. Examples include:

- [lcs.n](#) – short program computing the longest common substring
- [sql.n](#) – an example of database connectivity using Nemerle macros
- [suffix.n](#) – suffix trees
- [form.n](#) – a Windows.Forms example
- [myPoll.n](#) – an example of a CGI application connecting to a PostgreSQL database
- [power-race.n](#) – a ncurses-based ”game”. [Screenshot](#).
- [boyer-moore.n](#) – a Boyer-Moore algorithm

- [knuth-morris-pratt.n](#) – a Knuth-Morris-Pratt algorithm
- [nondec-subseq.n](#) – the longest non-decreasing subsequence
- [rachunki.n](#) – a billing generator for a conference
- [shift-or.n](#) – a shift-or text searching algorithm

These are also bundled with the source distribution.

1.4 Larger examples

There are also several more advanced examples there:

1.4.1 Sokoban solver

[Sokoban](#) – Sokoban solver by Bartosz Podlejski

1.4.2 Nemerle Course programs

Examples done by students during the Nemerle course at the CS Institute in the Wroclaw University.

- [ERA-SMS-Sender](#) – allows sending an SMS to a Polish GSM provider ERA (uses a POP3 client). By **Adrian Macal** *mel_on0 at o2 dot pl*.
- [backup-tool.n](#) – a program for performing backups, using simple scripts. By **Ryszard Trojnacki** *rysiek at menel dot com*.
- [freedb.org client](#) – a simple program to download information from [freedb.org](#) database for an audio CD currently in the drive under Linux. By **Wojtek Knapik** *d at hell dot art dot pl*.
- [swf-calculator.n](#) – a simple calculator using System.Windows.Forms. By **Marek Czajka** *marek_czajka at wp dot pl*.
- [huffman](#) – huffman compression and decompression. By **Ania Dwojak** *andzia200 at wp dot pl*.
- [getmxbyname.n](#) – a class to find out the MX record for a domain. By **Marcin Skórzewski** *pirol at o2 dot pl*.

1.4.3 ICFP'2004 Programming Contest entry

In June 2004 there was a [contest](#) to create finite state automata controlling the ants. Our entry is now available [here](#).

This year's contest page of our team is located here [ICFPC2005](#).

1.4.4 The Great Language Shootout Examples

[These](#) are the few examples ported from the [Win32 Computer Language Shootout](#). We didn't submit them yet, looking for volunteers to port more. When we have enough examples, we will also send them to the recently linked [renewed shootout](#).

1.4.5 ntrace

This [simple program](#) will trace memory leaks in programs written in C. By Jacek Śliwerski.

1.4.6 Sioux HTTP server

A **simplistic HTTP server**. The main idea behind it is to serve web applications written in Nemerle. It already served as a subscription server for the XVIII-th FIT (a local conference) and later for the **CSL 2004**.

1.5 External projects created using Nemerle

1.5.1 RiDL - Compiler tools

RiDL is a set of tools to simplify building compilers using Nemerle. It includes a lexical analyzer generator and a parser generator.

It is created by **Kojo Adams** and uses **BSD Licence**.

1.5.2 Speagram

Speagram is an eager purely functional language with a strong type system and unusual syntax resembling natural language.

1.5.3 Asper

Asper is a text editor and IDE for Nemerle written in Nemerle.

2 First Tutorial

2.1 Introduction

This tutorial describes the basics of programming in Nemerle. It is also a quick tour of some of the features and programming concepts that distinguish Nemerle from other languages.

We assume the reader is familiar with C#, Java or C++. But even if you are new to programming, you should find the code easy to understand.

2.1.1 Nemerle, .NET, and Mono

Nemerle is a *.NET* compatible language. As such, it relies heavily on the **.NET Framework**, which not only defines how critical parts of all .NET languages work, but also provides these services:

- A runtime environment, called the Common Language Runtime (CLR), which is shared among all .NET languages. This is similar to the Java VM
- A set of libraries, called the Base Class Libraries (BCL), which contain thousands of functions that perform a wide range of services required by programs

The BCL and CLR ensure that programs written in any .NET language can be easily used in any other .NET language. These features of language neutrality and interoperability make .NET an attractive platform for development.

Further, Nemerle is compatible with **Mono**, an open-source implementation of the published *.NET Common Language Infrastructure (CLI)* standard. This opens up the exciting possibility of writing .NET programs that not only work across different languages, but span different operating systems. With Mono, you can design programs that run on Linux, Mac OS X, and the BSD Unix flavors, as well as Windows.

So, while Nemerle is defined (and constrained) in part by the .NET Framework, it is very much its own language. It offers a unique combination of features not found in other .NET languages, which give it advantages of conciseness and flexibility that should prove attractive to a wide range of programmers, from students to seasoned developers.

2.1.2 Getting Started

To run the examples in this tutorial, you will need to **install Nemerle**. Hacker types will want to `!REPLACE_ME_PLEASE title="Download">download;/a</code> the source.`

2.2 Simple Examples

This section lists simple examples that look almost the same in C# (or Java, or C++).

2.2.1 Hello World

We start with a classic first example:

```
System.Console.WriteLine ("Hello world!");
```

To run this example:

- Write it with your favorite text editor and save it as `hello.n`
- Get to the console by running `cmd` in Windows, or the terminal window in Linux/BSD
- Run the Nemerle compiler by typing `ncc hello.n`
- The output goes to `out.exe`
- Run it by typing `out` or `mono out.exe` depending on your OS

Observe how an individual code statement ends with a semi-colon;

This program writes "Hello world!" to the console. It does this by calling `System.Console.WriteLine`, a function in the .NET Framework.

As this example shows, you can write a bunch of statements in a file (separated by semi-colons), and Nemerle will execute them. However, this example really isn't a proper, stand-alone program. To make it one, you need to wrap it in a class.

Classes: a First Look Lets expand our example to include a class. Enter these lines in your `hello.n` file:

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

Notice how blocks of code are grouped together using curly braces `{ }`, typical of C-style programs.

When you compile and run this program, you get the same results as before. So why the extra lines? The answer is that Nemerle, like most .NET languages, is object-oriented:

- `class Hello` simply means that you are defining a *class*, or type of object, named `Hello`. A class is a template for making objects. Classes can be used standalone, too, as is the case here.
- `static Main` defines a function `Main`, which belongs to class `Hello`. A function that belongs to a class is called a *method* of the class. So, here you would say "function `Main` is a method of class `Hello`."
- By convention, program execution starts at `static Main`. The keyword `static` means the method can be called directly, without first creating an object of type `Hello`. Static methods are the equivalent of public or module-level functions in non-object languages.

This example is much closer to what a C# programmer would write. The only difference is that in Nemerle we write the method's return type on the right, after the colon. So, `static Main():void` specifies that method `Main` returns `void`, or no usable type. This is the equivalent of a subroutine in Basic.

2.2.2 The Adder

Adder is a very simple program that reads and adds two numbers. We will refine this program by introducing several Nemerle concepts.

To start, enter and compile this code:

```
/* Our second example.
   This is a comment. */

using System;

// This is also a comment

public class Adder    // As in C#, we can mark classes public.
{
    public static Main () : void    // Methods, too.
    {
        /* Read two lines, convert them to integers and return their
           sum. */
        Console.WriteLine ("The sum is {0}",
                           // System.Int32.Parse converts string into integer.
                           Int32.Parse (Console.ReadLine ()) +
                           Int32.Parse (Console.ReadLine ()));
    }
}
```

When run, Adder lets you type in two numbers from the console, then prints out the sum.

As you can see, a lengthy statement can be continued on multiple lines, and mixed with comments, as long as it ends with a semi-colon;

The `using` declaration imports identifiers from the specified namespace, so they can be used without a prefix. This improves readability and saves typing. Unlike C#, Nemerle can also import members from classes, not only from namespaces. For example:

```
using System;
using System.Console;

public class Adder
{
    public static Main () : void
    {
        WriteLine ("The sum is {0}",
                   Int32.Parse (ReadLine ()) +
                   Int32.Parse (ReadLine ()));
    }
}
```

You probably noticed that the code that reads and converts the integers is needlessly duplicated. We can simplify and clarify this code by factoring it into its own method:

```
using System;

public class Adder
{
    // Methods are private by default.
    static ReadInteger () : int
    {
        Int32.Parse (Console.ReadLine ())
    }

    public static Main () : void
    {
        def x = ReadInteger (); // Value definition.
        def y = ReadInteger ();
        // Use standard .NET function for formatting output.
        Console.WriteLine ("{0} + {1} = {2}", x, y, x + y);
    }
}
```

Within the Main method we have defined two *values*, x and y. This is done using the `def` keyword. Note that we do not write the value type when it is defined. The compiler sees that `ReadInteger` returns an `int`, so therefore the type of x must also be `int`. This is called *type inference*.

There is more to `def` than just declaring values: it also has an impact on how the value can be used, as we shall see in the next section.

In this example we see no gain from using `def` instead of `int` as you would do in C# (both are 3 characters long :-). However `def` will save typing, because in most cases type names are far longer:

```
FooBarQuxxFactory fact = new FooBarQuxxFactory (); // C#
def fact = FooBarQuxxFactory (); // Nemerle
```

When creating objects, Nemerle does not use the `new` keyword. This aligns nicely with the .NET concept that all types, even simple ones like `int` and `bool`, are objects. That being said, simple types are a special kind of object, and are treated differently during execution than regular objects.

2.2.3 Counting Lines in a File

```
class LineCounter
{
    public static Main () : void
    {
        // Open a file.
        def sr = System.IO.StreamReader ("SomeFile.txt"); // (1)
        mutable line_no = 0; // (2)
        mutable line = sr.ReadLine ();
        while (line != null) { // (3)
            System.Console.WriteLine (line);
            line_no = line_no + 1; // (4)
            line = sr.ReadLine ();
        }; // (5)
        System.Console.WriteLine ("Line count: {0}", line_no);
    }
}
```

Several things about this example require further remarks. The first is the very important difference between the lines marked (1) and (2).

In line (1) we define an *immutable* variable, `sr`. Immutable means the value cannot be changed once it is defined. The `def` statement is used to mark this intent. This concept may at first seem odd, but quite often you will find the need for variables that don't change over their lifetime.

In (2) we define a *mutable* variable, `line_no`. Mutable values are allowed to change freely, and are defined using the `mutable` statement. This is the Nemerle equivalent of a variable in C#. All variables, mutable or not, have to be initialized before use.

In (3) we see a `while` loop. While the line is not null (end of file), this loop writes the line to the console, counts it, and reads the next. It works much like it would in C#. Nemerle also has `do ... while` loops.

We see our mutable counter getting incremented in (4). The assignment operator in Nemerle is `=`, and is similar to C#.

Lastly, in (5), we come to the end of our `while` loop code block. The line count gets written after the `while` loop exits.

2.3 Functional examples

This section introduces some of the more functional features of Nemerle. We will use the functional style to write some simple programs, that could easily be written in the more familiar imperative style, to introduce a few concepts of the functional approach.

Functional Programming: a First Look Functional programming (FP) is style in which you do not modify the state of the machine with instructions, but rather evaluate functions yielding newer and newer values. That is, the entire program is just one big expression. In purely functional languages (Haskell being the main example) you cannot modify any objects once they are created (there is no assignment operator, like `=` in Nemerle). There are no loops, just *recursive functions*. A recursive function calls itself repeatedly until some end condition is met, at which time it returns its result.

Nemerle does not force you to use FP. However you can use it whenever you find it necessary. Some algorithms have a very natural representation when written in functional style – for example functional languages are very good at manipulating tree-like data structures (like XML, in fact XSLT can be thought of as a functional language).

We will be using the terms *method* and *function* interchangeably.

2.3.1 Rewriting Line Counter without the loop

Let's rewrite our previous Line Counter example using a recursive function instead of the loop. It will get longer, but that will get fixed soon.

```
class LineCounterWithoutLoop
{
    public static Main () : void
    {
        def sr = System.IO.StreamReader ("SomeFile.txt");
        mutable line_no = 0;

        def read_lines () : void {           // (1)
            def line = sr.ReadLine ();
            when (line != null) {           // (2)
                System.Console.WriteLine (line); // (3)
                line_no = line_no + 1;      // (4)
                read_lines ()              // (5)
            }
        }
    }
}
```

```

};

read_lines ();    // (6)

    System.Console.WriteLine ("Line count: {0}", line_no); // (7)
}
}

```

In (1) we define a *nested method* called `read_lines`. This method simulates the `while` loop used in our previous example. It takes no parameters and returns a void value.

(2) If line wasn't null (i.e. it was not the last line), (3) we write the line we just read, (4) increase the line number, and finally (5) call ourself to read rest of the lines. The `when` expression is explained below.

Next (6) we call `read_lines` for the first time, and finally (7) print the line count.

The `read_lines` will get called as many times as there are lines in the file. As you can see this is the same as the `while` loop, just expressed in a slightly different way. It is very important to grok this concept of writing loops as recursion, in order to program functionally in Nemerle.

If you are concerned about performance of this form of writing loops – fear not. When a function body ends with call to another function no new stack frame is created. It is called a **tail call**. Thanks to it the example above is as efficient as the `while` loop we saw before.

In Nemerle the `if` expression always need to have the `else` clause. It's done this way to avoid stupid bugs with dangling-else:

```

// C#, misleading indentation hides real code meaning
if (foo)
    if (bar)
        m1 ();
else
    m2 ();

```

If you do not want the `else` clause, use `when` expression, as seen in the example. There is also `unless` expression, equivalent to `when` with the condition negated.

2.3.2 Rewriting line counter without mutable values

Our previous aim of rewriting line counter removed the loop and one mutable value. However one mutable value has left, so we cannot say the example is written functionally. We will now kill it.

```

class FunctionalLineCounter
{
    public static Main () : void
    {
        def sr = System.IO.StreamReader ("SomeFile.txt");
        def read_lines (line_no : int) : int { // (1)
            def line = sr.ReadLine ();
            if (line == null) // (2)
                line_no // (3)
            else {
                System.Console.WriteLine (line); // (4)
                read_lines (line_no + 1) // (5)
            }
        }
    };
}

```

```
        System.Console.WriteLine ("Line count: {0}", read_lines (0)); // (6)
    }
}
```

In (1) we again define a nested method called `read_lines`. However this time it takes one integer parameter – the current line number. It returns the number of lines in the entire file.

(2) If line we just read is null (that was last line), we (3) return the current line number as number of lines in entire file. As you can see **there is no return statement**. The return value of a method is its last expression.

(4) Otherwise (it was not last line) we write the line we just read. Next (5) we call ourselves to read the next line. We need to increase the line number, since it is the next line that we will be reading. Note that as a return value from this invocation of `read_lines` we return what the next invocation of `read_lines` returned. It in turn returns what the next invocation returned and so on, until, at the end of file, we reach (3), and final line count is returned through each invocation of `read_lines`.

In (6) we call the `read_lines` nested method, with initial line number of 0 to read the file and print out line count.

2.3.3 Type inference

We have already seen type inference used to guess types of values defined with `def` or `mutable`. It can be also used to guess type of function parameters and return type. Try removing the `: int` constraints from line marked (1) in our previous example.

Type inference only works for nested functions. Type annotations are required in top-level methods (that is methods defined in classes, not in other methods). This is design decision, that is here not to change external interfaces by accident.

It is sometimes quite hard to tell the type of parameter, from just looking how it is used. For example consider:

```
class HarderInference
{
    static Main () : int {
        def f (x) {
            x.Length
        };
        f ("foo");
    }
}
```

When compiling the `f` method we cannot tell if `x` is a string or array or something else. Nevertheless, we can tell it later (looking at `f` invocation) and Nemerle type inference engine does it.

If function with incomplete type information was not used or its type was ambiguous, compiler would refuse to compile it.

2.4 More info

Now, once you read through all this, please move to the [Grokking Nemerle](#) tutorial, that is much more complete. You can also have a look at [The Reference Manual](#) if you are tough.

3 Second Tutorial

We will write some example code, so you can see some common code patterns when writing programs in Nemerle.

3.1 Small app with array and Hashtable

Let us create an array and print its elements walking through it in loop.

```
using System.Console;

module MyApp
{
    Main () : void
    {
        // create an array containing strings (array [string])
        def elements = array ["{0}", "a {0} a", "{0}", "b {0}", "b {0}"];

        // iterate through elements of array
        for (mutable i = 0; i < elements.Length; i++)
            // print current value of 'i' with using format from array
            WriteLine (elements [i], i);
    }
}
```

The keys stored in *elements* array looks a little bit strange, but we will use *WriteLine* method as a good example of processing key together with value associated to it.

3.1.1 Counting unique elements

We will now make the collection of keys unique. For this purpose we will use generic dictionary type, which in Nemerle is called *Nemerle.Collections.Hashtable*.

```
using Nemerle.Collections;
using System.Console;

module MyApp
{
    Main () : void
    {
        // create an array containing strings (array [string])
        def elements = array ["{0}", "a {0} a", "{0}", "b {0}", "b {0}"];

        def hash = Hashtable ();

        // count the number of occurrences of each unique element
        foreach (e in elements)
        {
            unless (hash.Contains (e))
                hash[e] = 0;

            hash[e] += 1;
        }

        // print each element with number indicating its total count
        foreach (keypair in hash) {
            WriteLine (keypair.Key, keypair.Value);
        }
    }
}
```

For each entry in hashtable we store the number of how many times it occurred.

3.1.2 What about functional programming?

Ok, iterating over collection with *foreach* is nice, but it is still **too long**!

```
// print each element with number indicating its total count
foreach (keypair in hash)
    WriteLine (keypair.Key, keypair.Value);

// pass matching overload of WriteLine to iteration method
hash.Iter (WriteLine)
```

Hashtable have the special *Iter* overload, which takes a generic two-argument function, which will be called for every element stored in collection. The declaration of *Iter* looks like this:

```
class Hashtable ['a, 'b]
{
    public Iter (f : 'a * 'b -> void) : void
    { ... }
    ...
}
```

and after substituting *'a* and *'b* with actual types in our *hash*, we just use it as `Iter : (string * int -> void) -> void`, which takes `string * int -> void` function as the only argument.

And one more way of iterating over a *Hashtable* is to query *KeyValuePairs* property and use pairs of key and value in **foreach** statement:

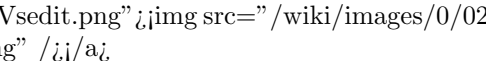
```
foreach( (key, value) in hash.KeyValuePairs)
    WriteLine(key, value);
```

4 Gtk text editor (tutorial)

4.1 Simple text editor in GTK#

In this tutorial you will see step-by-step how to build a simple text editor in Nemerle using *Gtk#*. *Gtk#* is a .Net binding for the **GTK+** GUI toolkit. It is an alternative to [Windows.Forms](#).

The final appearance of our application will be something like:

[http://wiki.nemerle.org/Image:Vsedit.png](http://wiki.nemerle.org/Image:Vsedit.png "Image:Vsedit.png") 
 alt="Image:Vsedit.png" width="312" height="229" longdesc="/Image:Vsedit.png" />

4.1.1 Compiling examples

In order to compile and run examples presented below, you need to install **Gtk#**. Assuming you have `pkg-config` installed, you can compile `gtk#` examples using

```
ncc example.n -pkg:gtk-sharp -out:example.exe
```

Otherwise you will need to copy `gtk-sharp.dll` to the current directory and use

```
ncc example.n -r:gtk-sharp.dll -out:example.exe
```

4.1.2 Part I creating a window

First we must create an application and display a window, which will be the base of our editor:

```
using System;
using Gtk;

class MainWindow : Window
{
    public this()
    {
        // set caption of window
        base ("Very Simple Editor");
        // resize windows to some reasonable shape
        SetSizeRequest (300,200);
    }
}

module SimpleEditor
{
    Main() : void
    {
        Application.Init();
        def win = MainWindow();

        // exit application when editor window is deleted
        win.DeleteEvent += fun (_) { Application.Quit () };

        win.ShowAll ();
        Application.Run();
    }
}
```

The *MainWindow* class is quite simple now, but we will add things to it in a moment.

SimpleEditor is a module with one method, *Main* - the entry point of our application. It is executed first when the program is run and creates the editor window.

One more thing to note is *win.DeleteEvent*, which is a signal triggered when the window is destroyed (for example user clicks close). We specify what should happen by attaching an anonymous function to the event. Here, we will simply quit the application.

4.1.3 Part II adding text input section

Now we need to add an area to the window where actual text can be written by the user.

Modify the *MainWindow* class to hold the *TextView* object:

```
class MainWindow : Window
{
    /// text input area of our window
    input : TextView;

    public this()
    {
        // set caption of window
        base ("Very Simple Editor");
        // resize windows to some reasonable shape
```

```
        SetSizeRequest (300,200);

        def scroll = ScrolledWindow ();
        input = TextView ();
        scroll.Add (input);

        // place scrolledwin inside our main window
        Add (scroll);
    }
}
```

In our window's constructor we must choose how the text input will be positioned. We use a Gtk container to encapsulate *TextView* into *ScrolledWindow*. This allows text in the text view to be easily viewed using scroll bars.

Add (scroll) adds the *ScrolledWindow* as part of our text editor window. So, now we have created working text input and viewing.

4.1.4 Part III a simple menu

But we still want to perform some more advanced operations in our editor. Ha, we will even need a menu! We will create a menu and put it into a vertical box. Instead of adding scrolling to the main window, put it into the box as well:

```
def menu = MenuBar ();
def mi = NMenuItem ("File");
menu.Append(mi);

def vbox = VBox ();
vbox.PackStart (menu, false, false, 0u);
vbox.PackStart (scroll, true, true, 0u);

// place vertical box inside our main window
Add (vbox);
```

In the last instruction we add the entire vertical box to the main window, above the text view.

NMenuItem, defined below, is the next class we will work with. We will extend it and add some actions to it. Here is the initial code:

```
class NMenuItem : MenuItem
{
    public name : string;

    public this(1 : string)
    {
        base(1);
        name = 1;
    }
}
```

For now the constructor only sets the caption of the base *MenuItem*.

4.1.5 Part IV adding submenus

We want our only menu item, *File*, to have some submenus for the tasks available in the editor.

Add a property to *NMenuItem*, which allows easy adding of child menus:

```
// class NMenuItem : MenuItem
// {

// this property allows us to set submenus of this menu item
public SubmenuList : list [NMenuItem]
{
    set
    {
        def sub = Menu();
        foreach (submenu in value) sub.Append (submenu);
        this.Submenu = sub;
    }
}

// }
```

SubmenuList simply iterates through the elements of the supplied list and adds each of them to the current instance's submenu.

Use this property in the constructor of *MainWindow*:

```
// def menu = MenuBar ();
// def mi = NMenuItem ("File");
mi.SubmenuList =
[
    NMenuItem ("Open"),
    NMenuItem ("Save as...")
];
// menu.Append(mi);
```

Note the commented out lines. We will uncomment these later.

4.1.6 Part V adding tasks to the submenus

Finally, we will attach actions to the items in our editor's menu. We will implement opening and saving files.

Add a second constructor to *NMenuItem*, which will accept an action function to be performed after choosing the given menu item:

```
public this(l : string, e : object * EventArgs -> void)
{
    base(l);
    name = l;
    this.Activated += e;
}
```

Like the previous one, this constructor sets the current menu item's caption, but it also attaches the given function to the *Activated* event of the item.

Now we will define a function which will handle opening and saving files by our editor. Add it to *MainWindow*:

```
// handler of opening and saving files
OnMenuFile (i : object, _ : EventArgs) : void
{
```

```

def mi = i :> NMenuItem;
def fs = FileSelection (mi.name);

when (fs.Run () == ResponseType.Ok :> int) match (mi.name)
{
  | "Open" =>
    def stream = IO.StreamReader (fs.FileName);
    input.Buffer.Text = stream.ReadToEnd();

  | "Save as..." =>
    def s = IO.StreamWriter(fs.FileName);
    s.Write(input.Buffer.Text);
    s.Close();

  | _ => ();
};
fs.Hide();
}

```

This method creates a window for selecting files (*Gtk FileSelection*), checks which menu item it was called from, and takes the appropriate action. Here we match on the name of *NMenuItem*, but you could also use something like **enums** to distinguish various menu items (or attach a specific handler to each of them).

Actions performed for opening and saving files are quite simple - we use *Buffer.Text* properties of *TextView* to get and set the contents of the editor.

The last thing to do is actually create the menu items using handlers. Uncomment the lines we entered in the earlier step to read:

```

def menu = MenuBar ();
def mi = NMenuItem ("File");
mi.SubmenuList =
[
  NMenuItem ("Open", OnMenuFile),
  NMenuItem ("Save as...", OnMenuFile)
];
menu.Append(mi);

```

4.1.7 Result

The result of these steps can be accessed [here](#) and it looks like this:

```

using System;
using Gtk;

class NMenuItem : MenuItem
{
  public name : string;

  public this(l : string)
  {
    base(l);
    name = l;
  }
}

```

```
public this(l : string, e : object * EventArgs -> void)
{
    base(l);
    name = l;
    this.Activated += e;
}

// this property allows us to set submenus of this menu item
public SubmenuList : list [NMenuItem]
{
    set
    {
        def sub = Menu();
        foreach (submenu in value) sub.Append (submenu);
        this.Submenu = sub;
    }
}
}

class MainWindow : Window
{
    /// text input area of our window
    input : TextView;

    public this()
    {
        // set caption of window
        base ("Very Simple Editor");
        // resize windows to some reasonable shape
        SetSizeRequest (300,200);

        def scroll = ScrolledWindow ();
        input = TextView ();
        scroll.Add (input);

        def menu = MenuBar ();
        def mi = NMenuItem ("File");
        mi.SubmenuList =
        [
            NMenuItem ("Open", OnMenuFile),
            NMenuItem ("Save as...", OnMenuFile)
        ];
        menu.Append(mi);

        def vbox = VBox ();
        vbox.PackStart (menu, false, false, 0u);
        vbox.PackStart (scroll, true, true, 0u);

        // place vertical box inside our main window
        Add (vbox);
    }

    // handler of opening and saving files
    OnMenuFile (i : object, _ : EventArgs) : void
}
```

```

    {
        def mi = i :> NMenuItem;
        def fs = FileSelection (mi.name);

        when (fs.Run () == ResponseType.Ok :> int) match (mi.name)
        {
            | "Open" =>
                def stream = IO.StreamReader (fs.FileName);
                input.Buffer.Text = stream.ReadToEnd();

            | "Save as..." =>
                def s = IO.StreamWriter(fs.FileName);
                s.Write(input.Buffer.Text);
                s.Close();

            | _ => ();
        };
        fs.Hide();
    }
}

module SimpleEditor
{
    Main() : void
    {
        Application.Init();
        def win = MainWindow();

        // exit application when editor window is deleted
        win.DeleteEvent += fun (_) { Application.Quit () };

        win.ShowAll ();
        Application.Run();
    }
}

```

5 System.Windows.Forms tutorial

Graphical User Interface (GUI) with Windows.Forms

Windows.Forms is the standard GUI toolkit for the .NET Framework. It is also supported by Mono. Like every other thing in the world it has its strong and weak points. Another GUI supported by .NET is [Gtk#](#). Which outguns which is up to you to judge. But before you can do that, get a feel for Windows.Forms by going through this tutorial.

Note 1: To compile programs using Windows.Forms you will need to include `-r:System.Windows.Forms` in the compile command. E.g. if you save the examples in the file `MyFirstForm.n`, compile them with:

```
ncc MyFirstForm.n -r:System.Windows.Forms -o MyFirstForm.exe
```

Note 2: I am not going to go too much into details of every single thing you can do with Windows.Forms. These can be pretty easily checked in the [Class reference](#). I will just try to show you how to put all the bricks together.

5.1 The first step

```
using System.Drawing;
using System.Windows.Forms;

class MyFirstForm : Form {
    public static Main() : void {
        Application.Run(MyFirstForm());
    }
}
```

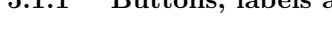
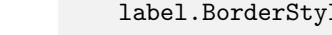

This is not too complex, is it? And it draws a pretty window on the screen, so it is not utterly useless, either :)

Let us now try to customize the window a little. All the customization should be placed in the class constructor.

```
using System.Drawing;
using System.Windows.Forms;

class MyFirstForm : Form {
    public this() {
        Text=&quot;My First Form&quot;; //title bar
        ClientSize=Size(300,300); //size (without the title bar) in pixels
        StartPosition=FormStartPosition.CenterScreen; //I'm not telling
        FormBorderStyle=FormBorderStyle.FixedSingle; //not resizable
    }

    public static Main() : void {
        Application.Run(MyFirstForm());
    }
}
```

[http://wiki.nemerle.org/Image:Form1.png](http://wiki.nemerle.org/Image:Form1.png "An empty Windows.Forms form.")  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form. <http://wiki/images/5/56/Form1.png>  An empty Windows.Forms form.

5.1.1 Buttons, labels and the rest

Wonderful. Now the time has come to actually display something in our window. Let us add a slightly customized button. All you need to do is to add the following code to the constructor of your Form class (i.e. somewhere between `public this() {` and the first `}` after it).

```
def button=Button();
    button.Text=&quot;I am a button&quot;;
    button.Location=Point(50,50);
    button.BackColor=Color.Khaki;
    Controls.Add(button);

def label=Label();
    label.Text=&quot;I am a label&quot;;
    label.Location=Point(200,200);
    label.BorderStyle=BorderStyle.Fixed3D;
    label.Cursor=Cursors.Hand;
    Controls.Add(label);
```

I hope it should pretty doable to guess what is which line responsible for. Perhaps you could only pay a little more attention to the last one. Buttons, labels, textboxes, panels and the like are all controls, and simply

defining them is not enough, unless you want them to be invisible. Also, the button we have added does actually nothing. But fear not, we shall discuss a bit of events very soon.

5.1.2 A simple menu

Menus are not hard. Just take a look at the following example:

```
def mainMenu=MainMenu();

def mFile=mainMenu.MenuItems.Add("&quot;File&quot;");
def mFileDont=mFile.MenuItems.Add("&quot;Don't quit&quot;");
def mFileQuit=mFile.MenuItems.Add("&quot;Quit&quot;");

def mHelp=mainMenu.MenuItems.Add("&quot;Help&quot;");
def mHelpHelp=mHelp.MenuItems.Add("&quot;Help&quot;");

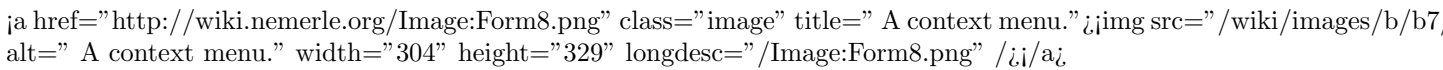
Menu=mainMenu;
```

It will create a menu with two drop-down submenus: *File* (with options *Don't Quit* and *Quit*) and *Help* with just one option, *Help*.

 A button, a label and a menu.

It is also pretty easy to add context menus to controls. Context menus are the ones which become visible when a control is right-clicked. First, you need to define the whole menu, and then add a reference to it to the button definition:

```
def buttonCM=ContextMenu();
def bCMWhatAmI=buttonCM.MenuItems.Add("&quot;What am I?&quot;");
def bCMWhyAmIHere=buttonCM.MenuItems.Add("&quot;Why am I here?&quot;");
...
def button=Button();
...
button.ContextMenu=buttonCM;
```

 A context menu.

5.1.3 Basic events

Naturally, however nice it all might be, without being able to actually serve some purpose, it is a bit pointless. Therefore, we will need to learn handling events. There are in fact two ways to do it. Let us take a look at the easier one for the beginning.

The first thing you will need to do is to add the *System* namespace:

```
using System;
```

You can skip this step but you will have to write *System.EventArgs* and so on instead of *EventArgs*.

Then, you will need to add a reference to the event handler to your controls. It is better to do it first as it is rather easy to forget about it after having written a long and complex handler.

```

button.Click+=button_Click;
...
mFileDont.Click+=mFileDont_Click;
mFileQuit.Click+=mFileQuit_Click;
...
mHelpHelp.Click+=mHelpHelp_Click;

```

Mind the `+=` operator instead of `=` used when customizing the controls.

Finally, you will need to write the handlers. Do not forget to define the arguments they need, i.e. an *object* and an *EventArgs*. Possibly, you will actually not use them but you have to define them anyway. You can prefix their names with a `_` to avoid warnings at compile time.

```

private button_Click(_sender:object, _ea:EventArgs):void{
    Console.WriteLine("&quot;I was clicked. - Your Button&quot;");
}

private mFileQuit_Click(_sender:object, _ea:EventArgs):void{
    Application.Exit();
}

```

This is the way you will generally want to do it. But in this very case the handlers are very short, and like all handlers, are not very often used for anything else than handling events. If so, we could rewrite them as lambda expressions which will save a good bit of space and clarity.

```

button.Click+=fun(_sender:object, _ea:EventArgs){
    Console.WriteLine("&quot;I wrote it. - button_Click as a lambda&quot;");
};

mFileQuit.Click+=fun(_sender:object, _ea:EventArgs){
    Application.Exit();
};

```

5.1.4 Cleaning up

After you are done with the whole application, you should clean everything up. In theory, the system would do it automatically anyway but it certainly is not a bad idea to help the system a bit. Especially that it is not a hard thing to do at all, and only consists of two steps.

The first step is to add a reference to *System.ComponentModel* and define a global variable of type *Container*:

```

using System.ComponentModel;
...

class MyFirstForm : Form {
    components : Container = null;
...

```

In the second and last step, you will need to override the *Dispose* method. It could even fit in one line if you really wanted it to.

```

protected override Dispose(disposing : bool) : void {
    when (disposing) when (components!=null) components.Dispose();
    base.Dispose(d);
}

```

```
}
```

5.2 The second step

You should now have a basic idea as to what a form looks like. The time has come to do some graphics.

5.2.1 Painting

There is a special event used for painting. It is called `Paint` and you will need to create a handler for it.

```
using System;
using System.Drawing;
using System.Windows.Forms;

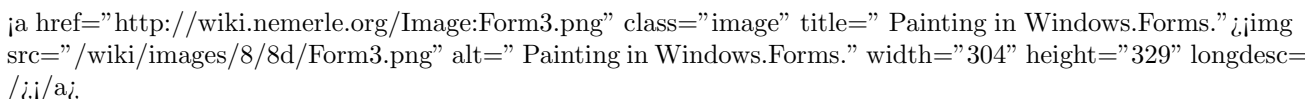
class MySecondForm : Form {
    public this() {
        Text="My Second Form";
        ClientSize=Size(300,300);

        Paint+=PaintEventHandler(painter);
    }

    private painter(_sender:object, pea:PaintEventArgs):void{
        def graphics=pea.Graphics;
        def penBlack=Pen(Color.Black);
        graphics.DrawLine(penBlack, 0, 0 , 150, 150);
        graphics.DrawLine(penBlack, 150, 150, 300, 0);
    }

    public static Main():void{
        Application.Run(MySecondForm());
    }
}
```

A reference to the painter method should be added in the form class constructor (it affects the whole window), but this time it is a **Paint**EventHandler. Same as **Paint**EventArgs in the handler itself. In the handler, you should first define a variable of type `PaintEventArgs.Graphics`. This is the whole window. This is where you will be drawing lines, circles, strings, images and the like (but not pixels; we will discuss bitmaps later on). To draw all those things you will need pens and brushes. In our example we used the same pen twice. But sometimes you will only need it once. In such case, there is no point in defining a separate variable for it. Take a look at the next example.

[http://wiki.nemerle.org/Image:Form3.png](http://wiki.nemerle.org/Image:Form3.png " Painting in Windows.Forms.")  Painting in Windows.Forms.

5.2.2 Overriding event handlers

Another way to define a handler is to override the default one provided by the framework. It is not at all complicated, it is not dangerous and in many cases it will prove much more useful. Take a look at the example overriding the `OnPaint` event handler:

```
protected override OnPaint(pea:PaintEventArgs):void{
    def g=pea.Graphics;
    g.DrawLine(Pen(Color.Black), 50, 50, 150, 150);
}
```

Naturally, when you override the handler, there is no need any longer to add a reference to it in the class constructor.

By the same token can you override other events. One of the possible uses is to define quite complex clickable areas pretty easily:

```
protected override OnPaint(pea:PaintEventArgs):void{
    def g=pea.Graphics;
    for (mutable x=0; x<300; x+=10) g.DrawLine(Pen(Color.Blue), x, 0, x, 300);
}

protected override OnMouseDown(meas:MouseEventArgs):void{
    when (meas.X%10<4 && meas.Button==MouseButtons.Left)
        if (meas.X%10==0) Console.WriteLine("&quot;You left-clicked a line.&quot;");
        else Console.WriteLine("&quot;You almost left-clicked a line ($(meas.X%10) pixels).&quot;");
}
```

[Easily creating a complex clickable image map.](http://wiki.nemerle.org/Image:Form7.png " Easily creating a complex clickable image map.")

5.2.3 A simple animation with double-buffering

What is an animation without double-buffering? Usually, a flickering animation. We do not want that. Luckily enough, double-buffering is as easy as abc in Windows.Forms. In fact, all you need to do is to set three style flags and override the OnPaint event handler instead of writing your own handler and assigning it to the event.

```
class Animation:Form{
    mutable mouseLocation:Point;           //the location of the cursor

    public this(){
        Text="&quot;A simple animation&quot;;
        ClientSize=Size(300,300);
        SetStyle(ControlStyles.UserPaint | ControlStyles.AllPaintingInWmPaint |
            ControlStyles.DoubleBuffer, true);           //double-buffering on
    }

    protected override OnPaint(pea:PaintEventArgs):void{
        def g=pea.Graphics;
        g.FillRectangle(SolidBrush(Color.DimGray),0,0,300,300); //clearing the window

        def penRed=Pen(Color.Red);
        g.FillEllipse(SolidBrush(Color.LightGray),
            mouseLocation.X-15, mouseLocation.Y-15, 29, 29);
        g.DrawEllipse(penRed, mouseLocation.X-15, mouseLocation.Y-15, 30, 30);
        g.DrawLine(penRed, mouseLocation.X, mouseLocation.Y-20,
            mouseLocation.X, mouseLocation.Y+20);
        g.DrawLine(penRed, mouseLocation.X-20, mouseLocation.Y,
            mouseLocation.X+20, mouseLocation.Y);
    }
}
```

```

protected override OnMouseMove(mea:MouseEventArgs):void{
    mouseLocation=Point(mea.X,mea.Y);
    Invalidate();    //redraw the screen every time the mouse moves
}

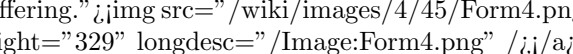
public static Main():void{
    Application.Run(Animation());
}
}

```

As you can see, the basic structure of a window with animation is pretty easy. *Main* starts the whole thing as customized in the constructor. The *OnPaint* event handler is called immediately. And then the whole thing freezes waiting for you to move the mouse. When you do, the *OnMouseMove* event handler is called. It checks the new position of the mouse, stores it in *mouseLocation* and tells the window to redraw (*Invalidate()*).

Try turning double-buffering off (i.e. comment the *SetStyle...* line) and moving the mouse slowly. You will see the viewfinder flicker. Now turn double-buffering back on and try again. See the difference?

Well, in this example you have to move the mouse slowly to actually see any difference because the viewfinder we are drawing is a pretty easy thing to draw. But you can be sure that whenever anything more complex comes into play, the difference is very well visible without any extra effort.

ja href="http://wiki.nemerle.org/Image:Form4.png" class="image" title=" A simple animation with double-buffering."> A simple animation with double-buffering.">

5.2.4 Bitmaps and images

I promised I would tell you how to draw a single pixel. Well, here we are.

```

protected override OnPaint(pea:PaintEventArgs):void{
    def g=pea.Graphics;
    def bmp=Bitmap(256,1);    //one pixel height is enough; we can draw it many times
    for (mutable i=0; i<256; ++i) bmp.SetPixel(i, 0, Color.FromArgb(i,0,i,0));
    for (mutable y=10; y<20; ++y) g.DrawImage(bmp, 0, y);
}

```

So, what do we do here? We define a bitmap of size 256,1. We draw onto it 256 pixels in colours defined by four numbers: alpha (transparency), red, green, and blue. But as we are only manipulating alpha and green, we will get a nicely shaded green line. Finally, we draw the bitmap onto the window ten times, one below another.

Mind the last step, it is very important. The bitmap we had defined is an offset one. Drawing onto it does not affect the screen in any way.

Now, that you know the *DrawImage* method nothing can stop you from filling your window with all the graphic files you might be having on your disk, and modifying them according to your needs.

```

class ImageWithAFilter:Form{
    img:Image=Image.FromFile(""/home/johnny/pics/doggy.png"");    //load an image
    bmp:Bitmap=Bitmap(img);    //create a bitmap from the image

    public this(){
        ClientSize=Size(bmp.Width*2, bmp.Height);
    }

    protected override OnPaint(pea:PaintEventArgs):void{
        def g=pea.Graphics;

```

```

for (mutable x=0; x<bmp.Width; ++x) for (mutable y=0; y<bmp.Height; ++y)
  when (bmp.GetPixel(x,y).R>0){
    def g=bmp.GetPixel(x,y).G;
    def b=bmp.GetPixel(x,y).B;
    bmp.SetPixel(x,y,Color.FromArgb(255,0,g,b));
  }
g.DrawImage(img,0,0);           //draw the original image on the left
g.DrawImage(bmp,img.Width,0);  //draw the modified image on the right
}
...
}

```

This example draws two pictures: the original one on the left and the modified one on the right. The modification is a very simple filter entirely removing the red colour.

[!\[\]\(72b4cf351241b08691672e806c1604b7_img.jpg\)](http://wiki.nemerle.org/Image:Form5.png "A simple graphic filter.")

5.2.5 Adding icons to the menu

Adding icons is unfortunately a little bit more complicated than other things in Windows.Forms.

You will need to provide handlers for two events: *MeasureItem* and *DrawItem*, reference them in the right place, and set the menu item properly.

```

mFileQuit.Click+=EventHandler( fun(_) {Application.Exit();} );
mFileQuit.OwnerDraw=true;
mFileQuit.MeasureItem+=MeasureItemEventHandler(measureItem);
mFileQuit.DrawItem+=DrawItemEventHandler(drawItem);

```

```

private measureItem(sender:object, miea:MeasureItemEventArgs):void{
  def menuItem=sender:>MenuItem;
  def font=Font("&quot;FreeMono&quot;;8); //the name and the size of the font
  miea.ItemHeight=(miea.Graphics.MeasureString(menuItem.Text,font).Height+5):>int;
  miea.ItemWidth=(miea.Graphics.MeasureString(menuItem.Text,font).Width+30):>int;
}

private drawItem(sender:object, diea:DrawItemEventArgs):void{
  def menuItem=sender:>MenuItem;
  def g=diea.Graphics;
  diea.DrawBackground();
  g.DrawImage(anImageDefinedEarlier, diea.Bounds.Left+3, diea.Bounds.Top+3, 20, 15);
  g.DrawString(menuItem.Text, diea.Font, SolidBrush(diea.ForeColor),
    diea.Bounds.Left+25, diea.Bounds.Top+3);
}

```

[!\[\]\(3a91434fb6b4bec5a2c52d3fbe2b9c14_img.jpg\)](http://wiki.nemerle.org/Image:Form6.png "A menu with an icon.")

6 Macros tutorial

Nemerle type-safe macros

6.1 What exactly is a macro?

Basically every macro is a function, which takes a fragment of code as parameter(s) and returns some other code. On the highest level of abstraction it doesn't matter if parameters are function calls, type definitions or just a sequence of assignments. The most important fact is that they are not common objects (e.g. instances of some types, like integer numbers), but their internal representation in the compiler (i.e. syntax trees).

A macro is defined in the program just like any other function, using common Nemerle syntax. The only difference is the structure of the data it operates on and the way in which it is used (executed at compile-time).

A macro, once created, can be used to process some parts of the code. It's done by calling it with block(s) of code as parameter(s). This operation is in most cases indistinguishable from a common function call (like $f(1)$), so a programmer using a macro would not be confused by unknown syntax. The main concept of our design is to make the usage of macros as transparent as possible. From the user point of view, it is not important if particular parameters are passed to a macro, (which would process them at the compile-time and insert some new code in their place), or to an ordinary function.

6.2 Defining a new macro

Writing a macro is as simple as writing a common function. It looks the same, except that it is preceded by a keyword `macro` and it lives at the top level (not inside any class). This will make the compiler know about how to use the defined method (i.e. run it at the compile-time in every place where it is used).

Macros can take zero (if we just want to generate new code) or more parameters. They are all elements of the language grammar, so their type is limited to the set of defined syntax objects. The same holds for a return value of a macro.

Example:

```
macro generate_expression ()
{
    MyModule.compute_some_expression ();
}
```

This example macro does not take any parameters and is used in the code by simply writing `generate_expression ();`. The most important is the difference between `generate_expression` and `compute_some_expression` - the first one is a function executed by the compiler during compilation, while the latter is just some common function that must return syntax tree of expressions (which is here returned and inserted into program code by `generate_expression`).

6.2.1 Compiling a simplest macro

In order to create and use a macro you have to write a library, which will contain its executable form. You simply create a new file `mymacro.n`, which can contain for example

```
macro m () {
    Nemerle.IO.printf ("compile-time\n");
    <[ Nemerle.IO.printf ("run-time\n") ]>;
}
```

and compile it with command

```
ncc -r Nemerle.Compiler.dll -t:dll mymacro.n -o mymacro.dll
```

Now you can use `m()` in any program, like here

```

module M {
    public Main () : void {
        m ();
    }
}

```

You must add a reference to `mymacro.dll` during compilation of this program. It might look like

```
ncc -r mymacro.dll myprog.n -o myprog.exe
```

6.2.2 Exercise

Write a macro, which, when used, should slow down the compilation by 5 seconds (use `System.Timers` namespace) and print the version of the operating system used to compile program (use `System.Environment` namespace).

6.3 Operating on syntax trees

Definition of function `compute_some_expression` might look like:

```

using Nemerle.Compiler.Parsetree;

module MyModule
{
    public mutable debug_on : bool;

    public compute_some_expression () : PExpr
    {
        if (debug_on)
            <[ System.Console.WriteLine ("Hello, I'm debug message") ]>
        else
            <[ () ]>
    }
}

```

The examples above show a macro, which conditionally inlines expression printing a message. It's not quite useful yet, but it has introduced the meaning of compile-time computations and also some new syntax used only in writing macros and functions operating on syntax trees. We have written here the `<[...]>` constructor to build a syntax tree of expression (e.g. `'()'`).

6.3.1 Quotation operator

`<[...]>` is used to both construction and decomposition of syntax trees. Those operations are similar to quotation of code. Simply, everything which is written inside `<[...]>`, corresponds to its own syntax tree. It can be any valid Nemerle code, so a programmer does not have to learn internal representation of syntax trees in the compiler.

```

macro print_date (at_compile_time)
{
    match (at_compile_time) {
        | <[ true ]> => MyModule.print_compilation_time ()
    }
}

```

```

    | _ => <[ WriteLine (DateTime.Now.ToString ()) ]>
  }
}

```

The quotation alone allows using only constant expressions, which is insufficient for most tasks. For example, to write function `print_compilation_time` we must be able to create an expression based on a value known at the compile-time. In next sections we introduce the rest of macros' syntax to operate on general syntax trees.

6.3.2 Matching subexpressions

When we want to decompose some large code (or more precisely, its syntax tree), we must bind its smaller parts to variables. Then we can process them recursively or just use them in an arbitrary way to construct the result.

We can operate on entire subexpressions by writing `$(...)` or `$ID` inside the quotation operator `<[...]>`. This means binding the value of `ID` or the interior of parenthesized expression to the part of syntax tree described by corresponding quotation.

```

macro for (init, cond, change, body)
{
  <[
    $init;
    def loop () : void {
      if ($cond) { $body; $change; loop() }
      else ()
    };
    loop ()
  ]>
}

```

The above macro defines function `for`, which is similar to the loop known from C. It can be used like this

```

for (mutable i = 0, i < 10, i++, printf ("%d", i))

```

Later we show how to extend the language syntax to make the syntax of `for` exactly as in C.

6.3.3 Base elements of grammar

Sometimes quoted expressions have literals inside of them (like strings, integers, etc.) and we want to operate on their value, not on their syntax trees. It is possible, because they are constant expressions and their runtime value is known at the compile-time.

Let's consider the previously used function `print_compilation_time`.

```

using System;
using Nemerle.Compiler.Parssetree;

module MyModule {
  public print_compilation_time () : PExpr
  {
    <[ System.Console.WriteLine ($(DateTime.Now.ToString () : string)) ]>
  }
}

```

Here we see some new extension of splicing syntax where we create a syntax tree of string literal from a known value. It is done by adding `: string` inside the `$(...)` construct. One can think about it as of enforcing the

type of spliced expression to a literal (similar to common Nemerle type enforcement), but in the matter of fact something more is happening here - a real value is lifted to its representation as syntax tree of a literal.

Other types of literals (`int`, `bool`, `float`, `char`) are treated the same. This notation can be used also in pattern matching. We can match constant values in expressions this way.

There is also a similar schema for splicing and matching variables of a given name. `$(v : name)` denotes a variable, whose name is contained by object `v` (of special type `Name`). There are some good [reasons](#) for encapsulating a real identifier within this object.

6.3.4 Constructs with variable amount of elements

You might have noticed, that Nemerle has a few grammar elements, which are composed of a list of subexpressions. For example, a sequence of expressions enclosed with `{ .. }` braces may contain zero or more elements.

When splicing values of some expressions, we would like to decompose or compose such constructs in a general way - i.e. obtain all expressions in a given sequence. It is natural to think about them as if a list of expressions and to bind this list to some variable in meta-language. It is done with special syntax `..`:

```
mutable exps = [ <[ printf ("%d ", x) ]>, <[ printf ("%d ", y) ]> ];
exps = <[ def x = 1 ]> :: <[ def y = 2 ]> :: exps;
<[ {.. $exps } ]>
```

We have used `{ .. $exps }` here to create the sequence of expressions from list `exps : list<Expr>`. A similar syntax is used to splice the content of tuples (`(.. $elist)`) and other constructs, like `array []`:

```
using Nemerle.Collections;

macro castedarray (e) {
  match (e) {
    | <[ array [.. $elements ] ]> =>
      def casted = List.Map (elements, fun (x) { <[ ($x : object) ]> });
      <[ array [.. $casted] ]>
    | _ => e
  }
}
```

If the exact number of expressions in tuple/sequence is known during writing the quotation, then it can be expressed with

```
<[ $e.1; $e.2; $e.3; x = 2; f () ]>
```

The `..` syntax is used when there are `e.i : Expr` for `1 <= i <= n`.

6.3.5 Exercise

Write a macro `rotate`, which takes two parameters: a pair of floating point numbers (describing a point in 2D space) and an angle (in radians). The macro should return a new pair - a point rotated by the given angle. The macro should use as much information as is available at the compile-time, e.g. if all numbers supplied are constant, then only the final result should be inlined, otherwise the result must be computed at runtime.

6.4 Adding new syntax to the compiler

After we have written the `for` macro, we would like the compiler to understand some changes to its syntax. Especially the C-like notation

```
for (mutable i = 0; i < n; --i) {
    sum += i;
    Nemerle.IO.printf ("%d\n", sum);
}
```

In order to achieve that, we have to define which tokens and grammar elements may form a call of `for` macro. We do that by changing its header to

```
macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")", body)
```

The `syntax` keyword is used here to define a list of elements forming the syntax of the macro call. The first token must always be a unique identifier (from now on it is treated as a special keyword triggering parsing of defined sequence). It is followed by tokens composed of operators or identifiers passed as string literals or names of parameters of macro. Each parameter must occur exactly once.

Parsing of syntax rule is straightforward - tokens from input program must match those from definition, parameters are parsed according to their type. Default type of a parameter is `Expr`, which is just an ordinary expression (consult Nemerle grammar in [Reference](#)). All allowed parameter types will be described in the extended version of reference manual corresponding to macros.

6.4.1 Exercise

Add a new syntactic construct `forpermutation` to your program. It should be defined as the macro

```
macro forp (i, n : int, m : int, body)
```

and introduce `syntax`, which allows writing the following program

```
mutable i = 0;
forpermutation (i in 3 to 10) Nemerle.IO.printf ("%d\n", i)
```

It should create a random permutation `p` of numbers `xj`, $m \leq x_j \leq n$ at the compile-time. Then generate the code executing body of the loop $n - m + 1$ times, preceding each of them with assignment of permutation element to `i`.

6.5 Macros in custom attributes

6.5.1 Executing macros on type declarations

Nemerle macros are simply plugins to the compiler. We decided not to restrict them only to operations on expressions, but allow them to transform almost any part of program.

Macros can be used within custom attributes written near methods, type declarations, method parameters, fields, etc. They are executed with those entities passed as their parameters.

As an example, let us take a look at `Serializable` macro. Its usage looks like this:

```
[Serializable]
class S {
    public this (v : int, m : S) { a = v; my = m; }
    my : S;
    a : int;
}
```

From now on, `S` has additional method `Serialize` and it implements interface `ISerializable`. We can use it in our code like this

```
def s = S (4, S (5, null));
s.Serialize ();
```

And the output is

```
<a>4</a>
<my>
  <a>5</a>
  <my>
    <null/>
  </my>
</my>
```

The macro modifies type `S` at compile-time and adds some code to it. Also inheritance relation of given class is changed, by making it implement interface `ISerializable`

```
public interface ISerializable {
  Serialize () : void;
}
```

6.5.2 Manipulating type declarations

In general, macros placed in attributes can do many transformations and analysis of program objects passed to them. To see `Serializable` macro's internals and discuss some design issues, let's go into its code.

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance, Nemerle.MacroTargets.Class,
  Inherited = true)]
macro Serializable (t : TypeBuilder)
{
  t.AddImplementedInterface (<[ ISerializable ]>)
}
```

First we have to add interface, which given type is about to implement. But more important thing is the phase modifier `BeforeInheritance` in macro's custom attribute. In general, we separate three [stages of execution](#) for attribute macros. `BeforeInheritance` specifies that the macro will be able to change subtyping information of the class it operates on.

So, we have added interface to our type, we now have to create `Serialize ()` method.

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers, Nemerle.MacroTargets.Class,
  Inherited = true)]
macro Serializable (t : TypeBuilder)
{
  /// here we list its fields and choose only those, which are not derived
  /// or static
  def fields = t.GetFields (BindingFlags.Instance | BindingFlags.Public %|
    BindingFlags.NonPublic | BindingFlags.DeclaredOnly);

  /// now create list of expressions which will print object's data
  mutable serializers = [];
```

```

/// traverse through fields, taking their type constructors
foreach (x : IField in fields) {
  def tc = x.GetMemType ().TypeInfo;
  def nm = Macros.UseSiteSymbol (x.Name);
  if (tc != null)
    if (tc.IsValueType)
      /// we can safely print value types as strings
      serializers = <[
        printf ("<%s>", $(x.Name : string));
        System.Console.Write ($(nm : name));
        printf ("</%s>\n", $(x.Name : string));
      ]>
      :: serializers
    else
      /// we can try to check, if type of given field also implements ISerializable
      if (x.GetMemType ().Require (<[ ttype: ISerializable ]>))
        serializers = <[
          printf ("<%s>\n", $(x.Name : string));
          if ($(nm : name) != null)
            $(nm : name).Serialize ()
          else
            printf ("<null/>\n");
          printf ("</%s>\n", $(x.Name : string));
        ]>
        :: serializers
      else
        /// and finally, we encounter case when there is no easy way to serialize
        /// given field
        Message.FatalError ("field '" + x.Name + "' cannot be serialized")
    else
      Message.FatalError ("field '" + x.Name + "' cannot be serialized")
  };
  /// after analyzing fields, we create method in our type, to execute created
  /// expressions
  t.Define (<[ decl: public Serialize () : void
    implements ISerializable.Serialize {
      .. $serializers
    }
  ]>);
}

```

6.5.3 Execution stages

Analysing object-oriented hierarchy and class members is a separate pass of the compilation. First it creates inheritance relation between classes, so we know exactly all base types of given type. After that every member inside of them (methods, fields, etc.) is being analysed and added to the hierarchy and its type annotations are resolved. After that also the rules regarding implemented interface methods are checked.

For the needs of macros we have decided to distinguish three moments in this pass at which they can operate on elements of class hierarchy. Every macro can be annotated with a stage, at which it should be executed.

- **BeforeInheritance** stage is performed after parsing whole program and scanning declared types, but before building subtyping relation between them. It gives macro a freedom to change inheritance hierarchy and operate on parse-tree of classes and members
- **BeforeTypedMembers** is when inheritance of types is already set. Macros can still operate on bare parse-trees, but utilize information about subtyping.

- **WithTypedMembers** stage is after headers of methods, fields are already analysed and in bound state. Macros can easily traverse entire class space by reflecting type constructors of fields, method parameters, etc. Original parse-trees are no longer available and signatures of class members cannot be changed.

Parameters of attribute macros Every executed attribute macro operates on some element of class hierarchy, so it must be supplied with an additional parameter describing the object, on which macro was placed. This way it can easily query for properties of that element and use compiler's API to reflect or change the context in which it was defined.

For example a method macro declaration would be

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers,
                    Nemerle.MacroTargets.Method)]
macro MethodMacro (t : TypeBuilder, f : MethodBuilder, expr)
{
  // use 't' and 'f' to query or change class-level elements
  // of program
}
```

Macro is annotated with additional attributes specifying respectively the stage in which macro will be executed and the macro target.

The available parameters contain references to class hierarchy elements that given macro operates on. They are automatically supplied by compiler and they vary on the target and stage of given macro. Here is a little table specifying valid parameters for each stage and target of attribute macro.

- Attribute macro targets and parameters
- | MacroTarget | MacroPhase.BeforeInheritance | MacroPhase.BeforeTypedMembers | MacroPhase.WithTypedMembers
- | Class | TypeBuilder | TypeBuilder | TypeBuilder
- | Method | TypeBuilder, ParsedMethod | TypeBuilder, ParsedMethod | TypeBuilder, MethodBuilder
- | Field | TypeBuilder, ParsedField | TypeBuilder, ParsedField | TypeBuilder, FieldBuilder
- | Property | TypeBuilder, ParsedProperty | TypeBuilder, ParsedProperty | TypeBuilder, PropertyBuilder
- | Event | TypeBuilder, ParsedEvent | TypeBuilder, ParsedEvent | TypeBuilder, EventBuilder
- | Parameter | TypeBuilder, ParsedMethod, ParsedParameter | TypeBuilder, ParsedMethod, ParsedParameter | TypeBuilder, MethodBuilder, ParameterBuilder
- | Assembly | (none) | (none) | (none)

The intuition is that every macro has parameter holding its target and additionally objects containing it (like TypeBuilder is available in most of the attribute macros).

After those implicitly available parameters there come standard parameters explicitly supplied by user. They are the same as for expression level macros.

6.6 Reference to more advanced aspects

6.6.1 Hygiene and alpha-renaming of identifiers

Problem with names capture Identifiers in quoted code (object code) must be treated in a special way, because we usually do not know in which scope they would appear. Especially they should not mix with variables with the same names from the macro-use site.

Consider the following macro defining a local function `f`

```
macro identity (e) { <[ def f (x) { x }; f($e) ]> }
```

Calling it with `identity (f(1))` might generate confusing code like

```
def f (x) { x }; f (f (1))
```

To preserve names capture, all macro generated variables should be renamed to their unique counterparts, like in

```
def f_42 (x_43) { x_43 }; f_42 (f (1))
```

Hygiene of macros The idea of separating variables introduced by a macro from those defined in the plain code (or other macros) is called ‘hygiene’ after Lisp and Scheme languages. In Nemerle we define it as putting identifiers created during a single macro execution into a unique namespace. Variables from different namespaces cannot bind to each other.

In other words, a macro cannot create identifiers capturing any external variables or visible outside of its own generated code. This means, that there is no need to care about locally used names.

The Hygiene is obtained by encapsulating identifiers in special `Name` class. The compiler uses it to distinguish names from different macro executions and scopes (for details of implementation consult [paper about macros](#)). Variables with appropriate information are created automatically by quotation.

```
def definition = <[ def y = 4 ]>;
<[ def x = 5; $definition; x + y ]>
```

When a macro creates the above code, identifiers `y` and `x` are tagged with the same unique mark. Now they cannot be captured by any external variables (with a different mark). We operate on the `Name` class, when the quoted code is composed or decomposed and we use `<[$(x : name)]>` construct. Here `x` is bound to an object of type `Name`, which we can use in other place to create exactly the same identifier.

An identifier can be also created by calling method `Macros.NewSymbol()`, which returns `Name` with a unique identifier, tagged with a current mark.

```
def x = Macros.NewSymbol ();
<[ def $(x : name) = 5; $(x : name) + 4 ]>
```

Controlled breaking hygiene Sometimes it is useful to generate identifiers, which bind to variables visible in place where a macro is used. For example one of macro’s parameters is a string with some identifiers inside. If we want to use these as real identifiers, then we need to break automatic hygiene. It is especially useful in embedding domain-specific languages, which reference symbols from the original program.

As an example consider a `Nemerle.IO.sprint (string literal)` macro (which have the syntax shortcut `$"some text $id "`). It searches given string literal for `$var` and creates a code concatenating text before and after `$var` to the value of `var.ToString ()`.

```
def x = 3;
System.Console.WriteLine ("My value of x is $x and I'm happy");
```

expands to

```
def x = 3;
System.Console.WriteLine ({
```

```
def sb = System.Text.StringBuilder ("My value of x is ");
sb.Append (x.ToString ());
sb.Append (" and I'm happy");
sb.ToString ()
});
```

Breaking of hygiene is necessary here, because we generate code (reference to x), which need to have the same context as variables from invocation place of macro.

To make given name bind to the symbols from macro usesite, we use `Nemerle.Macros.UseSiteSymbol (name : string) : Name` function, or special splicing target `usesite` in quotations. Their use would be like in this simplified implementation of macro

```
macro sprint (lit : string)
{
  def (prefix, symbol, suffix) = Helper.ExtractDollars (lit);
  def varname = Nemerle.Macros.UseSiteSymbol (symbol);
  <[
    def sb = System.Text.StringBuilder ($(prefix : string));
    sb.Append ($(varname : name).ToString ());
    // or alternatively $(symbol : usesite)
    sb.Append ($(suffix : string));
    sb.ToString ()
  ]>
}
```

Note that this operations is 'safe', that is it changes context of variable to the place where macro invocation was created (see paper for more details).

Unhygienic variables Sometimes it is useful to completely break hygiene, where programmer only want to experiment with new ideas. From our experience, it is often hard to reason about correct contexts for variables, especially when writing class level macros. In this case it is useful to be able to easily break hygiene.

Nemerle provides it with `<[$("id" : dyn)]>` construct. It makes produced variable break hygiene rules and always bind to the nearest definition with the same name.

7 ASP.NET (tutorial)

7.1 Preparations

In order to use Nemerle in your **ASP.NET** pages you first need to add following line

```
<compiler language="n;nemerle;Nemerle" extension=".n" warningLevel="1" compilerOptions=""
  type="Nemerle.Compiler.NemerleCodeProvider, Nemerle.Compiler, Version=0.9.1.0, Culture=neutral,
  />
```

in `<configuration><system.web><compilation>...` to your `machine.config` file (tested on mono, where this file can be found in `/etc/mono/2.0/`, you will need to use XSP2)

You may need to change the version. (Check `ncc -version`).

7.2 Example page

Now you can create `test.aspx` with following content:

```

<%@ language=Nemerle %>

<script runat="server">
  Page_Load(_ : object, _ : EventArgs) : void {
    Message.Text = $"You last accessed this page at: {(DateTime.Now)}";
  }

  EnterBtn_Click(_ : object, _ : EventArgs) : void
  {
    Message.Text = $"Hi {(Name.Text)}, welcome to ASP.NET!";
  }
</script>

<html>
  <form runat=server>

    <font face="Verdana">

      Please enter your name: <asp:textbox id="Name" runat=server/>
      <asp:button id="EnterBtn" text="Enter" Onclick="EnterBtn_Click" runat=server/>

    <p>
      <asp:label id="Message" runat=server/>
    </p>

    </font>

  </form>
</html>

```

And everything should work fine. This scenario was tested with xsp and mod_mono on mono 1.1.6, please let us know how it works on MS.NET + ISS.

Using a separate file for code is also possible and may be preferable in larger examples. This may be done by creating a file test.n for the code.

```

using System;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class Test : Page
{
  protected Page_Load(_ : object, _ : EventArgs) : void
  {
    Message.Text = $"You last accessed this page at: {(DateTime.Now)}";
  }
  protected EnterBtn_Click(_ : object, _ : EventArgs) : void
  {
    Message.Text = $"Hi {(Name.Text)}, welcome to ASP.NET!";
  }
}

```

and then removing the script section of test.aspx and replacing the top line with

```

<%@ language=Nemerle codefile="test.n" inherits="Test" %>

```

8 Remoting (tutorial)

This is a simple example that shows how to realize remoting in Nemerle. Essentially this is translation of the example presented in the [MonoHandbook](#).

Remoting allow clients to access objects in the server. And thus it is a good abstraction to distribute computing.

In this example both Server and Client share the definition of ServerObjects. The server program opens a port for conection and publish a ServerObject. Later the client program connects himself to the server port and access, remotely, the published objects and their methods.

The example shows the basic remoting usage, managing objects by reference and by value (through serialization).

8.1 Client

Save the following code in the file "Client.n".

```
using System;
using System.Net;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace RemotingTest
{
    class RemotingClient
    {
        static Main () :void
        {
            def ch = TcpChannel (0);
            ChannelServices.RegisterChannel (ch);

            // This gets the object from the server (a list of ServerObject)

            Console.WriteLine ("Getting instance ...");
            def remOb = Activator.GetObject (typeof (ServerList),
                "tcp://localhost:1122/test.rem");

            def list = remOb :>ServerList;

            // These are remote calls that return references to remote objects

            Console.WriteLine ("Creating two remote items...");
            def item1 = list.NewItem ("Created at server 1") :ServerObject;
            item1.SetValue (111); // another call made to the remote object
            def item2 = list.NewItem ("Created at server 2") :ServerObject;
            item2.SetValue (222);
            Console.WriteLine ();

            // Two objects created in this client app

            Console.WriteLine ("Creating two client items...");
            def item3 = ServerObject ("Created at client 1");
            item3.SetValue (333);
            def item4 = ServerObject ("Created at client 2");
            item4.SetValue (444);
            Console.WriteLine ();
        }
    }
}
```

```
// Object references passed to the remote list

Console.WriteLine ("Adding items...");
list.Add (item3);
list.Add (item4);
Console.WriteLine ();

// This sums all values of the ServerObjects in the list. The server
// makes a remote call to this client to get the value of the
// objects created locally
Console.WriteLine ("Processing items...");
list.ProcessItems ();
Console.WriteLine ();

// Passing some complex info as parameter and return value

Console.WriteLine ("Setting complex data...");
def cd = ComplexData (AnEnum.D, array ["some" :object, 22, "info"]);
def res = list.SetComplexData (cd) :ComplexData;
Console.WriteLine ("This is the result:");
res.Dump ();
Console.WriteLine ();

list.Clear ();
Console.WriteLine ("Done.");

ch.StopListening (null);

    }
}
}
```

8.2 Server

Save the following code in the file "Server.n".

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace RemotingTest
{
    public class RemotingServer
    {
        static Main () :int
        {
            Console.WriteLine ("Starting Server");
            def ch = TcpChannel (1122);
            ChannelServices.RegisterChannel (ch);

            def ser = ServerList ();
            _ = RemotingServices.Marshal (ser, "test.rem");
        }
    }
}
```

```
        Console.WriteLine ("Server Running ...");
        _ = Console.ReadLine ();

        ch.StopListening (null);

        0;
    }
}
}
```

8.3 ServerObjects

Save the following code in the file "ServerObjects.n".

```
using System;
using System.Runtime.Remoting;
using System.Collections;

namespace RemotingTest
{
    // A list of ServerObject instances
    public class ServerList : MarshalByRefObject
    {
        values : ArrayList;

        public this ()
        {
            values = ArrayList ();
        }

        public Add (v :ServerObject) :void
        {
            _ = values.Add (v);
            System.Console.WriteLine ("Added " + v.Name);
        }

        public ProcessItems () :void
        {
            System.Console.WriteLine ("Processing...");

            mutable total = 0;
            foreach (ob in values) total += (ob :> ServerObject).GetValue ();

            System.Console.WriteLine ("Total: " + total.ToString());
        }

        public Clear () :void
        {
            values.Clear ();
        }

        public NewItem (name :string) :ServerObject
        {
```

```
        def obj = ServerObject (name);
        Add (obj);
        obj;
    }

    public SetComplexData (data :ComplexData) :ComplexData
    {
        System.Console.WriteLine ("Showing content of ComplexData");
        data.Dump ();
        data;
    }
}

// A remotable object
public class ServerObject :MarshalByRefObject
{
    mutable _value :int;
    _name :string;

    public this (name :string)
    {
        _name = name;
    }

    public Name :string
    {
        get { _name; }
    }

    public SetValue (v :int) :void
    {
        System.Console.WriteLine ("ServerObject " + _name + ": setting " + v.ToString() );
        _value = v;
    }

    public GetValue () :int
    {
        System.Console.WriteLine ("ServerObject " + _name + ": getting " + _value.ToString());
        _value;
    }
}

public enum AnEnum { |A |B |C |D |E };

[Serializable]
public class ComplexData
{
    public Val :AnEnum;
    public Info : array [object];

    public this ( va:AnEnum, info : array [object])
    {
        Info = info;
    }
}
```

```

        Val = va;
    }

    public Dump () :void
    {
        System.Console.WriteLine ("Content:");
        System.Console.WriteLine ("Val: " + Val.ToString());
        foreach (ob in Info) System.Console.WriteLine ("Array item: " + ob.ToString());
    }
}
}

```

8.4 Compilation

Save the following code in the file "Makefile".

```

all:
    ncc -t:library ServerObjects.n -o ServerObjects.dll
    ncc -r:ServerObjects.dll -r:System.Runtime.Remoting.dll Client.n -o Client.exe
    ncc -r:ServerObjects.dll -r:System.Runtime.Remoting.dll Server.n -o Server.exe

```

Run "make" to compile the programs. Then execute the "Server.exe" in a console and "Client.exe" in another one.

8.5 Serializing Variants

By default custom attributes on **variants** are not passed to its variant options.

Thus using

```

[Serializable]
public variant AnEnum { |A |B |C |D |E };

```

instead of the proposed enum (which is directly serializable) will create an `System.Runtime.Serialization.SerializationException` error - you must explicitly declare each element of the variant as `Serializable`.

```

[Serializable]
public variant AnEnum {
    [Serializable] | A
    [Serializable] | B
    [Serializable] | C { x : string; }
    [Serializable] | D { x : int; }
    [Serializable] | E
};

```

8.5.1 Using a macro to mark all options with attribute

It is possible to use a macro `Nemerle.MarkOptions` to realize this in a compact way. For a little insight of how it works, there is its a full source, which fortunately is short.

```

using Nemerle.Compiler;
using Nemerle.Compiler.Parsertree;

[Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance,
                    Nemerle.MacroTargets.Class)]
macro MarkOptions (t : TypeBuilder, attribute)
{
  // iterate through members of this type and select only variant options
  foreach (ClassMember.TypeDeclaration
           (TopDeclaration.VariantOption as vo) in t.GetParsedMembers ())
  {
    // add given custom attribute to this variant option
    vo.AddCustomAttribute (attribute)
  }
}

```

Now you can use

```

[Serializable]
[Nemerle.MarkOptions (Serializable)]
public variant AnEnum {
  | A
  | B
  | C { x : string; }
  | D { y : int; }
  | E
}

```

Side note Remember that if you use variant instead of enum, the expression

```
AnEnum.B
```

has to be changed to

```
AnEnum.B ()
```

This is for keeping consistency with AnEnum.C ("Ala")

9 PInvoke (tutorial)

You can take advantage of native platform libraries from within Nemerle programs. The syntax is very similar to C#'s and other .NET languages. Here is the simplest example:

```

using System;
using System.Runtime.InteropServices;

class PlatformInvokeTest
{
  [DllImport("msvcrt.dll")]
  public extern static puts(c : string) : int;
}

```

```
[DllImport("msvcrt.dll")]
internal extern static _flushall() : int;

public static Main() : void
{
    _ = puts("Test");
    _ = _flushall();
}
}
```

As you can see we use *DllImport* attribute, which comes from *System.Runtime.InteropServices* namespace. Every method marked with this attribute should also have **extern** modifier. The concept is that the implementation of given method is substituted by call to unmanaged method from library specified inside *DllImport* attribute.

So, in above example, during execution of *Main* method first the *puts* function from *msvcrt.dll* will be called (printing a text to the standard output) and after that *_flushall* (making sure that all contents of buffer are printed).

For more see [P/Invoke tutorial](#) from MSDN.

10 Category:Tutorials

;

10.1 Articles in category "Tutorials"

There are 9 articles in this category.

- [ASP.NET \(tutorial\)](#)

10.1.1 B

- [Beers on the wall](#)

10.1.2 F

- [First Tutorial](#)

10.1.3 G

- [Gtk text editor \(tutorial\)](#)

10.1.4 P

- [PInvoke \(tutorial\)](#)

10.1.5 R

- [Remoting \(tutorial\)](#)

10.1.6 S

- [Second Tutorial](#)
- `¡REPLACE_ME_PLEASE title="System.Windows.Forms tutorial" ¿System.Windows.Forms tutorial|/a¿`

10.1.7 T

11 Tutorials and examples

11.1 Introduction

This page lists some snippets of Nemerle code you may want to look at. If you have any examples to be submitted here – please edit this page or contact us.

11.2 Short tutorials for writing example applications

The tutorials gathered here are meant to be used as an easy introduction to the the language, by stepping through the process of writing small applications.

- [Don't Panic! - Nemerle Basics Explained](#) – a simple tutorial covering basic concepts of the Nemerle language.
- [Hashtables and foreach](#) – presentation of how type inference makes code cleaner.
- Building a [text editor using GTK#](#).
- [System.Windows.Forms tutorial](#)
- [Macros tutorial](#) – the basics of meta-programming in Nemerle.
- Using [ASP.NET](#) with Nemerle.
- [Remoting](#) – presents a technique of accessing remote objects on the server.
- [PInvoking](#) – gives insight on how to use native (unmanaged) libraries from Nemerle.

There is also [Grokking Nemerle](#) – a "book" covering most of the language.

11.3 Snippets

You can find various examples of Nemerle code in [snippets](#) directory in Nemerle source tree. You are welcomed to transform them into tutorials. Examples include:

- [lcs.n](#) – short program computing the longest common substring
- [sql.n](#) – an example of database connectivity using Nemerle macros
- [suffix.n](#) – suffix trees
- [form.n](#) – a Windows.Forms example
- [myPoll.n](#) – an example of a CGI application connecting to a PostgreSQL database
- [power-race.n](#) – a ncurses-based "game". [Screenshot](#).
- [boyer-moore.n](#) – a Boyer-Moore algorithm
- [knuth-morris-pratt.n](#) – a Knuth-Morris-Pratt algorithm
- [nondec-subseq.n](#) – the longest non-decreasing subsequence
- [rachunki.n](#) – a billing generator for a conference
- [shift-or.n](#) – a shift-or text searching algorithm

These are also bundled with the source distribution.

11.4 Larger examples

There are also several more advanced examples there:

11.4.1 Sokoban solver

[Sokoban](#) – Sokoban solver by Bartosz Podlejski

11.4.2 Nemerle Course programs

Examples done by students during the Nemerle course at the CS Institute in the Wroclaw University.

- [ERA-SMS-Sender](#) – allows sending an SMS to a Polish GSM provider ERA (uses a POP3 client). By **Adrian Macal** *mel_on0 at o2 dot pl*.
- [backup-tool.n](#) – a program for performing backups, using simple scripts. By **Ryszard Trojnacki** *rysiek at menel dot com*.
- [freedb.org client](#) – a simple program to download information from [freedb.org](#) database for an audio CD currently in the drive under Linux. By **Wojtek Knapik** *d at hell dot art dot pl*.
- [swf-calculator.n](#) – a simple calculator using System.Windows.Forms. By **Marek Czajka** *marek_czajka at wp dot pl*.
- [huffman](#) – huffman compression and decompression. By **Ania Dwojak** *andzia200 at wp dot pl*.
- [getmxbyname.n](#) – a class to find out the MX record for a domain. By **Marcin Skórzewski** *pirol at o2 dot pl*.

11.4.3 ICFP'2004 Programming Contest entry

In June 2004 there was a [contest](#) to create finite state automata controlling the ants. Our entry is now available [here](#).

This year's contest page of our team is located here [ICFPC2005](#).

11.4.4 The Great Language Shootout Examples

[These](#) are the few examples ported from the [Win32 Computer Language Shootout](#). We didn't submit them yet, looking for volunteers to port more. When we have enough examples, we will also send them to the recently linked [renewed shootout](#).

11.4.5 ntrace

This [simple program](#) will trace memory leaks in programs written in C. By Jacek Śliwerski.

11.4.6 Sioux HTTP server

A [simplistic HTTP server](#). The main idea behind it is to serve web applications written in Nemerle. It already served as a subscription server for the XVIII-th FIT (a local conference) and later for the [CSL 2004](#).

11.5 External projects created using Nemerle

11.5.1 RiDL - Compiler tools

RiDL is a set of tools to simplify building compilers using Nemerle. It includes a lexical analyzer generator and a parser generator.

It is created by [Kojo Adams](#) and uses [BSD Licence](#).

11.5.2 Speagram

Speagram is an eager purely functional language with a strong type system and unusual syntax resembling natural language.

11.5.3 Asper

Asper is a text editor and IDE for Nemerle written in Nemerle.