Exercises
for Monday 1 May 2006

2005-04-26

Exercise C# 1 The purpose of the first four exercises is to get used to the C# compiler and to get experience with properties, operator overloading and user-defined conversions.

A Time value stores a time of day such as 10:05 or 00:45 as the number of minutes since midnight (that is, 605 and 45 in these examples). A struct type Time can be declared as follows:

```
public struct Time {
  private readonly int minutes;
  public Time(int hh, int mm) {
    this.minutes = 60 * hh + mm;
  }
  public override String ToString() {
    return minutes.ToString();
  }
}
```

Enter this declaration in a source file `TestTime.cs` and compile it using the C# compiler `csc`. The file should begin with the `using System;` directive. The next few exercises use this type.

In the same source file, add another class with a `Main` method in which you declare variables of type Time, assign values of type Time to them, and print the Time value using `Console.WriteLine`. Compile and run your program.

Exercise C# 2 In the Time struct type, declare a read-only property `Hour` returning the number of hours and a read-only property `Minute` returning the number of minutes. For instance, `new Time(23, 45).Minute` should be 45.

Modify the `ToString()` method so that it shows a Time in the format `hh:mm`, for instance `10:05`, instead of 605. You may use `String.Format` to do the formatting. Use these facilities in your `Main` method.

Exercise C# 3 In the Time struct type, define two overloaded operators:

- Overload (`+`) so that it can add two Time values, giving a Time value.
- Overload (`-`) so that it can subtract two Time values, giving a Time value.

It is convenient to also declare an additional constructor `Time(int)`. Use these facilities in your `Main` method. For instance, you should be able to do this:

```
Time t1 = new Time(9,30);
Console.WriteLine(t1 + new Time(1, 15));
Console.WriteLine(t1 - new Time(1, 15));
```

Exercise C# 4 In struct type Time, declare the following conversions:

- an implicit conversion from `int` (minutes since midnight) to Time
- an explicit conversion from Time to `int` (minutes since midnight)

Use these facilities in your `Main` method. For instance, you should be able to do this:

```
Time t1 = new Time(9,30);
Time t2 = 120;                      // Two hours
int m1 = (int)t1;
Console.WriteLine("t1={0} and t2={1} and m1={2}", t1, t2, m1);
Time t3 = t1 + 45;
```

Why is the addition in the initialization of `t3` legal? What is the value of `t3`?

Exercise C# 5 The purpose of this exercise and the next one is to understand the differences between structs and classes.

Try to declare a non-static field of type Time in the struct type Time. Why is this illegal? Why is it legal for a class to have a non-static field of the same type as the class?

Can you declare a static field `noon` of type Time in the struct type? Why?

Exercise C# 6 Make the `minutes` field of struct type Time `public` (and not `readonly`) instead of `private readonly`. Then execute this code:

```
Time t1 = new Time(9,30);
Time t2 = t1;
t1.minutes = 100;
Console.WriteLine("t1={0} and t2={1}", t1, t2);
```

What result do you get? Why? What result do you get if you change Time to be a class instead of a struct type? Why?

Exercise C# 7 The purpose of this exercise is to illustrate virtual and non-virtual instance methods.

In a new source file `TestMethods.cs`, declare this class that has a static method `SM()`, a virtual instance method `VIM()`, and a non-virtual instance method `NIM()`:

```
class B {
  public static void SM() { Console.WriteLine("Hello from B.SM()"); }
  public virtual void VIM() { Console.WriteLine("Hello from B.VIM()"); }
  public void NIM() { Console.WriteLine("Hello from B.NIM()"); }
}
```

Declare a subclass C of B that has a static method `SM()` that hides B’s `SM()`, has a virtual instance method `VIM` that overrides B’s `VIM`, and has a non-virtual instance method `NIM()` that hides B’s `NIM()`. Make C’s methods print something that distinguish them from B’s methods.

In a separate class (but possibly in the same source file), write code that calls the static methods of B and C.

Also, write code that creates a single C object and assigns it to a variable `b` of type B and a variable `c` of type C, and then call `b.VIM()` and `b.NIM()` and `c.VIM()` and `c.NIM()`. Explain the results.

Which of the methods `SM()` and `VIM()` and `NIM()` work as in Java?

Exercise C# 8 The purpose of this exercise is to illustrate delegates and (quite unrelated, really) the `foreach` statement.

In a new source file `TestDelegate.cs`, declare a delegate type IntAction that has return type `void` and takes as argument an `int`.

Declare a static method `PrintInt` that has return type `void` and takes a single `int` argument that it prints on the console.

Declare a variable `act` of type IntAction and assign method `PrintInt` (as a delegate) to that variable. Call `act(42)`.

Declare a method

```
static void Perform(IntAction act, int[] arr) { ... }
```

that applies the delegate `act` to every element of the array `arr`. Use the `foreach` statement to implement method `Perform`. Make an `int` array `arr` and call `Perform(PrintInt, arr)`.

Exercise C# 9 The purpose of this exercise is to illustrate variable-arity methods and parameter arrays.

Modify the `Perform` method above so that it can take as argument an IntAction and any number of integers. It should be possible to call it like this, for instance:

```
Perform(PrintInt, 2, 3, 5, 7, 11, 13, 17);
```