C# Exercises

This note contains a set of C# exercises originally developed by Peter Sestoft

Exercise C# 1.1 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();
  }
}
```

Create a VS2005 C# project called `TestTime`. Modify the `Main` method to 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.

The next few exercises use this type.

Exercise C# 1.2 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# 1.3 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# 1.4 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# 1.5 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# 1.6 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# 1.7 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# 1.8 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# 1.9 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);
```

The first two pages of exercises concern generic types and methods; the last page concerns attributes.

Exercise C# 2.1 The purpose of this exercise is to understand the declaration of a generic type in C# 2.0. The exercise concerns a generic struct type because structs are suitable for small value-oriented data, but declaring a generic class would make little difference.

A generic struct type Pair<T,U> can be declared as follows (C# Precisely example 182):

```
public struct Pair<T,U> {
  public readonly T Fst;
  public readonly U Snd;
  public Pair(T fst, U snd) {
    this.Fst = fst;
    this.Snd = snd;
  }
  public override String ToString() {
    return "(" + Fst + ", " + Snd + ")";
  }
}
```

(a) In a new source file, write a C# program that includes this declaration and also a class with an empty `Main` method. Compile it to check that the program is well-formed.

(b) Declare a variable of type Pair<String, `int`> and create some values, for instance `new Pair<String,int>("Anders", 13)`, and assign them to the variable.

(c) Declare a variable of type Pair<String, `double`>. Create a value such as `new Pair<String,double>("Phoenix", 39.7)` and assign it to the variable.

(d) Can you assign a value of type Pair<String,`int`> to a variable of type Pair<String,`double`>? Should this be allowed?

(e) Declare a variable `grades` of type Pair<String,`int`>[ ], create an array of length 5 with element type Pair<String,`int`> and assign it to the variable. (This shows that in C#, the element type of an array may be a type instance.) Create a few pairs and store them into `grades[0]`, `grades[1]` and `grades[2]`.

(f) Use the `foreach` statement to iterate over `grades` and print all its elements. What are the values of those array elements you did not assign anything to?

(g) Declare a variable `appointment` of type Pair<Pair<`int`,`int`>,String>, and create a value of this type and assign it to the variable. What is the type of `appointment.Fst.Snd`? This shows that a type argument may itself be a constructed type.

(h) Declare a method `Swap()` in Pair<T,U> that returns a new struct value of type `Pair<U,T>` in which the components have been swapped.

Exercise C# 2.2 The purpose of this exercise and the next one is to experiment with the generic collection classes of C# 2.0. Don't forget the directive `using System.Collections.Generic;`.

Create a new source file. In a method, declare a variable `temperatures` of type List<`double`>. (The C# collection type List<T> is similar to Java's ArrayList<T>). Add some numbers to the list. Write a `foreach` loop to count the number of temperatures that equal or exceed 25 degrees.

Write a method `GreaterCount` with signature

```
static int GreaterCount(List<double> list, double min) { ... }
```

that returns the number of elements of `list` that are greater than or equal to `min`. Note that the method is not generic, but the type of one of its parameters is a type instance of the generic type List<T>.

Call the method on your temperatures list.

Exercise C# 2.3 Write a generic method with signature

```
static int GreaterCount(IEnumerable<double> eble, double min) { ... }
```

that returns the number of elements of the enumerable `eble` that are greater than or equal to `min`. Call the method on an array of type `double[]`. Can you call it on an array of type `int[]`?

Now call the method on `temperatures` which is a List<double>. If you just call `GreaterCount(temperatures, 25.0)` you'll actually call the `GreaterCount` method declared in exercise 2.2 because that method is a better overload (more specific signature) than the new `GreaterCount` method. To call the new one, you must cast `temperatures` to type IEnumerable<double> — and that's legal in C#.

In C# it is legal to overload a method on type instances of generic types. You may try this by declaring also

```
static int GreaterCount(IEnumerable<String> eble, String min) { ... }
```

This methods must have a slightly different method body, because the operators (`<=`) and (`>=`) are not defined on type String. Instead, use method `CompareTo(...)`. Maybe insert a `Console.WriteLine(...)` in each method to be sure which one is actually called.

Exercise C# 2.4 The purpose of this exercise is to investigate type parameter constraints. You may continue with the same source file as in the previous two exercises.

We want to declare a method similar to `GreaterCount` above, but now it should work for an enumerable with any element type T, not just `double`. But then we need to know that values of type T can be compared to each other. Therefore we need a constraint on type T:

```
static int GreaterCount<T>(IEnumerable<T> eble, T x) where T : ... { ... }
```

(Note that in C# methods can be overloaded also on the number of type parameters; and the same holds for generic classes, interfaces and struct types). Complete the type constraint and the method body. Try the method on your List<double> and on various array types such as `int[]` and `String[]`. This should work because whenever T is a simple type or String, T implements IComparable<T>.

Exercise C# 2.5 Create a new source file `GenericDelegate.cs` and declare a generic delegate type Action<T> that has return type void and takes as argument a T value. This is a generalization of yesterday's delegate type IntAction.

Declare a class that has a method

```
static void Perform<T>(Action<T> act, params T[] arr) { ... }
```

This method should apply the delegate `act` to every element of the array `arr`. Use the `foreach` statement when implementing method `Perform<T>`.

Exercise C# 2.6 (Optional) As you know, C# does not have wildcard type parameters. However, most uses of wildcards in the parameter types of methods can be simulated using extra type parameters on the method. For instance, in the case of the `GreaterCount<T>(IEnumerable<T> eble, T x)` method, it is not really necessary to require that T implements IComparable<T>. It suffices that there is a supertype U of T such that U implements IComparable<U>. This would be expressed with a wildcard type in Java, but in C# 2.0 it can be expressed like this:

```
static int GreaterCount<T,U>(IEnumerable<T> eble, T x)
  where T : U
  where U : IComparable<U>
{ ... }
```

When you call this method, you may find that the C# compiler's type inference sometimes cannot figure out the type arguments to a method. In that case you need to give the type arguments explicitly in the methods call, like this:

```
int count = GreaterCount<Car,Vehicle>(carList, car);
```

Exercise C# 2.7 The purpose of this exercise is to illustrate the use and effect of a predefined attribute.

The predefined attribute Obsolete (see C# Precisely section 28) may be put on classes, methods, and so on that should not be used — it corresponds to the 'deprecated' warnings so well known from the Java class library.

Declare a class containing a method

```
static void AcousticModem() {
  Console.WriteLine("beep buup baap bzfttfsst %^@~#&&^@CONNECTION LOST");
}
```

Put an Obsolete attribute on the `AcousticModem` method and call the method from your `Main` method. What message do you get from the C# compiler? Does the message concern the declaration or the use of the `AcousticModem` method?

Exercise C# 2.8 The purpose of this exercise is to show how to declare a new attribute, how to put it on various targets, and how to detect at run-time what attributes have been put of a given target (in this case, a method).

Create a new source file. Declare a custom attribute BugFixed that can be used on class declarations, struct type declarations and method declarations. It must be legal to use BugFixed multiple times on each target declaration.

There must be two constructors in the attribute class: one taking both a bug report number (an `int`) and a bug description (a string), and another one taking only a description. (Presumably the latter is used when a bug does not get reported through the official channels). When no bug number is given explicitly, the number -1 (minus one) is used. The attribute class should have a `ToString()` method that shows the bug number and description if the bug number is positive, otherwise just the description.

It should be legal to use the BugFixed attribute like this:

```
class Example {
  [BugFixed(4, "Performance: Uses SortedDictionary")]
  [BugFixed(3, "Throws IndexOfOutRangeException on empty array")]
  [BugFixed("Performance: Uses repeated string concatenation in for-loop")]
  [BugFixed(2, "Loops forever on one-element array")]
  [BugFixed(1, "Spelling mistakes in output")]
  public static String PrintMedian(int[] xs) {
    /* ... */
    return "";
  }

  [BugFixed(67, "Rounding error in quantum mechanical simulation")]
  public double CalculateAgeOfUniverse() {
    /* ... */
    return 11.2E9;
  }
}
```

Write an additional class with a `Main` method that uses reflection to get the public methods of class Example, gets the BugFixed attributes from each such method, and prints them. If `mif` is a MethodInfoObject, then `mif.GetCustomAttributes(typeof(t), false)` returns an array of the type `t` attributes.

Some inspiration may be found in the full source code for C# Precisely example 208, which can be downloaded from the book's homepage `http://www.dina.kvl.dk/~sestoft/csharpprecisely/`.

There are probably too many exercises here. When you get tired of enumerables, jump to the last exercise so you get to use nullable types also.

Exercise C# 4.1 The purpose of this exercise is to illustrate the use of delegates and especially anonymous method expressions of the form `delegate(...) { ... }`.

Get the file `http://www.itu.dk/people/sestoft/csharp/IntList.cs`. The file declares some delegate types:

```
public delegate bool IntPredicate(int x);
public delegate void IntAction(int x);
```

The file further declares a class IntList that is a subclass of .Net's List<`int`> class (which is an arraylist; see C# Precisely section 24.4). Class IntList uses the delegate types in two methods that take a delegate as argument:

- `list.Act(f)` applies delegate `f` to all elements of `list`.
- `list.Filter(p)` creates a new IntList containing those elements `x` from `list` for which `p(x)` is true.

Add code to the file's `Main` method that creates an IntList and calls the `Act` and `Filter` methods on that list and various anonymous delegate expressions. For instance, if `xs` is an IntList, you can print all its elements like this:

```
xs.Act(Console.WriteLine);
```

This works because there is an overload of `Console.WriteLine` that takes an `int` argument and therefore conforms to the `IntAction` delegate type.

You can use `Filter` and `Act` to print only the even list elements (those divisible by 2) like this:

```
xs.Filter(delegate(int x) { return x%2==0; }).Act(Console.WriteLine);
```

Explain what goes on above: How many IntList are there in total, including `xs`?

Further, use anonymous methods to write an expression that prints only those list elements that are greater than or equal to 25.

An anonymous method may refer to local variables in the enclosing method. Use this fact and the `Act` method to compute the sum of an IntList's elements (without writing any loops yourself).

Note: If you have an urge to make this exercise more complicated and exciting, you could declare a generic subclass MyList<T> of List<T> instead of IntList, and make everything work for generic lists instead of just IntLists. You need generic delegate types Predicate<T> and Action<T>, but in fact these are already declared in the .Net System namespace.

Exercise C# 4.2 This exercise and the next one explore some practical uses of enumerables and the `yield` statement.

Declare a static method `ReadFile` to read a file and return its lines as a sequence of strings:

```
static IEnumerable<String> ReadFile(String fileName) { ... }
```

C# Precisely section 22.4 describes TextReader and example 153 shows how to create a StreamReader by opening a file. (The good student will of course use a `using` statement — C# Precisely section 13.10 — to bind the TextReader to make sure the file gets closed again, even in case of errors).

The `ReadFile` method should read lines from the TextReader, using the `yield return` statement to hand the lines to the 'consumer' as they are produced. The consumer may be a `foreach` statement such as
`foreach (String line in ReadFile("foo.txt")) Console.WriteLine(s);`.

Exercise C# 4.3 Declare a static method `SplitLines` that takes as argument a stream of lines (strings) and returns a stream of the words on those lines (also strings)

```
static IEnumerable<String> SplitLine(IEnumerable<String> lines) { ... }
```

C# Precisely example 191 shows how a regular expression (of class System.Text.RegularExpressions.Regex) can be used to split a string into words, where a ‘word’ is a non-empty contiguous sequence of the letters a–z or A–Z or the digits 0–9.

The `SplitLine` method should use a `foreach` loop to get lines of text from the given enumerable `lines`, and use the `yield return` statement to produce words.

It should be possible to e.g. find the average length of words in a file by combining the two methods:

```
int count = 0, totalLength = 0;
foreach (String word in SplitLines(ReadFile "foo.txt")) {
  count++;
  totalLength += word.Length;
}
double averageLength = ((double)totalLength)/count;
```

Note that in this computation, only a single line of the file needs to be kept in memory at any one time. In particular, the call to `ReadFile` does not read all lines from the file before `SplitLines` begin to produce words. That would have been the case if the methods had returned lists instead of enumerables.

Exercise C# 4.4 The purpose of this exercise and the next one is to emphasize the power of enumerables and the `yield` and `foreach` statements.

Declare a generic static method `Flatten` that takes as argument an array of IEnumerable<T> and returns an IEnumerable<T>. Use `foreach` statements and the `yield return` statement. The method should have this header:

```
public static IEnumerable<T> Flatten<T>(IEnumerable<T>[] ebles) { ... }
```

If you call the method as shown below, you should get 2 3 5 7 2 3 5 7 2 3 5 7:

```
IEnumerable<int>[] ebles = new IEnumerable<int>[3];
ebles[0] = ebles[1] = ebles[2] = new int[] { 2, 3, 5, 7 };
foreach (int i in Flatten<int>(ebles))
  Console.Write(i + " ");
```

Exercise C# 4.5 (If you enjoy challenges) Redo the preceding exercise without using the `yield` statement.

Exercise C# 4.6 The purpose of this exercise is to illustrate computations with nullable types over simple types such as `double`.

To do this, implement methods that work like SQL’s aggregate functions. We don’t have a database query at hand, so instead let each method take as argument an IEnumerable<`double?`>, that is, as sequence of nullable `double`s:

- `Count` should return an `int` which is the number of non-`null` values in the enumerable.
- `Min` should return a `double?` which is the minimum of the non-`null` values, and which is `null` if there are no non-`null` values in the enumerable.
- `Max` is similar to `Min` and there is no point in implementing it.
- `Avg` should return a `double?` which is the average of the non-`null` values, and which is `null` if there are no non-`null` values in the enumerable.
- `Sum` should return a `double?` which is the sum of the non-`null` values, and which is `null` if there are no non-`null` values. (Actually, this is weird: Mathematically the sum of no elements is 0.0, but the SQL designers decided otherwise. This design mistake will also make your implementation of `Sum` twice as complicated as necessary: 8 lines instead of 4).

When/if you test your method definitions, note that `null` values of any type are converted to the empty string when using `String.Format` or `Console.WriteLine`.