Introduction – Part II

Radu Nicolescu
Department of Computer Science
University of Auckland

24 Sep 2020

1. WebAPI with Carter
2. Basic functional morphisms
3. Nested functions
4. Type dynamic
5. The expando object
6. Named and default parameters
7. Try ...
8. Hashtables
9. Nullables
10. Elvis op and the pyramide of doom
11. Miscellany

WebAPI with Carter

- Carter : open source layer over ASP.NET
- Opinion : ASP.NET as it should have been created form start
- More functional style, fluent validation, etc
- Name : Sinatra, Nancy, Carter (Jay-Z); cf. Node.js express
- Github : `https://github.com/CarterCommunity/Carter`
- Intro : `https://www.hanselman.com/blog/TheOpenSourceCarterCommunityProjectAddsOpinionatedElega` `aspx`
- Docs required : `https://dotnet.microsoft.com/apps/aspnet/apis` `https://docs.microsoft.com/en-us/aspnet/web-api/`

Carter from scratch

- Once : install the template

```
dotnet new -i CarterTemplate
```

- Create skeleton project

```
dotnet new Carter -n MyCarterApp
```

- Build and run the created skeleton

```
dotnet run -p MyCarterApp.csproj
```

- Skeleton offers a GET function “Hello ...”

Carter testing

- Default ASP.NET ports: http 5000, https 5001
- Test: Postman (gui convenience)
- Test: curl (cmd automated)

```
> curl -s -S -X GET http://localhost:5000/
> curl -s -S -X GET https://localhost:5001/
```

Carter testing

- Test: httprepl (cmd ASP.NET) https://docs.microsoft.com/en-us/aspnet/core/web-api/http-repl?view=aspnetcore-3.1&tabs=windows

```
> httprepl
get http://localhost:5000
get https://localhost:5001
```

- Install (once) httprepl

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

Carter files – can be merged

- .....csproj : compiles all extant .cs files
- Program.cs : main entry point – can change urls here
  e.g. ports 5000 → 8000
- Startup.cs : ASP.NET configuration
- ....Module.cs : actual custom code (GET, PUT, POST, etc)

Carter – custom url configuratiion

- Program.cs : original (default ports 5000, 5001)
  note expression lambda

```
var host = Host.CreateDefaultBuilder (args)
    .ConfigureWebHostDefaults (webBuilder =>
          webBuilder.UseStartup<Startup>()  )
    .Build();
```

- Program.cs : change default urls – note statement lambda

```
var host = Host.CreateDefaultBuilder (args)
    .ConfigureWebHostDefaults (webBuilder => {
         webBuilder.UseStartup<Startup> ();
         webBuilder.UseUrls (
             "http://localhost:8081",
             "https://localhost:8082");  })
    .Build();
```

Carter – skeleton module code

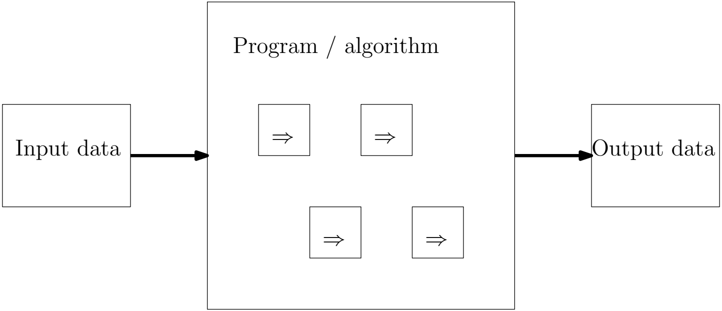
- HomeModule.cs

```
public class HomeModule : CarterModule {
    public HomeModule () {
        Get("/", async (req, res) =>
            await res.WriteAsync ("Hello ..."));
}}
```

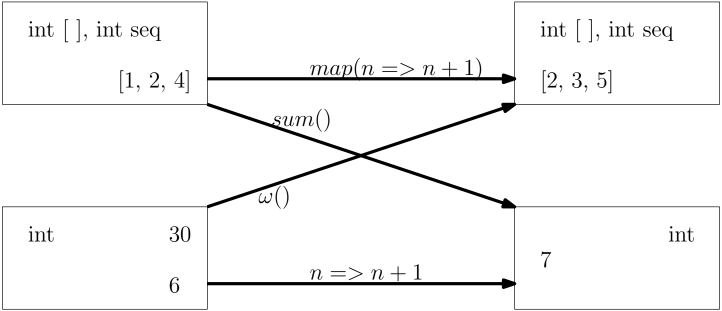
- Get registers a GET method (lambda) for the base URLs
- Complex routing: urls extensions, query string parameters, ...
- Other methods: Post for POST, Put for PUT, ...
  – conventions
- Conversions: req.Bind<X>(), res.AsJson (x)
  – also fluent validations
- Self describing : OpenAPI / Swagger

Basic functional morphisms (transformations)

- obj $\rightsquigarrow$ obj : n $\Rightarrow$ n + 1
- obj seq $\rightsquigarrow$ obj seq : map (n $\Rightarrow$ n+1), filter (...)
- obj seq $\rightsquigarrow$ obj : sum (), fold (...), reduce (...)
- obj $\rightsquigarrow$ obj seq : unfold (...)
- recursion – no explicit loops



Basic functional morphisms (transformations)



- hylomorphisms : map (n ⇒ n+1), filter (...), ...
- catamorphisms : sum (), fold (...), reduce (...), ...
- anamorphisms : ω (), ...

Nested functions

- Functions nested in other functions – a FP feature

```
int Outer () {
    int s = 0;
    void Inner (int i) {
        if (i > 0) {
            s += i;
            Inner (i-1);
        }
    }
    Inner (4);
    return s;
}
```

- Result: $\boxed{10}$

ReadAllLines with nested functions

```
string[] ReadAllLines (TextReader inp) {
    var s = new List<string> ();
    void Inner () {
        var line = inp.ReadLine ();
        if (line != null) {
            s.Add (line);
            Inner ();
        }
    }
    Inner ();
    return s.ToArray ();
}
```

Type dynamic

- Dynamic types transgress the usual OO class and interface hierarchy
- Only the actual object capabilities count, not its type credentials
- This is also known as "duck typing"

  "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

  `en.wikipedia.org/wiki/Duck_typing`
- `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/dynamic`

Type dynamic

- For example, consider two totally unrelated types which implement a method Talk (same name and signature)

```
class Cat { public void Talk() { meow } }

class Duck { public void Talk() { quack } }
```

- And method Test that looks like almost applicable to instances of both classes

```
void Test(??? pet) {
    pet.Talk();
}
```

- Problem: what should we declare for the type of pet ???, to keep the compiler happy?

Type dynamic

- We declare pet **dynamic**!

```
void Test(dynamic pet) {
    pet.Talk();
}
```

- Then this will work!

```
    var moggy = new Cat();
    Test(moggy);

    var donald = new Duck();
    Test(donald);
```

- The capability checks are now the responsibility of the runtime

The expando object

- An object with dynamically increasing capabilities (at runtime)!
- `http://www.c-sharpcorner.com/Blogs/9439/expandoobject-in-C-Sharp-4-0.aspx` `https://msdn.microsoft.com/en-us/library/system.dynamic.expandoobject(v=vs.110).aspx`
- This creates an empty expando object (no visible properties or methods) – must be typed as dynamic

```
dynamic p = new System.Dynamic.ExpandoObject();
```

- This dynamically expands it with two properties, X and Y

```
p.X = 10;
p.Y = 20;
Console.WriteLine("X={0}, Y={1}", p.X, p.Y);
```

The expando object

- This dynamically expands it with one method, MoveX

```
p.MoveX = (Action<int>)
     ((int deltax) => { p.X += deltax; });
```

- We'll talk later about such inline functions (don't worry for now)
- Just keep in mind that this works!

```
p.MoveX(100);
Console.WriteLine(”X={0}, Y={1}”, p.X, p.Y);
```

- Note: an expando object is implemented as a dynamic dictionary...

Named and default parameters

- Consider this method

```
void F(int x, int y=20, string z="abc") { ... }
```

- Parameter x is mandatory
- Parameter y is optional: if not given, the default value 20 is assumed
- Parameter z is optional: if not given, the default value "abc" is assumed
- Parameters can be given by position (as traditionally) and/or by name

Named and default parameters

- Consider this method

```
void F(int x, int y=20, string z="abc") { ... }
```

- Possible calls

```
    F(10, 15, "xyz");
    F(y:25, z:"xyz", x:10);

    F(100);
    F(100, z:"klm");
```

- Named and default parameters are frequently used in some framework, e.g. MVC

Try ...

- Consider this frequently used coding pattern, which avoids exception handling

```
    int r;
    bool b;

    b = int.TryParse("123", out r);
    if (b) // use r
    else   // ...

    b = int.TryParse("abc", out r);
    if (b) // ...
    else   // r is useless, 0
```

- The API contains quite a few similar **bool** TryX(..., **out** r) methods

Hashtable based collections

Hashtables are data structures based on hash functions, which are often more efficient than other structures (such as arrays or trees), in the sense that, on average, elementary operations (retrievals, insertions, deletions) take constant time, $O(1)$.

The following hashtable based structures will be often used in our topics (please make yourself familiar with these):

- HashSet<T>
- Dictionary<TKey, TValue>
- Lookup<TKey, TValue>

HashSet – cf. LINQ Distinct ()

HashSet<T> class provides a high performance set of elements of type T.

A set is a collection that contains no duplicate elements, and whose elements are in no particular order (although you can list them in some order).

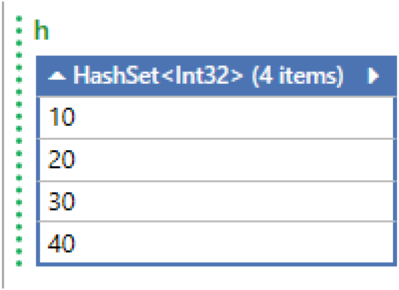
$$\{elem_0, elem_1, elem_2, \ \ldots\}$$

HashSet<T> methods:

```
bool Add(T elem)
bool Contains(T elem)
bool Remove(T elem)
```

HashSet Example

```
var h = new HashSet<int> (new[]  {
            10, 20, 30, 40, 30, 20, 10, });
```



```
h
HashSet<Int32> (4 items)
10
20
30
40
```

Dictionary – cf. LINQ ToDictionary ()

Dictionary<TKey, TValue> class provides a mapping from a set of keys to a set of values (TKey → TValue).

Each addition to the dictionary consists of a pair (key, value), where keys are unique.

$$\{k_0 \to v_0, k_1 \to v_1, k_2 \to v_2, \ \ldots\}$$

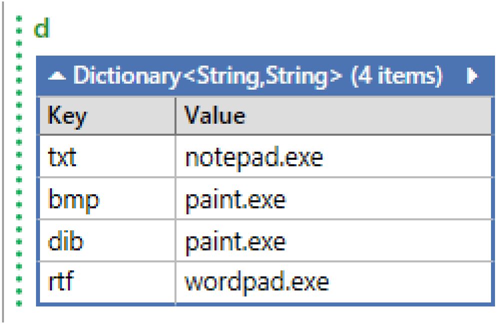
Dictionary<TKey, TValue> methods:

```
void Add(TKey key, TValue value) // exception if key exists

TValue this[TKey key] { get; set; }   // set can replace
                                 // get exception if key does not exist

bool TryGetValue(TKey key, out TValue value)    // !
bool Remove(TKey key)
```

Dictionary Example



d

Dictionary<String,String> (4 items)

Key	Value
txt	notepad.exe
bmp	paint.exe
dib	paint.exe
rtf	wordpad.exe

Lookup – cf. LINQ ToLookup ()

Lookup<TKey, TValue> resembles a Dictionary<TKey, TValue>.

The difference is that a dictionary maps keys to single values, whereas a lookup maps keys to enumerable collections of values. Also, lookups are immutable, i.e. once created, cannot be changed.

$$\boxed{\left\{ k_0 \to \{v_{00}, v_{01}, \ldots\}, k_1 \to \{v_{10}, v_{11}, \ldots\}, k_2 \to \{v_{20}, v_{21}, \ldots\}, \ \ldots \right\}}$$

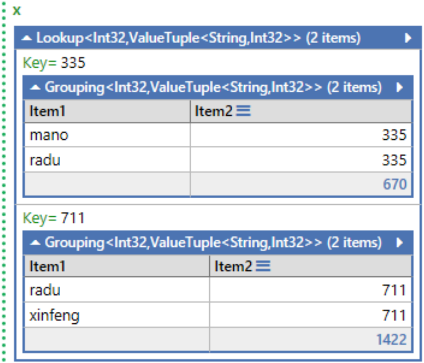
Lookup<TKey, TValue> indexer (read-only):

```
IEnumerable<TValue> this[TKey key] { get; }
                           // Item in VB, ...

// example
IEnumerable<Student> students = lookup["335"];
```

Lookup Example

```
var t = new[] { ("mano", 335), ("radu", 335),
          ("radu", 711),    ("xinfeng", 711),};
var x = t.ToLookup (p => p.Item2);
```



x

Lookup<Int32,ValueTuple<String,Int32>> (2 items)

Key= 335

Grouping<Int32,ValueTuple<String,Int32>> (2 items)

Item1	Item2 ≡
mano	335
radu	335
	670

Key= 711

Grouping<Int32,ValueTuple<String,Int32>> (2 items)

Item1	Item2 ≡
radu	711
xinfeng	711
	1422

Nullable types—Overview

- SQL $\boxed{\text{x } \textbf{int} \ \textbf{null}}$: **null** $\neq$ 0!
- C#'s nullable types provide complete and integrated support for nullable forms of all **struct** types (technically, this includes all *numeric* types).
- useful for smooth SQL data integration
  - one exception: in C#, **null**==**null** is **true**
  - while in standard SQL NULL=NULL is UNKNOWN (which practically, in many tests, is equivalent to FALSE)
- Nullable types are constructed using the generic type Nullable<T> or (as a shorthand) the T? type modifier.
- Each nullable type has the properties HasValue and Value.
- Additional operator: ??

Nullable types

```
int? x = 7;
int? y = null;
int i = 3;
if (x.HasValue) { i = x.Value; }
if (y.HasValue) { i = y.Value; }
//      .HasValue can be replaced by != null,
//      unless this operator is overridden
```

```
int i = 7;
int? x = i;                //  int → int?
double? y = x;             //  int? → double?
int? z = (int?)y;          //  double? → int?
int j = (int)z;            //  int? → int
                           //  exception if z==null
```

Nullable types

```
int? z = x ?? y;
int i = y ?? −1;
int? w = x + y;
```

The result of x??y is x, if x is non-**null**; otherwise is y.

The result of x+y is **null**, if any of the operands is **null**.

Note: Nullable<**int**> offers coherent extensions for all arithmetic operations available on **int**.

Extra null operators

Meet the following operators:

- ?? : null coalescing operator (similar to Elvis operator ?: used eg in Kotlin)
- ?. : null conditional operator aka safe navigation operator – for properties and methods
- ?[ ] : null conditional operator aka safe navigation operator – for arrays

Safe navigation – scenario

Consider the following class:

```
class Person {
    public string Name { get; set; }
    public Person Father { get; set; }
}
```

Problem: get the name of the grandfather of a given Person x

You write:

```
string Grand (Person x) {
    return x.Father.Father.Name;
}
```

What's wrong? The Father pointer may be **null** (and also the Name) – possible exceptions

Safe navigation – quick and dirty solution

Instead of:

```
return x.Father.Father.Name;
```

You write:

```
if (x != null && x.Father != null
              && x.Father.Father != null) {
    return x.Father.Father.Name;
} else {
    return null;
}
```

What's now wrong? Possible repeated computations of Fasther and possibly some side-effects!

.Father could be a method call like .GetFather()

Safe navigation – Pyramide of doom “solution”

You may want to write:

```
if (x != null) {
    var f = x.Father;   // called once only
    if (f != null) {
        var g = f.Father;   // called once only
        if (g != null) {
            return g.Name;
        } else {
            return null;
        }
    } else {
        return null;
    }
} else {
    return null;
}
```

Would you do this? Better solution?

Safe navigation – solution!

You can write ☺:

```
    return x?.Father?.Father?.Name;
```

Note: this is the default semantics in Objective-C!

```
    return x.Father.Father.Name;
```

Safe navigation and null coalescing!

Problem: write "Unspecified" if either p is null or its name is null:

```
Person p = new Person { Name = "Dr Evil" };
              new Person();  // Name == null
              // null

var n1 = p == null? "Unspecified":
         (p.Name == null? "Unspecified": p.Name);

var n2 = p?.Name ?? "Unspecified";

WriteLine ($"{n1} {n2}");
```

Properties

Structured accessors, i.e. (getter, setter) pair

You write (inside a class definition):

```
    private int a;                  // private backing field

    public int A {                  // public property
        get { return a; };          // getter
        set { a = value; }          // setter
     }
```

Conceptually equivalent code:

```
    private int a;
    public int getA () { return a; }
    public void setA (int value) { a = value; }
```

Properties

Usage (right-hand vs left-hand side):

```
x = t.A;   // x = t.getA();
t.A = y;   // t.setA(y);
```

Increment A/a as a property

```
t.A = t.A + 1;
```

Compare to incrementation via accessors

```
t.setA (t.getA() + 1);
```

Many frameworks require public properties instead of public fields.

Auto-Properties

Automatically generated accessors
Most simple – no filters, no side effects

You write (inside a class definition):

```
    public int A { get; set; }
```

Compiler generates:

```
    private int a;
    public int getA () { return a; }
    public void setA (int value) { a = value; }
```

Usage (right-hand vs left-hand side):

```
x = t.A;          // x = t.getA();
t.A = y;          // t.setA(y);
t.A = t.A + 1;    // t.setA (t.getA() + 1);
```

Auto-Properties

Read-only – only from outside

```
    public int B { get; private set; }
```

Truly read-only – initialised in constructor

```
    public int C { get; }
```

Truly read-only – initialised locally

```
    public int D { get; } = 4;
```

Truly read-only – initialised locally
This style can also be used for methods

```
    public int E => 5;
```

using static

WriteLine is a static method of class System.Console

Traditional:

```
using System;
...

    Console.WriteLine (”This is the Console!”);
```

New – use static methods as top-level functions:

```
using static System.Console;
...

    WriteLine (”Where is the Console?”);
```

string interpolation

Note the $ sign before the format string:

```
var greet1 = "Hello";
var greet2 = "Goodbye";

WriteLine ("{0}, {1}!", greet1, greet2); // traditional

WriteLine ($"{greet1}, {greet2}!"); // new
```