# Procedural

We’ll use modern c#, but try to stick to a readable style (using var), but using code similar to that used when an approach was common – so “for …” in procedural rather than “foreach…”.

Sidebar – we aren’t using struct, because that doesn’t allow us to change it after creation, which is something we’ll come back to later.

We are going to use public, and members not properties.

# Object Orientated

## Simple objects

## Inheritance

## Interfaces

## Complex custom object tree - DTOs

# Functional Concepts

## Introduction

### What’s that

We are going to look at the following and try to explain them.

* Setoid
* Semigroup
* Monoid
* Functor
* Applicative
* Monad

We’ll then look at why that’s a good idea.

### What it’s not

It’s always nice to try and explain why something is a good idea and what the point of it is, it helps with the learning. With these principles it’s a little like trying to explain to someone starting to learn to code why it’s a good idea to have a kinda little box with a value on that we give a name and call it a variable. Sometimes you just have to go with it, and things become clearer later, so you aren’t going to get the “why” up front. We’ll be sure to move on to that later though.

This is NOT going to be a mathematically, algebraically, category theoryally strict definition of what we discuss – strict correctness too often means many corollaries, qualifications and generally illegibility so as to make the subject impenetrable to the beginner. We may note as a detail some of the strict things we are glossing over, but not all of them!

|  |
| --- |
| * We’ll talk later about associativity. This can be explained by saying that for some operation “.”, then   (x.y).z == x.(y.z)  ... Or ... the whole 5 screenfulls of various exotic symbols that is  <https://en.wikipedia.org/wiki/Associative_property> |

Not being strict means we’ll possibly bust a learning principle called *Primacy* (<https://en.wikipedia.org/wiki/Principles_of_learning#Primacy>). This holds that if you don’t teach it right first time, it’s much harder to shake the wrong first understanding later. It’s especially important in safety-related teaching where stress and confusion can result in subconsciously reverting to those first-taught principles, despite later teaching and experience.

|  |
| --- |
|  |

* Say a sub-aqua diver is learning in a shallow training pool and is taught that if their oxygen supply cuts they should stand up and remove the mask. Years later with lots of more advanced training and experience, they may get into difficulty. It’s possible they may react instinctively by kicking their legs to stand up – and possibly ripping off the mask. That would be primacy at work.

If theoretical strictness and primacy are your kinda thing, probably best to find another guide!

### Show me in C#

We’ll show the ideas in C# code whenever possible. The idea is to give a familiar context for an “average” C# programmer to what is being discussed. It’s not the intention to give the best code or perfect code, but it will be valid C# code, which is available for download.

There’s always a way to write it better - where everyone has their own definition of “better”. In fact from experience if there are *n* programmers in a room, they will probably come up with at least *2n+1* ways to improve any given bit of code.

So the C# given is illustrative of the ideas, not a strict definition or a template for all to follow.

## Setoid

### What’s that

A setoid is a type together with an understanding of which items are “equivalent”. A definition for “equivalent” may be “equals” – all the member values of two objects of the type match exactly – but it doesn’t have to be this.

The understanding of equivalence may vary depending on the usage even for the same type of object. So there may be more than one setoid relating to a given type. See the product examples below.

### Explain the name

Setoid – some rules about the things that can be in a “set” such as a container or collection (let’s not get too strict about what a “set” is).

### What it’s not

Despite its name, it’s not a kind of set, container or collection, or a subtype of one of those - It’s not some sort of fancy alternative to a list.

It’s not a thing that’s in a collection (but in a collection where we care about setoids, the items will be of the type belonging to a setoid).

It’s not a type, it’s not a way of understand the equivalence between two objects of a type (it’s the two together).

Functions/delegates/methods don’t form setoids. There can be a collection or container of them, but there’s no useful understanding of equivalence - apart from exhaustive trial and error, how could it be determined that f(x) => x² is equivalent to f(x) => x \* x ?

C# mashes this a bit because it treats functions/delegates/methods as an object which have an “Equals()” method, but this is a reference equals only. It also uses a kludge to enable Equals() to be called on Value types, which is handy for us to do examples using Ints and Floats, but muddies the waters a bit when trying to relate the concepts in C#.

### Examples

* When looking at products, if we are doing a stock-take we may wish to consider each item unique based on serial number.
* Or if compiling a catalogue, we may consider all the same product type and package quantity as equivalent.
* Customers on a website may be uniquely identified by the credit card number they use irrespective of what name, spelling or address they use at the time of order.
* More generally if an object has any form of unique id, the equivalence test can be shortcutted so it doesn’t look at every member value, just the unique id.

### Show me in C#

There’s very little to this code-wise. Strictly you could write this.



But in C# the object base class kinda implements the setoid interface definition, and it’s probably better all round in C# not to bother and to just override the Equals() method of the base object, though you’ll need to cast to your type and add null protection, such as



This is the simple case just using the default Equals(). If there’s a need for different understandings of equivalence for the same type in different use cases, it may be necessary to define and call different methods to test equivalence when using the class as a setoid in different situations.

### Deep dive

Some Haskell examples are given in <https://hackage.haskell.org/package/setoid-0.1.0.0/docs/Data-Setoid.html>

### Get the picture

So here's what a setoid isn't, and is, in pictures.

### Setoid.png

Figure - Setoid

## Semigroup

### What’s that

It’s a collection of items of a type belonging to a setoid together with an *associative* operation.

Associative means that for an operation denoted by “.”, that

(x.y).z == x.(y.z)

And when we say “==”, what we mean is the understanding of equivalence that the setoid has.

But it’s the two things together

* a collection of items of the type which belongs to the setoid
* an associative operation

which make a semigroup.

Strictly, the collection and the operation should also be “closed” meaning that for every

z = x.y

with x and y from the collection, then z will also be from the collection.

This is a more theoretical issue than a practical one for a developer. If the type of the setoid is integer, then this together with the addition operation seems like it would be a semigroup – it's our first example below.

But in C# you get into the issue of maxint so it isn’t strictly a semigroup – what's MaxInt+10?. It's either a value outside of the bounds of Int32, or if we try to be clever it's -9, but then we find that the operation is no longer associative.

Mostly the closure bit can be ignored from the functional concepts perspective we are looking at, though as always it will need to be considered when choosing your types based on the range of expected values.

### Explain the name

This is probably the clearest-named thing we will look at in functional concepts.

You won’t be surprised to hear it’s got less rules to abide by than a “group” (which we aren’t going to look at).

But it’s got one more rule than a ... *Magma* ... mathematicians eh? Go figure!

### What it’s not

It’s not the nature of the collection – dictionary, list etc.

It’s not the type of what’s in the collection (that’s the type which belongs to the semigroup’s setoid).

It’s not the collection. It’s not the operation. (It’s the two together).

### Examples

* Positive Integers (1,2,3….n) together with addition form a semigroup.

(1 + 2) + 3 == 1 + (2 + 3)

3 + 3 == 1 + 5

6 == 6



* Strings together with concatenation form a semigroup.

("hello" + "there") + "world" == "hello" + ("there" + "world")

"hellothere" + "world" == "hello" + "thereworld"

"hellothereworld" == "hellothereworld"



* Positive integers together with subtraction are, perhaps surprisingly, NOT a semigroup since the operation is not associative

(3 - 2) – 1 == 3 – (2 – 1)

1 – 1 == 3 – 1

0 == 2



This is also why negative integers together with addition are not a semigroup.

* OK, so integers together with subtraction did not surprise you? Well we've already seen that integers and addition do form a semigroup. Then would it surprise you that on a computer, floating point numbers and addition do not form a semigroup, since floating point addition is not associative?

(0.001 + 0.001) + 0.003 == 0.001 + (0.001 + 0.003)

0.002 + 0.003 == 0.001 + 0.004



Don't believe it? Here's the output from Visual Studio 2017's C# interactive window (View -> Other Windows -> C# interactive):

> 0.001f + 0.001f

0.002

> 0.002f + 0.003f

0.005

> 0.001f + 0.003f

0.004

> 0.001f + 0.004f

0.00500000035

> (0.001f + 0.001f) + 0.003f == 0.001f + (0.001f + 0.003f)

False

There's more detail on this at <https://www.quora.com/Is-floating-point-addition-commutative-and-associative> and

### Show me in C#

Let's make the assumption that the collection of the type which belongs to the setoid needed to form the semigroup can be any and all objects of the type which belongs to the setoid.

If we didn't make this assumption, we'd have to also pass in the collection of objects of the type which belongs to the setoid. That wouldn't be hard to code – another property on the interface of type IEnumerable<T> is all. But often we will want the collection to be all possible values of the type which belongs to the Setoid – e.g. all integers. C# doesn't give us a sensible usable way of specifying that, so let's pretend shall we?

We can then make use of a semigroup in C# with the following interface definition:



OK, let's actually define a semigroup class so we can test some of the examples above.



and let's define some operations to test with



Now let's make an actual semigroup and use the method to test if it's valid. We'll use a little trick to generate a collection of integers from 1 to 100 to test with.



Nice. So let's try the float addition semigroup. We'll beef up the enumerable trick to give as a collection of float values.



The reason it's false is the lack of associativity of the float add operation on a computer.

Well we've come this far, so time for a confession – something was deliberately skipped over above when the float addition was pronounced non-associative. Our judgement on that was based on

0.002 + 0.003 == 0.001 + 0.004



Because it actually evaluated to

0.005 == 0.00500000035

**%%% CHECK THIS – c.f. “talk of Associative as far as Isomorphism” (jjelfs) %%%**

Except ... what is "=="? Let's go right back to the beginning in the Semigroup first section "What's that":

And when we say “==”, what we mean is the understanding of equivalence that the setoid has.

Above we read "==" as "equals" so *clearly* the evaluation above wasn't true. But we can define a semigroup with a setoid that has a different definition of "equivalent". Let's use the usual approach for float of checking if two values are within a tolerance. Now we aren't using a value type, the Add operation will have to be a wee bit more complicated and dig the value out of the object. This is something you may recognise in later sections. ;-)



Now we can use this to create and test a new semigroup. We need to play with the enumerable trick a bit more to create our collection of our setoid Floatish, but it's straightforward linq stuff.



Cool. Is it starting to feel like an actual usable concept now? Where to use it still isn't obvious, but at least it's looking usable!

### Deep dive

<https://en.wikipedia.org/wiki/Semigroup>

<http://blog.ploeh.dk/2017/11/27/semigroups/>

<https://en.wikipedia.org/wiki/Group_(mathematics)>

<https://en.wikipedia.org/wiki/Magma_(algebra)>

### Get the picture

## Monoid

### What’s that

It’s a semigroup, where the collection of items includes an “empty” (aka Identity) value for the associative operation.

Recall that a semigroup is

a collection of items of a type belonging to a setoid together with an associative operation.

An empty/identity value is one for which if E is the Empty value, for all x in the collection:

E.x == x.E == x

In E.x == x, E is known as a “left identity” and in x.E == x it’s a “right identity”. If it’s both it’s known as a “two-sided identity”, but usually just as an identity.

Once again, as with the semigroup, when we say “==”, what we mean is the understanding of equivalence that the setoid has.

* Examples of identity values for associative operations are:
* 0 for integer addition - 0 + x == x for all x
* 1 for integer multiplication - 1 \* x == x for all x

So the complete description of a monoid is:

* a collection of items of the type which belongs to the setoid
* … which includes an identity value
* together with an associative operation

### Explain the name

It seems to be subject to a fair bit of debate[1], but probably just to do with having a single associative operation, or possibly a single identity value – which itself is often denoted by “1”.

Yup, setoids and semigroups have a single operation too.

### What it’s not

This section is very similar to that for a semigroup. In fact there’s not so many practical cases where you’d have a semigroup which isn’t a monoid – for this there’d need to be a semigroup which didn’t have an identity value. The introduction of semigroup was really just a place on the path taking us to monoid.

It’s not the nature of the collection – dictionary, list etc.

It’s not the type of what’s in the collection (that’s the type which belongs to the semigroup’s setoid).

It’s not the collection. It’s not the operation. It’s not the identity value. (It’s the three together).

### Examples

* Positive Integers (1,2,3….n) together with addition and identity value 0 form a monoid.
* Strings together with concatenation and identity value of the empty string (“”) form a monoid.
* Integers together with multiplication and identity value 1 form a monoid.

### Show me in C#

We can define a monoid by extending the definition of a semigroup



Let’s make a monoid class so we can test some examples

We’ll copy the AccociativeOperation and IsValidSmeigroup() members from the SemiGroup class so it’s all in one place here, but we could also have decided to have our Monoid class inherit from the previous Semigroup class.

We’ll also reuse an operation we met in the semigroup chapter:



So the monoid class definition is:



Let’s try a non-valid monoid:



As the comment says, the identity value 0 is not part of the collection:

A valid monoid would be:



### Deep dive

<https://en.wikipedia.org/wiki/Monoid>

### Get the picture

## Functor

### What’s that

A functor is something that can map a function over the content which is within it. The content is said to be within a “context” in the functor.

It has a method which takes as a parameter a function which it then applies to all items in its context. It will then return a functor of the same form, but this will contain in its context the results of applying the function to each of its items. The type of the items in the return functor may be different to the type of those in the original functor – this is down to the signature of the function applied.

Convention names the functor’s method “fmap”.

The context which the items are held in may be some kind of collection, or it may just be that the item which the function is targeting is within some structure within the functor.

### Explain the name

It’s something that knows how to apply a function to its content.

### What it’s not

It’s not a function or a type of function, it’s something that knows how to apply a function which knows nothing about the structure of the context within the functor.

### Examples

The functor may be a collection type itself such as a list, dictionary etc – as long as it has an “fmap” method that can apply a function to all the items within the collection.

It may also be something that adds some functionality around a value. The canonical example of this is the Haskell/elm “Maybe” type. This is something similar to C#’s nullable type, though more useable.

### Show me in C#

Functor<TA>.Fmap(MyAtoBFunction()) -> Functor<TB>

### Deep dive

### Get the picture

## Applicative

### What’s that

### Explain the name

### What it’s not

### Examples

### Show me in C#

### Deep dive

### Get the picture

## Monad

### What’s that

### Explain the name

### What it’s not

### Examples

### Show me in C#

### Deep dive

### Get the picture

|  |  |
| --- | --- |
| [1] | [Online]. Available: https://math.stackexchange.com/questions/156952/why-the-terminology-monoid. |