About the course

Writing correct software is a challenge.

The first and foremost source of complexity is of course the ability of a programmer to interpret a real-world phenomenon and translate it into code. Some of the difficulty lies indeed in the translation from human language to a programming language: human language is less precise and makes heavier use of an implied context, whereas computer languages are much more precise (*nothing* can be left to deduction) and moreover have only a basic mathematical context to build upon.

In this course, we will focus on patterns that guide us towards the construction of well-defined software. Our ambition is to reach the point where we write code which, thanks to type safety, composition, and referential transparency, **just works**. We hope to show that the usual frantic cycle of writing code, testing, despairing is not all there is to software development, and that programming can actually be a source of reliable reasoning which produces quality results without much trial and error.

The issue of correctness

To make matters worse, programming is often done with an *imperative mindset* and *without thinking much about types*. The implications of this reach much farther than we are often led to believe. Let us explore these two points in depth.

Extraneous imperative thoughts

Imperative programming focuses on defining **how things are computed**. For example, suppose we were writing a program which, given a number n, must compute a string of n asterisks.

For example, given `n=3`, we would expect `***` as result.

A typical imperative implementation would then be:

```
def f(n):
  s = ""
  while n > 0:"
    s = s + "*"
    n = n - 1
```

There is nothing particularly wrong with this implementation, but it does impose some cognitive overhead which we usually learn to ignore. Still, it is there, in the form of lurking hidden complexity. This hidden complexity is, in this case, embodied by the existence of variable `s`: it is nowhere to be found in the original problem description, which only talks about `n` and the resulting string.

In our effort to focus about *how we should compute*, we have introduced a new concept (`s`) which is completely extraneous to what the original problem contained. This means that the code we are writing now has to deal with more than just the problem, because the solution itself contains parts of the problem.

Data races

A source of dangers commonly practiced in imperative programming is *shared mutable state*. Sharing mutable state is very easy to accomplish, as we will now illustrate with a contrived example. Let us first define a simple data structure to wrap integer numbers. This will be the basis for our shared state:

```
class Int:
  def __init__(self):
    self.value = 0
  def incr(self):
    self.value = self.value + 1
  def decr(self):
    self.value = self.value - 1
```

Consider now a `Counter` class using an instance of `Int` in its constructor, and which enumerates even numbers:

```
class EvenCounter:
  def __init__(self):
    self.i = Int()
  def tick():
    self.i.incr()
    self.i.incr()
```

We might expect that the class defined above will always have an even value

Of course we might have more counters, for example a vanilla counter which only increments in steps of one:

```
class RegularCounter:
  def __init__(self):
    self.i = Int()
  def tick():
    self.i.incr()
```

Let us now imagine a program where two counters are initialized and then subsequently used:

```
ec = EvenCounter()
rc = RegularCounter()

... ec.tick() ...
... rc.tick() ...
```

We can always expect that the behavior of both counters will respect their implementation, and that reading their implementation is sufficient to build a mental model which *precisely tracks all facts* about how these objects behave.

Unfortunately, we can easily break this. For example, we could share the state between the two counters:

```
ec = EvenCounter()
rc = RegularCounter()

ec.i = rc.i # written by junior developer with good intentions

... # lots of code in between

ec.tick() # ec is now at 2, which is an even number as expected
rc.tick() # ec is now at 3, which is completely unexpected!

...
```

Investigation of the problem might lead us to further look into `EvenCounter`, given that it looks like the class is showing unexpected behavior. To make matters much worse, we could imagine that `ec` and `rc` are used in different threads, and as such the race condition might only present itself in some strange situation based on a combination of external factors.

In our case we have written very little code, but usually the shared state will be a bigger and more complex data structure, and there will be a lot of code, spread over many files, declaring and using this data structure. Each file may look correct and innocent, and only the sum of all parts of the program will lead to the bug manifesting itself. Testing isolated parts of the system will not help, since *the parts are correct*.

This sort of bug, often called a *data race* or a *race condition* is very difficult to diagnose, let alone fix. Moreover, the traditional imperative toolbox of locks adds to the complexity of the situation, given that locks might solve some of the issues, but also cause extra problems such as deadlocks.

Issues with uninteresting type safety

Another common source of issues in mainstream programming practice is the detailed representation of irrelevant information when it comes to types and type safety.

Types are seen by many developers as a tool for (verbosely) specifying trivial information which is not particularly valuable, but which some primitive compilers need in painful detail in order to allow us to run our programs.

An example of a typical class definition in a statically typed language could be:

```
class Person {
  private string name;
  private string surname;
  private Date birth_date;
  public Person(string name, string surname, Date birth_date) {
    this.name = name;
    this.surname = surname;
    this.birth_date = birth_date;
  }
  ...
}
```

In about ten lines of code we have said very little, but still, in a Java-like language there are no alternatives to this rite of passage before we have a chance to just use a new datatype. Moreover, some languages even require initialization to look as follows:

```
Person p = new Person("James", "Semaj", new Date(1, 1, 2001));
```

Once again we are confronted with verbosity and repetition of information: `Person` appears both left and right of `p`, which is pointless: saying it once should be enough. This is slowly evolving with the introduction of type inference, but we are still a long way from usable type systems.

A better way

Types are not uniquely meant to represent concrete information which will mirror reality. The usual definition of classes to model `Person`, `Customer`, `Car`, and other concrete entities we encounter in reality is not really the strength of types.

Types are annotations that define much more than which data and functions we can expect to find in a variable or symbol: types allow us to model all sorts of provable properties of data, and can be used to model even dynamic abstract information such as effects or exceptions.

A typical example of this would be defining a datatype to represent stateful processes. A stateful process takes as input an initial state of arbitrary type `s`, and returns as output both a new state `s` and some value of an arbitrary type `a` which is the result of the process.

We could model this as a type:

```
type Process s a = s → (a*s)
```

Where `a*s` denotes a pair containing both an `a` and an `s`. Moreover, we could model the fact that processes may fail, therefore a process will not always return `a*s`, but might also return `e`, which is an arbitrary type representing an error:

```
type FailableProcess e s a = s → (a*s + e)
```

`a*s+e` here denotes a value which might either be `a*s`, or `e`.

Notice that we are using types to model abstract notions of processes, instead of concrete notions of simple data structures.

Moreover, we could then use such type definitions to run and combine processes. For example, running two processes in sequence will produce both their results:

```
after : Process e s a * Process e s b → Process e s (a*b)
```

And much more. Depending on the programming language, it will be easier or harder to express such abstract concepts in our programs. Unfortunately, if we do not model our domain in terms of such abstract concepts, then it does not matter how far the language can take us: we will not even try.

Moving beyond

In this course we will focus on a series of fundamental properties of code which are easy to verify, in order to ensure that our libraries are well built, and that they do not suffer from the issues we discussed.

We will begin by modelling every construct we discuss by means of statically typed, "strong" definitions which a good compiler will be able to verify for us. We will then show the essential aspects of computation without sharing state, and ensure that our constructs can be composed in infinitely many new forms in order to achieve higher and higher complexity, without suddenly encountering race conditions or similar issues.

We will discuss a series of abstract constructs, known as *functors* and *monads*, and their general properties which, if present, ensure that the implementation will be reliable and composable. Moreover, we will show that these constructs can be used in practice, and indeed we will provide multiple examples of commercial, enterprise-grade libraries which are based on these constructs.

About the chosen language

The main language used in this text will be TypeScript. TypeScript combines an expressive and elegant type system, decent type-inference, higher order functions, and is a mainstream language used in many companies. Its type system is even Turing-Complete, meaning that it can express some very articulated concepts. Nevertheless, such a choice of language is quite peculiar given the topic, and indeed one might expect to encounter more academically inclined languages such as Haskell, or even Agda or Coq. The reason we will go with TypeScript is to make it immediately, unambiguously clear that the used concepts can and must be applied in practice, and must not be seen in isolation from the practice of software engineering when building actual software. Using a more research-oriented language would most certainly lead us to more elegant constructs and a fuller representation of all they stand for. Unfortunately, lack of applicability in realistic contexts would also give the false impression that there exists a gap between the world of theory and practice, which is actually not the case.

TypeScript is our attempt at harmonizing theory and practice, so that they may both show their rich interaction and mutual benefit.

The computational model we refer to is essentially based on types and functions.

Types are given as primitives, for example `int`, `float`, `string`, etc. We will later see how to extend our types with new custom types.

Given two arbitrary types a and b, we can define a function f that takes as input values of type a and which returns values of type b. We denote such a function with the suggestive notation:

```
f : a → b
```

A simple wrapper

Given the centrality of functions, let us define a wrapper datatype for functions. We will need it through the whole course.

```
type Fun<a,b> = { f:(i:a) ⇒ b }
```

Let us also define a lifting operator which takes as input a regular function and returns our (not yet, soon to be) enriched function definition:

```
let Fun = function<a,b>(f:(_:a)⇒b) : Fun<a,b> { return { f:f } }
```

At this point we can move on to some basic examples.

First examples

When presenting examples, we will mostly manipulate simple arithmetic functions in order to keep the discussion simple, yet showing how the concept are implemented. Occasionally we will move on to a more articulated case study.

Let us begin with a couple of functions to increment and double numbers:

```
let incr = Fun<number,number>(x ⇒ x + 1)
let double = Fun<number,number>(x ⇒ x * 2)
```

We could also define a function to negate a boolean value:

```
let negate= Fun<boolean,boolean>(x ⇒ !x)
```

At this point we could always define a function which "bridges the gap" between integers and booleans, for example by checking whether or not it is even:

```
let is_even = Fun<number,boolean>(x ⇒ x % 2 == 0)
```

Composition

The underpinning principle that we will try to carry around during the whole course is that of *composition*. The basic idea of composition is that, given two entities of the same sort, we can compose them into a new one.

This is not necessarily our direct practical experience in real-life: combining two tomatoes does not always yield a bigger tomato, but combining two bowls of tomato sauce does certainly require a bigger bowl. This vegetable-oriented example suggests that liquids, being flexible and having no structure, can be combined together into more of the same, whereas solid objects cannot. Abstract concepts such as the ones we manipulate in the context of programming, on the other hand, often offer us the possibility to combine. Numbers would be the first example: given two numbers, say `10` and `2`, we can combine them as follows:

```
10 + 2 = 12
10 / 2 = 5
10 * 2 = 20
```

Notice that in all cases, the composition has led us to a new number. We might conclude that numbers are quite a flexible medium, given that it allows us infinitely many ways to compose numbers while still obtaining numbers as results.

Composition also has a distinct advantage. Instead of forcing us to carry multiple (connected) concepts together, we can compose them and carry just one. This makes it easier for us to reason, because instead of having many separate objects and concepts, we now have a single one which combines the interesting essence of all of the original objects.

Throughout the whole course, we will focus on how far composition can take us: we will try to find structures which support composition in computing, and thanks to this composition we will build large architectures from smaller components with very little effort. This goal of building large architectures from smaller components is **the essence of software engineering**. Whence the name of the course.

Function composition

Given two functions, `f` and `g`, it is sometimes possible to compose them into a new function. The usual way to perform such a composition would be to create a new function which feeds its input to `f`, and the output of `f` to `g`. The output which `g` then produces should become the output of the composition.

For this to be possible, though, a condition must hold: the two functions must be "compatible". If the output of `f` is, for example, a `string`, but the input of `g` is an `int`, then the composition makes no sense: moreover, passing an argument of type `string` where an `int` is expected will produce no reasonable result, and in most programming languages this will result in a crash.

We encode this by saying that, given $f : a \to b$ and $g : b \to c$, no matter what the types a, b, and c are, we can generate a new function $(f;g) : a \to c$. We read this function as *f, then g*.

Implementing function composition:

Let us define a `then` operator which takes as input two functions and build a new one which is the composition of the two:

```
let then = function<a,b,c>(f:Fun<a,b>, g:Fun<b,c>) : Fun<a,c> {
  return Fun<a,c>(a ⇒ g.f(f.f(a)))
}
```

The composition of two functions simply creates a new function (notice the call to `Fun<a,c>`, which indeed creates a new function), which will invoke first `f`, and then `g`.

Instead of using this operator right away, let us ensure that the experience of a programmer composing functions is sufficiently smooth. We now extend the Fun type definition so that it also offers a `then` method:

```
type Fun<a,b> = { f:(i:a) ⇒ b, then:<c>(g:Fun<b,c>) ⇒ Fun<a,c> }
```

Of course, creating a new function must also define `then`. Here we use the implicit binding of `this` to our advantage:

```
let Fun = function<a,b>(f:(_:a)⇒b)) : Fun<a,b> {
  return {
    f:f,
    then:function<c>(this:Fun<a,b>, g:Fun<b,c>) : Fun<a,c> {
      return then(this,g) }
  }
}
```

Note that in TypeScript it is not necessary to specify the type of `this` when declaring a new method, but, unlike in Java or C#, it is always required to define `this` when defining its implementation We can now compose existing functions, for example by saying:

```
let incr_twice = incr.then(incr)
let double_twice = double.then(double)
let incr_then_double = incr.then(double)
let my_f = incr.then(is_even)
```

and so on.

Of course, we can now invoke any of these composed functions:

```
console.log(incr_twice.f(3))
console.log(double_twice.f(5))
```

etc.

Composition is not limited to a single step. We can further compose the result of a composition, in order to add even more steps. As long as the input and output types match as requested, then we can keep composing. For example, we could write the following compositions in more steps:

```
let f = incr.then(double.then(incr.then(is_double)))
```

Identity function

A special mention must go to a special function which exhibits a unique behavior: the identity function. The identity function is an apparently useless function which, when given a parameter, simply returns it right away without modification. This function thus does absolutely nothing.

The identity function on, for example, integers, would be defined as:

```
let id_num = Fun<number,number>(x ⇒ x)
```

Given that the identity function does not care what the input is (it just returns it right away, so it could be anything and the identity function would still work exactly the same way), we can give a similar definition for strings, arrays, and in general all imaginable data structures. Since all of these definitions are, in effect, the same, we could just define identity once and for all possible types, in the form of a generic function:

```
let id = function<a>() : Fun<a,a>(x ⇒ x)
```

This way, to *get* the identity function, we would need to invoke `id<number>()`. This would give us the identity function back for the given generic argument.

The uniqueness of the identity function is its behavior with respect to composition. Composing the identity function with any other function will produce no change whatsoever, meaning that `f.then(id())` and `id().then(f)` are exactly the same as just `f`.

The identity function is thus the *neutral element* of composition, that is it behaves similarly to 0 with respect to addition, and to 1 with respect to product$0 + x = x + 0 = x$ and $1 \times x = x \times 1 = x$, no matter what x was.

Referential transparency

The closing note of this chapter discusses an important requirement for function composition.

Let us take a short step back. Composition in general implies that the properties of the entities being composed are somehow preserved by the composition. For example, if our starting point were two bowls of tomato sauce, it would be very odd if pouring them both in a larger bowl would result in, say, the same amount of substance transmuted into milk.

Similarly, when we say that `2 + 3 = 5`, then we could conclude that somehow `5` preserves the properties necessary to define both `2` and `3`. Indeed, `+` preserves the basic properties associated with counting, so our target does indeed hold.

When composing functions, we must look for similar properties that are somehow preserved. The fundamental properties of functions is quite simple. Consider, for example, the following program:

```
x = f(3)
...
y = f(3)
```

Our intuitive expectation when reading code is *determinism*: code will behave the same as long as the _surrounding circumstances_ remain the same. This predictability would therefore lead us to conclude that `x` and `y` must be the same.

Unfortunately, this is not always the same. For example, suppose that `f` were implemented as follows:

```
let counter = 0
let f = function(x:number) ⇒ {
  counter = counter + 1
  if (counter > 1000) return -1
  else return x + 2
}
```

This implementation would be quite hard to debug: lots of calls would simply return the input plus two, but all of a sudden, and only after the program has been running long enough, it would start giving a different result.

The function is therefore non-deterministic, in that its output does not depend only on its input. Functions in which the output only depends on the input are called _pure_, and enjoy the property of _referential transparency_.

Referential transparency ensures that a function will always behave the exact same way when called with the same parameters. In other words we could also say that, with referential transparency, it is always possible to replace every occurrence of an *expression* in the program with its value without altering the result.

In order to better understand this, consider the following simple imperative program

```
def foo(x):
  print(x)
  x = x + 1
  print(x)
```

and imagine that we are calling `foo(5)`. In the first line of the function the value of x is 5. After re-assigning `x` its value will be 6. If we now try to replace `x` in the last `print` statement we would insert 6. This means that, in the first `print` the expression `x` is replaced with 5, and the very same expression in the second print is replaced with 6. This means that this program is not referentially transparent. Indeed the value that we use to replace `x` depends on the state of the program. This means that the order of the computation changes the final result of the program. Indeed if we swapped the variable assignment with the last print, we would print 5 twice and then assign 6 to variable `x`.

On the other hand, you know from previous courses that, in functional programming languages, it is always possible to replace an expression with the same value for all its occurrences in the program because its value does not depend on a specific state of the program.

Is it really relevant?

Let us consider the exact opposite of referential transparency, and build a program which is non-deterministic and unpredictable per default.

Let us suppose we have a `car` object, which can be used for driving on the highway or to be repaired by a mechanic.

We could then write the following program (in pseudo-code):

```
car = new Car( ... )

drive = new Thread(
  while True:
     car.accelerate()
)

repair = new Thread(
   car.open_bonnet()
   while car.engine_broken():
     car.repair_engine()
   car.close_bonnet()
)

drive.start()
repair.start()
```

By running such a program, we would encounter a variable number of issues: if we are very lucky, then the `repair` thread will be done before the `drive` thread actually starts. If we are very unlucky, the `drive` thread will accelerate with an

open bonnet and a mechanic trying to work on the engine. Such a situation is not only wrong in code, but would also constitute a source of life-threatening danger if attempted in real-life.

Moreover, when the threads of a program are started only when given situations are encountered (for example upon the user's request), then we would encounter yet another layer of complexity and unpredictability: only when many unfortunate circumstances align (a specific sequence of user inputs, timings, etc.) would we witness the bug, and often not on the developer's machine.

Referentially transparent architectures

A software architecture based on referential transparency will be focused on *transformations* of a state, which tracks all we know about a program, along a *pipeline*, which tracks all we want to do to the state.

The pipeline is composed by using operators such as `then`, and only when we actually start the pipeline with some initial state that computation will start.

Using `then` we will accumulate all desired operations, plugins, etc. (potentially also dynamically based on user input). The constructs composed are all referentially transparent, so we expect no strange interactions: each construct simply performs its own processing of (parts of) the state, and returns an output for the next stages of the pipeline. No invisible lines connect different parts of the pipeline, such as the sharing of the `car` in the threads we have seen above. All interactions are given by the position of elements in the pipeline, and by which parts of the state they get to see.

Referential transparency ensures that our code will behave in a consistent and predictable way, no matter the circumstances, and that the behavior of each component will behave the same, given the same input, notwithstanding other external circumstances. Testing and debugging becomes easier, and the quality of our software will start resembling the quality of other, longer established engineering disciplines such as mechanical or architectural engineering.

i

i

i

?

?

?

?

?

?

?

?

?

i

i

i

Select the **FALSE** sentence about a referentially-transparent function

- [ ] Its result only dependes on the input arguments.
- [ ] Its result does not depend on the point of the program it is invoked from
- [ ] Its calls can always be replaced by its result in any part of the program.
- [x] The result of the function might change depending on the point of the program it is invoked from.

Given the following functions:

```
let incr = Func<number, number>(x ⇒ x + 1)
let convert = Func<number, string>(x ⇒ String(x))
```

Which of the following expressions is equivalent to the result of their composition

```
incr.then(convert)
```

?

- [ ]
```
String(x + 1)
```

- [ ] The compiler throws a type error.

- [ ]
```
Func<number, string>(x ⇒ String(x))
```

- [x]
```
Fun<number, string>(x ⇒ String(x + 1))
```

Given the following functions

```
let convert = Fun<number, string>(x ⇒  String(x))
let square = Fun<number, number>(x ⇒ x * x)
```

Which expression is equivalent to their composition?

```
convert.then(square)
```

- [ ]
```
Fun<number, number>(x ⇒ x * x)
```

- [x] That composition is invalid

- [ ]
```
x * x
```

- [ ]
```
String(x * x)
```

Given the following functions:

```
incr = Fun<number, number> (x ⇒ x + 1)
double = Fun<number, number>(x ⇒ x * 2)
```

Which of the following expressions is equivalent to `incr.then(double)`?

- [x] `Fun<number, number>(x ⇒ (x + 1) * 2)`
- [ ] `x ⇒ (x + 1) * 2`
- [ ] The composition is invalid
- [ ] (x + 1) * 2

Given the following functions

```
let incr = Fun<number, number>(x ⇒ x + 1)
let square = Fun<number, number>(x ⇒ x * x)
```

which of the following expressions is equivalent to `incr.then(square).f(4)`?

- [ ]
```
Fun<number, number>(x ⇒ 25)
```

- [x] 25

- [ ]
```
Fun<number, number>(x ⇒ (x + 1) * (x + 1))
```

- [ ]
```
x ⇒ (x + 1) * (x + 1)
```

Given the following functions

```
let convert = Fun<number, string>((x: number) ⇒ String(x))
let incr = Fun<number, number>((x: number) ⇒ x + 1)
```

What is the type of the following expression?

```
incr.then(incr).then(convert).f(5)
```

- [x] string
- [ ] The expression contains a type error
- [ ] Fun<int, Func<string, int>>
- [ ] Fun<int, string, int>

Given the following functions

```
let convert = Fun<number, string>((x: number) ⇒ String(x))
let incr = Fun<number, number>((x: number) ⇒ x + 1)
```

What is the type of the following expression?

```
incr.then(convert).then
```

- [ ] The expression contains a type error
- [x] Fun<string, c> => Fun<number, c>
- [ ] string
- [ ] Fun<Fun<string, c>, Fun<number, c>>

Given the following functions

```
let convert = Fun<number, string>((x: number) ⇒ String(x))
let incr = Fun<number, number>((x: number) ⇒ x + 1)
```

What is the type of the following expression?

```
incr.then(convert).then(incr)
```

- [ ] Fun<string, number>
- [ ] number
- [x] The expression contains a type error
- [ ] Fun<number, Fun<string, number>>