@AndrzejWasowski@scholar.social
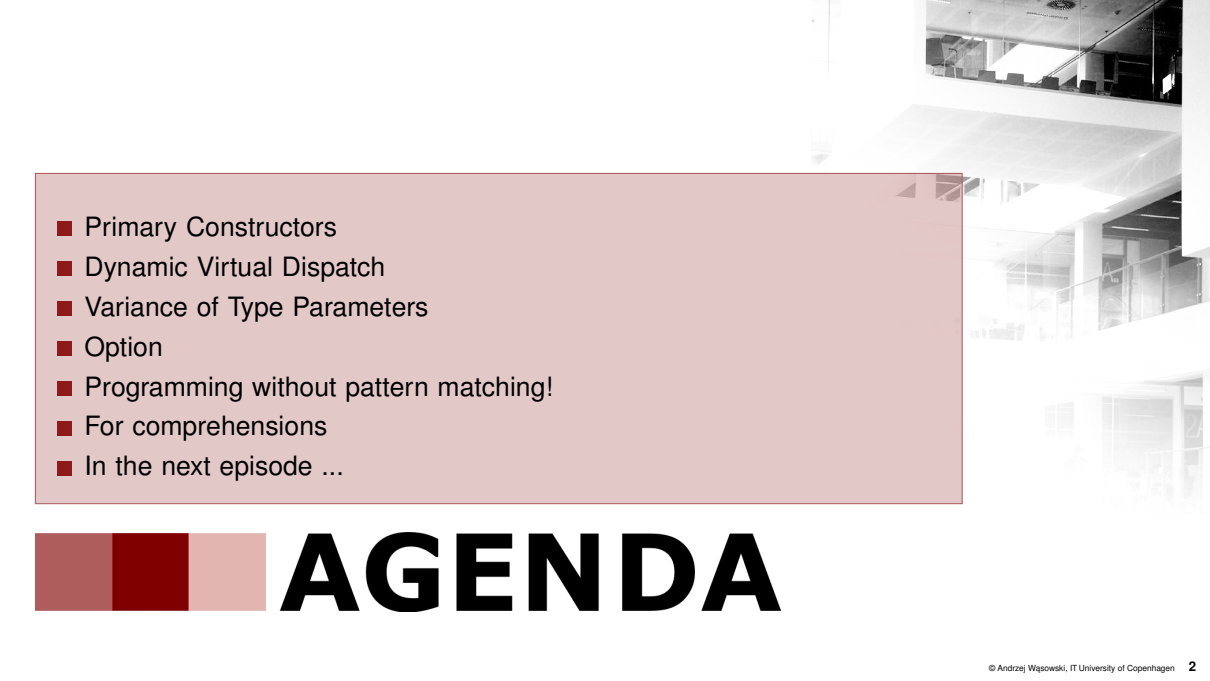**Andrzej Wąsowski**
Florian Biermann

Advanced
Programming

Partial Computations: The Option Type

- Primary Constructors
- Dynamic Virtual Dispatch
- Variance of Type Parameters
- Option
- Programming without pattern matching!
- For comprehensions
- In the next episode ...

AGENDA

The Primary Constructor

```scala
class Person(val name: String, val age: Int):
  println("Just constructed a person")
  def description =
    s"$name is $age years old"
```

```java
class Person {
  private String name;
  private int age;
  public String name() { return name; }
  public int age() { return age; }

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
    System.out.println("Just constructed a person");
  }

  public String description ()
  { return name + "is " + age + " years old"; }
}
```

- Parameters become fields
- 'val' parameters become values, 'var' become variables
- If no parameter list, primary constructor takes none
- Constructor initializes fields and executes top-level statements of the class
- Like for all functions, parameters can take default values, reducing the need for overloading
- Note: primary constructors are used with case classes
- In F# and C# 12 as well

[Horstmann 2012, Chpt. 5.7] explains the primary constructors in Scala

Mentimeter: Dynamic Dispatch in Java

1687 9725

```java
1 class Printable              { void hello() { print ("printable "); }}
2 class Triangle extends Printable{ void hello() { print ("triangle ");}}
3 class Square extends Printable  { void hello() { print ("square "); }}
4 ...
5 Square x = new Square ()
6 Printable y = new Triangle ()
7 x.hello ();
8 ((Printable)x).hello ();
9 y.hello ();
10 ((Printable)y).hello ();
```
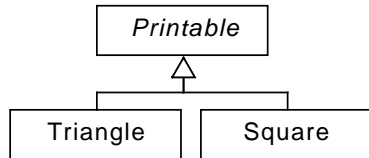

- printable printable printable printable
- square printable triangle printable
- square printable printable printable
- square square triangle triangle
- square square printable printable
- The program will crash, or fail to type check

In Scala,
like in Java*,
all instance methods are virtual
(dynamically dispatched)
unlike in C# and C++

*and Python and JavaScript

Programs as data
Higher-order functions, polymorphic types, and type inference

Niels Hallenberg

Thursday 2019-09-17

Originally by Peter Sestoft

Variance in type parameters

- Assume Student subtype of Person

```
void PrintPeople(IEnumerable<Person> ps) { ... }
```

```
IEnumerable<Student> students = ...;
PrintPeople(students);
```

> Java and C# 3 say
> NO: Ill-typed!

- C# 3 and Java:
  - A generic type is *invariant* in its parameter
  - I<Student> is *not* subtype of I<Person>
- Co-variance (co=with):
  - I<Student> is subtype of I<Person>
- Contra-variance (contra=against):
  - I<Person> is subtype of I<Student>

Co-/contra-variance is unsafe in general

- Co-variance is unsafe in general

```
List<Student> ss = new List<Student>();
List<Person> ps = ss;
ps.Add(new Person(...));
Student s0 = ss[0];
```

Wrong!

Because would allow writing Person to Student list

- Contra-variance is unsafe in general

```
List<Person> ps = ...;
List<Student> ss = ps;
Student s0 = ss[0];
```

Wrong!

Because would allow reading Student from Person list

- But:
  - co-variance OK if we *only read (output)* from list
  - contra-variance OK if we *only write (input)* to list

Co-variance in interfaces (C# 4)

- When an I<T> only produces/*outputs* T's,
  it is safe to use an I<Student>
  where an I<Person> is expected
- This is co-variance
- Co-variance is declared with the **out** modifier

```
interface IEnumerable<out T> {
  IEnumerator<T> GetEnumerator();
}
interface IEnumerator<out T> {
  T Current { get; }
}
```

- Type T can be used only in *output* position;
  e.g. not as method argument (input)

Contra-variance in interfaces (C# 4)

- When an I<T> only consumes/*inputs* T's,
  it is safe to use an I<Person>
  where an I<Student> is expected
- This is contra-variance
- Contra-variance is declared with **in** modifier

```
interface IComparer<in T> {
  int Compare(T x, T y);
}
```

OK! – a Compare method working on Persons will also work on Students.

- Type T can be used only in *input* position;
 e.g. not as method return type (output)

Variance of Type Parameters (condensed, in Scala)

- Write `A <: B` to say that `A` is a **subtype of** `B` (values of `A` fit where `B`s are expected)
- **Example**: if class `A` extends a class `B` then `A <: B`. Same for traits.
- Assume a generic type `T[B]`;
  `B` is a **covariant** parameter of `T` if for each `A <: B` we have that `T[A] <: T[B]`
  So we can use `T[A]` values where `T[B]`s are expected
- In Scala write `T[+B]` to specify that B is a covariant type parameter (so + is `out` in C#).
- Covariance common in pure programs. Scala lists are covariant (`List[+B]`).
- `A` is a **contra-variant** parameter of `T` if whenever `A <: B` we have that `T[B] <: T[A]`
- Contra-variance is needed if `A` is send in subsequently.
  In Scala, write `T[-A]` to specify contra-variance (- is `in` of C#)
- **Invariance** means that there is no automatic subtypes of generic type T;
  Invariance is default in Scala (when you omit the -/+), like in C#
- Java and C# generics **also** support variance. In F# : type constraints.
- Java has covariant arrays (problem). Scala has invariant arrays.

Why arrays shouldn't be covariant: http://stackoverflow.com/questions/6684493/why-are-arrays-invariant-but-lists-covariant
[Odersky et al. 2014, Chpt. 19] explains variance annotations in Scala in detail

The Problem with Covariance of Java Arrays

```java
1 class A {};
2 class B extends A {};
3 class C extends A {};

5 class Variance {

7   static void problem () {
8     B[] b = { new B() };
9     A[] a = b;
10    a[0] = new C();
11  }
12 };
```
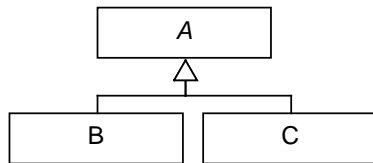
- All type checks compile
- Runtime type error in line 10. Why?
- Covariance is not always desirable.
- Covariance is good for **immutable containers** storing elements of the parameter type.

Code in `option/src/main/java/adpro/variance/Variance.java` and the corresponding test in `option/src/test/scala/adpro/variance/VarianceSpec.scala`
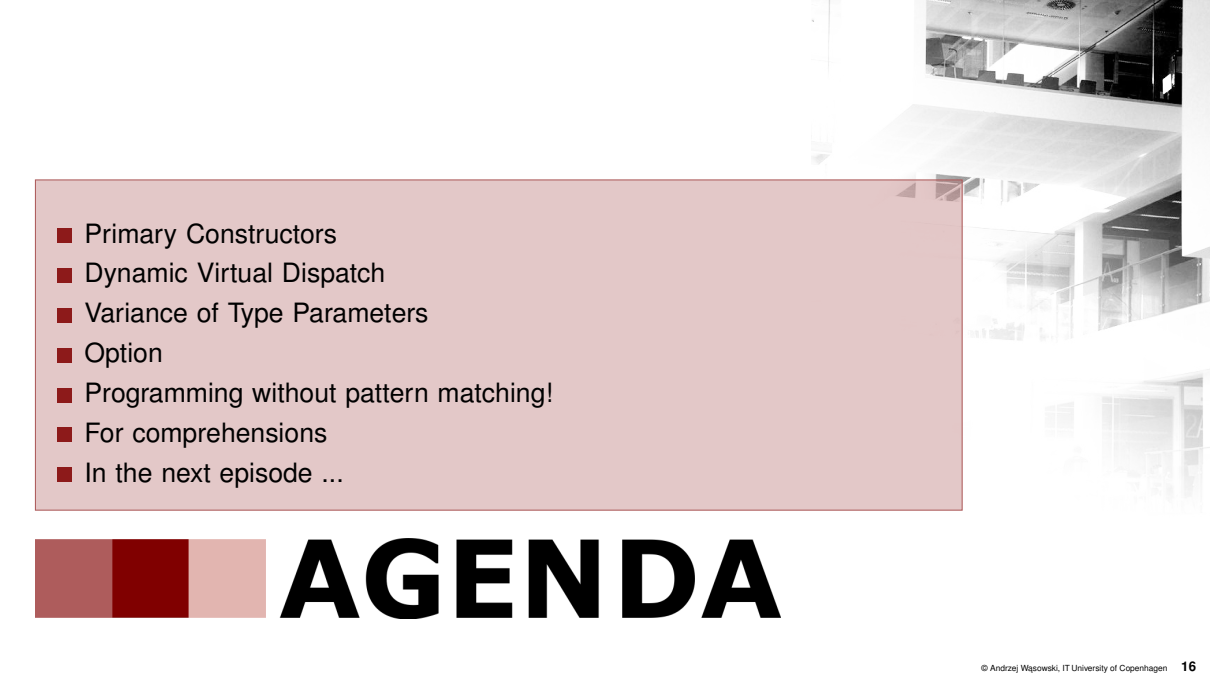
Contra-variance

```scala
1 class Cell[-T](init: T):
2   private var current = init
3   def get = current
4   def set(x: T) { current = x }

6 object Cell:
7   val c1: Cell[String] = Cell[String]("abc")
8   val c2: Cell[Any] = c1
9   c2.set(1)
10  val s: String = c1.get
```

- **If `Cell` covariant**: l. 9 would assign a string to integer (like with our Java example)
- Can do things to `Cell[Any]` that we cannot do to `Cell[String]` (assigning a number)
- Scala compiler detects this in l. 4 (`T` used in a contravariant position, on a value that will be assigned). It detects the **wrong design** of the Cell (if covariant).
- So `Cell` **contravariant**, note the `[-T]` annotation.
- Compiler flags the assignment in l. 8 (**wrong use** of the `Cell`)

Example from Odersky. Spoon. Venners. **Programming in Scala**. 2nd Edition, chapter 19

Mentimeter: Variance of Type Parameters

1687 9725

```
1 abstract class A

3 abstract class B extends A

5 // Will the following code type check if T is
6 // (a) invariant,
7 // (b) covariant,
8 // (c) contravariant ?

10 val T[A] = new T[B]
```

- Primary Constructors
- Dynamic Virtual Dispatch
- Variance of Type Parameters
- Option
- Programming without pattern matching!
- For comprehensions
- In the next episode ...

# AGENDA

A Partial Function

(and how to fix it)

```
1 def mean(xs: List[Double]): Double = xs match
2   case Nil => throw new ArithmeticException ("empty list")
3   case _   => xs.sum / xs.length
4             // _.sum and _.length are standard library methods on sequences
```

What is the domain and co-domain of the function above?

```
5 enum Option[+A]:
6   case Some(get: A)
7   case None
```
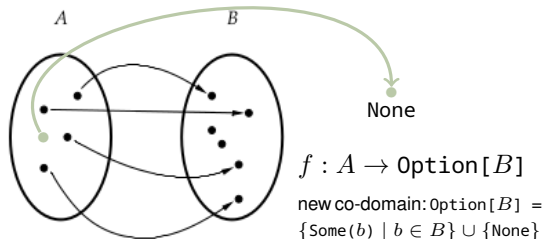
```
8 def mean (xs: List[Double]): Option[Double] =
9   xs match
10    case Nil => None
11    case _   => Some(xs.sum / xs.length)
```
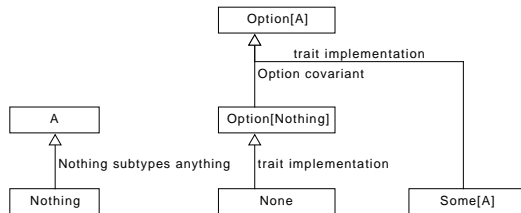

Referentially transparent (!), but we still need to figure out how to **defer** error processing
(like with exception handling)

Partial and Total Functions: Definitions

A **function** $f : A \rightarrow B$ is a binary relation on sets A and B such that for every $a \in A$ there exists precisely one such $b \in B$ that $(a, b) \in f$.

$f : A \rightarrow \texttt{Option}[B]$

new co-domain: $\texttt{Option}[B] = \{\texttt{Some}(b) \mid b \in B\} \cup \{\texttt{None}\}$

- A function f is **total**, if for each $a \in A$ there exists a $b \in B$ such that $f(a) = b$
- A function is **partial** otherwise.

- **Computations are functions**
- If an argument value for a call is **illegal** (crash, exception) then **partial function**
- Non-FP languages handle partiality with **exceptions**
- Advantage: handle the missing values **separately**, do not confuse errors with results.
- Scala has exceptions, but we don't use them in this course. **Why?**
- Need another way to handle partiality, but **keep the main advantage of exceptions**
- Idea: store the result in a special value by **growing the domain**, to contain the failure, and provide an API for handling failures non-locally.

Option in the Standard Library Methods (Examples)

How other types use Option

- `Option` is defined in the standard library
- In the course we make our own implementation for pedagogical purposes.
- `trait Map[K, +V]`
    - `def get(key: K): Option[V]`
      Optionally returns the value associated with a key

    - `def find(p: ((K, V)) =>Boolean): Option[(K, V)]`
      Finds the first element of the collection satisfying a predicate, if any
- `class List[+A]`
    - `def headOption: Option[A]` — Optionally selects the first element.
    - `def lastOption: Option[A]` — Optionally selects the last element.

Option API

What Option itself offers

```
1 enum Option[+A]:
2   def map[B](f: A => B): Option[B]
3   def flatMap[B](f: A => Option[B]): Option[B]
4   def filter(f: A => Boolean): Option[A]
5   def getOrElse[B >: A](default: => B): B
```

- Implement these functions in **homework** exercises
- Let's try using them (2 × Mentimeter 1687 9725)
    - List(1,2,3).headOption.map {_ /10.0} ?
    - List().headOption.map {_ /10.0} ?
- An interesting type parameter on getOrElse, with a **constraint** on B
    - Get a value of **any type** B from an Option[A], if B is a **super-type** of A (so implicit upcasting, as needed)
    - Another case of interesting interplay between object-oriented and functional programming type systems
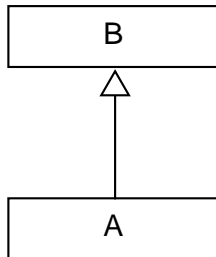
getOrElse type constraint

Localized Error Handling in the Option Monad

```
1 list.headOption
2     .map { _ / 10.0 }
3     .map { _ + 2 }
4     .flatMap { something that can fail }
5     .map { something that cannot fail }
6     ...
```

- A **failure** can occur in line 1 (or in line 4)
- The entire code is written **ignoring** a possible failure, like with exceptions
- All the computation steps are **in the Option monad** (informal for now)
- Handling the error is done **arbitrarily far** (maybe in a different function) by deciding what to do, if None is received.
- A default or error value (like -1 in C) can be injected with getOrElse:

```
6     ...
7     .getOrElse(-1)
```
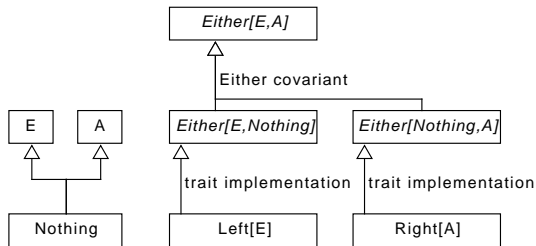
What does this compute?

```
1 List(2,3,4)
2   .headOption
3   .filter { _ % 2 == 0 }
4   .map { _ / 2 }
```

Either: Failures with diagnostic info

Recall that exceptions carry failure data objects

```
1 enum Either[+E,+A]:
2   case Left(value: E)
3   case Right(value: A)
```

- Two time parameters: left (error, E) and right (aka correct, value, A)
- Mnemonic: right is synonym for correct, which is a synonym for succesful

If you need to **grow the failure** info along the call-stack fashion, then E should be a **collection**, for instance a Either[List[Msg]], where Msg is the error message type.

For-Yield Comprehensions

```scala
1 // the original input
2 Some(4)
3 .flatMap { x1 => (if x1%2==0 then Some(x1/2) else None)
4   .flatMap { x2 => Some(x2 + 1)
5     .map { x3 => x3.toString } } }
```

```scala
1 // break lines differently
2 Some(4).flatMap { x1 =>
3  (if x1%2==0 then Some(x1/2) else None).flatMap { x2 =>
4    Some(x2 + 1).map { x3 =>
5      x3.toString } } }
```

```scala
1 // flush right
2                           Some(4).flatMap { x1 =>
3 (if x1%2==0 then Some(x1/2) else None).flatMap { x2 =>
4                      Some (x2+1).map { x3 =>
5                               x3.toString } } }
```

- For-yield comprehensions correspond to Haskell's do-notation or F# computation expressions.
- Work for any type with `map` and `flatMap`
- Other functions (like filter) also integrated
- Not to be confused with other uses of `for` in Scala (mostly impure loops iterating over collections)

```scala
1 for
2   x1 <- Some(4)
3   x2 <- if x1%2==0 then Some(x1/2) else None
4   x3 <- Some(x2+1)
5 yield x3.toString
```

https://docs.scala-lang.org/scala3/book/control-structures.html#for-expressions

List is also monadic

For-comprehensions work on lists, too

```scala
1 for
2   x <- List(3, 4, 5)
3   incremented = x + 1
4   duplicated <- List(incremented, incremented)
5                 if incremented % 2 == 0
6 yield duplicated // map identity
```

- **Mentimeter 1687 9725:** What is the result of the above?
- **Exercise:** Rewrite the above code using map and flatMap.
- '<-' translates to `flatMap`
- '=' translates to `map`
- `if` translates to `filter`
- 'yield' translates to `map`

F# uses Computation Expressions

```
1 // F#
2 seq {
3   for x in 1..10 do
4     yield x                      // map
5     yield! seq { for i in 1..x -> i} // flatMap
6 }
```

- Exercise: translate this to Scala using for-yield and using just map/flatmap?
- Key difference: implemented **explicitly**,
- Scala for–yield is **general**, for any **user defined type** that supports monadic API (flatMap and map, and ...)
- **Generators** are a similar construct in Python; Also the **with** blocks resemble monadic progamming (but are not strictly such)

Source of the example: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/sequences, where you will also find more about F# sequences.

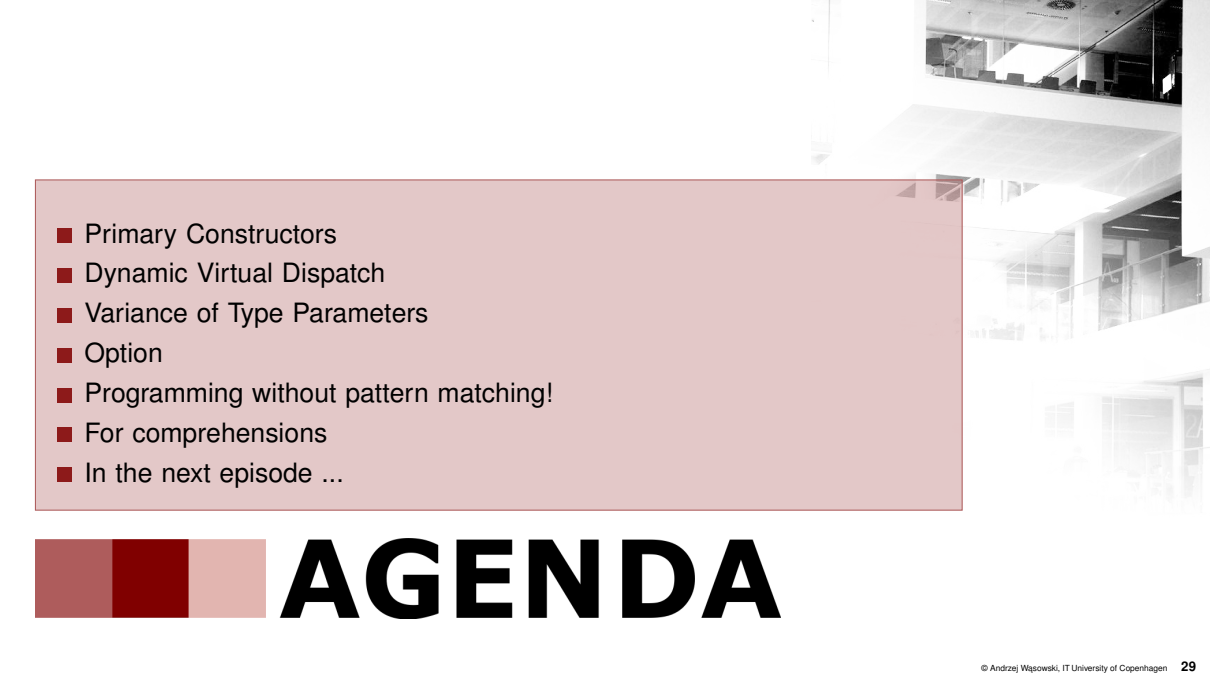# Our For-Yield Example in F#

```
1 type OptionBuilder() =
2     member _.Yield(x : 'a) : 'a option =
3         Some x

5     member _.Bind(m : 'a option, f : 'a -> 'b option) : 'b option =
6         Option.bind f m

8 let option = OptionBuilder ()

10 option {
11     let! x1 = Some 4
12     let! x2 = if x1 % 2 = 0 then Some (x1 / 2) else None
13     let! x3 = Some (x2 + 1)
14     yield x3.ToString()
15 }
```

Implemented **explicitly**! Not inferred from type as in Scala.

Blogs

- A blog about a related type in Kotlin `Result`. The author adds `flatMap` to the standard library as an extension, as it was needed, but missing. `https://bijukunjummen.medium.com/kotlin-result-type-for-functional-exception-handling-da7e8ba5490`

- Primary Constructors
- Dynamic Virtual Dispatch
- Variance of Type Parameters
- Option
- Programming without pattern matching!
- For comprehensions
- In the next episode ...

AGENDA

In the next episode . . .

- We implement a **lazy list library**
- A very **nice week**, **beautiful** ideas, **simple** but **powerful** API
- We learn **call-by-name** and **laziness**
- **Happy reading! and See you next week!**