# Software development with Scala

Vladyslav Pekker

Hochschule Konstanz
University of Applied Sciences

# Part I

# Introducing Scala

# Outline

1 Scalable Language

2 Syntax

# Scalable Language

Scala stands for "scalable language".
Scala translates from Latin to English as "stairs".
Scala's logo shows a circular staircase.

# Scalable Language

Scala is scalable in a sense of:

- Syntax: growing with the demands of its users
- FP: hacking small scripts
- OO: developing enterprise applications

# Growing the Language

If you were in need of a `type` of `Polynomial` in Java you would write it yourself and end up with something like this:

**Java**

```
Polynomial sum = firstPolynomial.plus(secondPolynomial)
```

In Scala the code would look like this:

**Scala**

```
val sum = firstPolynomial + secondPolynomial
```

Your library would feel as if it was in the language from the beginning.

# Small Scripts

## Run scripts on windows (Paste this into a *.bat* or *.cmd* file)

```
::#!
@echo off
call scala \%0 \%*
goto: eof
::!#

// The script starts here
```

## Run scripts on unix

```
#!/bin/sh
exec scala "$0" "$@"
!#

// The script starts here
```

# Small Scripts

## Get title of the Scala page on wiki

```
def title = xmlTitle.text
def xmlTitle = scala.xml.XML loadString stringTitle.get
def stringTitle = lines find { _ startsWith "<title>" }
def lines = io.Source.fromURL(url).getLines
def url = wiki + scalaLang
def wiki = "http://en.wikipedia.org/wiki/"
def scalaLang = "Scala_(programming_language)"

println(title) // Scripts must end with an expression
```

# Enterprise applications

# Outline

1 Scalable Language

2 Syntax

# Reading Scala

**The types are specified next to an identifier separated by a colon**

| Scala | Java |
|---|---|
| `x: Int` | `int x` |
| `getBeerPrice(): Int` | `int getBeerPrice()` |
| `setBeerPrice(price: Int): Unit` | `void setBeerPrice(int price)` |

# Reading Scala

## The types are specified next to an identifier separated by a colon

| Scala | Java |
|---|---|
| `x: Int` | `int x` |
| `getBeerPrice(): Int` | `int getBeerPrice()` |
| `setBeerPrice(price: Int): Unit` | `void setBeerPrice(int price)` |

## Only 3 keywords to remember and you are ready to go

- `var` declares a variable
- `val` declares a value (immutable variable)
- `def` defines a function

# Reading Scala

## First Acquaintance

```
scala> val pi: Double = 3.14;
pi: Double = 3.14 // interpreter response

scala> def calculateCircumference(r: Int): Double = {
     |   return 2 * pi * r;
     | }
calculateCircumference: (r: Int)Double // interpreter response

scala> var circ: Double = calculateCircumference(10);
circ: Double = 62.800000000000004 // interpreter response
```

# Writing Scala

## Second opinion

```
val pi = 3.14
def calculateCircumference(r: Int) = 2 * pi * r
var circ = calculateCircumference(10)
```

- semicolons are inferred
- most types are inferred
- last expression is implicitly returned
- single expression in the body means no need for curly braces

# Type Systems

## What is a type system?

# Type Systems

## What is a type system?

Type system = Proof system.
The type checker verifies proofs.

# Type Systems

## What is a type system?

Type system = Proof system.
The type checker verifies proofs.

## What does "static typing" mean?

# Type Systems

## What is a type system?

Type system = Proof system.
The type checker verifies proofs.

## What does "static typing" mean?

The types are verified at compile time.

# Type Systems

## What is a type system?

Type system = Proof system.
The type checker verifies proofs.

## What does "static typing" mean?

The types are verified at compile time.

## What does "dynamic typing" mean?

# Type Systems

## What is a type system?

Type system = Proof system.
The type checker verifies proofs.

## What does "static typing" mean?

The types are verified at compile time.

## What does "dynamic typing" mean?

The types are verified at runtime.

# Two Principles of Type Theory

**Progress**

If a term is well-typed it either evaluates or is already a value.

# Two Principles of Type Theory

**Progress**

If a term is well-typed it either evaluates or is already a value.

Progress is the connection between the static typing and the runtime.

# Two Principles of Type Theory

**Progress**

If a term is well-typed it either evaluates or is already a value.

Progress is the connection between the static typing and the runtime.

**Preservation**

If a term is well-typed and it evaluates, the result has the same type.

# Two Principles of Type Theory

**Progress**

If a term is well-typed it either evaluates or is already a value.

Progress is the connection between the static typing and the runtime.

**Preservation**

If a term is well-typed and it evaluates, the result has the same type.

Types never change!

# Two Principles of Type Theory

## Example

| `name.length()` | expression |
|---|---|
| Has type: `Int` | preservation |
| Is value? `false` | progress |

# Two Principles of Type Theory

## Example

| `name.length()` | expression |
|---|---|
| Has type: `Int` | preservation |
| Is value? `false` | progress |
| `"Vlad".length()` | expression |
| Has type: `Int` | preservation |
| Is value? `false` | progress |

# Two Principles of Type Theory

## Example

| | |
|---|---|
| `name.length()` | expression |
| Has type: `Int`<br>Is value? `false` | preservation<br>progress |
| `"Vlad".length()` | expression |
| Has type: `Int`<br>Is value? `false` | preservation<br>progress |
| `4` | literal |
| Has type: `Int`<br>Is value? `true` | preservation<br>progress |

# Type Inference

## Where do types come from?

Nothing in the type theory forces the typing to be explicit! Types come from the compiler and only the compiler cares about them. Programmers care only about the runtime.

# Type Inference

## Where do types come from?

Nothing in the type theory forces the typing to be explicit! Types come from the compiler and only the compiler cares about them. Programmers care only about the runtime.

## What is type inference?

Type inference is the act of type reconstruction in order to verify, that your program is well-typed. Type reconstruction is compiler's job, not yours!

# Type Inference

**What is *local* type inference?**

Type inference narrowed down to small chunks of code. E.g. declarations.

# Type Inference

## What is *local* type inference?

Type inference narrowed down to small chunks of code. E.g. declarations.

## Local type inference in Scala

```
def foo() = {
   val name = "Vlad"
   val length = name.length()
   println(length)
}
```

# Type Inference

## Local type inference in Scala

```
def add5(x) = x + 5
```

# Type Inference

## Local type inference in Scala

```
def add5(x) = x + 5
```

`x` is a formal parameter. There is no right hand side to the declaration. It is virtually impossible to examine the whole code base for the callers of `add5` in order to infer the type of `x`.

# Type Inference

## Local type inference in Scala

```
def add5(x) = x + 5
```

`x` is a formal parameter. There is no right hand side to the declaration. It is virtually impossible to examine the whole code base for the callers of `add5` in order to infer the type of `x`.

## Local type inference in Scala

```
def add5(x: Int) = x + 5
```

# Type Inference

## Local type inference in Scala

```
def factorial(x: Int) = {
   if (x <= 1) 1
   else x * factorial(x - 1)
}
```

# Type Inference

## Local type inference in Scala

```
def factorial(x: Int) = {
   if (x <= 1) 1
   else x * factorial(x - 1)
}
```

## Local type inference in Scala

```
def factorial(x: Int): Int = {
   if (x <= 1) 1
   else x * factorial(x - 1)
}
```

# Type Inference

JVM has its own type system.
Scala's type system has to be conform to JVM's type system.
Scala has local type inference.
Scala cannot have global type inference, because of the way JVM's type system works. (JVM has a nominal type system. A *structural* type system is required for global type inference.)

# Immutability

## What does "immutable" stand for?

Immutability is just another word for preservation. Immutability can be applied in different contexts.

# Immutability

## What does "immutable" stand for?

Immutability is just another word for preservation. Immutability can be applied in different contexts.

```
val name = "Vlad"
```

# Immutability

## What does "immutable" stand for?

Immutability is just another word for preservation. Immutability can be applied in different contexts.

```
val name = "Vlad"
```

## Contexts of immutability

The type `String` is immutable.

- `name` will always point to a `String`
- `"Vlad"` will always be a `String`

# Immutability

## What does "immutable" stand for?

Immutability is just another word for preservation. Immutability can be applied in different contexts.

```
val name = "Vlad"
```

## Contexts of immutability

The type `String` is immutable.

- `name` will always point to a `String`
- `"Vlad"` will always be a `String`

The value of the reference is immutable
(`name` will always point to `"Vlad"`)

# Immutability

## What does "immutable" stand for?

Immutability is just another word for preservation. Immutability can be applied in different contexts.

```
val name = "Vlad"
```

## Contexts of immutability

The type `String` is immutable.

- `name` will always point to a `String`
- `"Vlad"` will always be a `String`

The value of the reference is immutable
(`name` will always point to `"Vlad"`)
The value of the object is immutable
(`"Vlad"` will always be `"Vlad"`)

# Mutability

```
var person = new Person("Vlad")
```

# Mutability

```
var person = new Person("Vlad")
```

## Contexts of mutability

The type `Person` is still immutable

- `person` will always point to a `Person`
- `Person("Vlad")` will always be a `Person`

# Mutability

```
var person = new Person("Vlad")
```

## Contexts of mutability

The type `Person` is still immutable

- `person` will always point to a `Person`
- `Person("Vlad")` will always be a `Person`

The value of the reference is mutable
(`person` might point to another person someday)

# Mutability

```
var person = new Person("Vlad")
```

## Contexts of mutability

The type `Person` is still immutable

- `person` will always point to a `Person`
- `Person("Vlad")` will always be a `Person`

The value of the reference is mutable
(`person` might point to another person someday)
The value of the object is mutable
(`Person("Vlad")` might change internally)

# Mutability

```
var person = new Person("Vlad")
```

## Contexts of mutability

The type `Person` is still immutable

- `person` will always point to a `Person`
- `Person("Vlad")` will always be a `Person`

The value of the reference is mutable
(`person` might point to another person someday)
The value of the object is mutable
(`Person("Vlad")` might change internally)
The type of the object might change!
(`Person("Vlad")` might be casted to `Parent("Vlad")`)

# Immutability vs. Mutability

Avoid mutability at all cost, but do not demonize it!
For now this means only one thing: Try to use `val` instead of `var`.

# “Everything” is an expression

# What is not an expression?