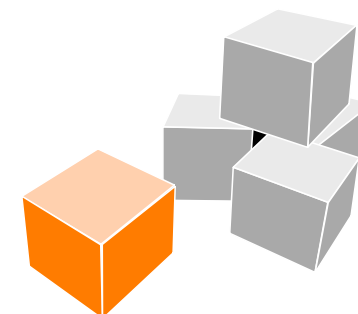


# Concepts of Programming Languages

## This Course in a Nutshell

Prof. Dr. Guido Salvaneschi



***Software  
Technology  
Group***

*TU Darmstadt | FB Informatik*

# Outline

- (Not so) few organisational data
- Why study programming languages?
- What exactly will we study and how?
- Excursus: Basics of Scala for Java Programmers

## **ADMINISTRIVIA**

# Contact

- Instructor: Guido Salvaneschi
  - salvaneschi@cs.tu-darmstadt.de
- Teaching assistants
  - Mirko Köhler: koehler@cs.tu-darmstadt.de
  - Aditya Oak: oak@cs.tu-darmstadt.de



- TAs are much faster in answering emails ;)
- Good news: you probably don't need emails at all (see policy on the forum)

# Administrative

- This information is also available at:
  - [http://www.stg.tu-darmstadt.de/teaching/courses/ws\\_2017\\_\\_18/copl\\_1/copl\\_ss14\\_index\\_2.en.jsp](http://www.stg.tu-darmstadt.de/teaching/courses/ws_2017__18/copl_1/copl_ss14_index_2.en.jsp)



- SVN: course content (slides, code, etc.).
  - <https://repository.st.informatik.tu-darmstadt.de/teaching/copl/2018/>
  - user and password: RBG login



# Administrative

- SVN: HTML version
  - <https://repository.st.informatik.tu-darmstadt.de/teaching/copl/2018/index.html>
  - **THE place for all announcements, course content, etc.**
  - **Please keep an eye on it!**

## Concepts of Programming Languages

### Announcements

- 2017-09-22
  - Please note, that the Lecture is on **Tuesday** and the only exercise is on **Thursday**
  - This year the details of how to work on and hand in the exercises will change compared to last year, details follow.

### Lectures

- 2017-10-17 Lecture 01

### Exercises

- [Exercise Folder \(with all tasks and solutions\)](#)
  - You can checkout the **Exercise** folder with SVN and use it in your favourite IDE. IntelliJ can load the **build.sbt** file. Eclipse requires you to execute the command **sbt eclipse** in the Exercise folder. See <http://www.scala-sbt.org/index.html> for instructions on installing **sbt**.

### Organization

#### Lecture

Tuesday 11:30–13:20 in S202/C110 Starting 2016-10-17

#### Exercise

# Administrative

- Forum:
  - <http://d120.de/forum/viewforum.php?f=300>
  - Subscribe to the RSS feed to get notifications!
  - Basically everything that is not strictly personal
    - **Organisational** questions
    - **Technical** questions
  - Don't send an email
    - The answer in the forum can be beneficial to others
  - You can often get an answer from a colleague
    - Please keep it a friendly place!

# Administrative

- Graded assignments (homework)
  - Will be on the Coursera platform.
  - All details in the first exercise next Thursday (and on the website)
  - Stay tuned.
- Exercises (from October 18th)
  - Thursday 11:40–13:20 in S202/C120
  - These are in addition to the Coursera stuff.  
A TA will be in the room to help you with these
  - The exercise this week is a quick intro to the Coursera platform.
  - The schedule lecture/exercise can change depending on the week



# Administrative

- 16 Oct - Lecture
- 18 Oct - Exercise - Setting up Scala & homework platform.
- 23 Oct - Lecture on Scala
- 25 Oct - Exercise 01
- 30 Oct - Lecture
- 1 Nov - Lecture
- 6 Nov Exercise 02
- 8 Nov Exercise 03
- 13 Nov - Lecture
- 15 Nov - Exercise 04
- ...

Two lectures  
in a row

# Administrative

- Written exam
- Date: Mo, 11. Mar. 2019 13:00-15:00
  - Implement/extend interpreters
  - “Theory”
  - Understanding programs under different semantics
  - More about this towards the end of the course...
- Bonus for graded assignments:  
You can improve your grade by up to 0.7  
You cannot pass the exam because of the bonus
- Details on the course website

# Extra: Other Courses on PL

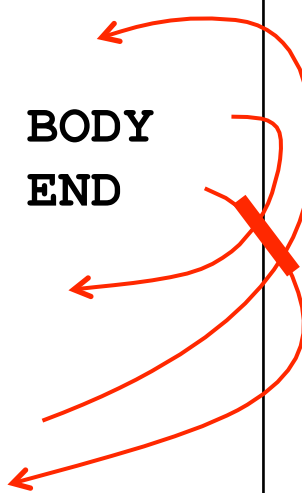
- Design and Implementation of Modern Programming Languages (Seminar)
  - Scientific reading and writing on PL topics
- Implementation of Modern Programming languages (Project)
  - Project on PL topics
  - Compilers, VMs, language design, patterns, ...
- Kick-off: Thursday, October 18 2018, 15:00 in S2|02 A213

## **WHY PROGRAMMING LANGUAGES?**

# Programming Languages (PLs) Are ...

- ... **a powerful instrument to control complexity in design**
  - A means for instructing a computer to perform tasks
  - Frameworks within which we organize ideas about problem domains
- “Better” languages increase our ability to deal with complex problems by providing ways to describe things more directly.

```
    i = 1
TEST: if i < 4
      then goto BODY
      else goto END
BODY: print(i)
      i = i + 1
      goto TEST
END:
```



```
i = 1
while ( i < 4 ) {
    print(i)
    i = i + 1
}
```

# The Goal of this Course

Provide insight into the **core concepts** of PLs

- What do concepts mean
  - how do they work
  - how can we implement them
  - how do they interact with each other
- Which concepts should we use
  - E.g., can we build a full language from a minimal “core”
- Concepts are the building blocks of new languages

# Why Study PL Design?

Many reasons

- Better PLs enable more **productive** software development
- Insights into PL design applicable to **API** design
- Many computational domains can be considered PLs
  - Example: solving polynomial equations
  - Specify computation in some notation
  - Implementation embodies computation rules for polynomials
  - Report result in some notation
- **Modelling** techniques for computer systems and technologies for development of new programming languages overlap each other.

# What Language Aspects Do We Study?

Every programming language consists of four elements:

1. Syntax: structure of programs
2. Semantics: meaning associated to syntax
3. Libraries: reusable computations
4. Idioms used by programmers of that language

Can you make examples for each of those?

Which of these elements is the most important for the study of PLs?



# What Language Aspects Do We Study?

*Which of the code fragments is most similar:*

<code>a[25] + 5</code>	(Java)
<code>(+ (vector-ref a 25) 5)</code>	(Scheme)
<code>a[25] + 5</code>	(C)

But semantics can be different.  
Think about writing out of the array boundaries...

# How to Study Semantics?

- **Informal specs** and language surveys
- **Formal specs**: operational, denotational, axiomatic semantics, ... if at all, will only take a brief look in this course
- **Interpreter semantics** (cousin of operational semantics)
  - Explain a language by writing an interpreter for it
  - By telling the computer what it means to execute a concept we thoroughly understand it ourselves
- We'll interleave **language surveys** and **interpreters**
  - Inductive versus deductive learning

# Interpreter Semantics

- An interpreter that defines a language cannot be “wrong”. It **defines** the meaning
- Assigning ‘+’ another meaning than addition is not wrong, at most it is unconventional
  - Can you make an example?
- Only, when given another specification of a language, one can speak about the correctness of the interpreter relative to the specification
  - Do two virtual machines implement the same “meaning”?
  - Think of Javascript in different browsers

# Some Interpreter Terminology

- **Host language** (or **base language**):  
the language, in which the interpreter is implemented.  
In our case: Scala
- **Interpreted language** (or **object language**):  
the language that the interpreter evaluates
  - We assume that we already understand the host language
- **meta-interpreter** or **meta-circular interpreter**:  
host language == interpreted language

# Note...

Although we will write and study interpreters for particular languages that we define ...

they have the fundamental structure of interpreters for  
**any expression-oriented language that can be used to write  
programs on a sequential machine.**

# Tentative Overview of the Course Topics

- Arithmetic expressions
- First-order and first-class functions
- Lazy evaluation
- Recursion
- State
- Continuations
- OO concepts
- Garbage collection
- Reactive Programming?
- Formal specification of semantics
- Type systems
- ...

# Administrative?

- How to pass the exam?
- **Read** the course material
- while(true){  
    **Understand** the course material  
} // How is interpreter X designed to support feature Y ?
- **Play** with interpreters!
  - Most exercises at the exam are about adding a feature to an interpreter

# About this course

- What this course **IS**
  - About **concepts** in PL
  - About interpreters
  - About deeply understanding PL abstractions
  - About bridging with more formal approaches
- What this course **IS NOT**
  - A course on Scala
  - A course on a specific PL
  - A course on PL theory (operational semantics, type systems, ...)



**BUT... IS THIS  
“IMPORTANT”?**

# More about semantics

```
[5, 12, 9, 2, 18, 1, 25].sort();
```

→ [1, 12, 18, 2, 25, 5, 9]

```
[5, 12, 9, 2, 18, 1, 25].sort(function(a, b){  
    return a - b;  
});
```

# More about semantics

```
var a = 1;
```

```
function four() {  
  if (true) {  
    var a = 4;  
  }  
  alert(a);  
}
```

```
four()
```

# JavaScript

When Netscape hired Brendan Eich in April 1995, he was told that he had 10 days to create and produce a working prototype of a programming language that would run in Netscape's browser. Back then, the pace of Web innovation was furious, with **Microsoft suddenly making the Internet the focus of its Windows 95 operating system release in response to Netscape's emerging browser and server products.**

Netscape got so much attention from Microsoft at that time because **Netscape considered the Web browser and server as a new form of a distributed OS rather than just a single application.** Once Mosaic debuted in 1993, the Web became portable across Windows, Macintosh, and Unix and gave software developers the hope that they could develop applications for all of these environments.

But **HTML wasn't sufficient by itself** to define a new application development environment or OS. To cement the portable OS concept, the Web (and Netscape) needed portable programming languages.

**Sun's Java language seemed to be the solution for portable heavyweight applications.** A compiled language that produced byte code and ran in the Java virtual machine, Java supported rich object-oriented patterns adopted from C++ and seemed likely to be able to achieve performance similar to C++ and C. Java was the Web's answer to Microsoft's Visual C++.

Knowing that Java was a rich, complex, compiled language aimed at professional programmers, Netscape and others also wanted a lightweight interpreted language to complement Java. This language would need to appeal to nonprofessional programmers much like Microsoft's Visual Basic and interpretable for easy embedding in webpages.

Charles Severance, "**JavaScript: Designing a Language in 10 Days**", Computer vol. 45 no. undefined, p. 7-8, Feb., 2012

# ECMAScript

- The ECMAScript specification is a standardized specification of a scripting language developed by Brendan Eich of Netscape;
- Initially it was named Mocha, later LiveScript, and finally JavaScript.
- In December 1995, Sun Microsystems and Netscape announced JavaScript in a press release.
- In March 1996, Netscape Navigator 2.0 was released, featuring support for JavaScript.

# JavaScript: How did it go?



# JavaScript

- Demo - video
- Talk by Gary Bernhardt from CodeMash 2012
- <https://www.youtube.com/watch?v=AU2Rhq5eWa4>



**WAT**





# JavaScript

- The next 20 years:
- Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. **An Operational Semantics for JavaScript**. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS '08)*, G. Ramalingam (Ed.). Springer-Verlag, Berlin, Heidelberg, 307-325. DOI=[http://dx.doi.org/10.1007/978-3-540-89330-1\\_22](http://dx.doi.org/10.1007/978-3-540-89330-1_22)
- Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. **The essence of javascript**. In *Proceedings of the 24th European conference on Object-oriented programming (ECOOP'10)*, Theo D'Hondt (Ed.). Springer-Verlag, Berlin, Heidelberg, 126-150.
- Daejun Park, Andrei Stefanescu, and Grigore Roşu. 2015. **KJS: a complete formal semantics of JavaScript**. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 346-356. DOI: <http://dx.doi.org/10.1145/2737924.2737991>
- ...

## A SHORT INTRO TO SCALA

# Driving Forces for Scala's Design

Goal: Better PL support for component software

Two hypotheses:

1. PLs for component software should be **scalable**
  - The same concepts describe small and large parts
  - Rather than adding lots of primitives, focus on abstraction, composition, and decomposition
2. **Unification** of OO and functional programming can provide scalable support for components

Adoption is key for testing the hypotheses ==>  
Scala interoperates with Java

# Scala's Adoption

Linked in.

foursquare®



yammer™

SIEMENS



Atlassian

tomTom®

xerox



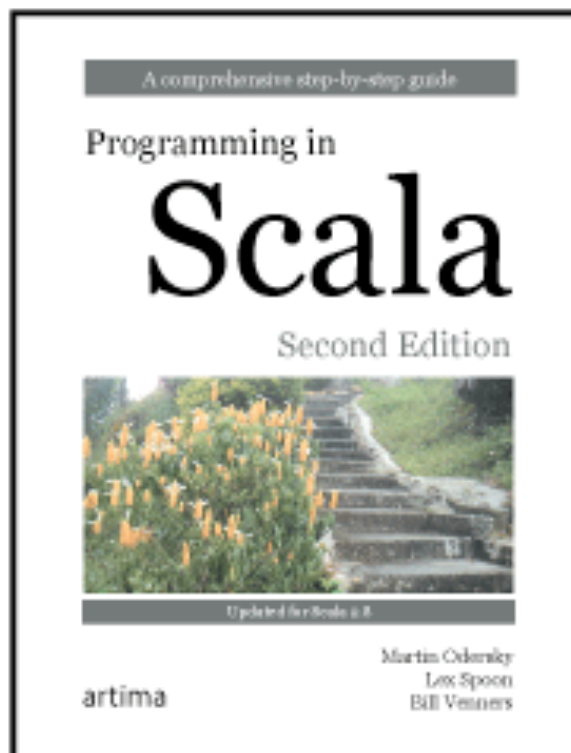
UBS

theguardian



# The Scala Language

- We will present only as much Scala as needed.
- For learning Scala, refer to books and online courses:  
[https:// www.coursera.org/course/progfun](https://www.coursera.org/course/progfun)



More books:

<http://www.scala-lang.org/node/959>

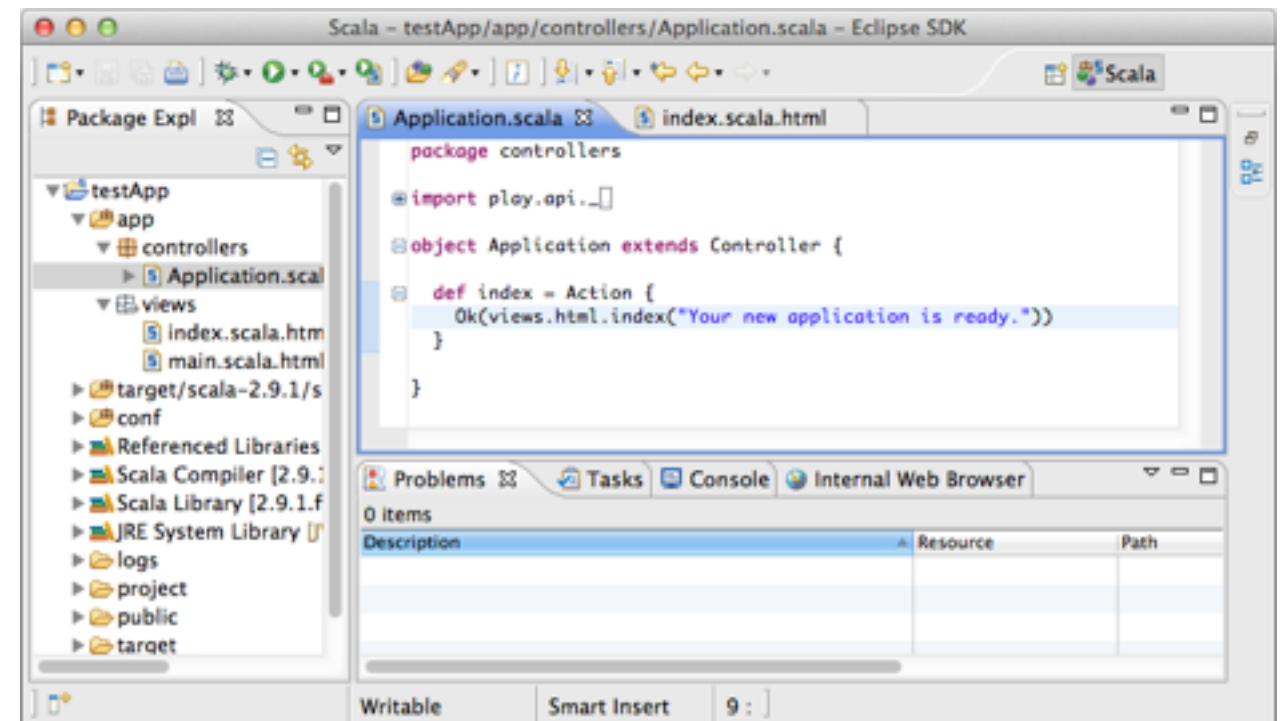
Quick entrance for Java Programmers:

<http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html>

# Start Right Away!

You will need both:

- SBT
  - [www.scala-sbt.org](http://www.scala-sbt.org)
  - Command line tool
  - Create projects
  - Upload projects to Coursera
- Scala IDE
  - Eclipse: [scala-ide.org](http://scala-ide.org)
  - NetBeans



# Some Features of Scala

- Everything is an object
- Classes
- Case classes and pattern matching

# Everything is an Object

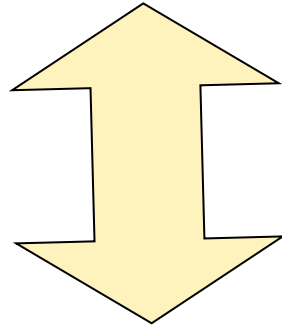
- Numbers are objects
- Functions are objects



# Scala Numbers are Objects

1 + 2 \* 3 / x

**Infix operator notation for method calls**



+, \*, ... are identifiers referring to methods of number objects.

(1).+((((2).\*(3))./(x)))

# Operator Notation

Operator notation is not limited to methods like `+` that look like operators in other languages.

**You can use any method in infix operator notation.**

```
val str = "hello"  
str indexOf 'o'
```

**Prefix and postfix operator notations are supported too.**

```
scala> -7  
res12: Int = -7  
  
scala> (7).unary_-  
res13: Int = -7
```

```
scala> val s = "Hello World!"  
s: String = Hello World!  
  
scala> s toLowerCase  
res16: String = hello world!  
  
scala> s.toLowerCase  
res17: String = hello world!
```

# Scala Functions are Objects

```
object Timer {  
  def oncePerSecond(callback: () => Unit) {  
    while (true) { callback(); Thread sleep 1000 }  
  }  
  def timeFlies() {  
    println("time flies like an arrow...")  
  }  
  def main(args: Array[String]) {  
    oncePerSecond(timeFlies)  
  }  
}
```

What does this do?

Explain all features not available in Java used in the example.

# Some Features of Scala

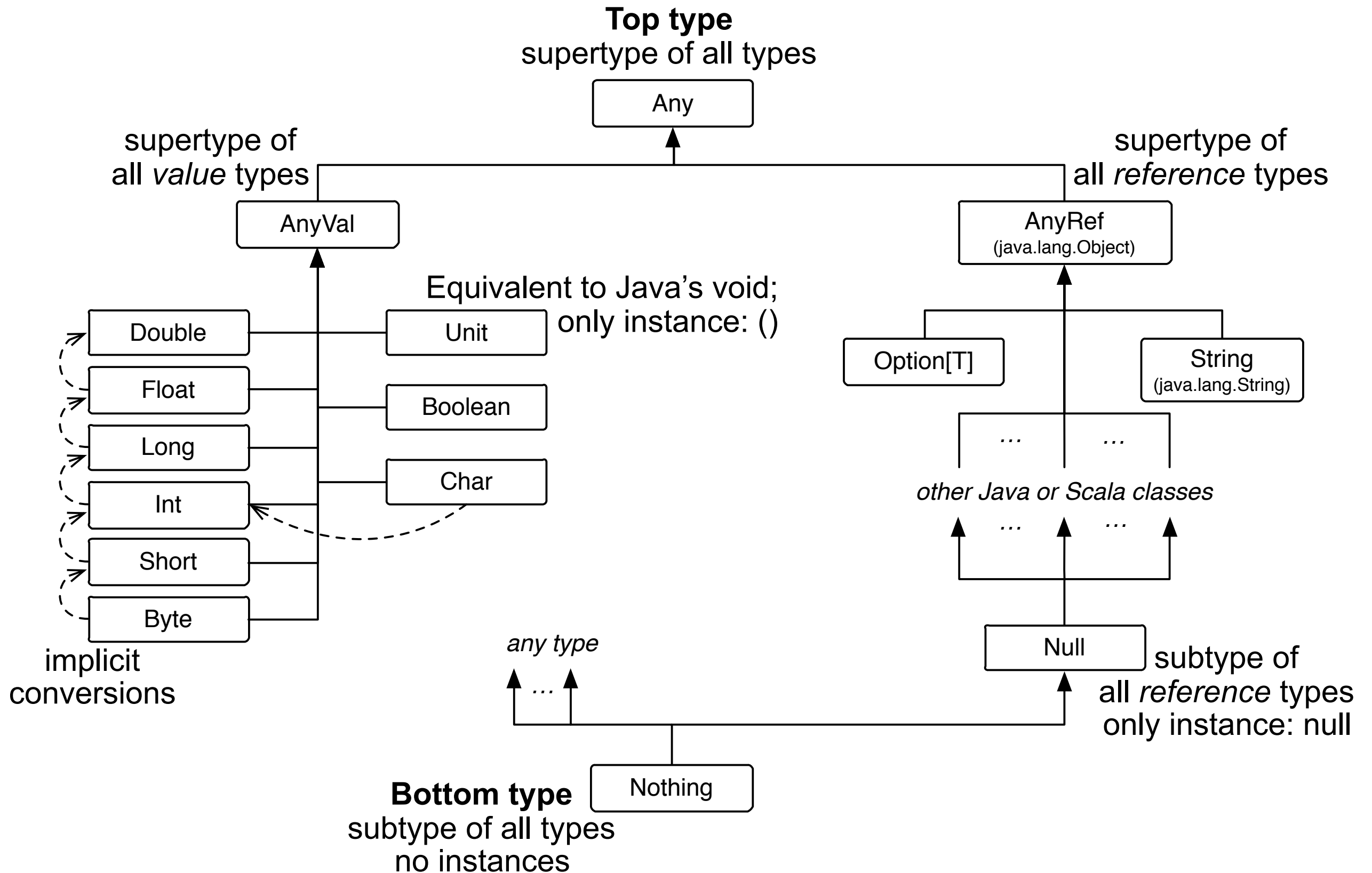
- Everything is an object
- Classes and Inheritance
- Case classes and pattern matching

# Classes and Inheritance

```
class Complex(real: Double, imaginary: Double) {  
  def re = real  
  def im = imaginary  
  
  override def toString() =  
    "" + re + (if (im < 0) "" else "+") + im + "i"  
}  
  
object ComplexNumbers {  
  def main(args: Array[String]) {  
    val c = new Complex(1.2, 3.4)  
    println(c.toString)  
    println("imaginary part: " + (c.im))  
  }  
}
```

Explain all features not available in Java used in the example.

# Scala's Type Hierarchy



# Some Features of Scala

- Everything is an object
- Classes
- Case classes and pattern matching
- Implicit conversions

# Algebraic Data Types (ADTs)

- An ADT is a data type whose values are data are made up of
  - a constructor name
  - subterm values from other datatypes

**Typename**

```
= Con1  t_11 ... t_1k1  |  
   Con2  t_21 ... t_2k2  |  
       ...  
   Con_n t_n1 ... t_nkn
```

**Pattern matching** to:

- distinguish between values defined with different constructors of an ADT
- extract the subparts of a complex ADT



# Case Classes for Algebraic Data Types (ADTs)

- Scala's case classes model regular, non-encapsulated ADT as objects and enable pattern matching on such objects
- Especially valuable when processing recursive (e.g., tree) structures

# Case Classes for Arithmetic Expressions

```
abstract class Tree  
  
case class Leaf(n: Int) extends Tree  
case class Node(left: Tree, right: Tree) extends Tree
```

Tree values:

```
Node(Leaf(3), Leaf(4))  
Node(Node(Leaf(3), Leaf(4)), Leaf(7))
```

# Case Classes vs. “normal” Classes (1)

- Factory methods are automatically available for case classes: `Leaf(3)` instead of `new Leaf(3)`
- Instances of case classes can be decomposed into their parts (constructor parameters) through pattern matching

# Pattern Matching on Case Classes

Basic idea:

- Attempt to match a value to a series of patterns
- As soon as a pattern matches, extract and name various parts of the value,
- Evaluate code that makes use of these named parts

```
abstract class Tree
case class Leaf(n: Int) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

def sum(t: Tree): Int = t match {
  case Leaf(n) => n
  case Node(left, right) => sum(left) + sum(right)
}
```

# Question

```
abstract class Tree
case class Leaf(n: Int) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

def sum(t: Tree): Int = t match {
  case Leaf(n) => n
  case Node(left, right) => sum(left) + sum(right)
}
```

- Do we really need case classes?
- Couldn't we define sum as a method of Tree and its subclasses?
- Wouldn't this be more OO conform?

## Case Classes vs. “normal” Classes (2)

- Getter functions automatically defined for constructor parameters
- Default definitions for methods `equals` and `hashCode` that work on the structure of the instances and not on their identities

# What is next?

- Modeling rudimentary languages to start with
  - A language for arithmetic
  - A language with names
- Add functions with various degrees of complexity
  - Think of C functions vs. Java 8 lambdas

**QUESTIONS?**