CS 186: Scala

Vikram Sreekanti

Project 2 Overview

- What are we doing?
  - UDF Caching!
  - What does that mean?
    - First: partition to disk
    - Then: evaluate the UDF over the particular partition while caching the results of the execution
- How should I approach this?
  - Task 1: Implement DiskPartition
    - this is only concerned with a *single* partition on disk
  - Task 2: do phase 1 of external hashing
    - builds on task 1
  - Task 3: do in-memory UDF caching
    - think memoization from CS 61A
  - Task 4: put it all together — disk-partitioned UDF caching

Getting Set Up

Example code:

github.com/viksree/ScalaTutorial

SBT:

Windows: http://www.scala-sbt.org/0.13/tutorial/Installing-sbt-on-Windows.html

Mac: http://www.scala-sbt.org/0.13/tutorial/Installing-sbt-on-Mac.html

What is Scala?

- Functional language
  - Basic idea: treats computation as the evaluation of mathematical functions.
    - Lot of powerful ideas behind this, but we don't have time to get into them.
- JVM-based
  - Basic idea: runs on Java Virtual Machine
    - Just like Java, Clojure, etc.
- Developed by Typesafe
  - Gained a lot of popularity in the last 5 years
    - Used by Twitter, Spark (Databricks), etc.
    - Important systems language of the future! (?)

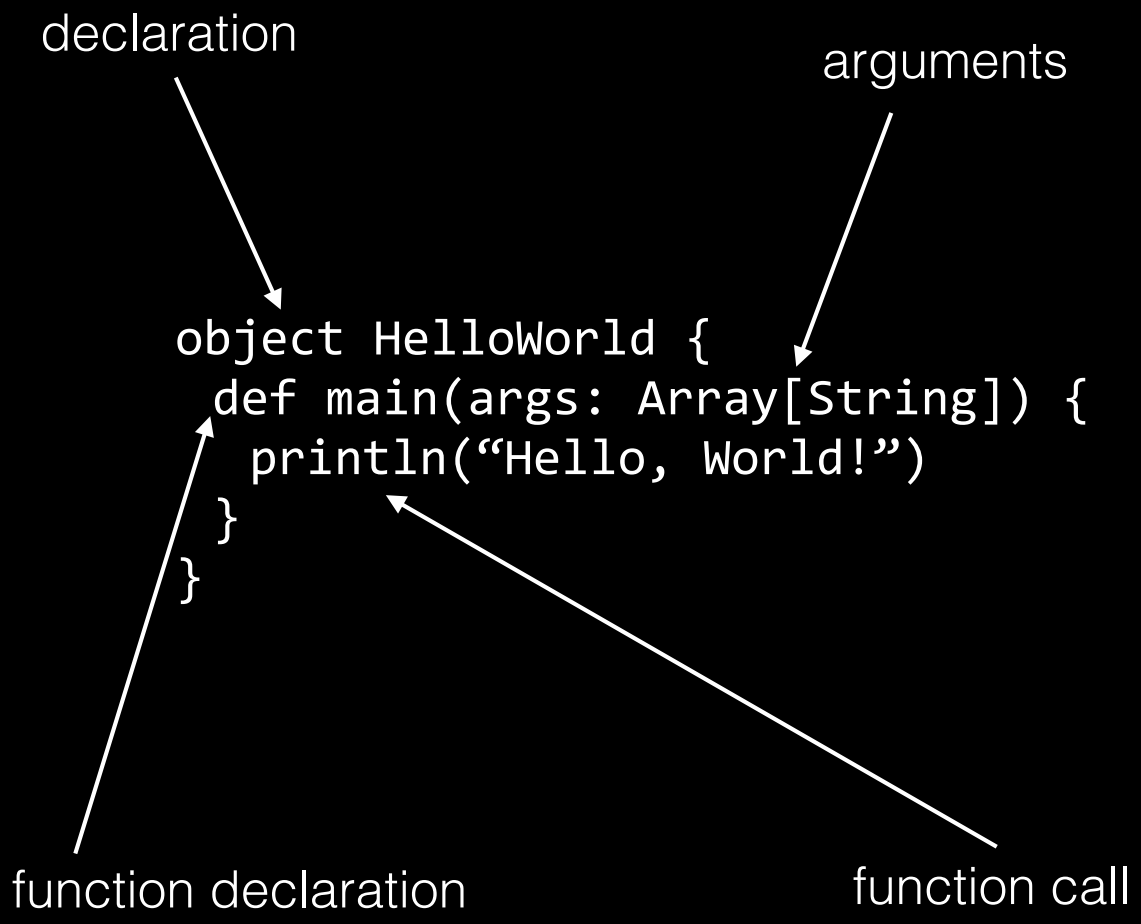
val vs. var and Type Inference

- `val` is used to denote a value that will not change.
  - This exists for compiler optimization reasons.
  - If something *can* be declared a `val`, then it should be.
- `var` is used to denote a value that can change.

- Variables do not need to be declared to have any type (think Python). Scala will infer the type.
  - However, if they are declared to have a type, the compiler will optimize for that type.
  - How do you declare a type?
    - `val x: Int = 5`

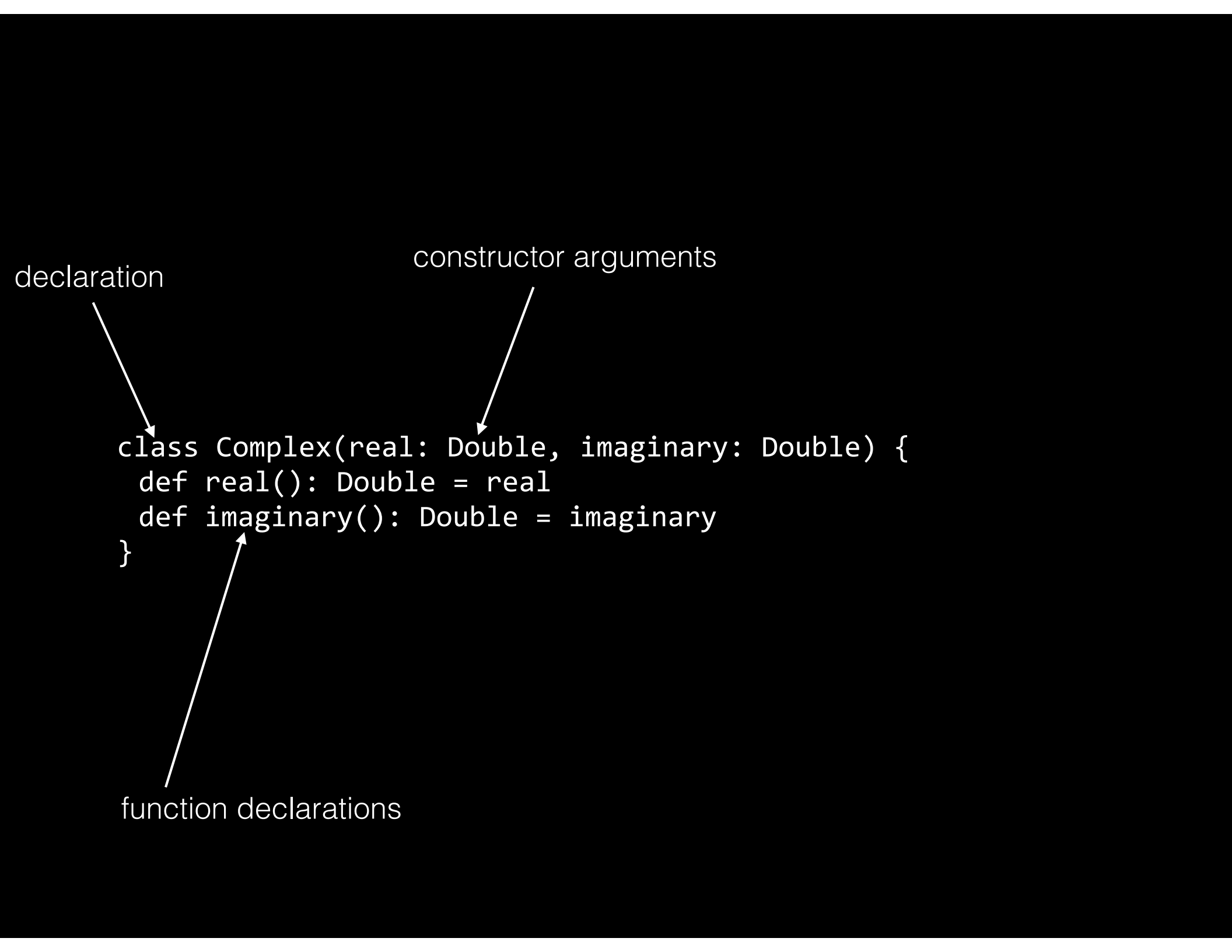
Objects

- Objects in Scala are *singleton instances.*
  - Declares both a class and a single instance of that class.
  - Similar to Java singleton paradigm but built-in to the language.
- Often used for Utils (see `CS186Utils.scala`) or object factories.
  - This is because there are no static methods in Scala.
  - In Java, util methods and object factories are often declared as static methods.
- Do not confuse these with instances of classes. These are also *objects*, but they are objects in the traditional OOP sense.


declaration
arguments
object HelloWorld {
def main(args: Array[String]) {
println(“Hello, World!”)
}
}
function declaration
function call

Classes

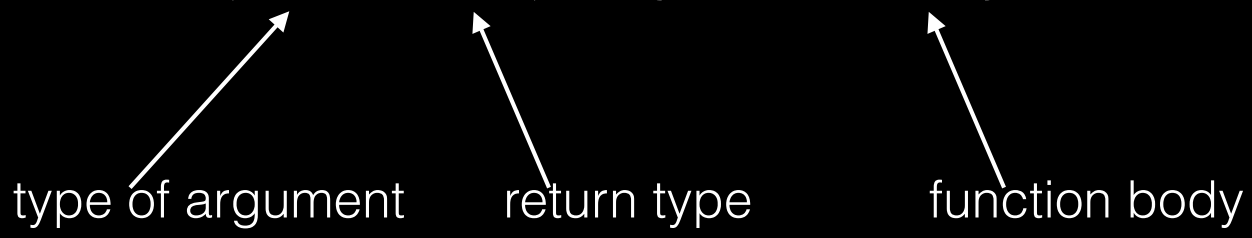
- These are just like Java classes!
- … but the syntax is mildly different.


declaration
constructor arguments
class Complex(real: Double, imaginary: Double) {
def real(): Double = real
def imaginary(): Double = imaginary
}
function declarations

Anonymous Functions (Closures)

- Just like Python, Scala has higher-order functions.
  - Functions can be declared within other functions, passed as arguments, put in variables, etc.

```
val func: (Int => Int) = { x => x + 1}
```



Case Classes

- Case classes are a special type of class in Scala.
  - Don't need to use the new keyword.
  - Getter functions are automatically defined.
  - Default hashcode and equals methods are provide, which work on the data in the case class.
  - Default toString method is provide, which represents the data in the class.
  - They can be used for case matching (more on this in a bit).

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var (n: String) extends Tree
case class Const(v: Int) extends Tree
```

Case Matching

- Think switch-cases in Java or C.
  - Similar idea, much more versatile syntax and functionality.

```
// simple case matching
def isEqualTo2(x: Int): Boolean = x match {
  case 2 => True
  case _ => False
}

// casting
var temp: JavaArrayList[Row] = null
temp = in.readObject() match {
  case value: JavaArrayList[Row] => value
  case _: Throwable => throw new RuntimeException(. . .)
}
```

Traits

- Like Java interfaces, but they can contain code.