Learning Scala?

Learn the *f*undamentals

Craig Tataryn

@craiger

- The Basement Coders Podcast
  - basementcoders.com
- Winnipeg JVM Programming Group
  - wjpg.ca
- GRIND Software Inc.
  - grindsoftware.com

- Became interested about four years ago

- Became interested about four years ago
- Scala was becoming really popular

- Became interested about four years ago
- Scala was becoming really popular
- Thought it was just syntax I’d have to learn

- Became interested about four years ago
- Scala was becoming really popular
- Thought it was just syntax I'd have to learn
- Wrong.

- Became interested about four years ago
- Scala was becoming really popular
- Thought it was just syntax I'd have to learn
- Wrong.
- Turns out it was a whole new paradigm

Write Scala like Java

FALSE^H^H^H^H^H

I humbly disagree

- Might work great at first

- Might work great at first
- You’ll have to use a library at some point

```
m map { t => val (s, i) = t; (s, i+1) }
```

Simple concepts **big impact**

Scala Basics

Scala Basics

```
obj.someProp
```

```
def someProp:String = {
  //getter code
}
```

Scala Basics

```
obj.someProp

obj.someProp = someVal
```

```
def someProp:String = {
  //getter code
}

def someProp_=(someVal:String) {
  //setter code
}
```

Scala Basics

```
obj.someProp

obj.someProp = someVal

val someVar:SomeType = someVal
```

```
def someProp:String = {
  //getter code
}

def someProp_=(someVal:String) {
  //setter code
}
```

Scala Basics

```
obj.someProp

obj.someProp = someVal

val someVar:SomeType = someVal

class SomeClass(arg1:SomeType)

new SomeClass(someVal)
```

```
def someProp:String = {
   //getter code
}

def someProp_=(someVal:String) {
   //setter code
}
```

Scala Basics

```
obj.someProp

obj.someProp = someVal

val someVar:SomeType = someVal

class SomeClass(arg1:SomeType)

new SomeClass(someVal)

case class SomeClass(arg1:String)

new SomeClass(“Hi”).arg1
```

```
def someProp:String = {
  //getter code
}

def someProp_=(someVal:String) {
  //setter code
}
```

Scala Basics

```
obj.someProp

obj.someProp = someVal

val someVar:SomeType = someVal

class SomeClass(arg1:SomeType)

new SomeClass(someVal)

case class SomeClass(arg1:String)

new SomeClass(“Hi”).arg1

val someVar = SomeClass(“no new!”)
```

```
def someProp:String = {
   //getter code
}

def someProp_=(someVal:String) {
   //setter code
}
```

Scala Basics

```
obj.someProp

obj.someProp = someVal

val someVar:SomeType = someVal

class SomeClass(arg1:SomeType)

new SomeClass(someVal)

case class SomeClass(arg1:String)

new SomeClass(“Hi”).arg1

val someVar = SomeClass(“no new!”)

def someFunc(a1:SomeType,...):SomeReturnType = {
    //...
}
```

```
def someProp:String = {
  //getter code
}

def someProp_=(someVal:String) {
  //setter code
}
```

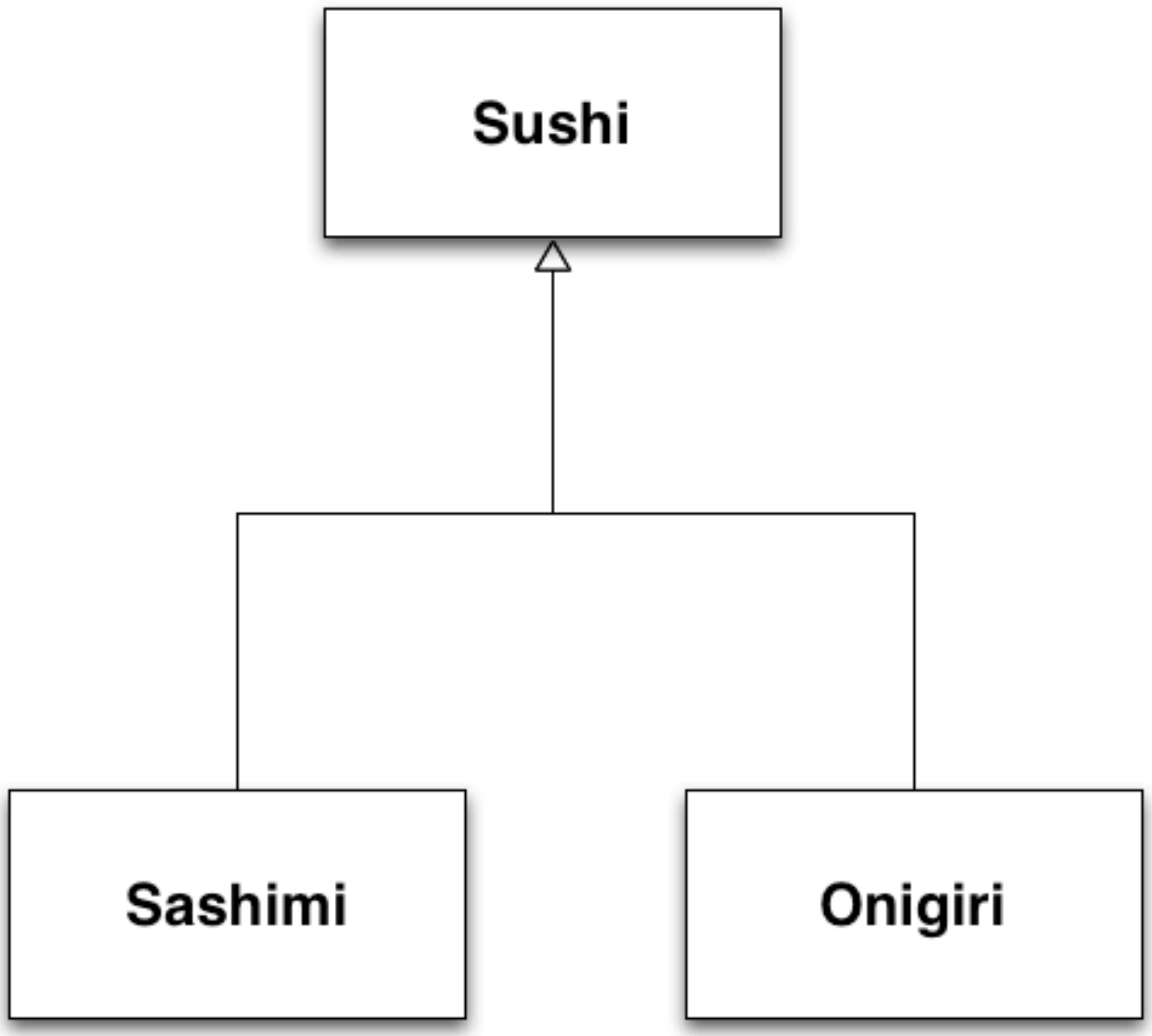
(Tuple)

Tuple

- Fundamental Scala data type
- Part of the syntax
- A container for other data types

Bento

(Sashimi, Onigiri)



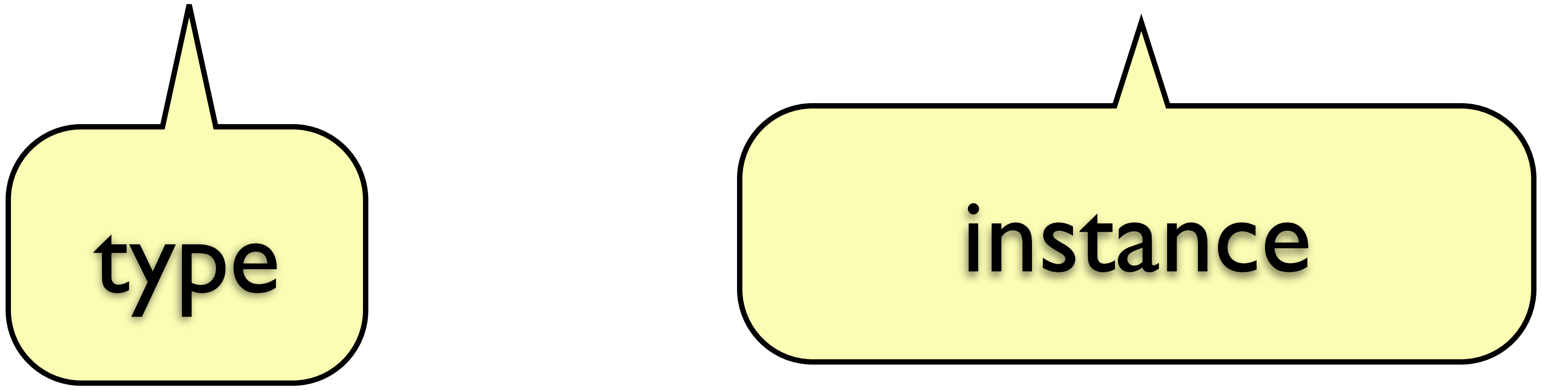
```
val bento:(Sushi, Sushi) = (new Sashimi, new Onigiri)
```

Tuple

- So Tuples are primitive types?
- Nope!
- Just classes
- Special syntax for
  - type definition
  - instantiation

Tuple

```
var bento:(Sushi, Sushi) = (new Sashimi, new Onigiri)
```



type

instance

Tuple

- What does a Tuple class look like?
- There are 22 of them as of 2.11.0

Tuple

- What does a Tuple class look like?
- There are 22 of them as of 2.11.0

```
var bento:Tuple2[Sushi, Sushi] = new Tuple2(new Sashimi, new Onigiri)
```

Tuple

- What does a Tuple class look like?
- There are 22 of them as of 2.11.0

```
var bento:Tuple2[Sushi, Sushi] = new Tuple2(new Sashimi, new Onigiri)

case class Tuple2[T1,T2](_1:T1, _2:T2) {
   //...
}
```

Tuple

- What does a Tuple class look like?
- There are 22 of them as of 2.11.0

```
var bento:Tuple2[Sushi, Sushi] = new Tuple2(new Sashimi, new Onigiri)

case class Tuple2[T1,T2](_1:T1, _2:T2) {
   //...
}


var first = bento._1
```

Tuple

- What does a Tuple class look like?
- There are 22 of them as of 2.11.0

```
var bento:Tuple2[Sushi, Sushi] = new Tuple2(new Sashimi, new Onigiri)

case class Tuple2[T1,T2](_1:T1, _2:T2) {
   //...
}


var first = bento._1


var (first,second) = bento
```

Essential Syntax

1. Functions that take exactly one parameter

```
class Util {
  def echo(s:String)
}
val util = new Util()
```

1. Functions that take exactly one parameter

    - Doesn’t need a *dot* preceding the method

```
class Util {
  def echo(s:String)
}
val util = new Util()


util echo(“Hello”)
```

1. Functions that take exactly one parameter

```
class Util {
  def echo(s:String)
}
val util = new Util()
```

- Doesn’t need a *dot* preceding the method

```
util echo(“Hello”)
```

- Can omit parentheses

```
util echo “Hello”
```

1. Functions that take exactly one parameter

```
class Util {
  def echo(s:String)
}
val util = new Util()
```

- Doesn’t need a *dot* preceding the method

```
util echo(“Hello”)
```

- Can omit parentheses

```
util echo “Hello”
```

- Or use braces instead

```
util echo {
  “Hello”
}
```

```
class Util {
  def echo(s:String)
}
val util = new Util()
```

1. Functions that take exactly one parameter
    - Doesn’t need a *dot* preceding the method

```
util echo(“Hello”)
```

    - Can omit parentheses

```
util echo “Hello”
```

    - Or use braces instead

```
util echo {
  “Hello”
}
```

2. Return value of a function is...

1. Functions that take exactly one parameter

```
class Util {
  def echo(s:String)
}
val util = new Util()
```

- Doesn’t need a *dot* preceding the method

```
util echo(“Hello”)
```

- Can omit parentheses

```
util echo “Hello”
```

- Or use braces instead

```
util echo {
  “Hello”
}
```

2. Return value of a function is...

- The last executable expression

```
def echo(s:String) = {
  s
}
```

Removing the Syntactic Sugar

```
m map { t => val (s, i) = t; (s, i+1) }
```

↓

```
m.map({ t => val (s, i) = t; (s, i+1) })
```

Functions as Types

- Not something we ~~are~~ were used to

- Not something we ~~are~~ were used to
- Functions, like Tuples, have special syntax in Scala for:
  - Type definition

- Not something we ~~are~~ were used to
- Functions, like Tuples, have special syntax in Scala for:
  - Type definition
  - Instantiation (aka *Function Literals*)

Type Definition

- Function Types are based on
  - The number and the type of the parameters
  - The return type of the function

Function Type Defs

Function Type Defs

- Function that takes an Int and returns an Int

```
Int => Int
```

```
val addOne: Int => Int = ...
```

Function Type Defs

- Function that takes an Int and returns an Int

  ```
  Int => Int
  ```

  ```
  val addOne: Int => Int = ...
  ```

- Function that takes two Ints and returns a String

Function Type Defs

- Function that takes an Int and returns an Int

  ```
  Int => Int
  ```

  ```
  val addOne: Int => Int = ...
  ```

- Function that takes two Ints and returns a String

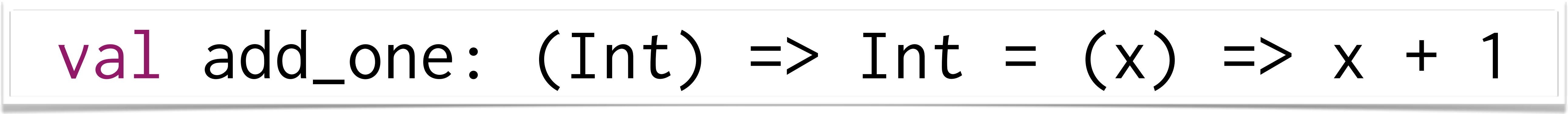
  ```
  (Int,Int) => String
  ```

  ```
  val concat: (Int, Int) => String = ...
  ```

Function Instantiation

Function Instantiation

Best described by showing Function literals



```scala
val add_one: (Int) => Int = (x) => x + 1
```

Function Instantiation

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

Function Instantiation

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which

Function Instantiation

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int

Function Instantiation

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int

Function Instantiation

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It’s implementation is as follows

Function Instantiation

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It’s implementation is as follows
  - The parameter will be named x

Function Instantiation

Best described by showing Function literals

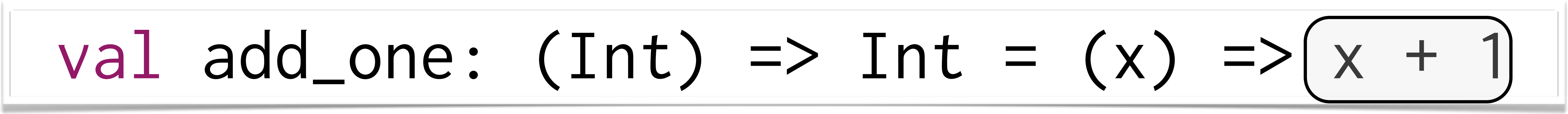
```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It’s implementation is as follows
  - The parameter will be named x
  - It will return x + 1

Function Instantiation

Best described by showing Function literals



```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It’s implementation is as follows
  - The parameter will be named x
  - It will return x + 1

How does it work?

How does it work?

- Just like Tuples, Functions have an underlying class
- Also a syntax convention that dictates:

How does it work?

- Just like Tuples, Functions have an underlying class
- Also a syntax convention that dictates:
  - If a class/object has a function called "apply"

How does it work?

- Just like Tuples, Functions have an underlying class
- Also a syntax convention that dictates:
  - If a class/object has a function called "apply"
    - An instance of that class can be called as if it is a function

How does it work?

- Just like Tuples, Functions have an underlying class
- Also a syntax convention that dictates:
  - If a class/object has a function called "apply"
    - An instance of that class can be called as if it is a function
  - There are 22 such basic Functions

- So when you do this

```
val squareIt:Int=>Int = x => x*x
```

- So when you do this

```
val squareIt:Int=>Int = x => x*x
```

- Scala converts it to this:

- So when you do this

```
val squareIt:Int=>Int = x => x*x
```

- Scala converts it to this:

```
val squareIt = new Function1[Int,Int]() {

    def apply(x:Int):Int = x*x

}
```

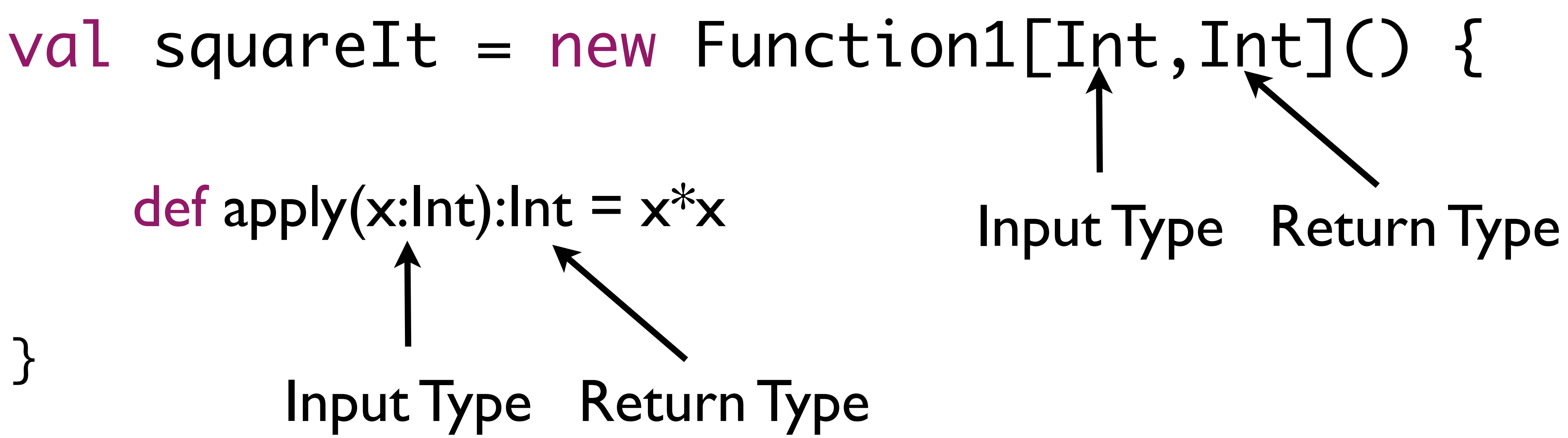
Input Type Return Type

Input Type Return Type

- So when you do this

```
val squareIt:Int=>Int = x => x*x
```

- Scala converts it to this:



```
val squareIt = new Function1[Int,Int]() {

    def apply(x:Int):Int = x*x

}
```

Input Type Return Type

Input Type Return Type

- Both can be called like this:

```
squareIt(1)
```

Call by Name

- A way to pass a literal function as a code block
- Makes your function look like it’s part of the language itself

```
def transaction(code: => Boolean) = {
   //connect to DB, grab a connection, start transaction
   //...
   //execute the code
   code
    //if things went ok commit, if not rollback
}

transaction {

   execute(“INSERT INTO SOME_TABLE...”)

}
```

```
def transaction(code: => Boolean) = {
  //connect to DB, grab a connection, start transaction
  //...
  //execute the code
  code
    //if things went ok commit, if not rollback
}

transaction {

  execute(“INSERT INTO SOME_TABLE...”)

}
```

Type Inference

Type Inference

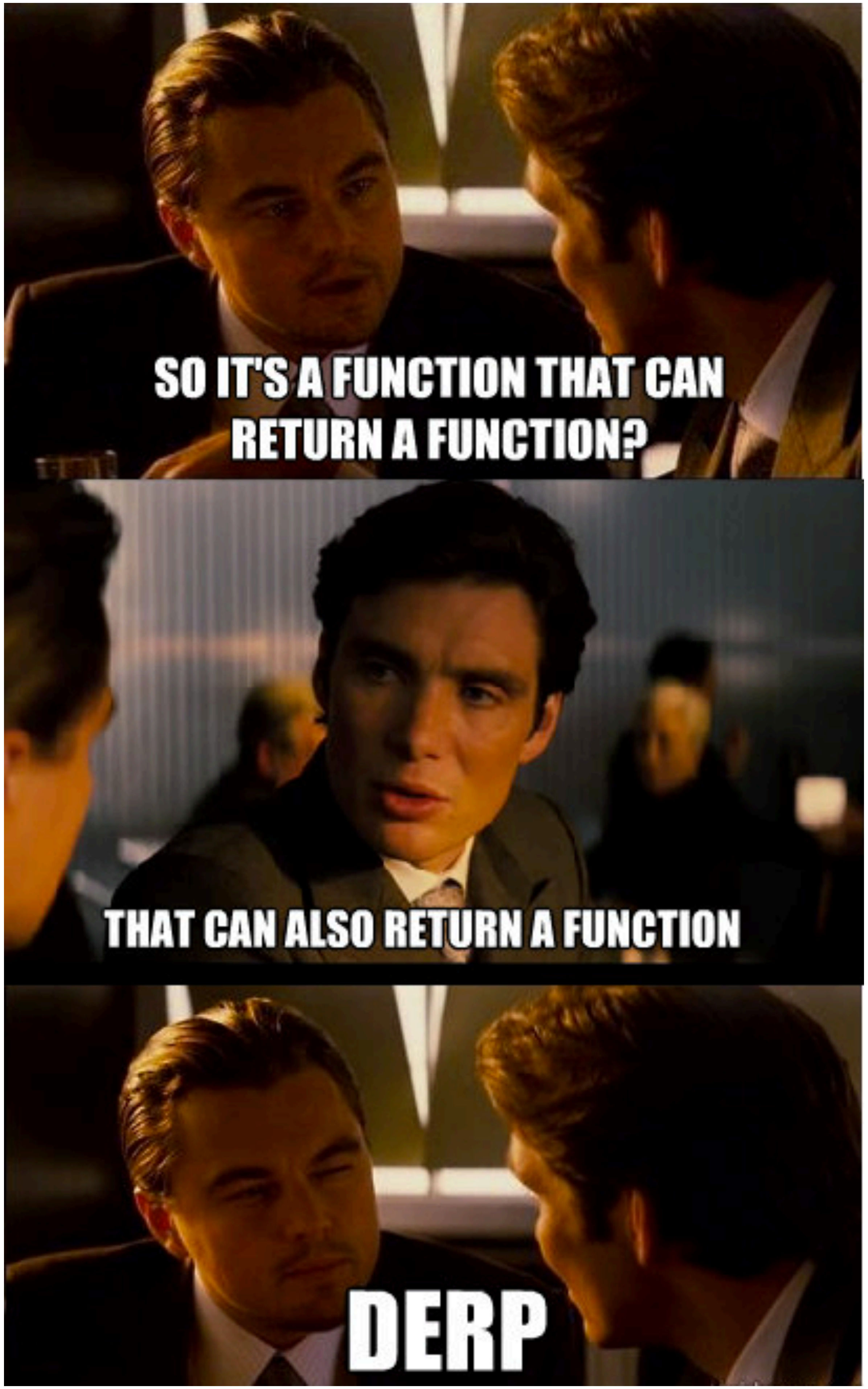
- Where Scala can infer a type, it will
- For instance, declaring a variable
  - `val s = "I'm a string, Duh!"`
  - `val m = new HashMap[String,Int]`

  -
    ```
    def add(a:Int, b:Int) = {
       a + b
    }
    ```

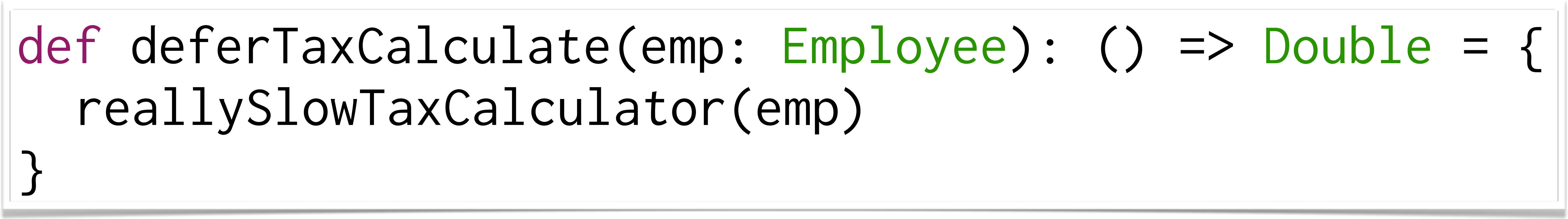
Higher-order Functions

Higher-order

- A function that can
  - Be passed a function
  - Return a function

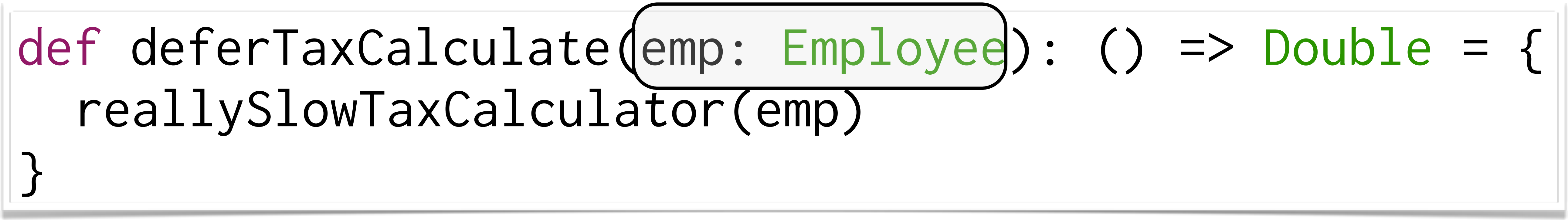

SO IT'S A FUNCTION THAT CAN RETURN A FUNCTION?
THAT CAN ALSO RETURN A FUNCTION
DERP

- We know about function types
- We know about function literals
- We can now construct a Higher-order function



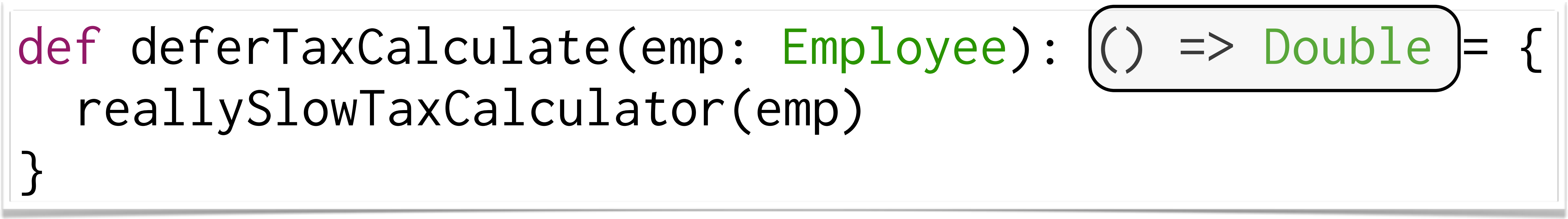
```
def deferTaxCalculate(emp: Employee): () => Double = {
  reallySlowTaxCalculator(emp)
}
```

“Pass an Employee and I’ll pass back a Function that you can call later which doesn’t take any parameters but returns a Double”



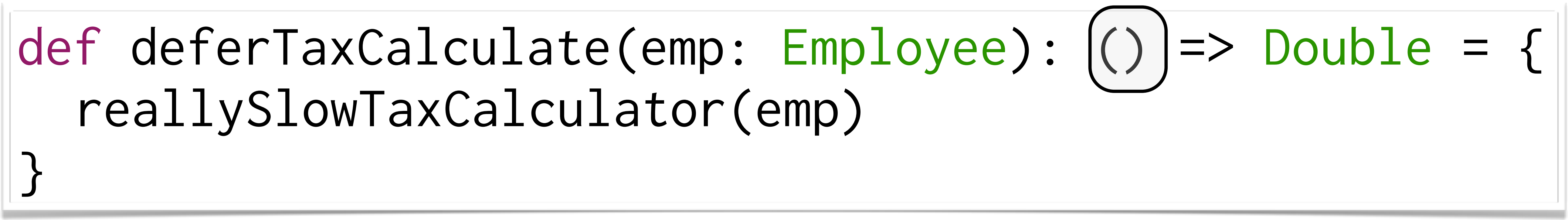
```
def deferTaxCalculate(emp: Employee): () => Double = {
  reallySlowTaxCalculator(emp)
}
```

“Pass an Employee and I’ll pass back a Function that you can call later which doesn’t take any parameters but returns a Double”



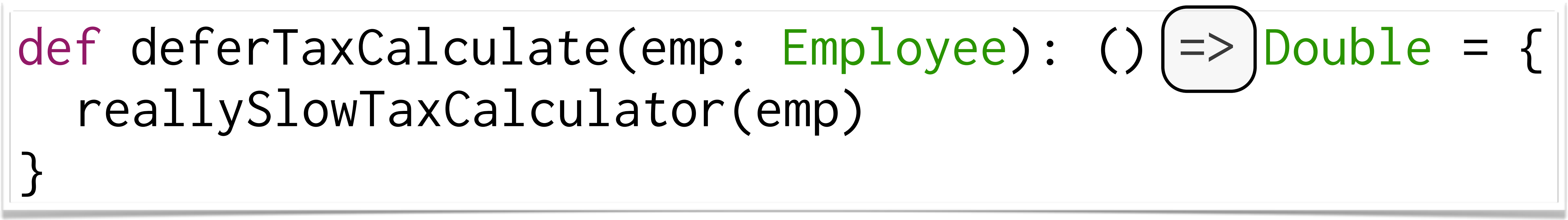
```scala
def deferTaxCalculate(emp: Employee): () => Double = {
  reallySlowTaxCalculator(emp)
}
```

"Pass an Employee and I'll pass back a Function that you can call later which doesn't take any parameters but returns a Double"



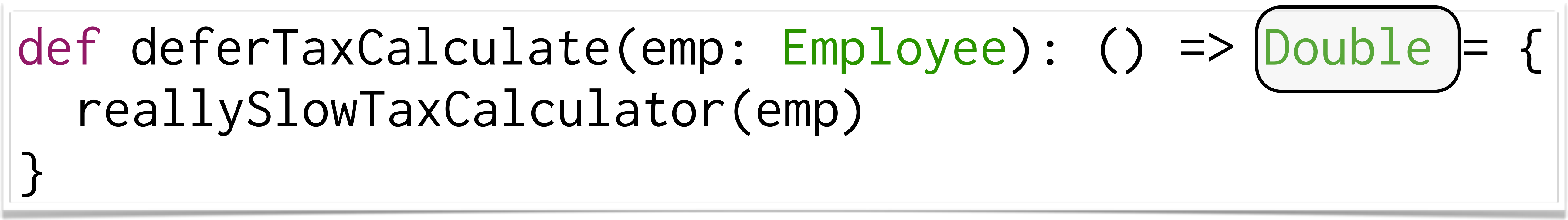
```
def deferTaxCalculate(emp: Employee): () => Double = {
  reallySlowTaxCalculator(emp)
}
```

“Pass an Employee and I’ll pass back a Function that you can call later which doesn’t take any parameters but returns a Double”



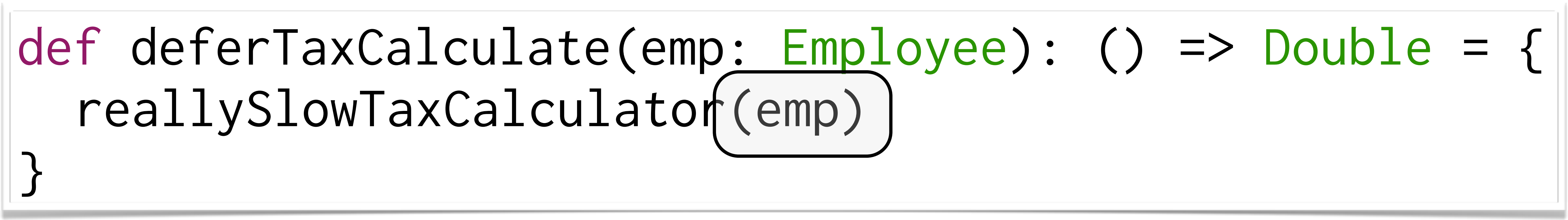
```
def deferTaxCalculate(emp: Employee): () => Double = {
  reallySlowTaxCalculator(emp)
}
```

“Pass an Employee and I’ll pass back a Function that you can call later which doesn’t take any parameters but returns a Double”



```scala
def deferTaxCalculate(emp: Employee): () => Double = {
  reallySlowTaxCalculator(emp)
}
```

“Pass an Employee and I’ll pass back a Function that you can call later which doesn’t take any parameters but returns a Double”



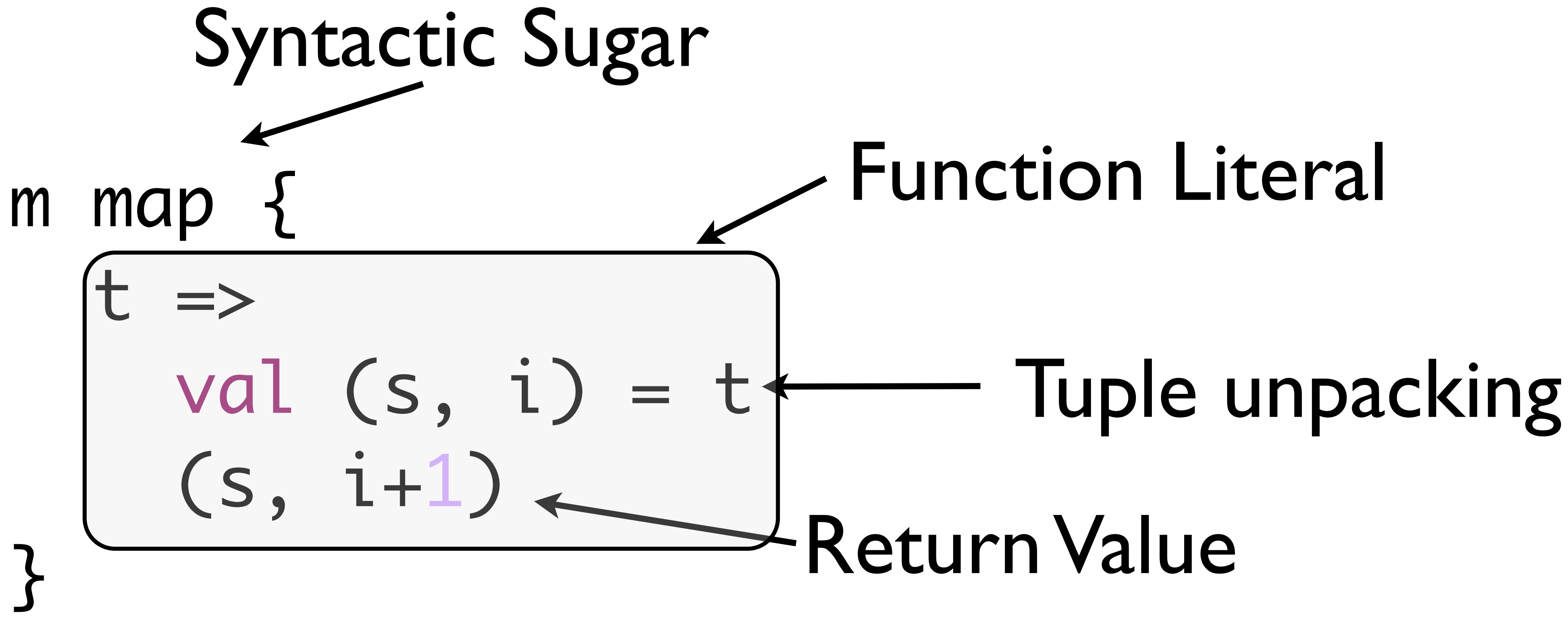
```scala
def deferTaxCalculate(emp: Employee): () => Double = {
  reallySlowTaxCalculator(emp)
}
```

“Pass an Employee and I’ll pass back a Function that you can call later which doesn’t take any parameters but returns a Double”

Closure!

```
m map { t => val (s, i) = t; (s, i+1) }
```

```
m map { t => val (s, i) = t; (s, i+1) }
```



```
m map {
  t =>
    val (s, i) = t
    (s, i+1)
}
```

Syntactic Sugar

Function Literal

Tuple unpacking

Return Value

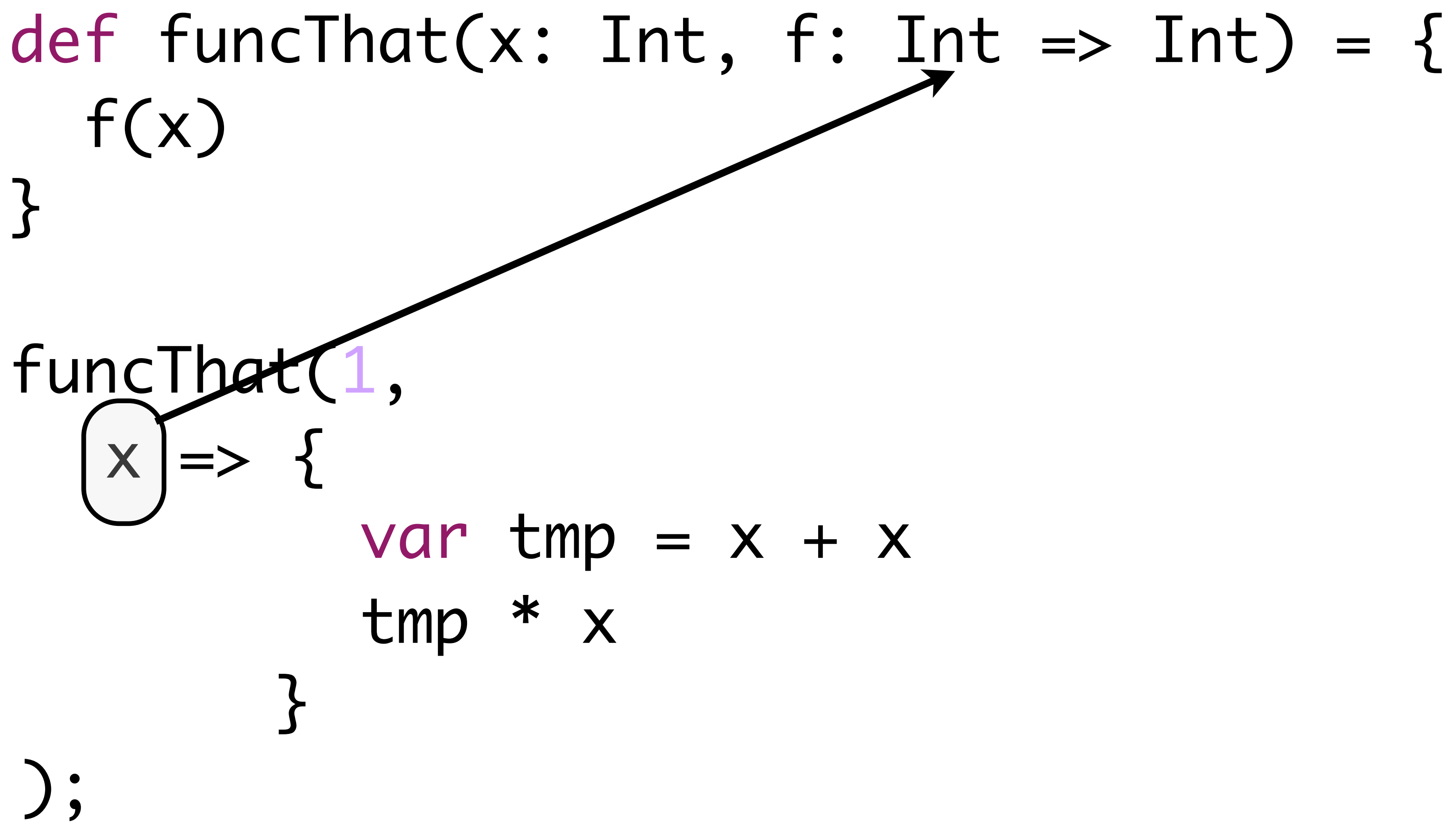
`map` is a function on the instance `m` that accepts a Function

Type Inference++

```
def funcThat(x: Int, f: Int => Int) = {
  f(x)
}

funcThat(1,
  x => {
          var tmp = x + x
          tmp * x
       }
);
```

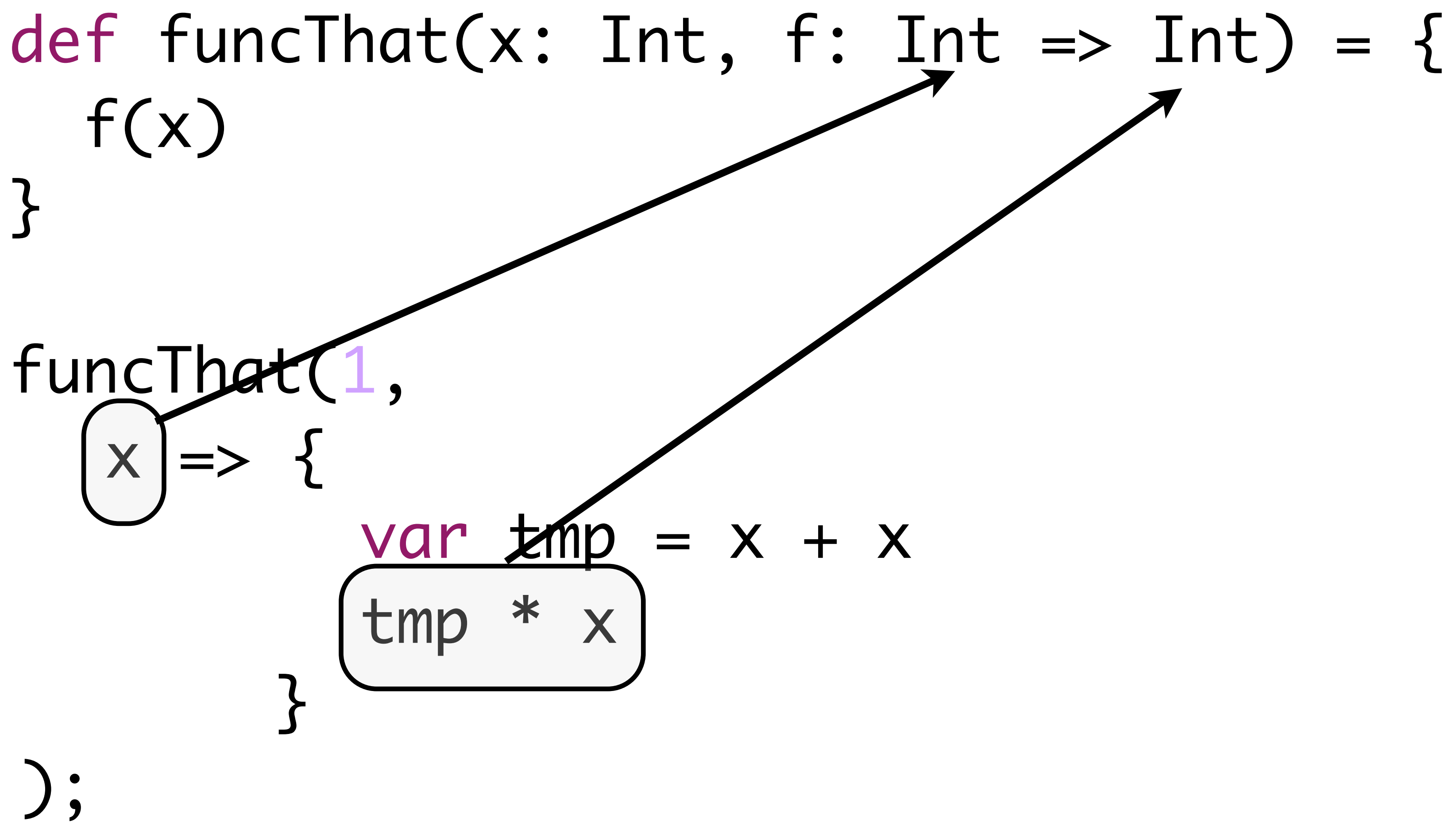
Type Inference++



```
def funcThat(x: Int, f: Int => Int) = {
  f(x)
}

funcThat(1,
  x => {
         var tmp = x + x
         tmp * x
       }
);
```

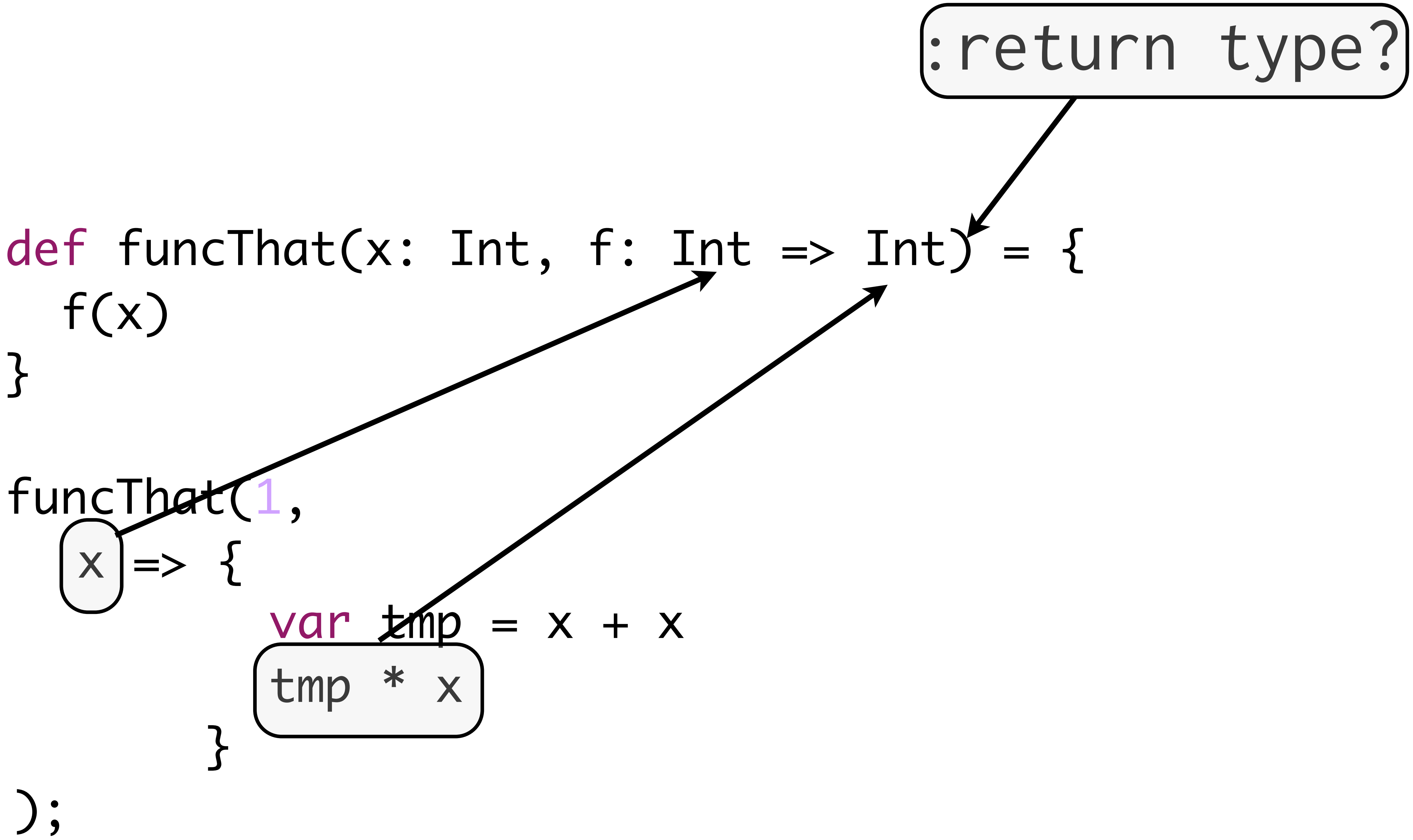
Type Inference++



```
def funcThat(x: Int, f: Int => Int) = {
  f(x)
}

funcThat(1,
  x => {
         var tmp = x + x
         tmp * x
       }
);
```

Type Inference++



```
def funcThat(x: Int, f: Int => Int) = {
  f(x)
}

funcThat(1,
  x => {
         var tmp = x + x
         tmp * x
       }
);
```

:return type?

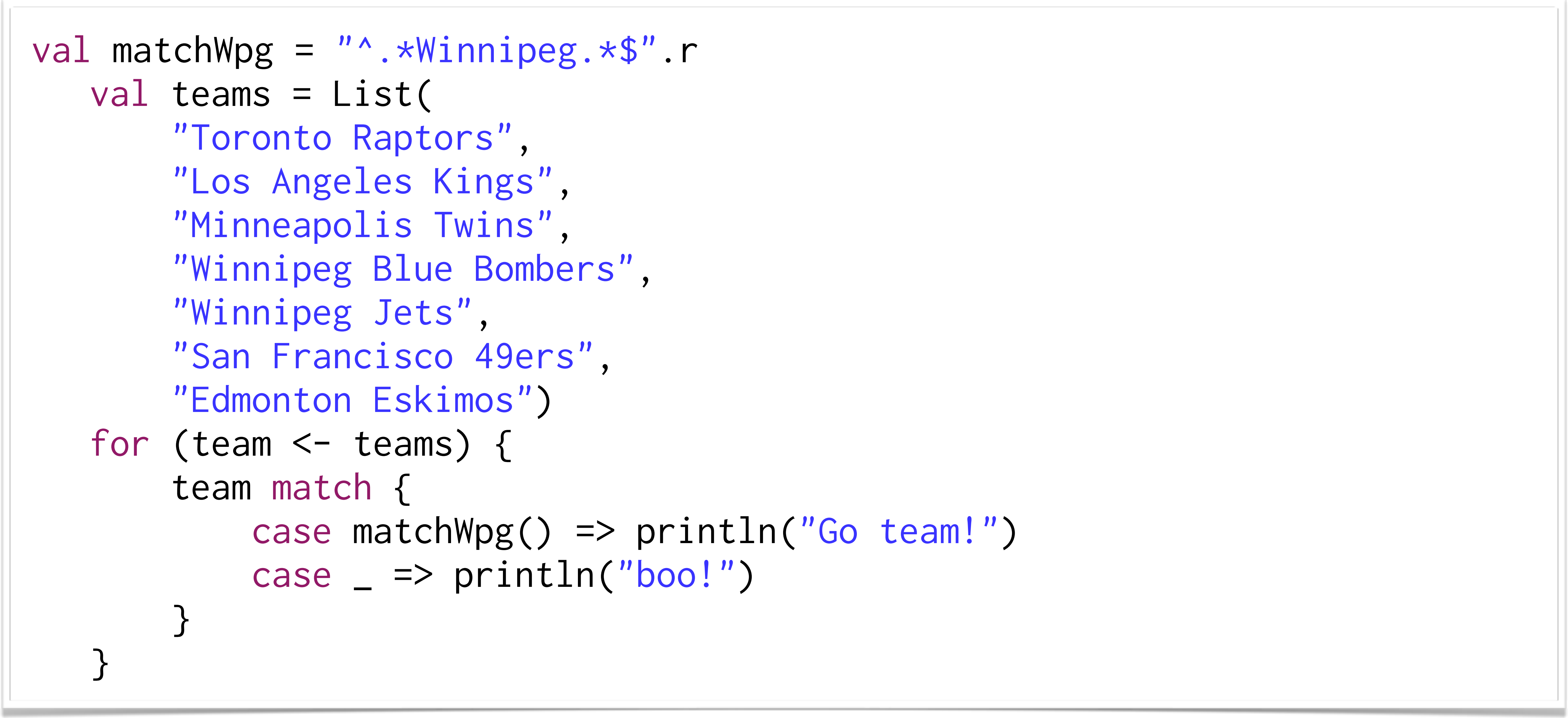
Pattern Matching

Pattern Matching

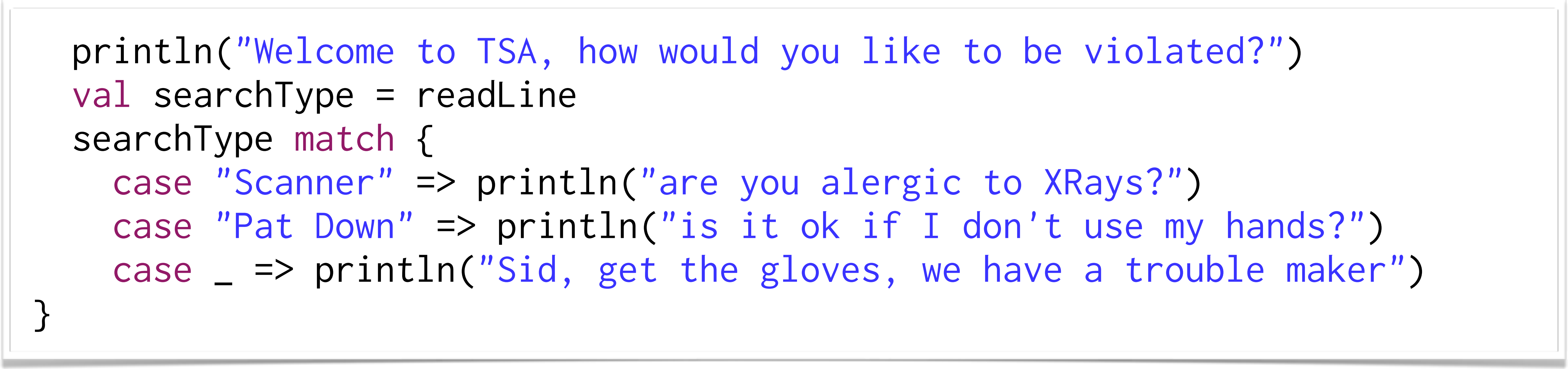
- We aren’t talking about Regular Expressions
    - Although they fall into this category
- Pattern matching in Scala is
    - Extremely expressive
    - Completely flexible

Expressiveness

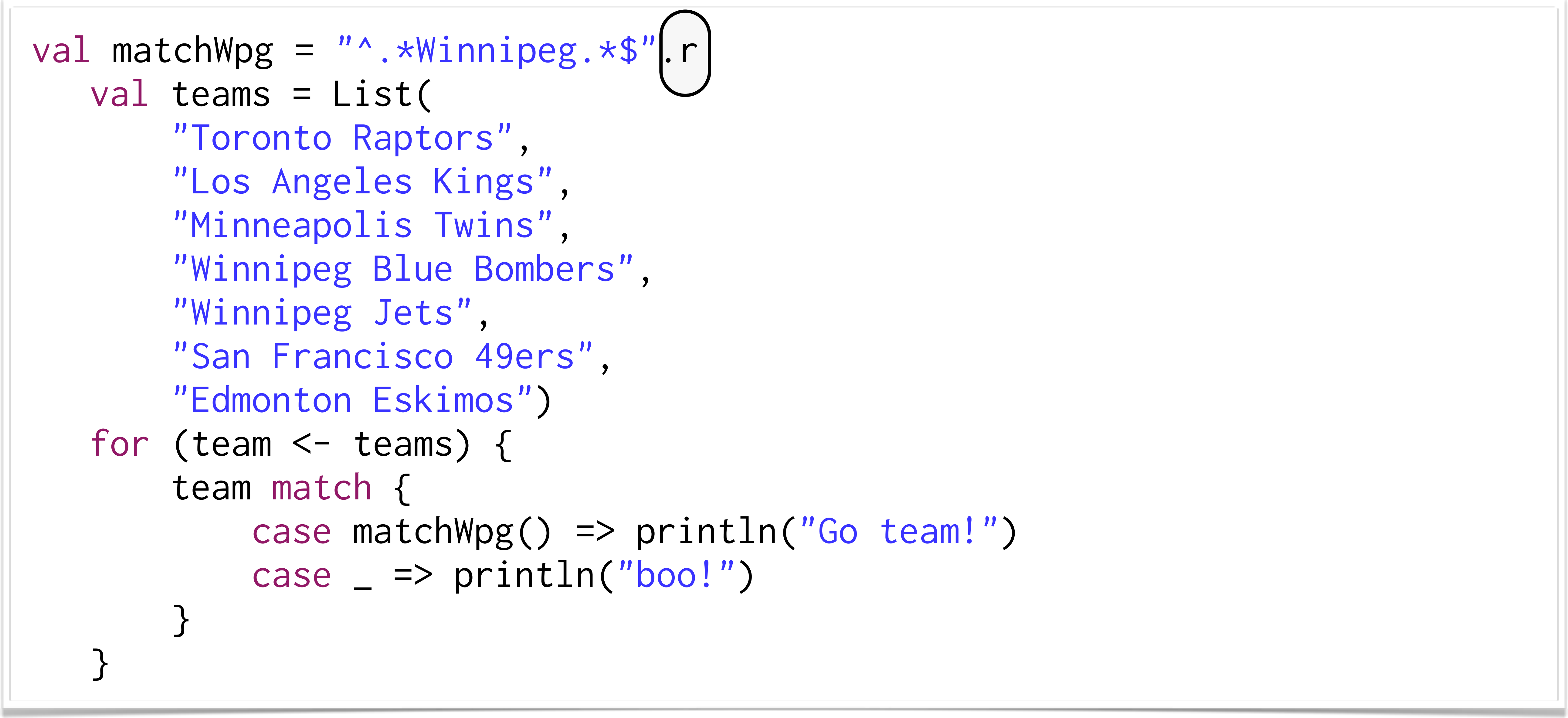
- Pattern Matching can seems so adhoc
- The following are all valid uses of the match/ case construct



```
val matchWpg = "^.*Winnipeg.*$".r
    val teams = List(
        "Toronto Raptors",
        "Los Angeles Kings",
        "Minneapolis Twins",
        "Winnipeg Blue Bombers",
        "Winnipeg Jets",
        "San Francisco 49ers",
        "Edmonton Eskimos")
    for (team <- teams) {
        team match {
            case matchWpg() => println("Go team!")
            case _ => println("boo!")
        }
    }
```



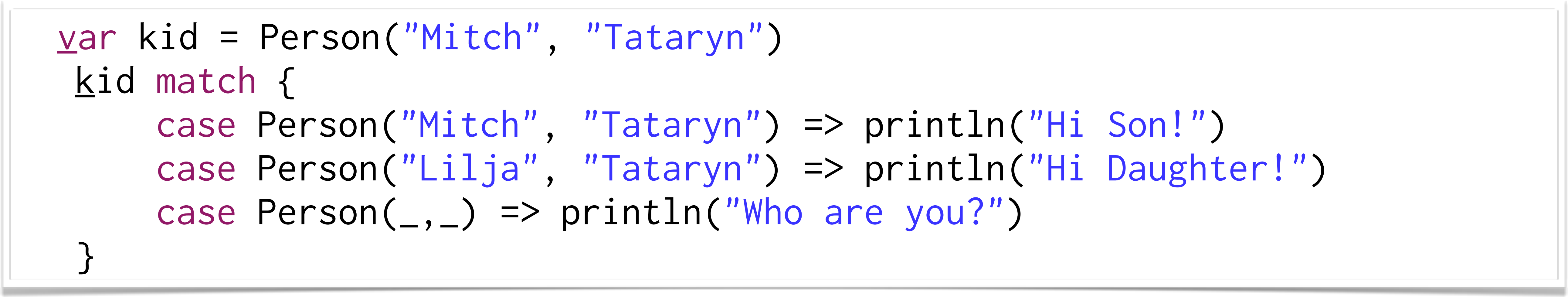
```
  println("Welcome to TSA, how would you like to be violated?")
  val searchType = readLine
  searchType match {
    case "Scanner" => println("are you alergic to XRays?")
    case "Pat Down" => println("is it ok if I don't use my hands?")
    case _ => println("Sid, get the gloves, we have a trouble maker")
}
```



```
val matchWpg = "^.*Winnipeg.*$".r
   val teams = List(
       "Toronto Raptors",
       "Los Angeles Kings",
       "Minneapolis Twins",
       "Winnipeg Blue Bombers",
       "Winnipeg Jets",
       "San Francisco 49ers",
       "Edmonton Eskimos")
   for (team <- teams) {
       team match {
           case matchWpg() => println("Go team!")
           case _ => println("boo!")
       }
   }
```

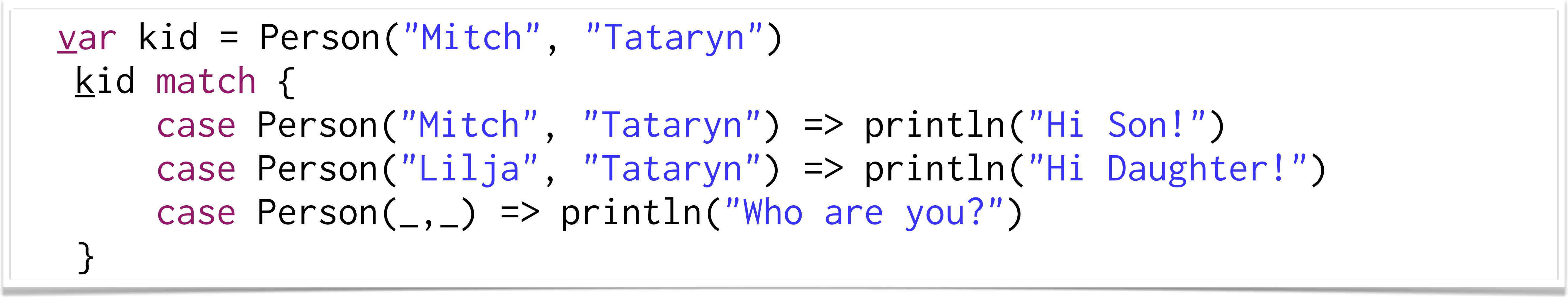
Match on Foo

Match on Foo

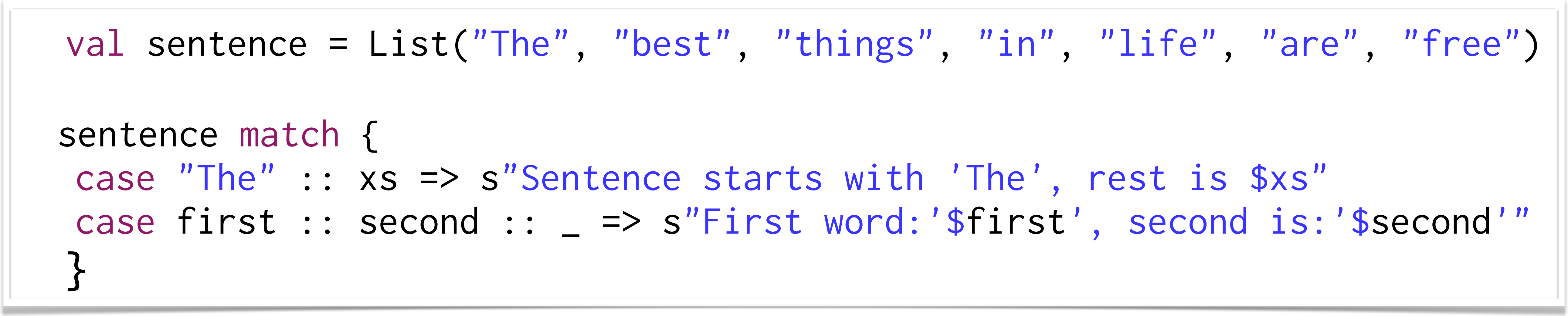


```
var kid = Person("Mitch", "Tataryn")
 kid match {
     case Person("Mitch", "Tataryn") => println("Hi Son!")
     case Person("Lilja", "Tataryn") => println("Hi Daughter!")
     case Person(_,_) => println("Who are you?")
 }
```

Match on Foo



```
var kid = Person("Mitch”, "Tataryn”)
 kid match {
      case Person("Mitch”, "Tataryn”) => println("Hi Son!”)
      case Person("Lilja”, "Tataryn”) => println("Hi Daughter!”)
      case Person(_,_) => println("Who are you?”)
 }
```



```
 val sentence = List("The”, "best”, "things”, "in”, "life”, "are”, "free”)

sentence match {
 case "The” :: xs => s”Sentence starts with 'The', rest is $xs”
 case first :: second :: _ => s”First word:'$first', second is:'$second'”
 }
```

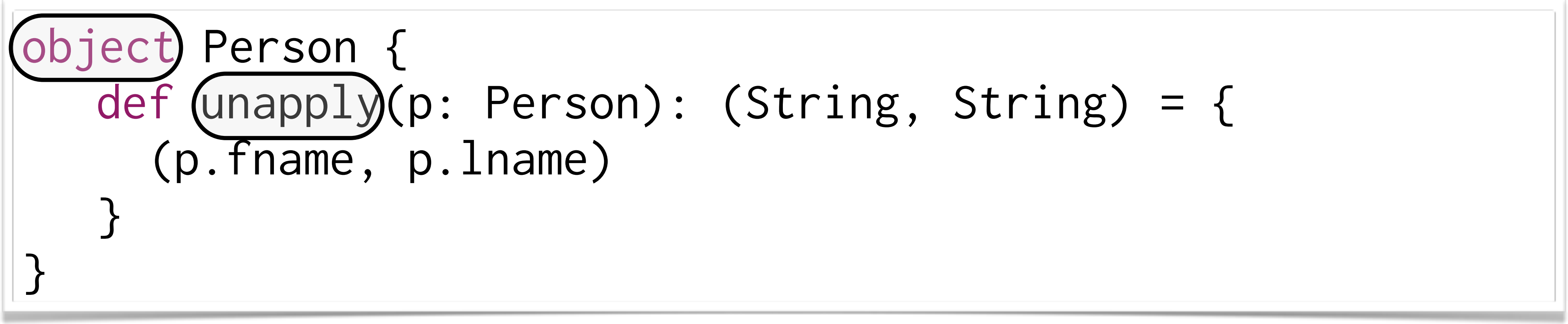
Flexibility

- Behind-the-scenes Pattern matching expects an `unapply` method

Flexibility

- Behind-the-scenes Pattern matching expects an `unapply` method

For our `Person` example:

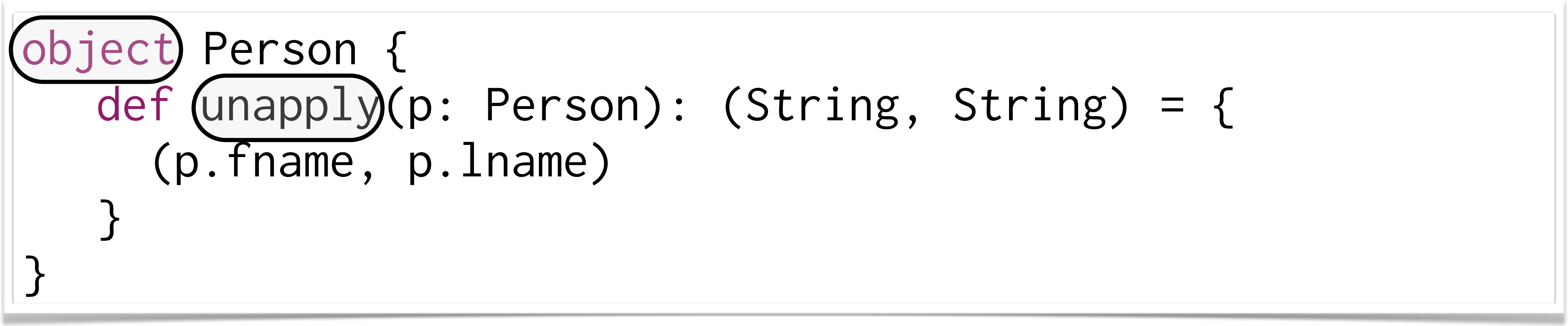


```
object Person {
   def unapply(p: Person): (String, String) = {
     (p.fname, p.lname)
   }
}
```

Flexibility

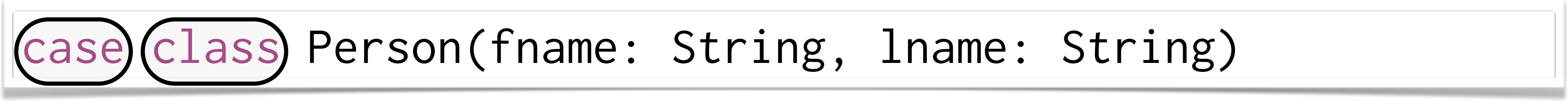
- Behind-the-scenes Pattern matching expects an `unapply` method

For our `Person` example:



```
object Person {
   def unapply(p: Person): (String, String) = {
     (p.fname, p.lname)
   }
}
```

Or:



```
case class Person(fname: String, lname: String)
```

Fundamentals

- Syntax Rules
- Tuples
- Function Types and Literals
- Pattern Matching
- not shown: *implicits*

Trivia

Trivia

- What’s one of the special syntax rules for functions that accept exactly one parameter?

Trivia

- What’s one of the special syntax rules for functions that accept exactly one parameter?
- A Function Type is comprised of what two things?

Craig Tataryn

Craig Tataryn

@craiger

- Review the fundamentals presented
- You'll be in good shape
- http://tataryn.net/tag/scala/
- https://github.com/ctataryn/LearningScala.git