Example 9

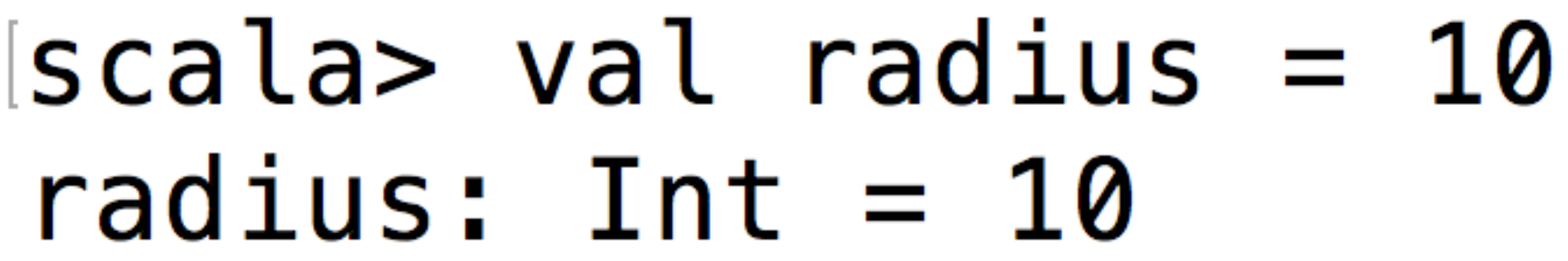
Statements v Expressions

Statements v Expressions

Let's draw a distinction between these two - Scala will make a lot more sense once we do

Statements

are units of code that do not return a value



```
scala> val radius = 10
radius: Int = 10
```

Statements

are units of code that do not return a value

```
scala> println("hello world")
hello world
```

Statements v Expressions

are units of code that do not return a value

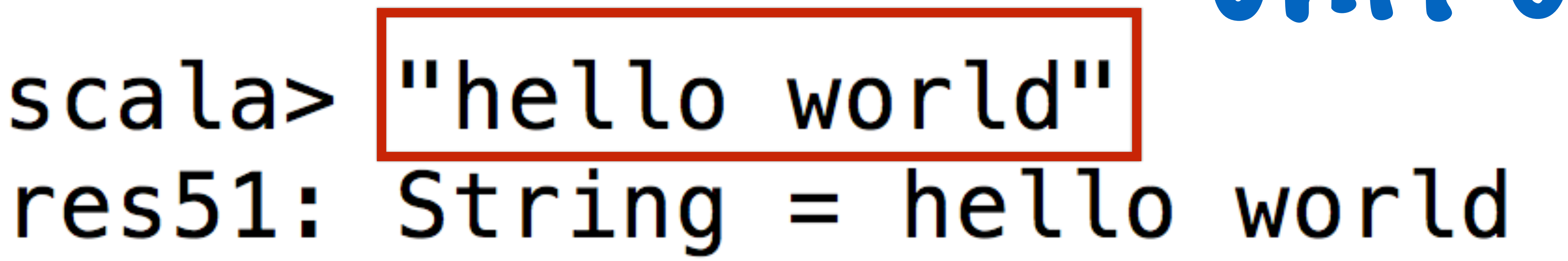
Expressions

are units of code that return a value

Expressions

are units of code that return a value

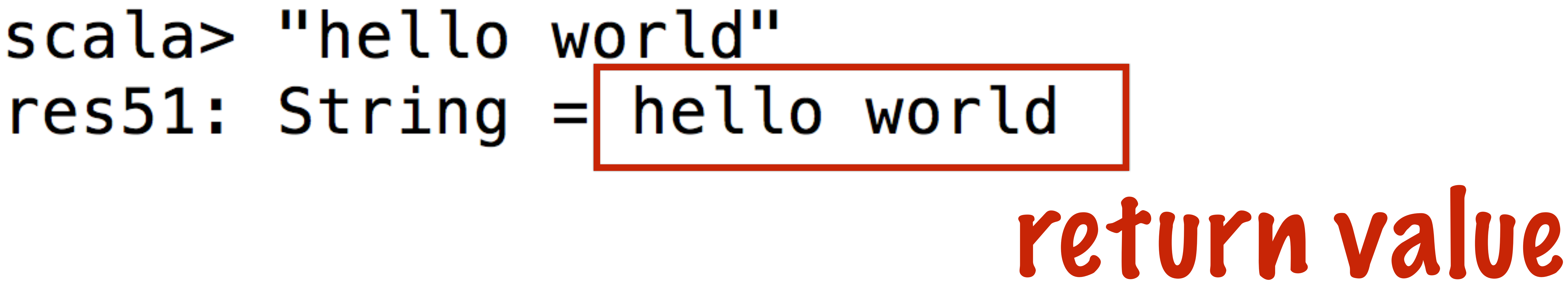
unit of code



```
scala> "hello world"
res51: String = hello world
```

Expressions

are units of code that return a value

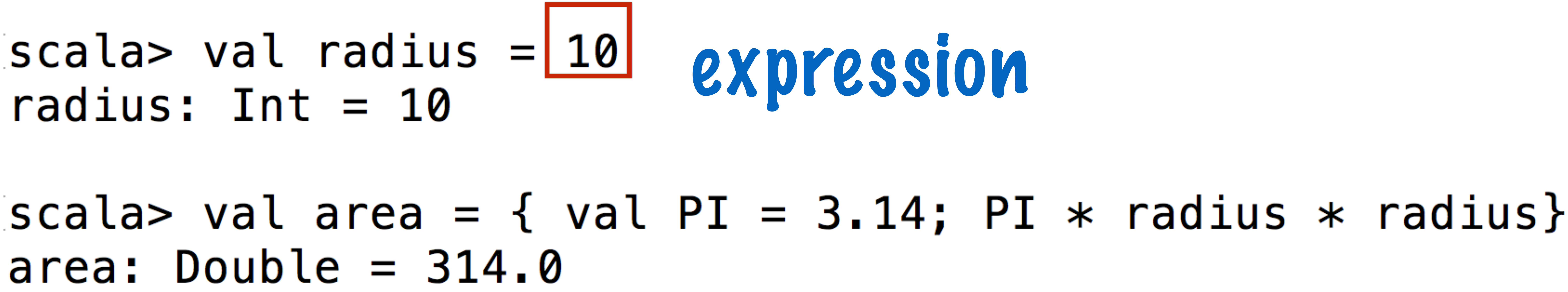


```
scala> "hello world"
res51: String = hello world
```

return value

Expressions

are units of code that return a value

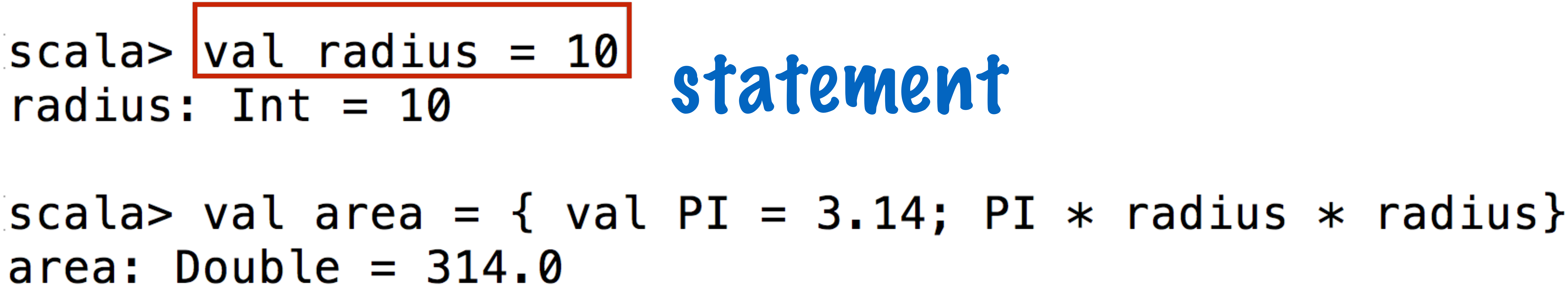


```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

expression

Expressions

are units of code that return a value

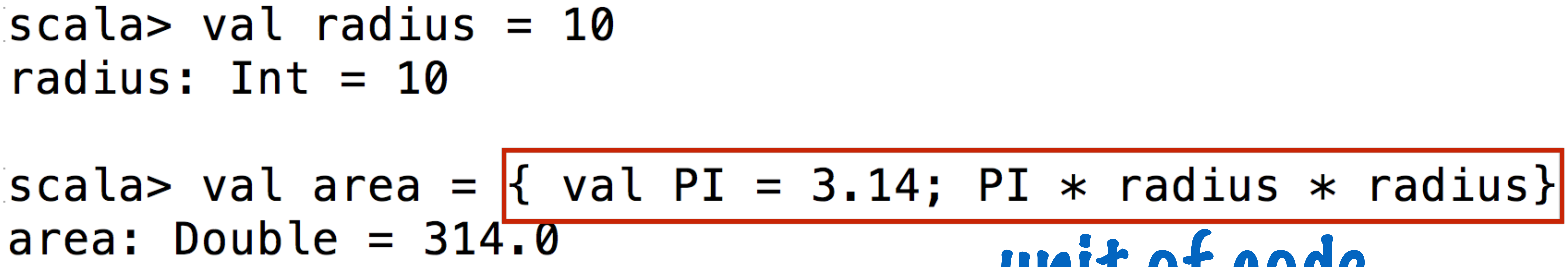


```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

statement

Expressions

are units of code that return a value

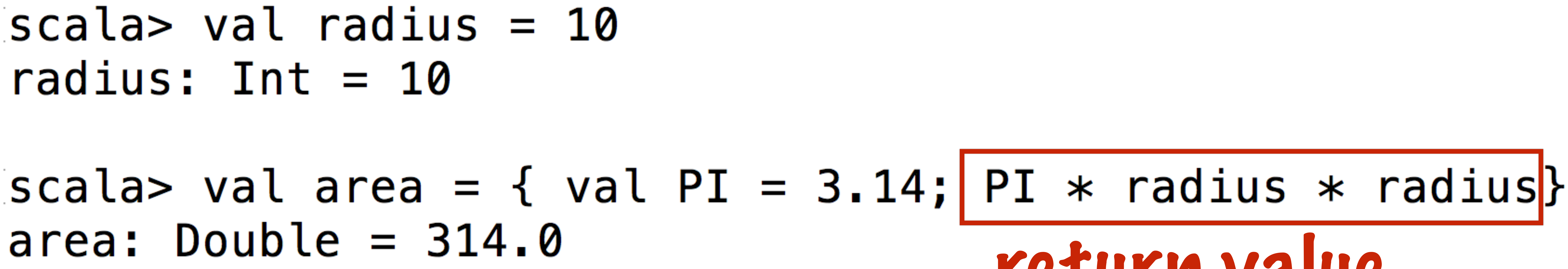


```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

unit of code

Expressions

are units of code that return a value

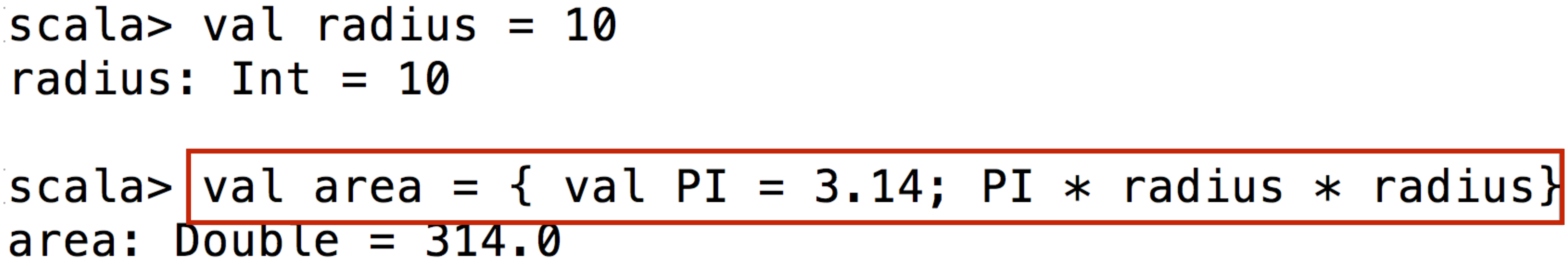


```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

return value

Expressions

are units of code that return a value

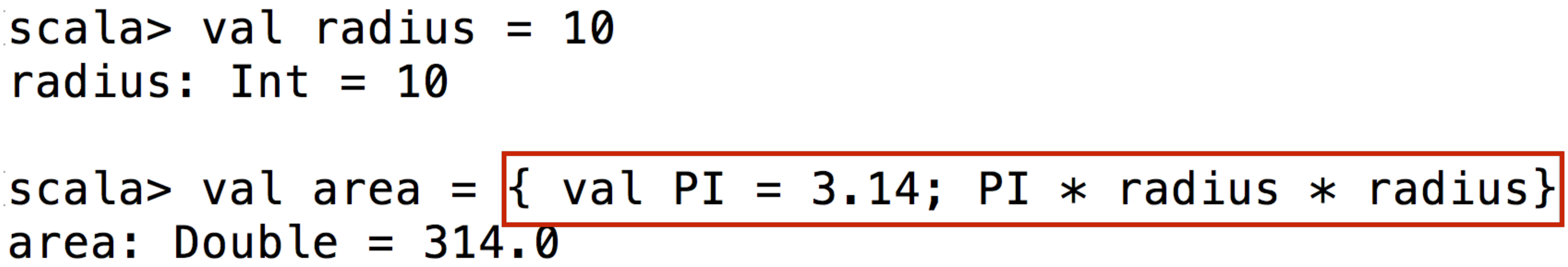


```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

statement

Expressions

are units of code that return a value



```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

"expression block"

A bit of code enclosed in {}

Expressions

are units of code that return a value

```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

The last expression in a block is the return value for the entire block

"expression block"

A bit of code enclosed in {}

“Expression block”

A bit of code enclosed in {}

The last expression in a block is the return value for the entire block

The last expression in a block is the return value for the entire block

The last expression in a block is the return value for the entire block

“Expression block”

A bit of code enclosed in {}

The last expression in a block is the return value for the entire block

Expressions

are units of code that return a value

```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

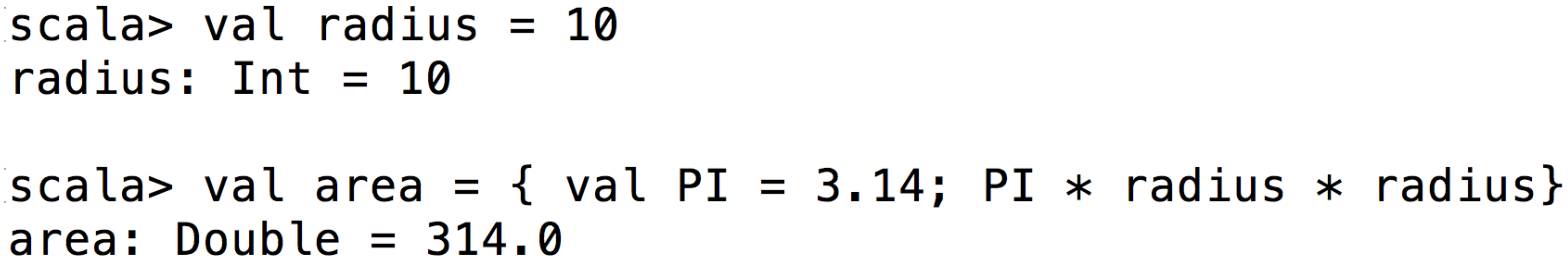
The last expression in a block is the return value for the entire block

"expression block"

A bit of code enclosed in {}

Expressions

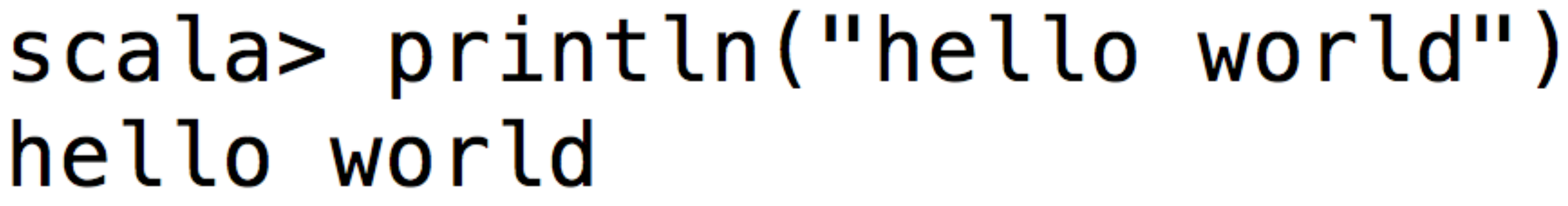
are units of code that return a value



```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

Statements

are units of code that do not return a value



```
scala> println("hello world")
hello world
```

Statements v Expressions

Why does this matter?

Because many constructs that are statements in Java are expressions in Scala

Many constructs that are statements in Java are expressions in Scala

`if/else`

`for` loops

(But not `while` loops)

`match`

(Think Java `case` statement, but way more powerful)

Many constructs that are statements in Java are expressions in Scala

`if/else`

`for` loops

(But not `while` loops)

`match`

(Think Java `case` statement, but way more powerful)

By changing these to expressions (with return values) Scala makes functional programming far easier

By changing these to expressions (with return values) Scala makes functional programming far easier

Why? because expressions can be "composed" (chained to each other) - but statements can't!

"composition of functions"

Example 10

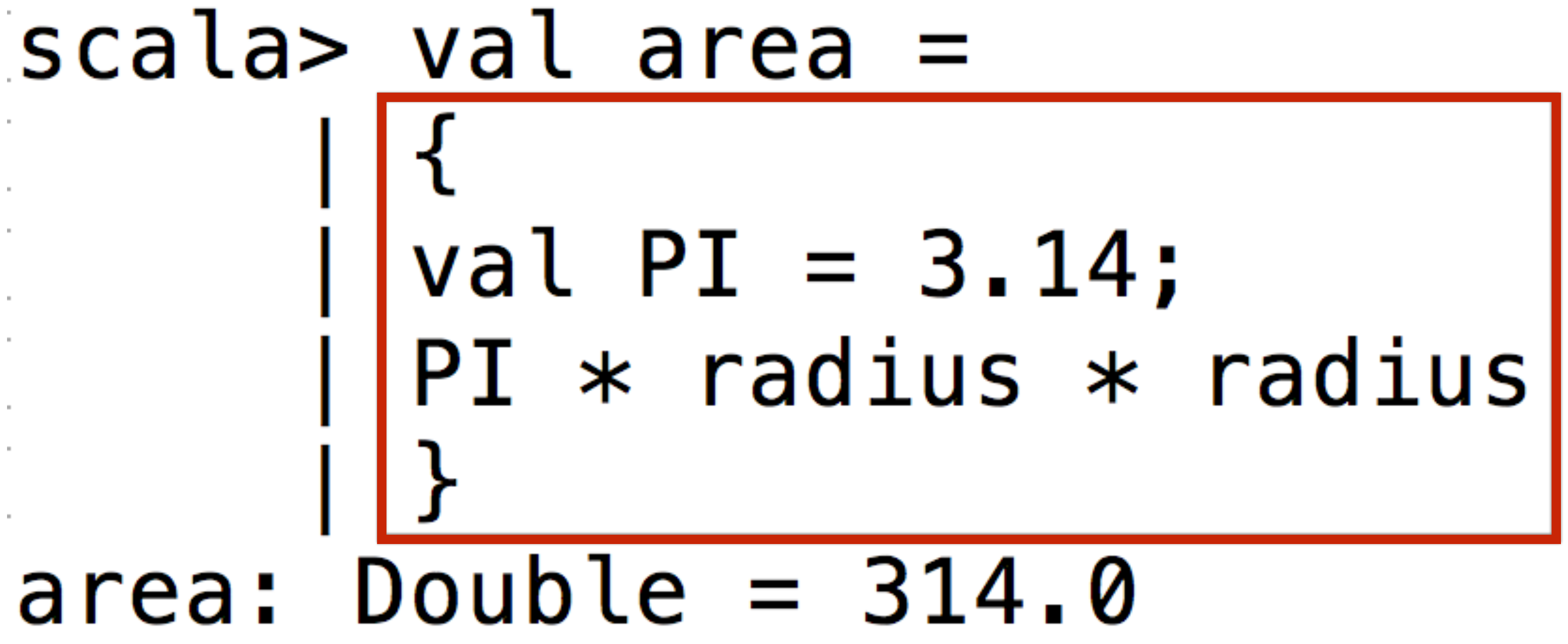
Defining Values and Variables via Expressions

Defining Values and Variables via Expressions

An expression is a unit of code that returns a value

We can take that value and use it to define a variable or a value

An expression is a unit of code that returns a value

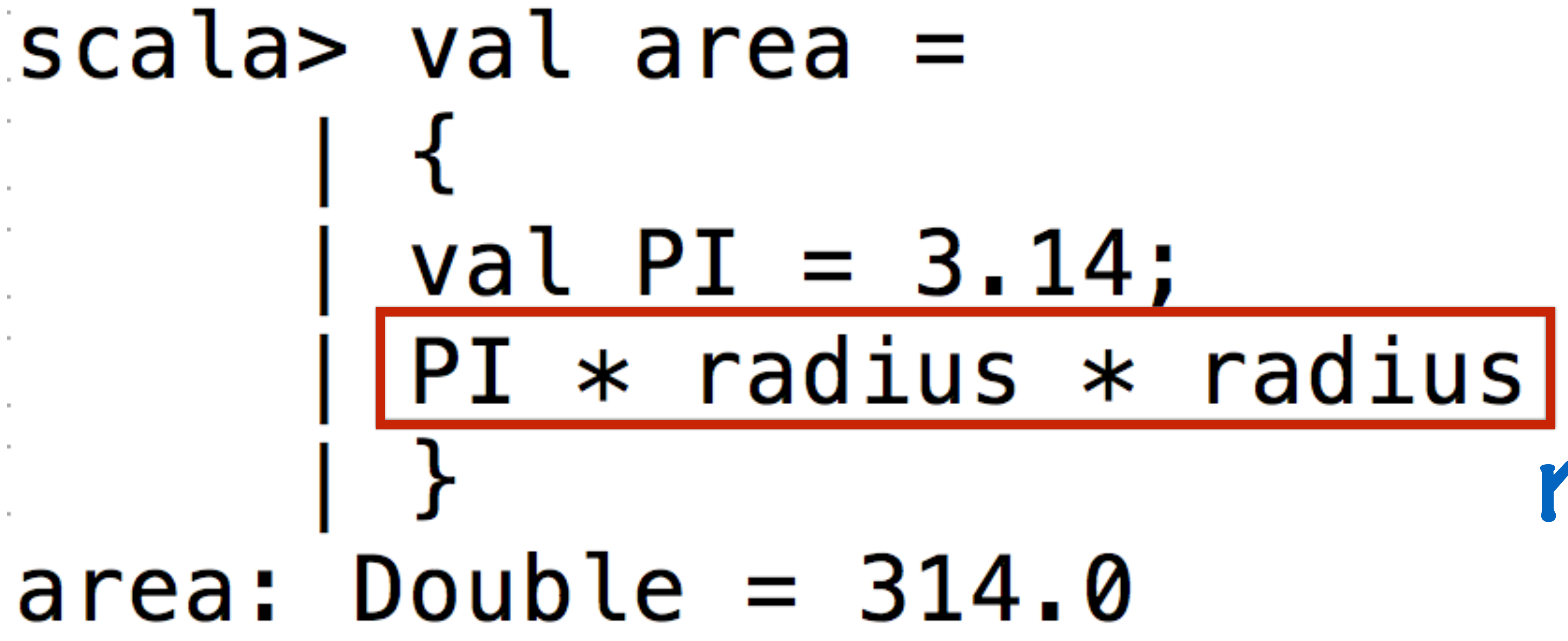


```
scala> val area =
     | {
     | val PI = 3.14;
     | PI * radius * radius
     | }
area: Double = 314.0
```

expression block

We can take that value and use it to define a variable or a value

An expression is a unit of code that returns a value



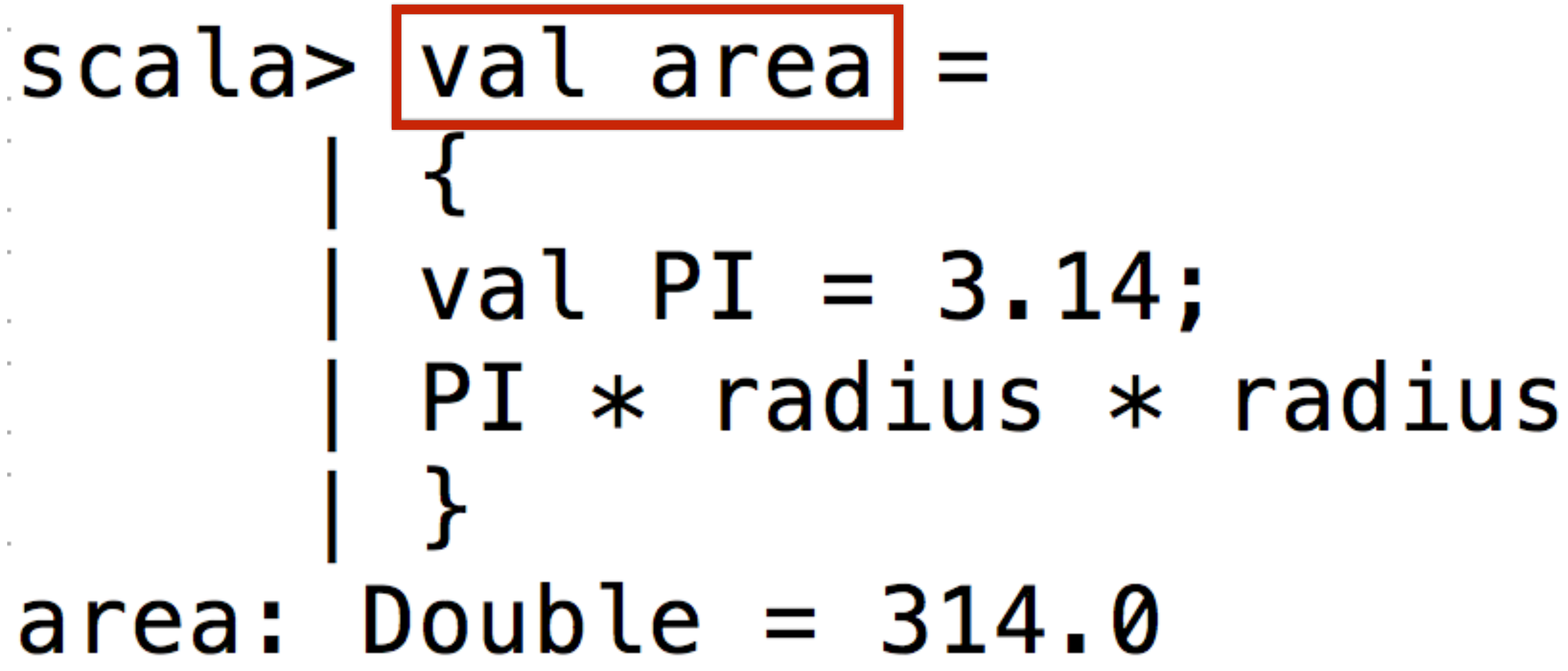
```
scala> val area =
     | {
     | val PI = 3.14;
     | PI * radius * radius
     | }
area: Double = 314.0
```

return value of that expression block

We can take that value and use it to define a variable or a value

An expression is a unit of code that
returns a value

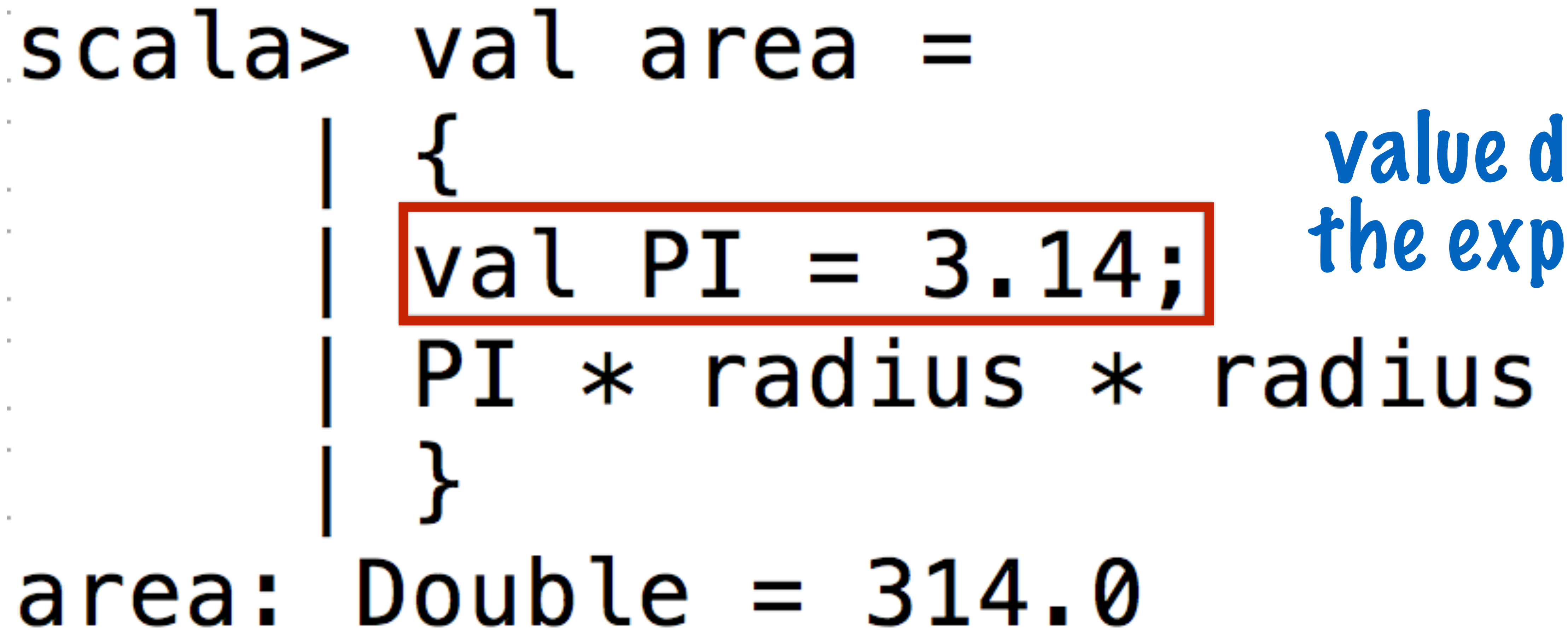
value defined using
the expression



```
scala> val area =
     | {
     | val PI = 3.14;
     | PI * radius * radius
     | }
area: Double = 314.0
```

We can take that value and use it to
define a variable or a value

An expression is a unit of code that returns a value

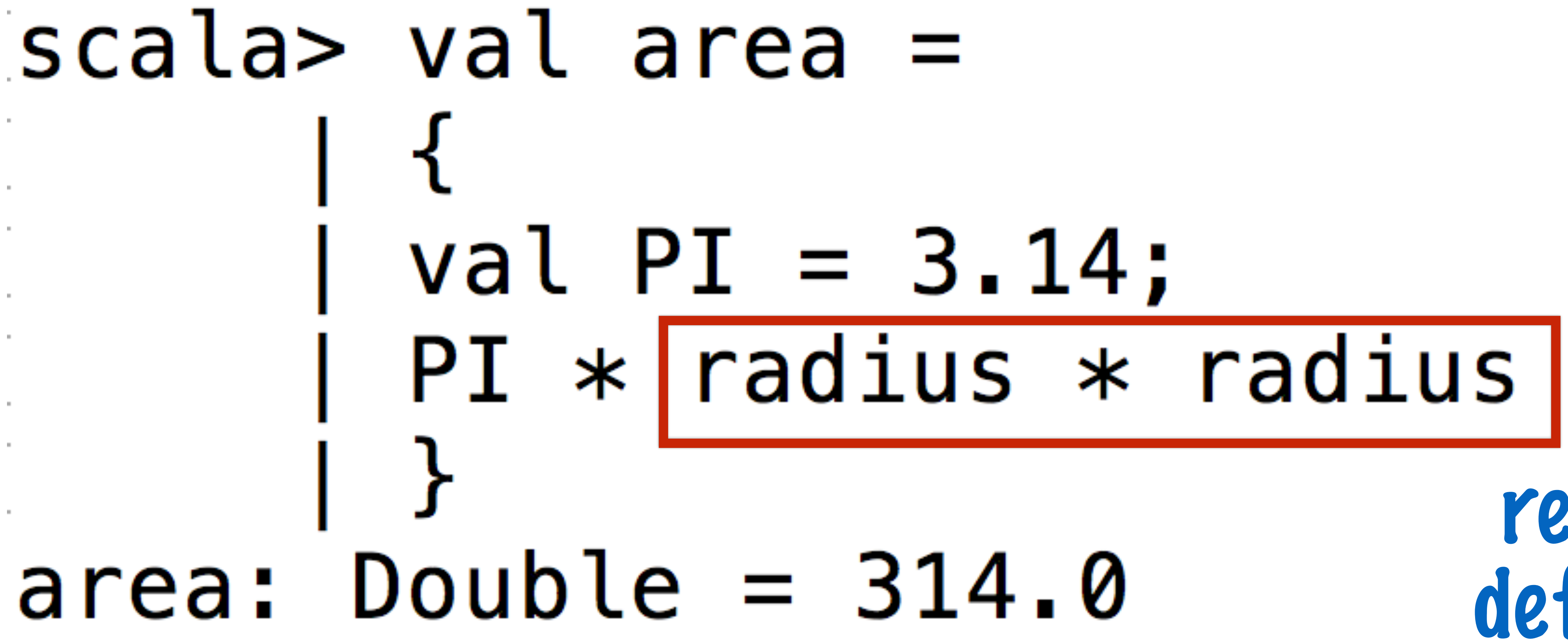


```
scala> val area =
     | {
     | val PI = 3.14;
     | PI * radius * radius
     | }
area: Double = 314.0
```

value defined inside the expression block

We can take that value and use it to define a variable or a value

An expression is a unit of code that returns a value

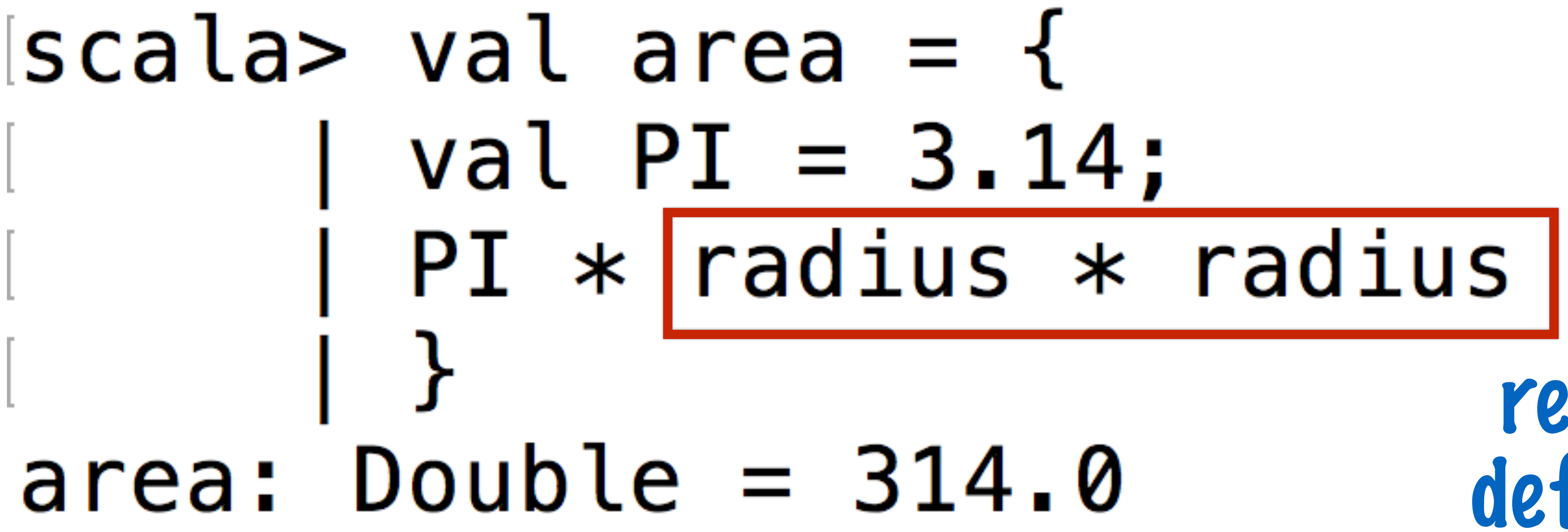


```
scala> val area =
     | {
     | val PI = 3.14;
     | PI * radius * radius
     | }
area: Double = 314.0
```

reference to previously defined value or variable

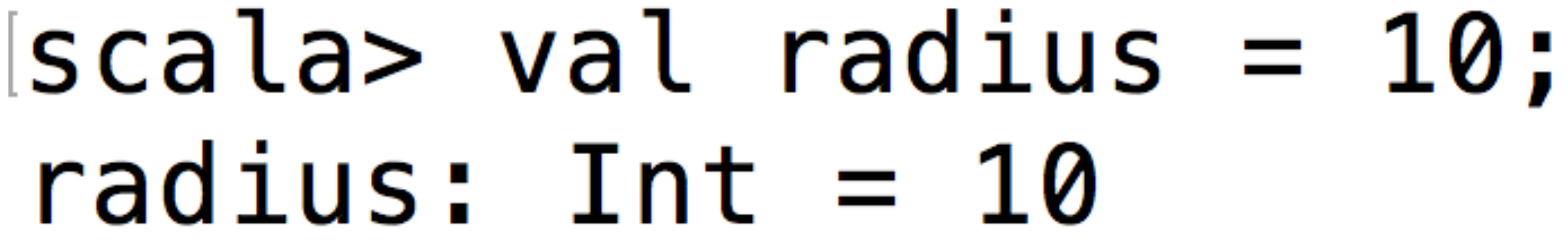
We can take that value and use it to define a variable or a value

An expression is a unit of code that returns a value



```
scala> val area = {
     | val PI = 3.14;
     | PI * radius * radius
     | }
area: Double = 314.0
```

reference to previously defined value or variable



```
scala> val radius = 10;
radius: Int = 10
```

We can take that value and use it to define a variable or a value

Defining Values and Variables via Expressions

An expression is a unit of code that returns a value

We can take that value and use it to define a variable or a value

Defining Values and Variables via Expressions

An expression is a unit of code that returns a value

Why does this matter?

We can take that value and use it to define a variable or a value

Defining Values and Variables via Expressions

Why does this matter?

Expressions are "r-values" which mean that they can be assigned and composed (chained)

Statements are not "r-values" - they just sit there and do their thing

Why does this matter?

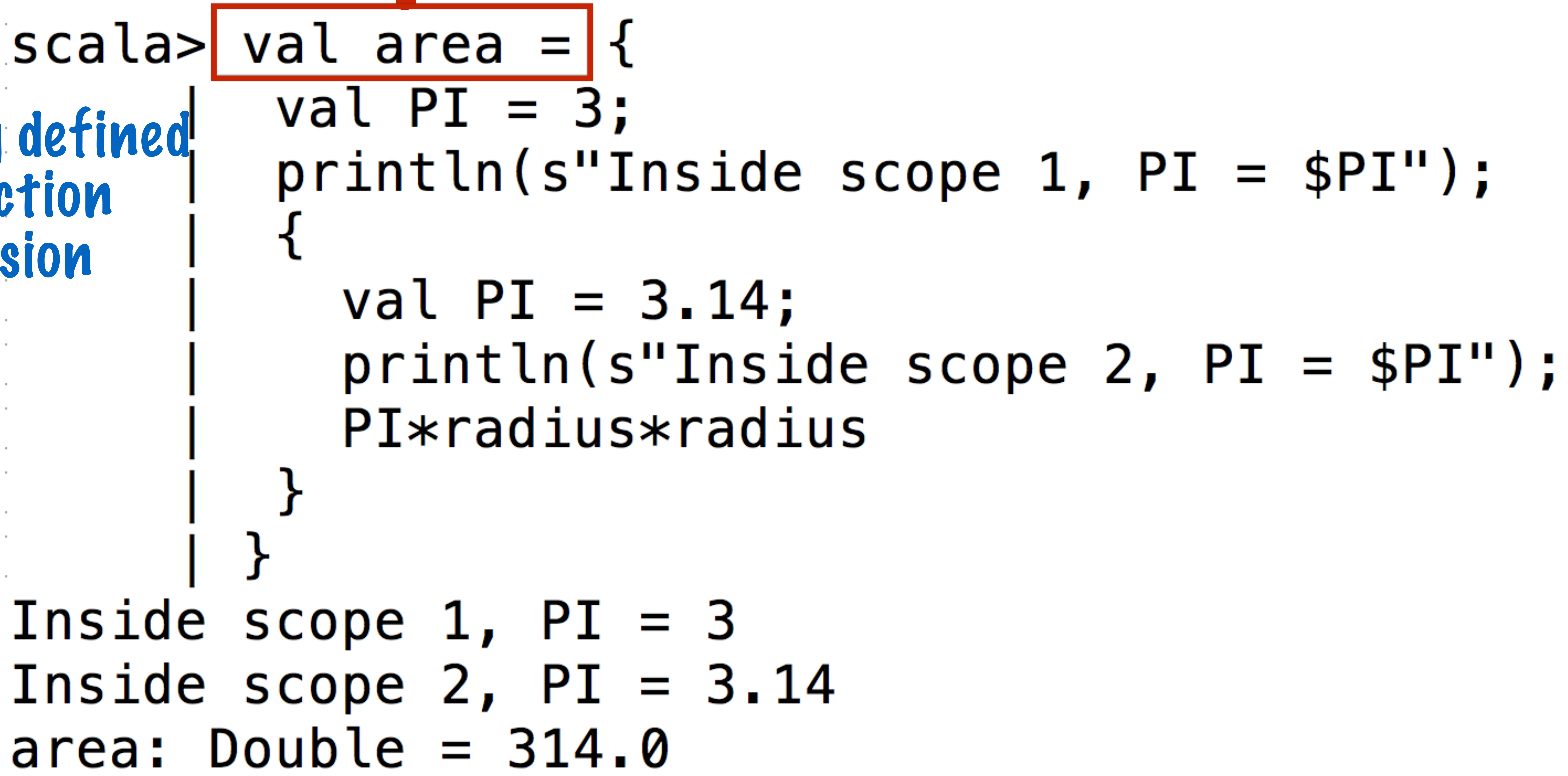
Expressions are “r-values” which mean that they can be assigned and composed (chained)

By expanding the possible r-values in code, Scala enables functional programming

Example 11

Nested Scopes in Expression Blocks

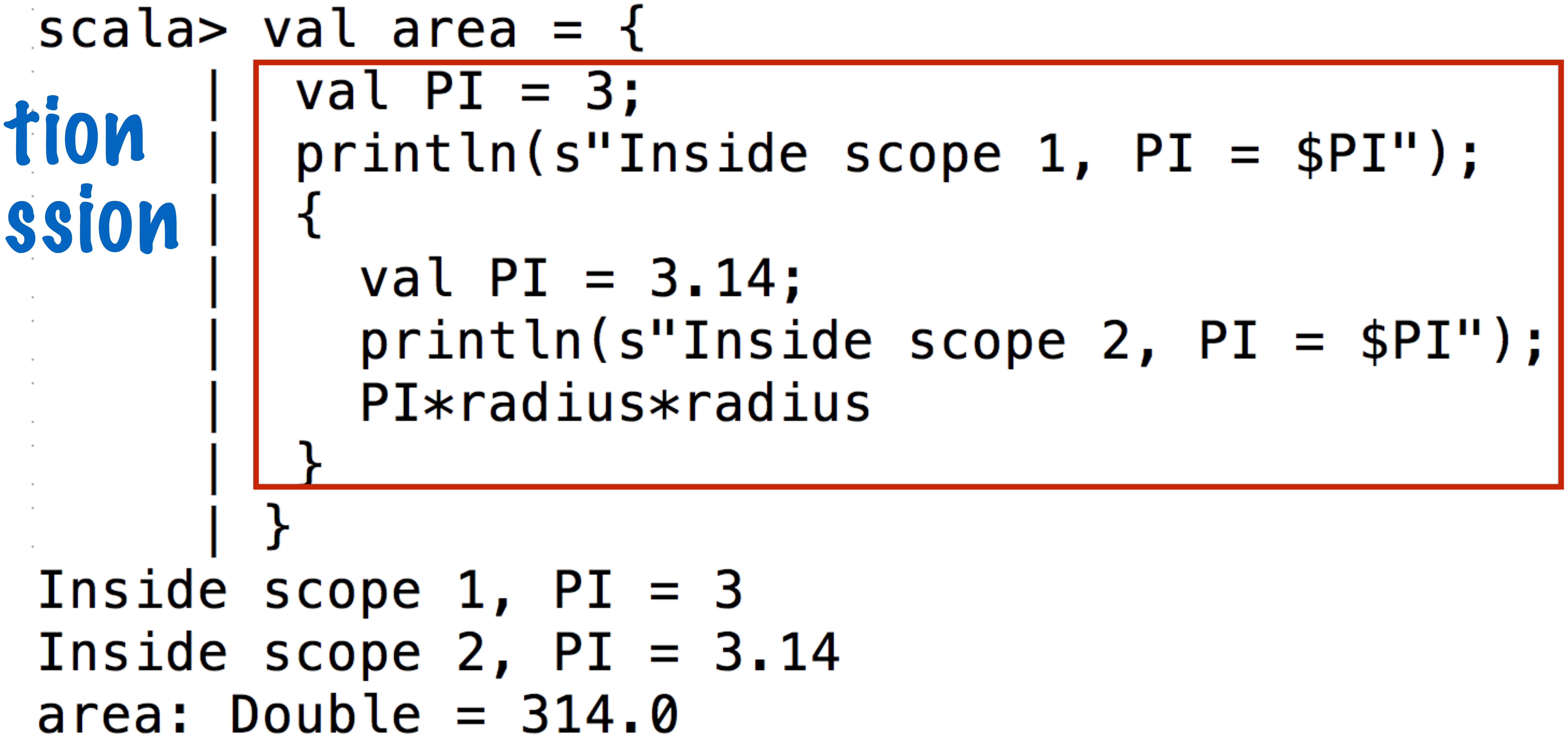
Nested Scopes in Expression Blocks



```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

value being defined
via function
expression

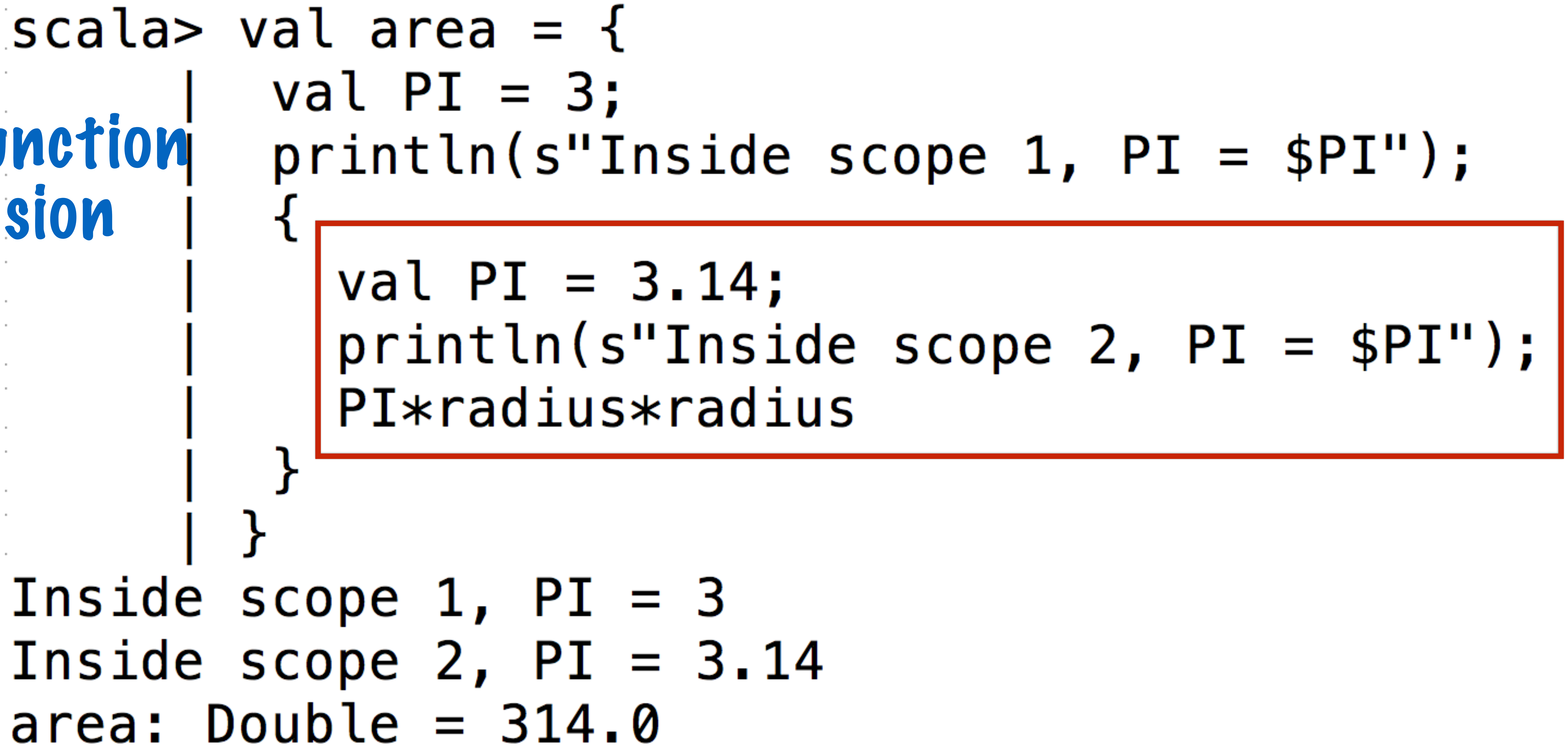
Nested Scopes in Expression Blocks



```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

function expression

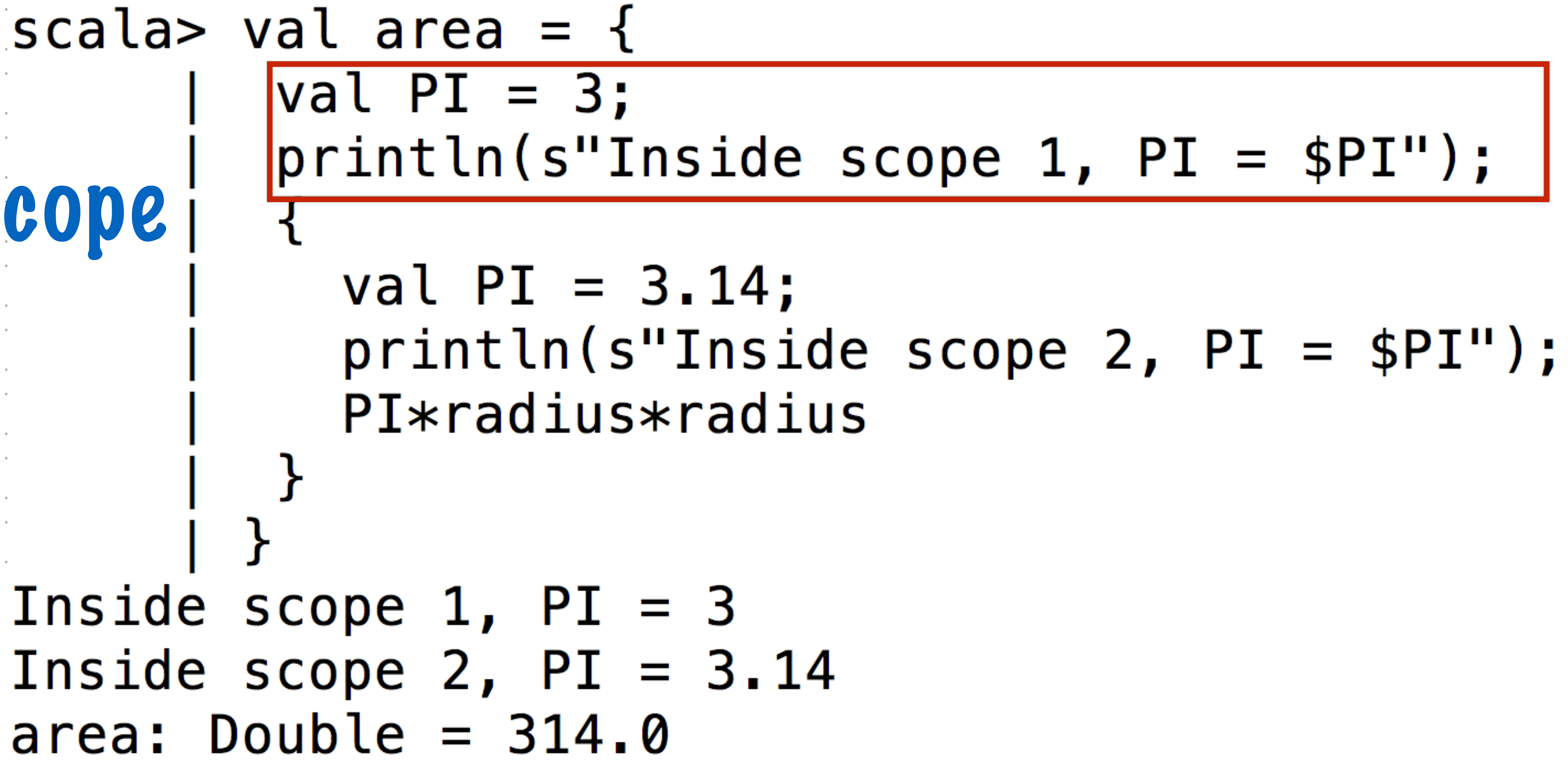
Nested Scopes in Expression Blocks



```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

nested function expression

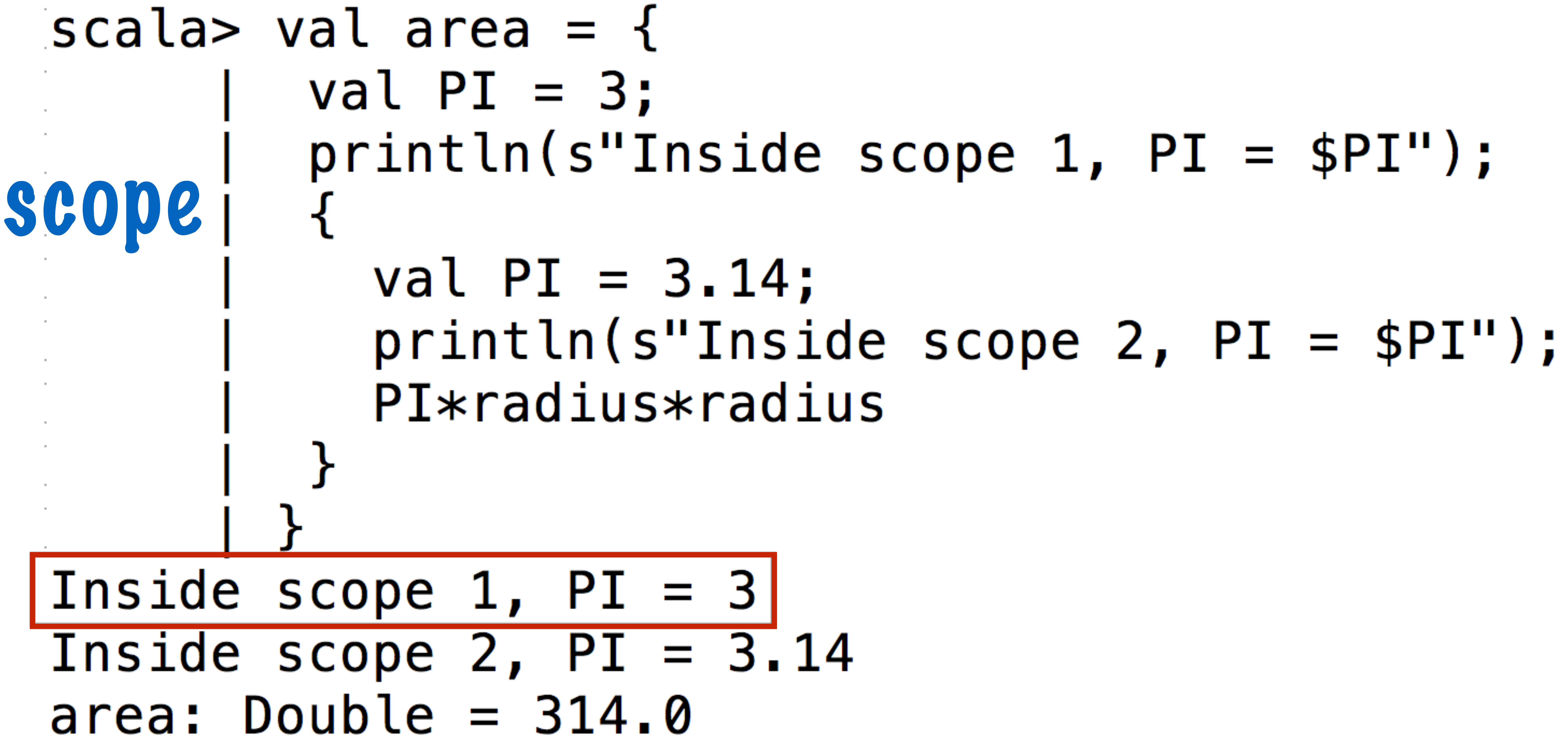
Nested Scopes in Expression Blocks



```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

outer scope

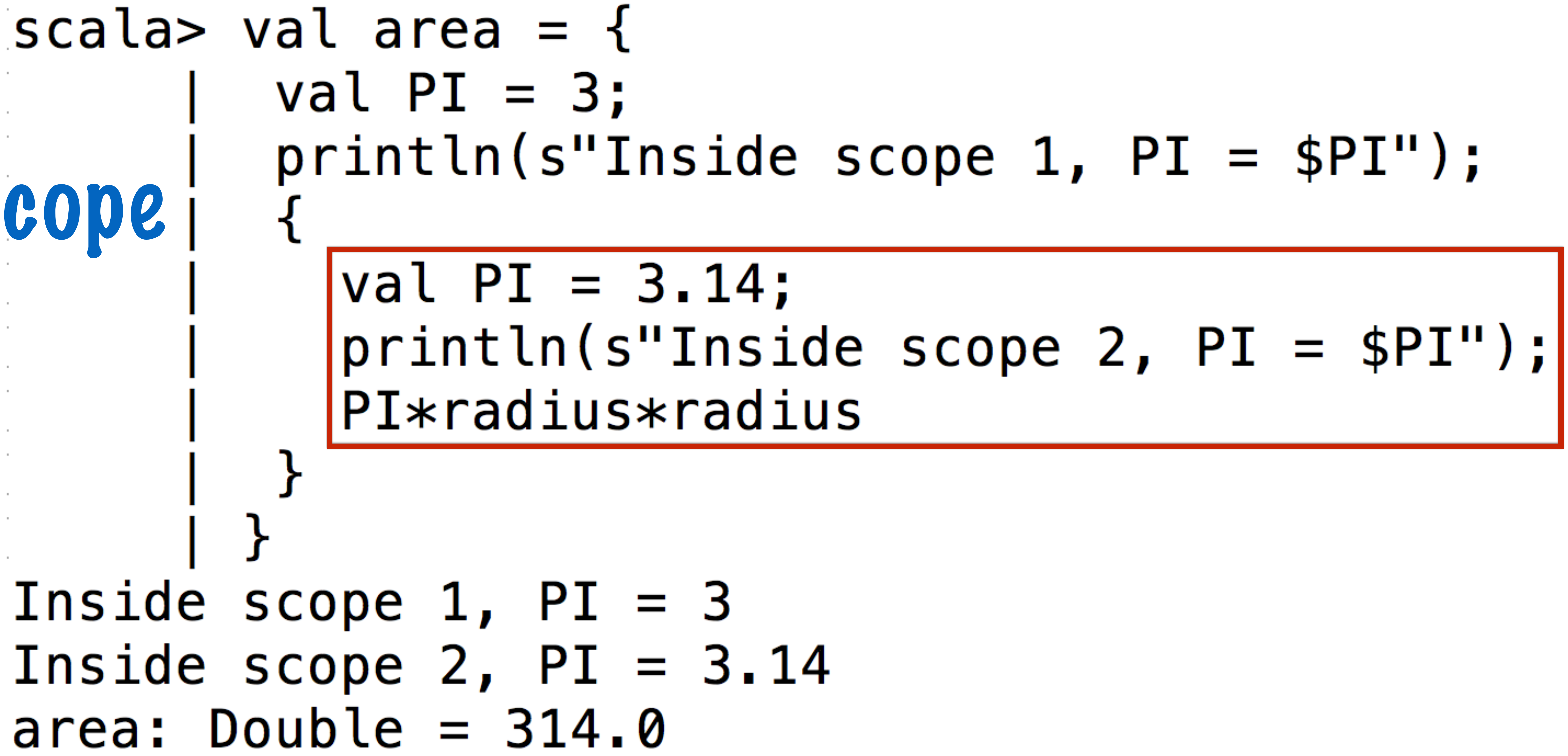
Nested Scopes in Expression Blocks



outer scope

```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

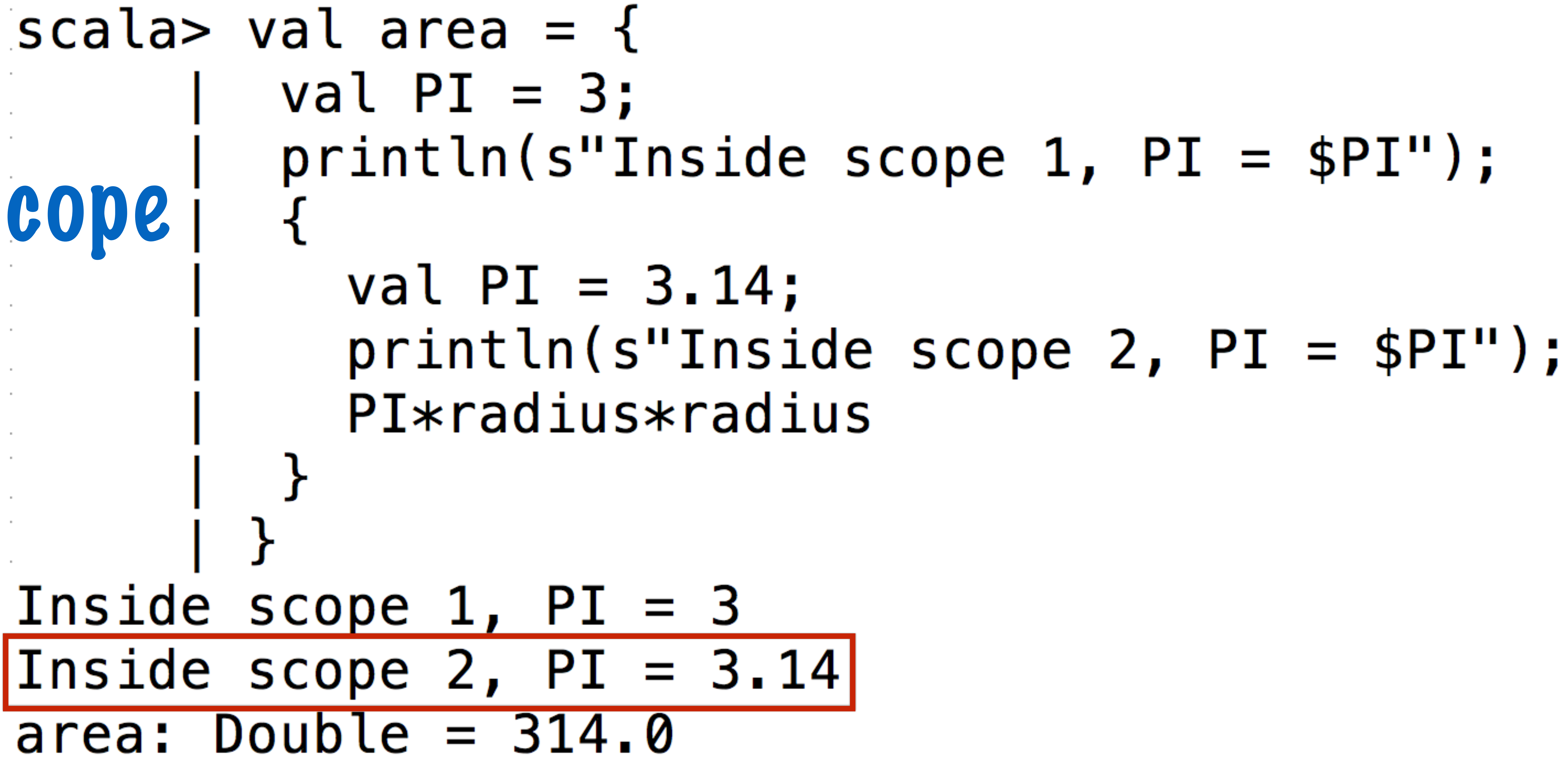
Nested Scopes in Expression Blocks



inner scope

```
scala> val area = {
     |  val PI = 3;
     |  println(s"Inside scope 1, PI = $PI");
     |  {
     |    val PI = 3.14;
     |    println(s"Inside scope 2, PI = $PI");
     |    PI*radius*radius
     |  }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

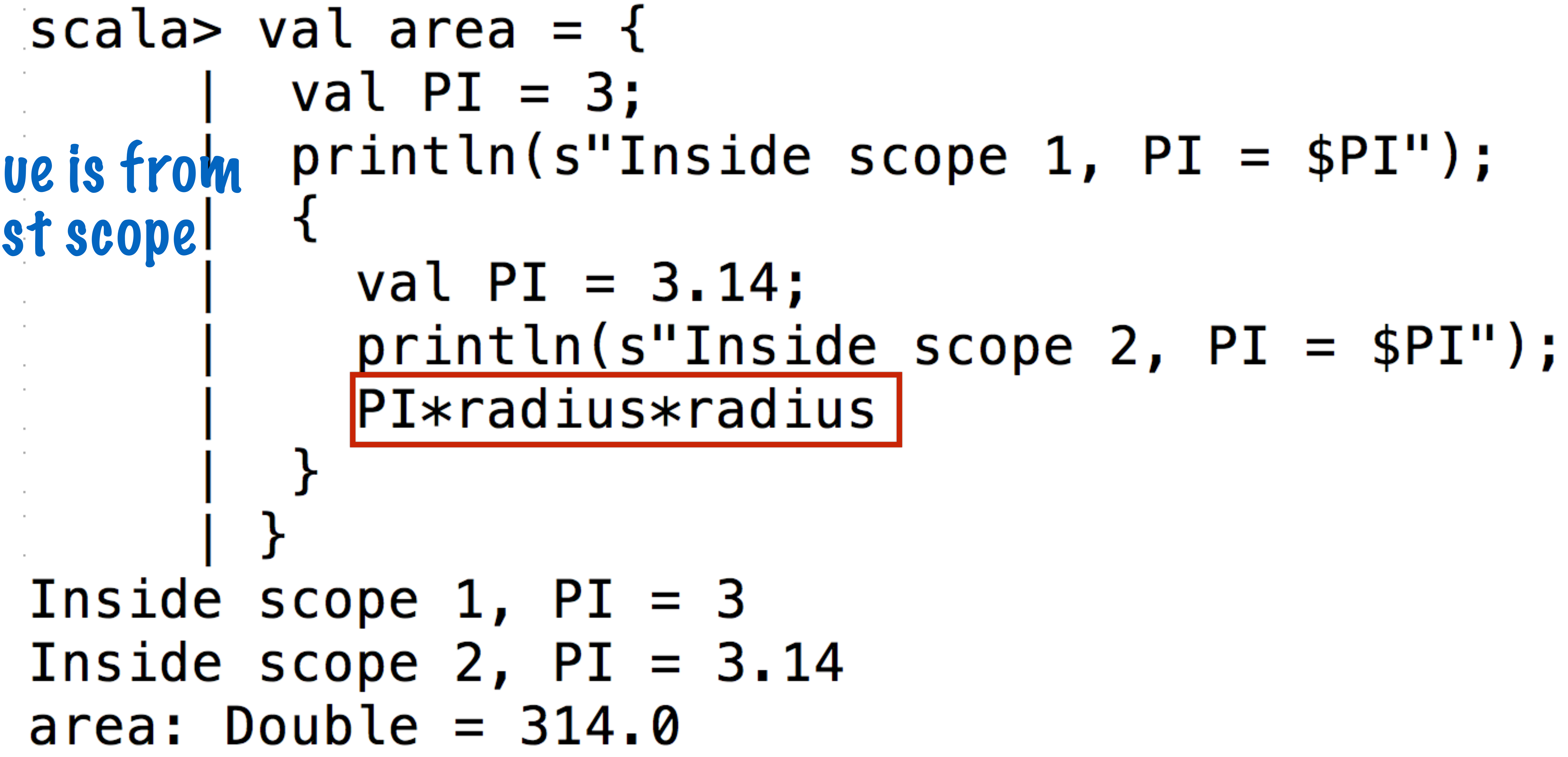
Nested Scopes in Expression Blocks



```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

inner scope

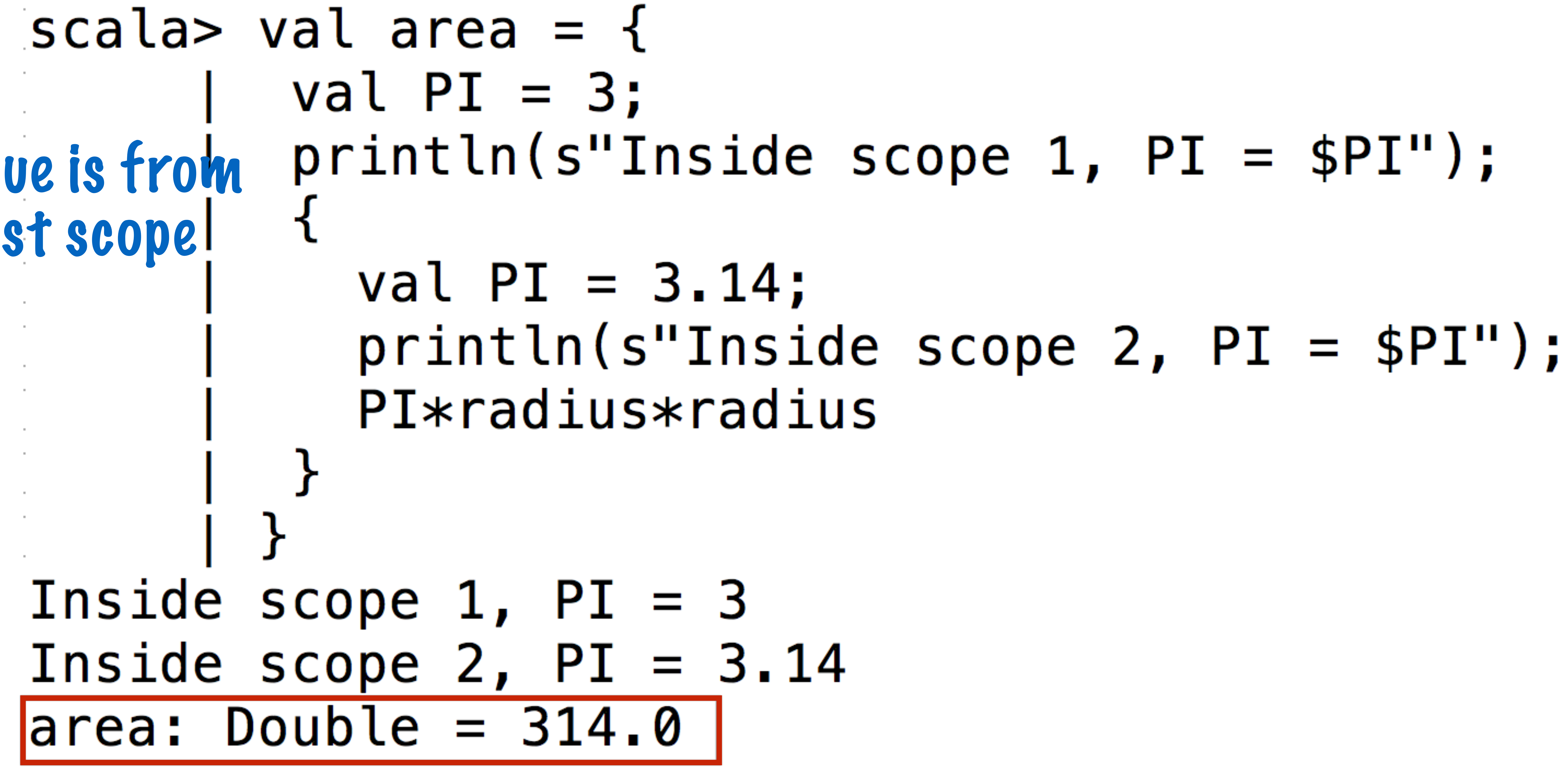
Nested Scopes in Expression Blocks



```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

return value is from innermost scope

Nested Scopes in Expression Blocks

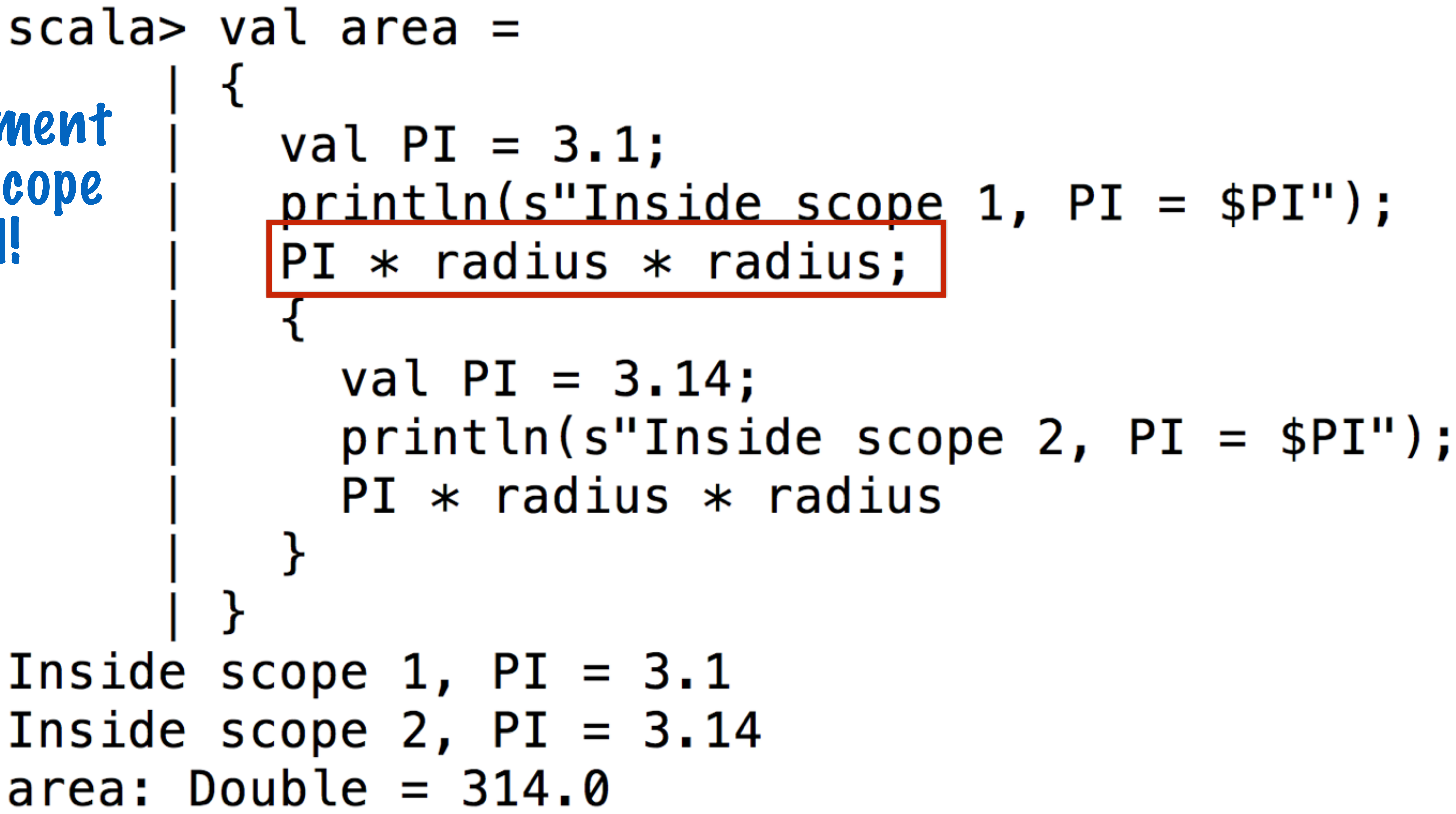


```
scala> val area = {
     |   val PI = 3;
     |   println(s"Inside scope 1, PI = $PI");
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI*radius*radius
     |   }
     | }
Inside scope 1, PI = 3
Inside scope 2, PI = 3.14
area: Double = 314.0
```

return value is from innermost scope

Nested Scopes in Expression Blocks

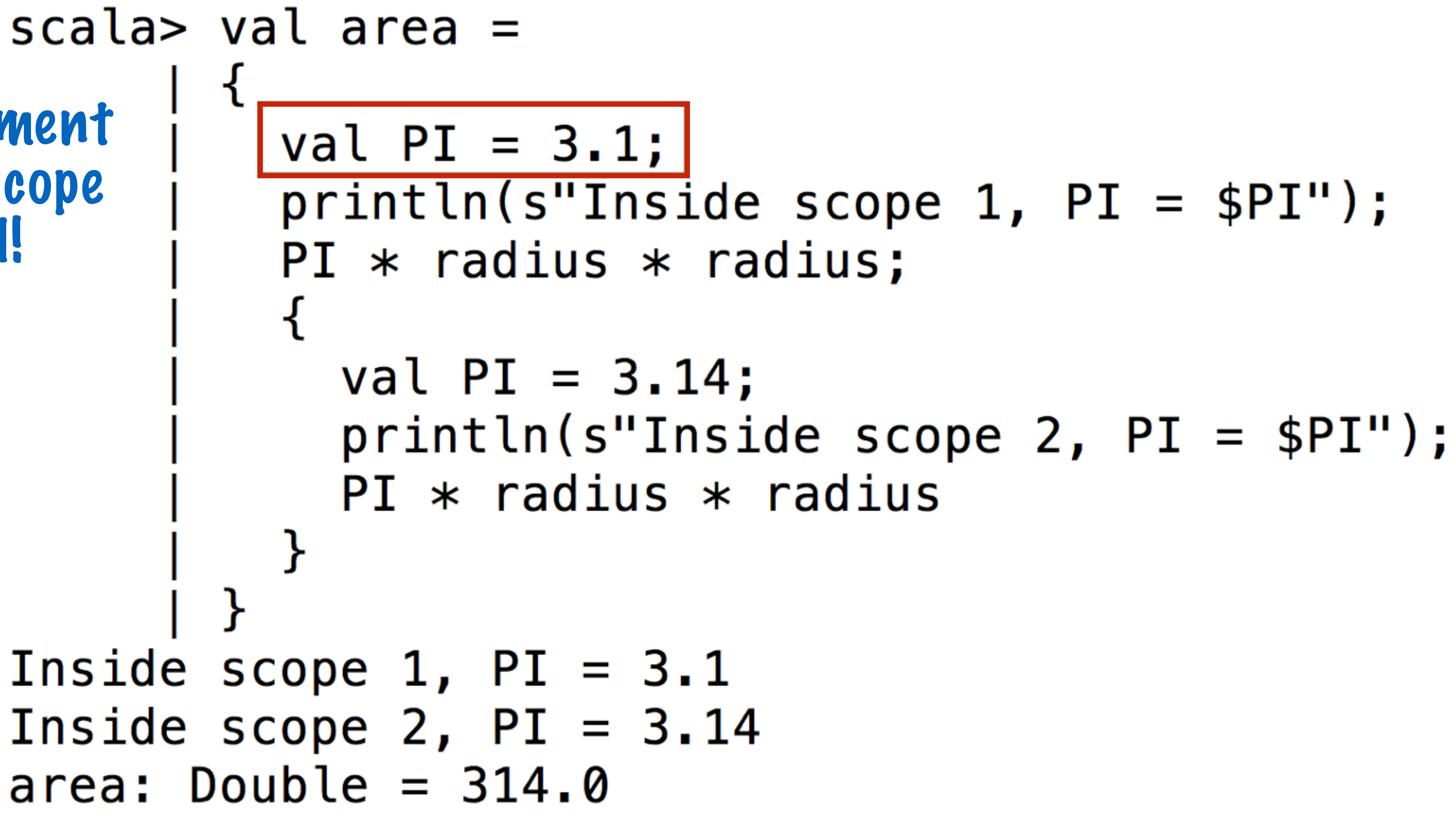
return statement
from outer scope
is ignored!



```
scala> val area =
     | {
     |   val PI = 3.1;
     |   println(s"Inside scope 1, PI = $PI");
     |   PI * radius * radius;
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI * radius * radius
     |   }
     | }
Inside scope 1, PI = 3.1
Inside scope 2, PI = 3.14
area: Double = 314.0
```

Nested Scopes in Expression Blocks

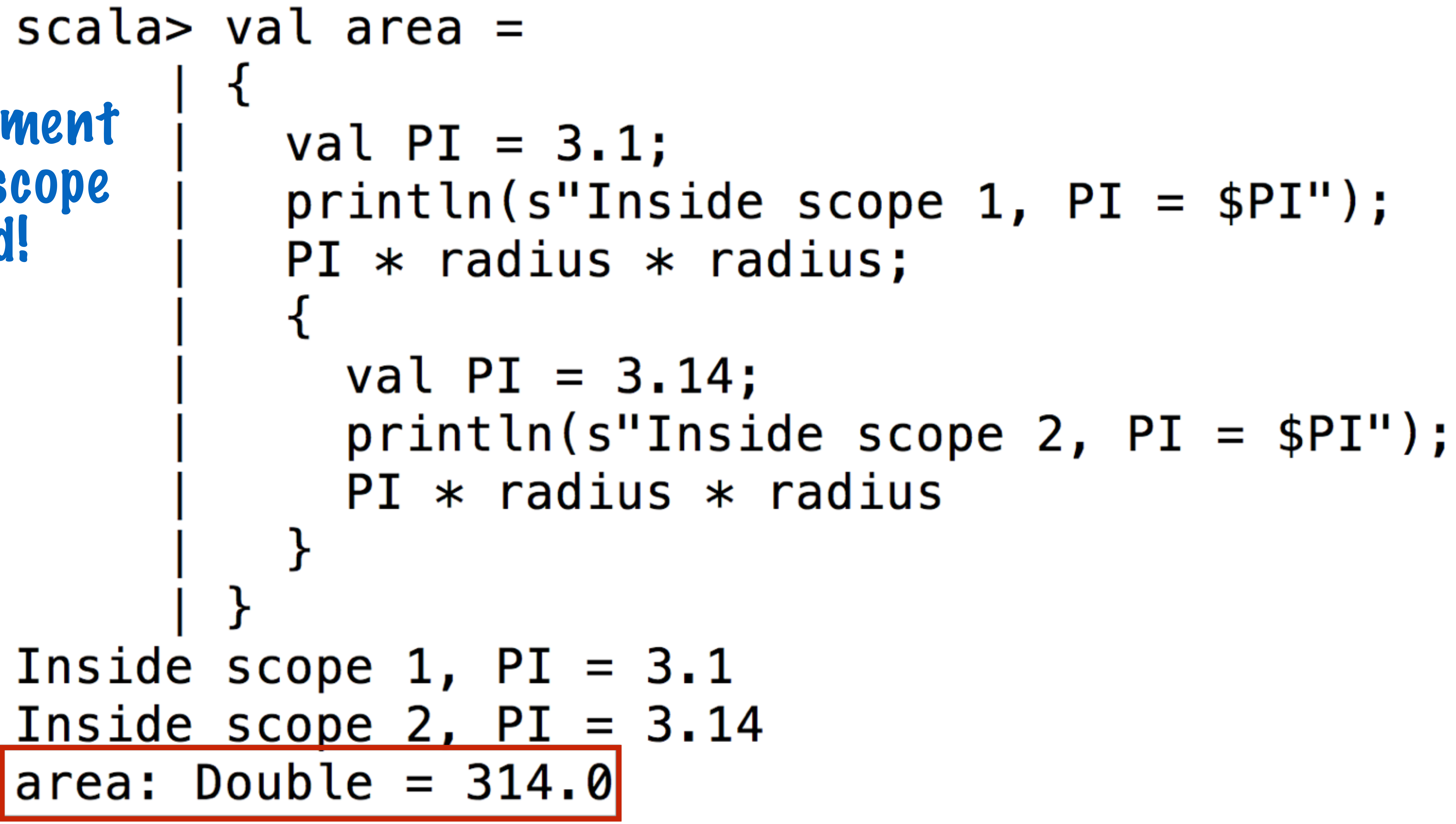
return statement from outer scope is ignored!



```
scala> val area =
     | {
     |   val PI = 3.1;
     |   println(s"Inside scope 1, PI = $PI");
     |   PI * radius * radius;
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI * radius * radius
     |   }
     | }
Inside scope 1, PI = 3.1
Inside scope 2, PI = 3.14
area: Double = 314.0
```

Nested Scopes in Expression Blocks

return statement from outer scope is ignored!



```
scala> val area =
     | {
     |   val PI = 3.1;
     |   println(s"Inside scope 1, PI = $PI");
     |   PI * radius * radius;
     |   {
     |     val PI = 3.14;
     |     println(s"Inside scope 2, PI = $PI");
     |     PI * radius * radius
     |   }
     | }
Inside scope 1, PI = 3.1
Inside scope 2, PI = 3.14
area: Double = 314.0
```

Nested Scopes in Expression Blocks

Why does this matter?

Because functions in Scala are merely named, reusable expression blocks

if you can have nested expression blocks, you can have nested functions too..

Because functions in Scala are merely named, reusable expression blocks

Nested functions

"First Class Functions"

Closures

These crucial Scala language features depend on nested scopes in expression blocks!

Example 12

`If/Else` expression blocks

If/Else expression blocks

Java, C#, C, C++ have If/Else **statements**

Scala has If/Else **expressions**

(like Excel, btw)

If/Else expression blocks

```
if (boolean expression)
    { expression block #1 }
else
    { expression block #2}
```

The entire if/else construct is an expression block!

If/Else expression blocks

expression block!

```
if (boolean expression)
    { expression block #1 }
else
    { expression block #2}
```

If/Else expression blocks

expression block!

```
if (boolean expression) true?
    { expression block #1 }
else
    { expression block #2}
```

return value of entire block depends on the return value of the boolean expression

If/Else expression blocks

expression block!

```
if (          true          )
   { expression block #1 }
else
   { expression block #2}
```

return value of expression block #1

If/Else expression blocks

expression block!

```
if (boolean expression) false?
    { expression block #1 }
else
    { expression block #2}
```

return value of entire block depends on the return value of the boolean expression

If/Else expression blocks

expression block!

```
if (        false          )
   { expression block #1 }
else
   { expression block #2}
```

return value of expression block #2

If/Else expression blocks

expression block!

if (boolean expression)
{ expression block #1 }

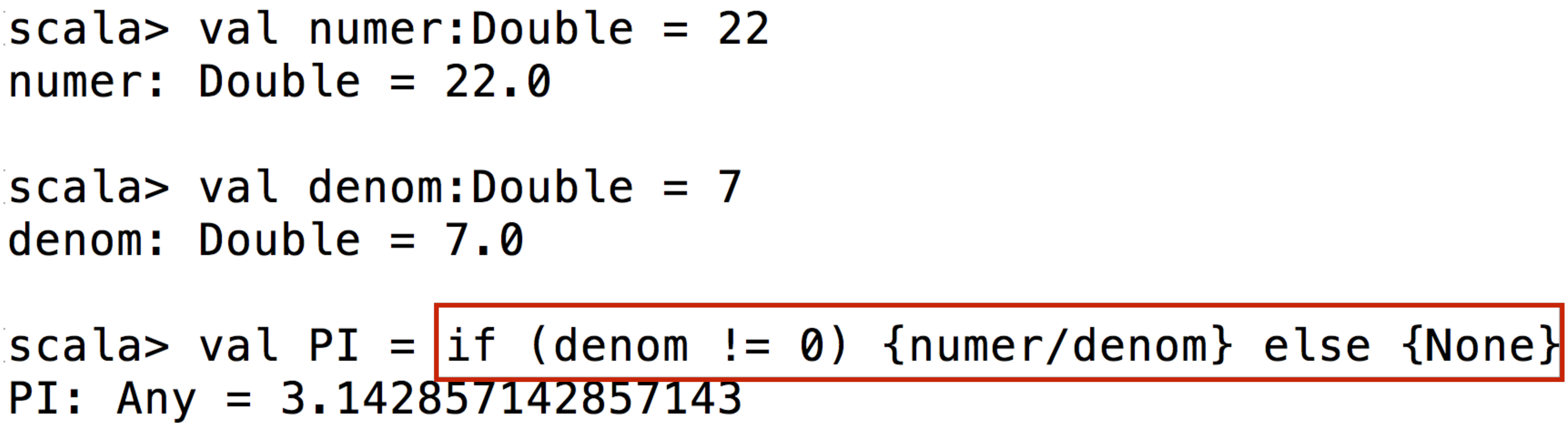
Btw, an if without an else will return Nothing if the boolean expression evaluates to false

If/Else expression blocks

```
if (boolean expression)
    { expression block #1 }
else
    { expression block #2}
```

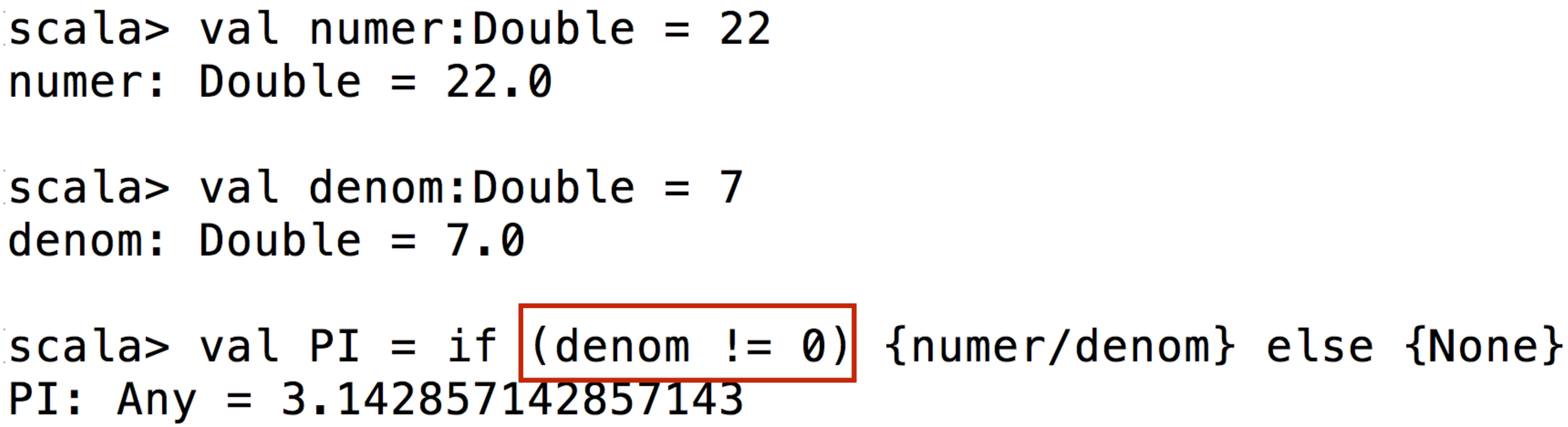
The entire if/else construct is an expression block!

If/Else expression blocks



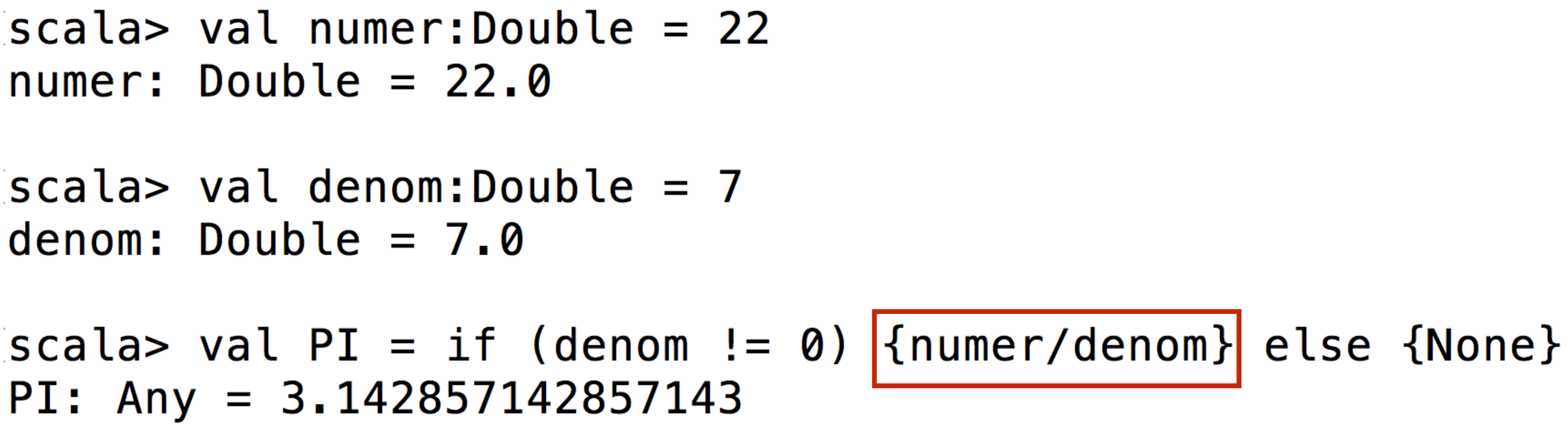
```
scala> val numer:Double = 22
numer: Double = 22.0

scala> val denom:Double = 7
denom: Double = 7.0

scala> val PI = if (denom != 0) {numer/denom} else {None}
PI: Any = 3.142857142857143
```

If/Else expression blocks



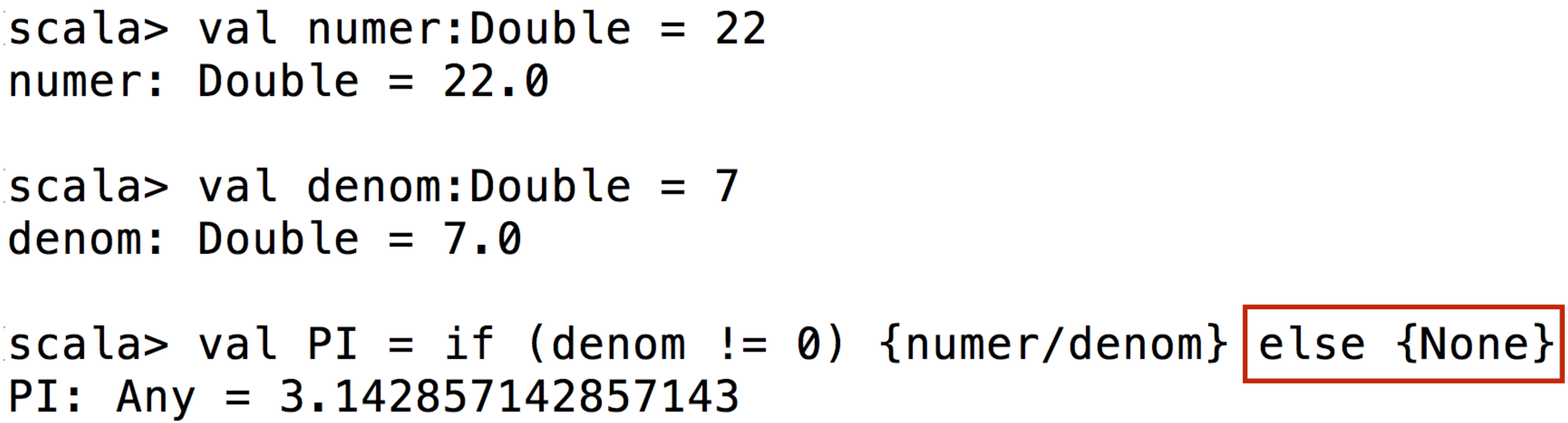
```
scala> val numer:Double = 22
numer: Double = 22.0

scala> val denom:Double = 7
denom: Double = 7.0

scala> val PI = if (denom != 0) {numer/denom} else {None}
PI: Any = 3.142857142857143
```

If/Else expression blocks



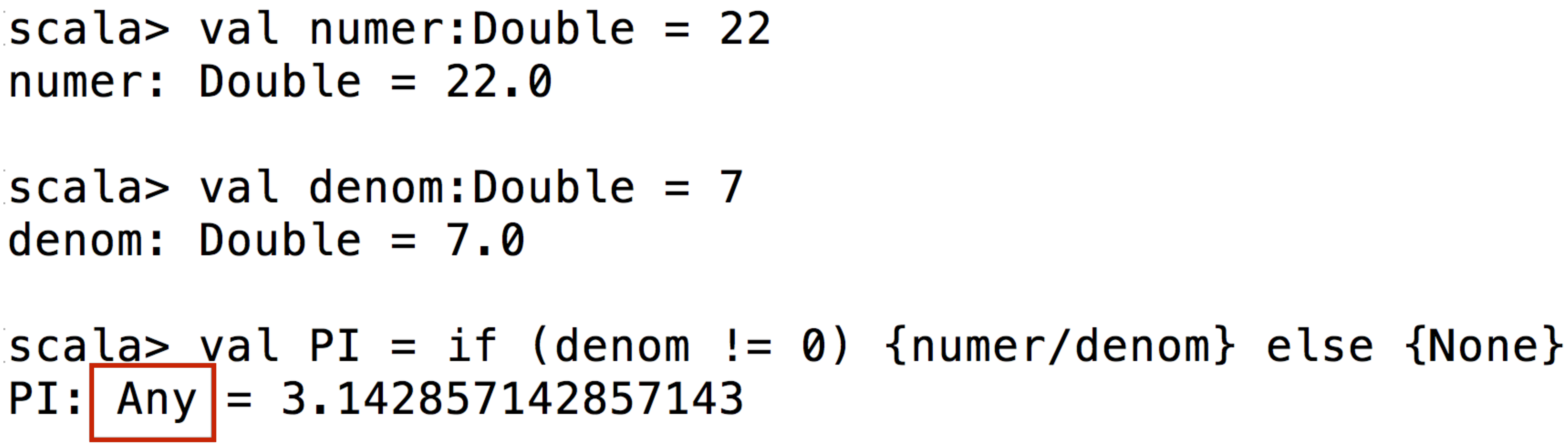
```
scala> val numer:Double = 22
numer: Double = 22.0

scala> val denom:Double = 7
denom: Double = 7.0

scala> val PI = if (denom != 0) {numer/denom} else {None}
PI: Any = 3.142857142857143
```

If/Else expression blocks



```
scala> val numer:Double = 22
numer: Double = 22.0

scala> val denom:Double = 7
denom: Double = 7.0

scala> val PI = if (denom != 0) {numer/denom} else {None}
PI: Any = 3.142857142857143
```

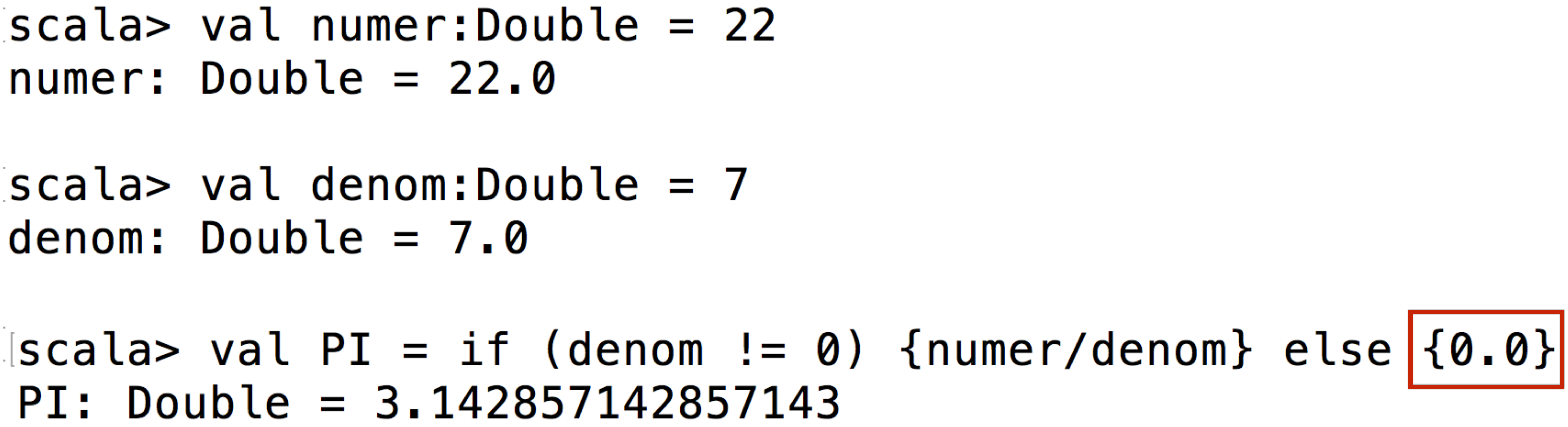
If/Else expression blocks



```
scala> val numer:Double = 22
numer: Double = 22.0

scala> val denom:Double = 7
denom: Double = 7.0

scala> val PI = if (denom != 0) {numer/denom} else {None}
PI: Any = 3.142857142857143
```

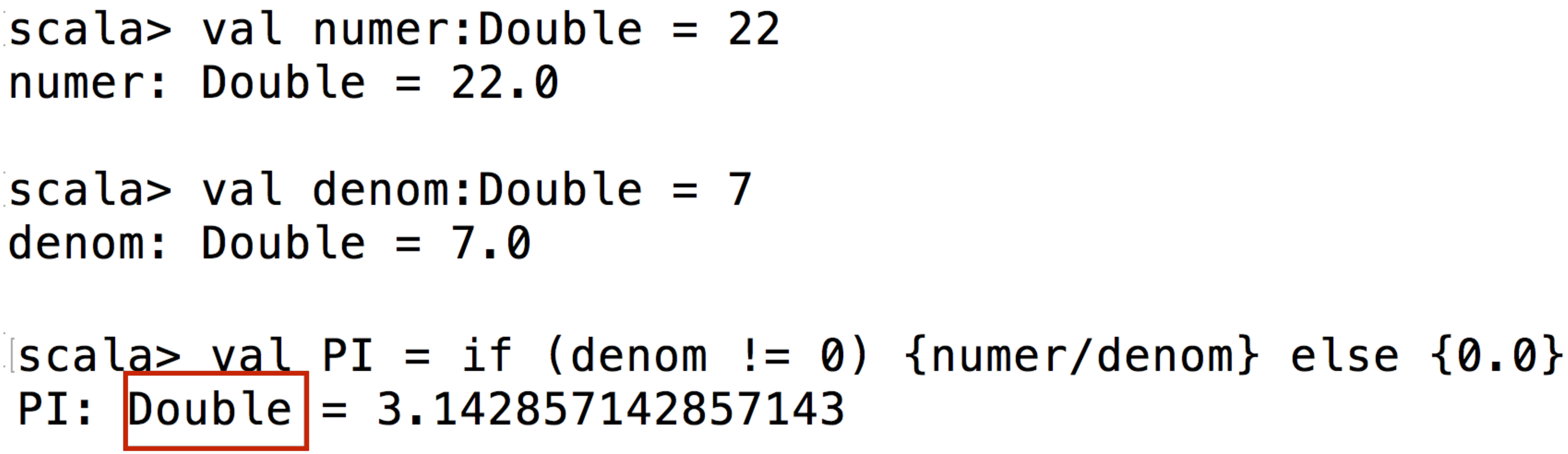
Type inference at work!

If/Else expression blocks



```
scala> val numer:Double = 22
numer: Double = 22.0

scala> val denom:Double = 7
denom: Double = 7.0

scala> val PI = if (denom != 0) {numer/denom} else {0.0}
PI: Double = 3.142857142857143
```

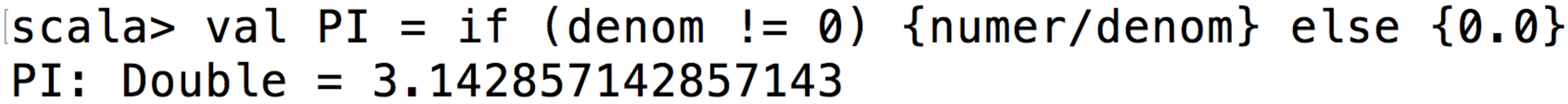
If/Else expression blocks



```
scala> val numer:Double = 22
numer: Double = 22.0

scala> val denom:Double = 7
denom: Double = 7.0

scala> val PI = if (denom != 0) {numer/denom} else {0.0}
PI: Double = 3.142857142857143
```

Type inference at work!

If/Else expression blocks



```
scala> val PI = if (denom != 0) {numer/denom} else {0.0}
PI: Double = 3.142857142857143
```

The entire if/else construct is an expression block!

If/Else expression blocks

```
if (boolean expression)
    { expression block #1 }
else
    { expression block #2}
```

The entire if/else construct is an expression block!

If/Else expression blocks

Why does this matter?

```
if (boolean expression)
    { expression block #1 }
else
    { expression block #2}
```

The entire if/else construct is an expression block!

If/Else expression blocks

Why does this matter?

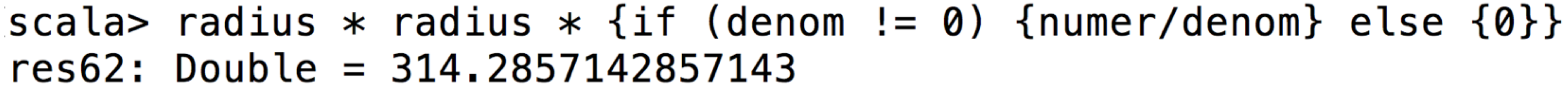
Java, C#, C, C++ have If/Else statements

Scala has If/Else expressions

Scala has If/Else expressions

Why does this matter?

Scala has cleverly transformed If/Else constructs into r-values..allowing them to be functionally composed!



```
scala> radius * radius * {if (denom != 0) {numer/denom} else {0}}
res62: Double = 314.2857142857143
```

Scala has cleverly transformed If/Else constructs into r-values..allowing them to be functionally composed!

Scala has cleverly transformed If/Else constructs **into r-values**..allowing them to be functionally composed!

Scala has cleverly transformed If/Else constructs into r-values..allowing them to be functionally composed!

Example 13

match expressions

`match` expressions

Java, C#, C, C++ have switch statements

Scala has match expressions

these are actually more powerful, and more widely used than if/else expressions

match expressions

Unlike in Java, only zero or one 'case' clauses will evaluate to true

No fall-through, no 'break'

there is a catch-all though

matches can be on value, but also on type and with additional conditions

match expressions

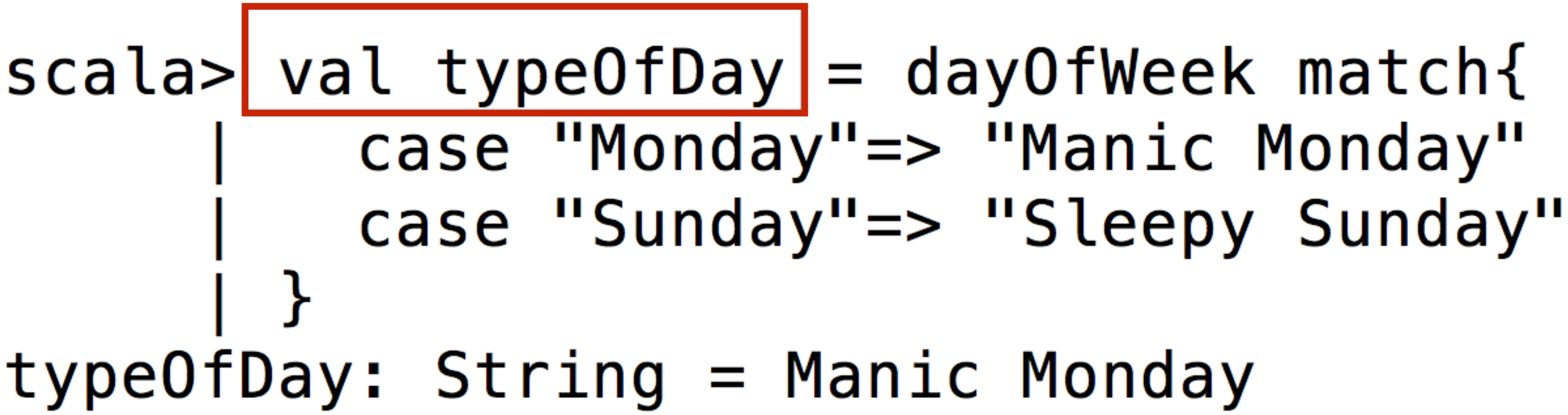
Match expressions are expressions like any others -

use them to initialise values or variables

or “compose” them to create functional chains!

match expressions

use them to initialise values or variables



```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

A value, to be initialised via a match expression

match expressions

use them to initialise values or variables

```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The match expression

match expressions

use them to initialise values or variables

```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The identifier to be matched

match expressions

use them to initialise values or variables

The identifier to be matched

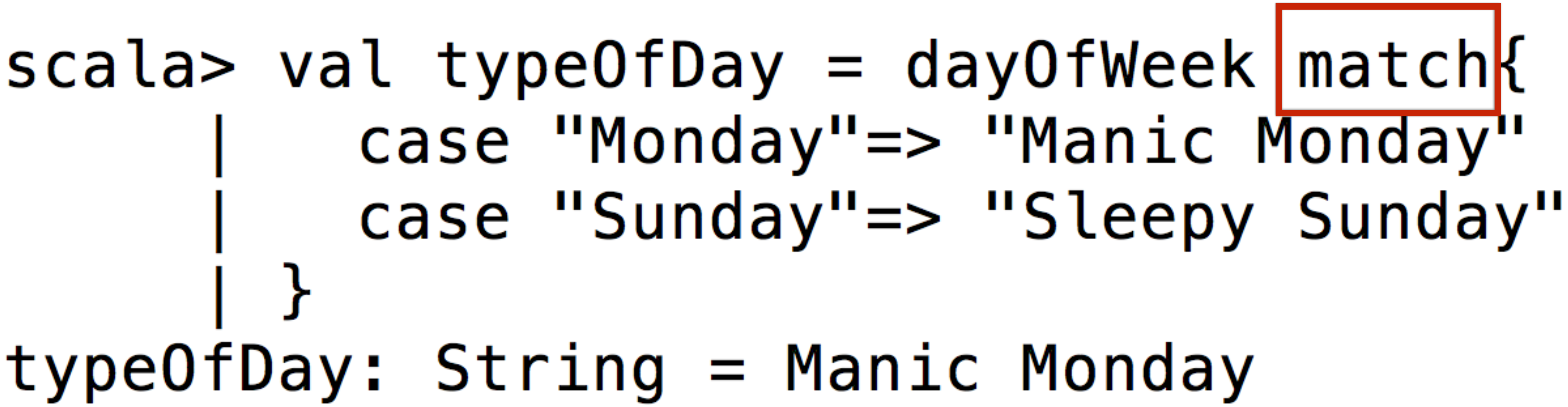
```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

(needs to be previously defined, somewhere)

```
scala> val dayOfWeek = "Monday"
dayOfWeek: String = Monday
```

match expressions

use them to initialise values or variables

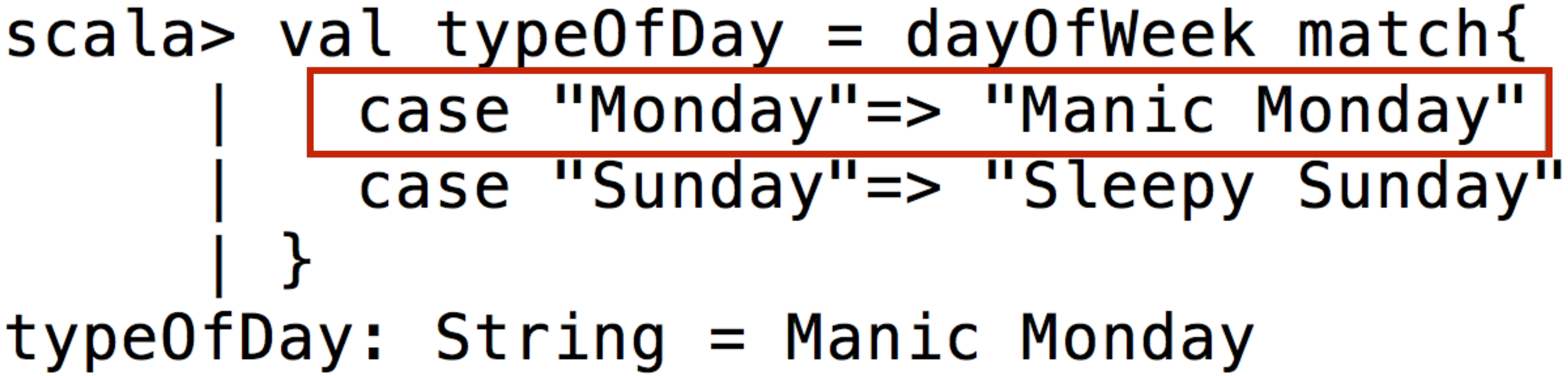


```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The match keyword

match expressions

use them to initialise values or variables

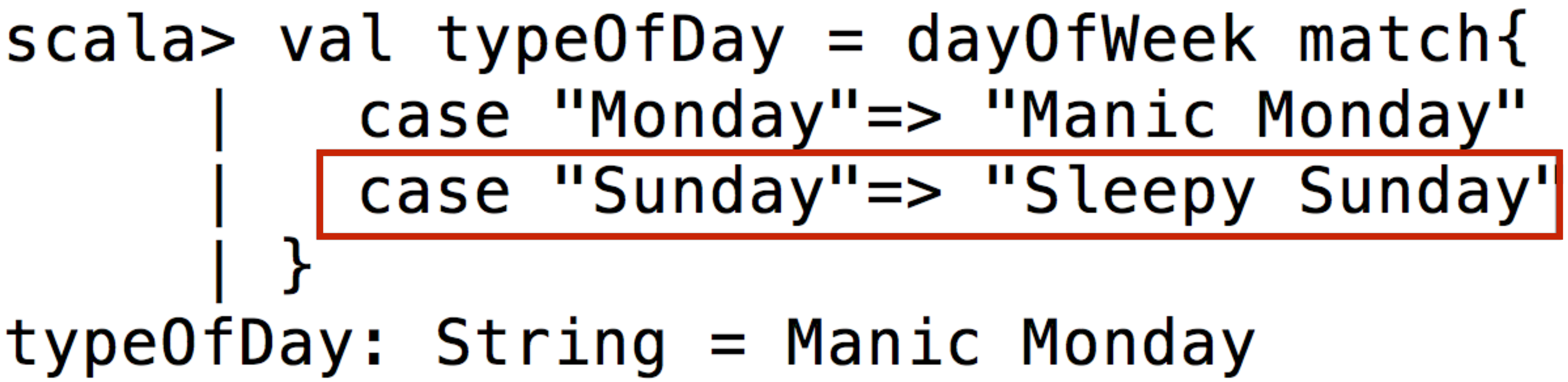


```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

A set of cases, none or one of which will be matched

match expressions

use them to initialise values or variables

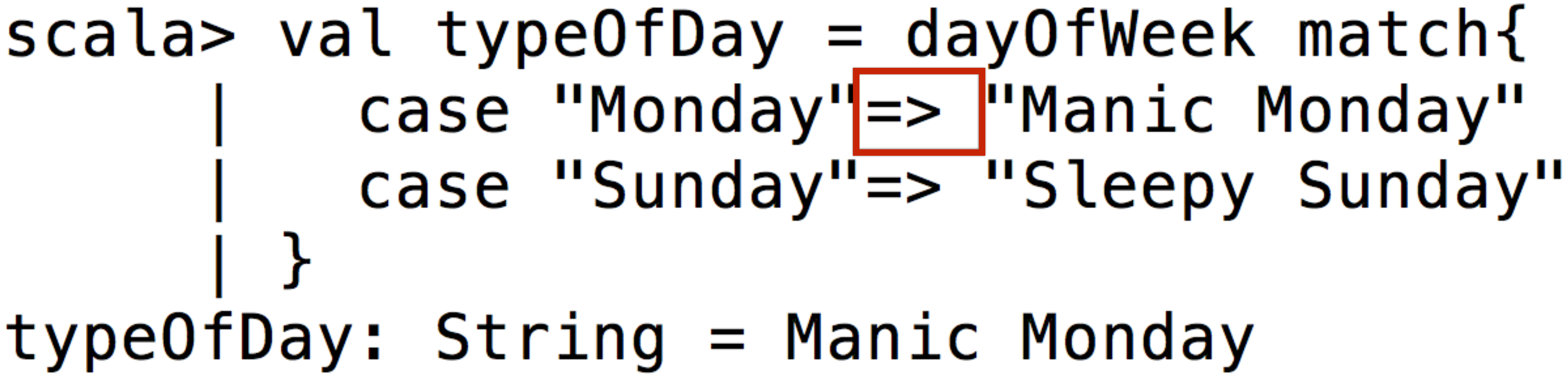


```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

A set of cases, none or one of which will be matched

match expressions

use them to initialise values or variables

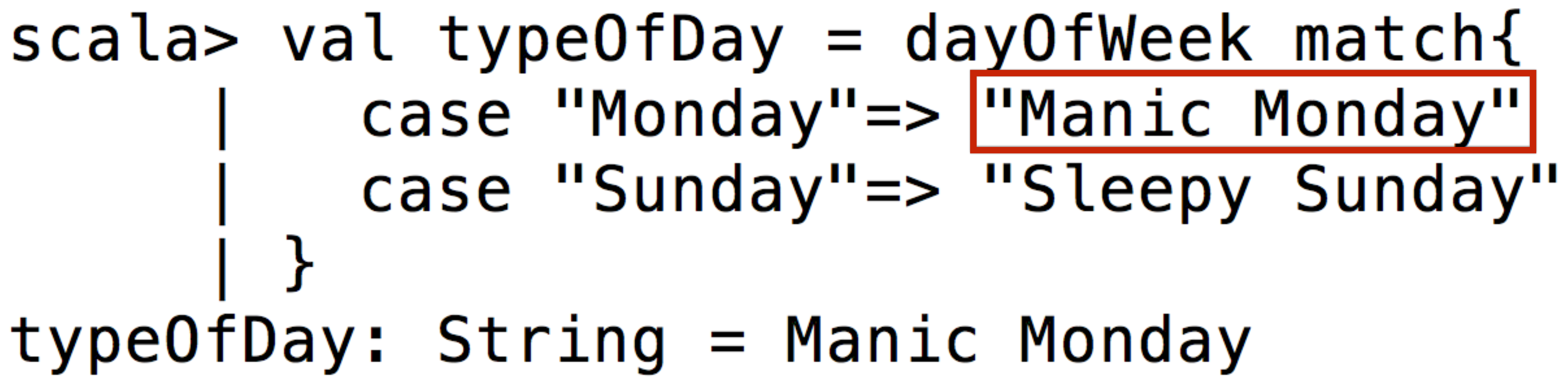


```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The => is followed by an expression that will be returned if that case is satisfied

match expressions

use them to initialise values or variables

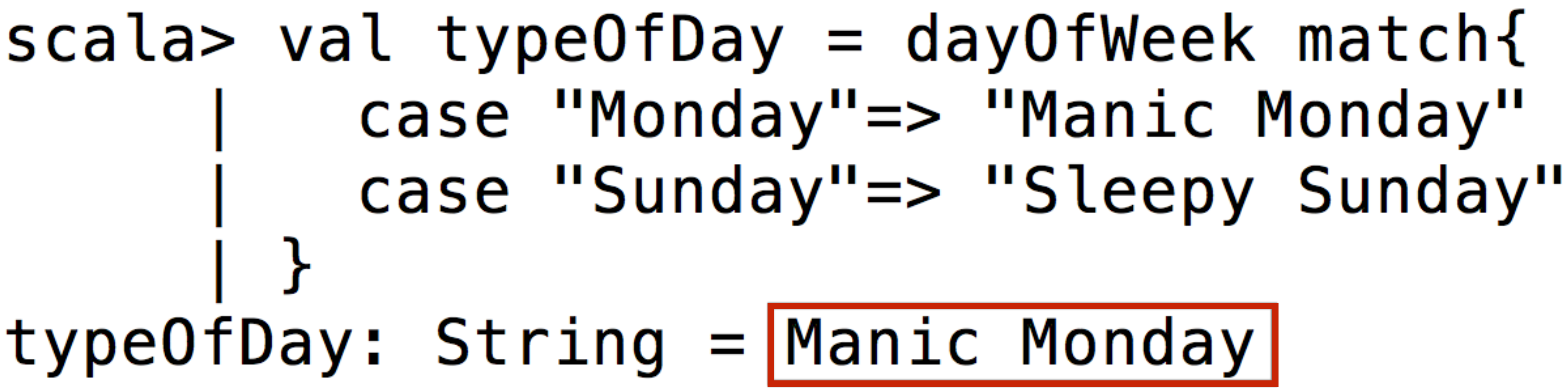


```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

expression that will be returned if that case is satisfied

match expressions

use them to initialise values or variables



```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The value returned by the match expression

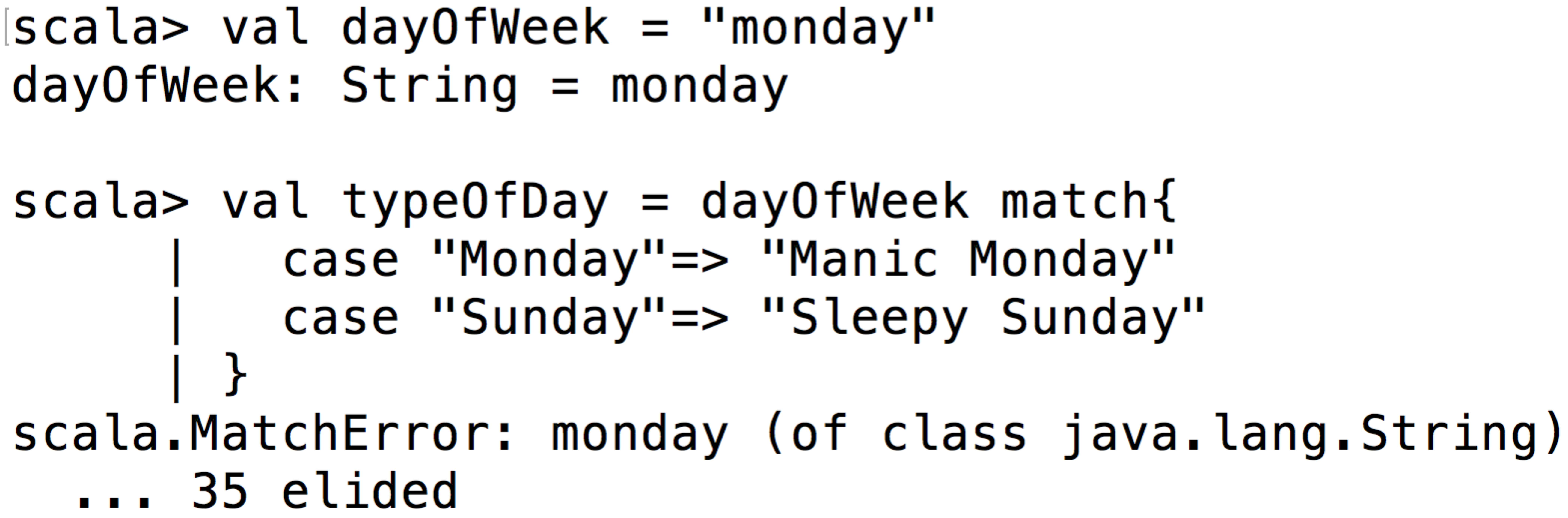
match expressions

use them to initialise values or variables

What if no case matches? What value will be assigned?

The result is a Scala.MatchError

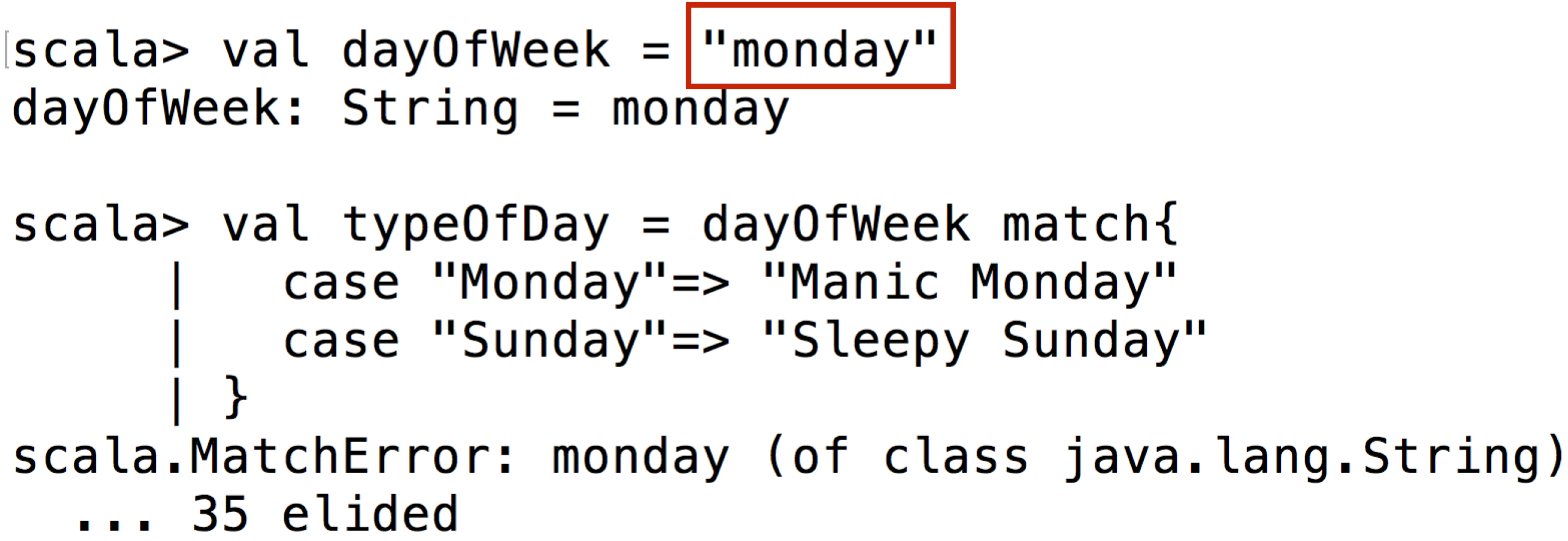
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

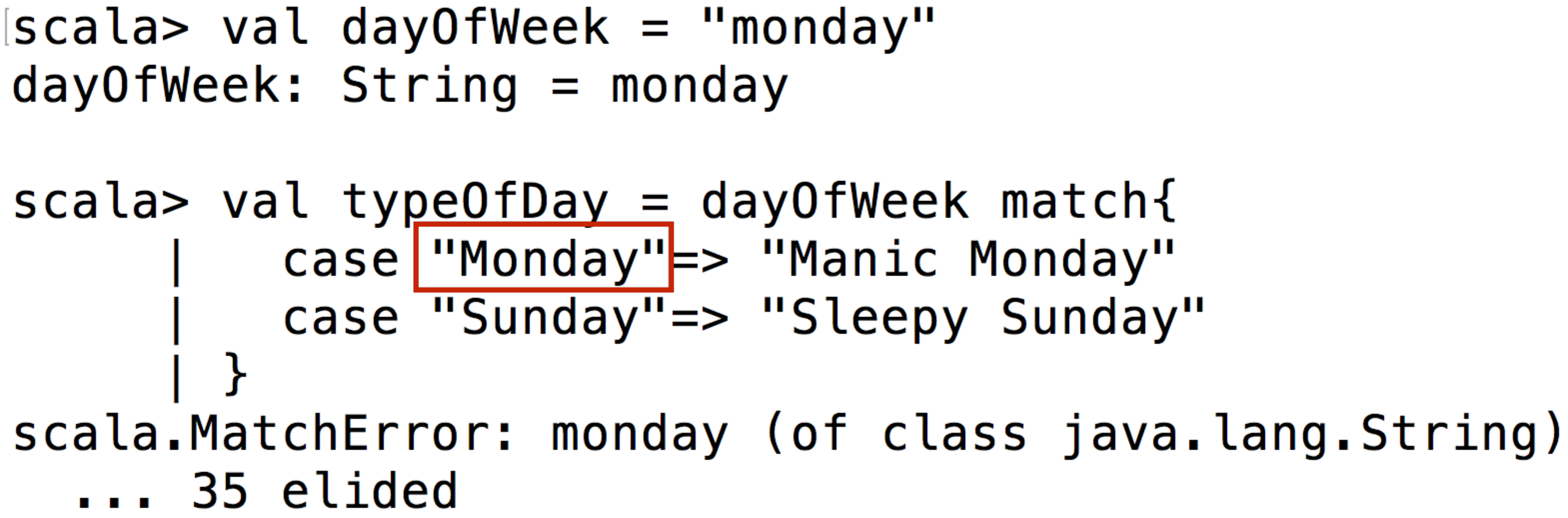
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

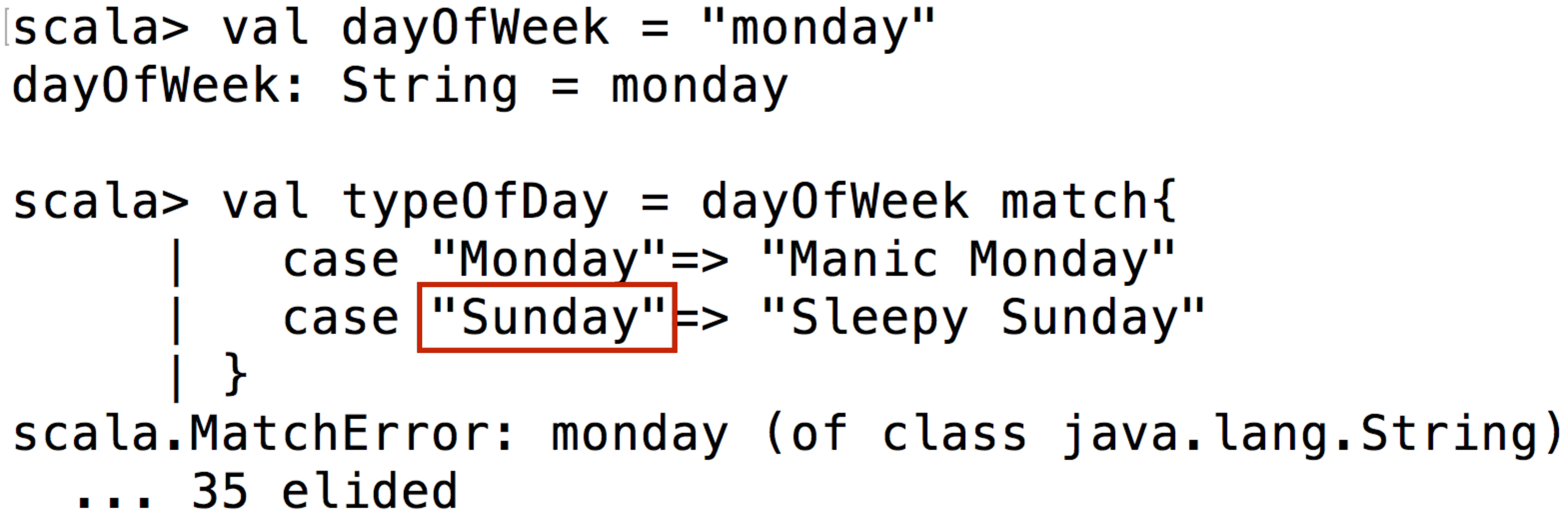
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

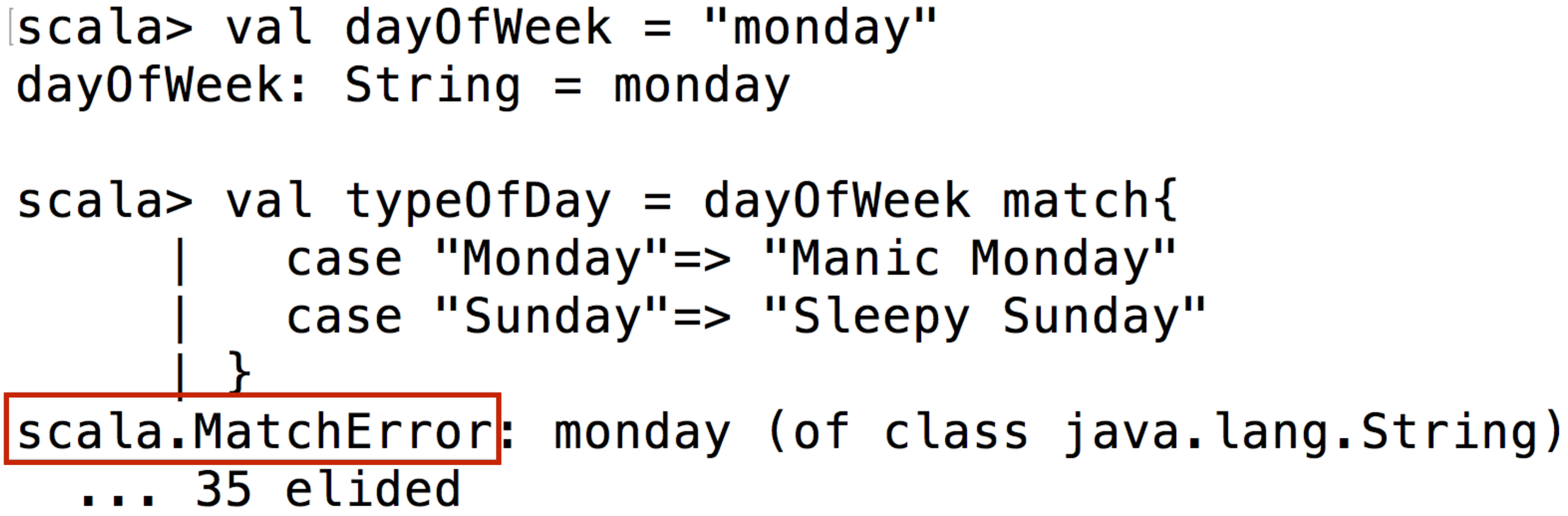
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

match expressions

use them to initialise values or variables

What if no case matches? What value will be assigned?

The result is a Scala.MatchError

match expressions

Match expressions are expressions like any others -

use them to initialise values or variables

or “compose” them to create functional chains!

match expressions

Match expressions are expressions like any others -

use them to initialise values or variables

or "compose" them to create functional chains!

Match expressions used more widely than if/else statements in Scala - they are a very popular language construct!!

Match expressions are a very popular language construct

And that's why they matter a great deal in Scala!

(far more than `switch` statements matter in Java)

Example 14

match expressions: Pattern guards & OR-ed expressions

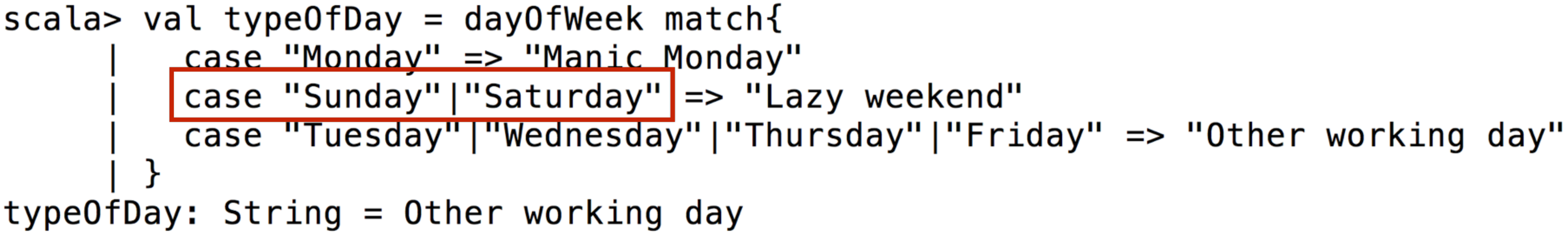
match expressions: Pattern guards & OR-ed expressions

There are two ways to add conditions to individual case clauses in a match expression

Pattern guards

OR-ed expressions

OR-ed expressions



```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Sunday"|"Saturday" => "Lazy weekend"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     | }
typeOfDay: String = Other working day
```

The boolean expressions in a case statement can be OR-ed together

OR-ed expressions

```
scala>  val dayOfWeek = "Friday"
dayOfWeek: String = Friday
```

```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Sunday"|"Saturday" => "Lazy weekend"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     | }
typeOfDay: String = Other working day
```

It just works!

OR-ed expressions

```
scala>  val dayOfWeek = "Friday"
dayOfWeek: String = Friday
```

```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Sunday"|"Saturday" => "Lazy weekend"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     | }
typeOfDay: String = Other working day
```

It just works!

OR-ed expressions

```
scala>  val dayOfWeek = "Friday"
dayOfWeek: String = Friday
```

```
scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Sunday"|"Saturday" => "Lazy weekend"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     | }
typeOfDay: String = Other working day
```

It just works!

OR-ed expressions

```
scala>  val dayOfWeek = "Friday"
dayOfWeek: String = Friday
```

```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Sunday"|"Saturday" => "Lazy weekend"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     | }
typeOfDay: String = Other working day
```

It just works!

OR-ed expressions

```
scala>  val dayOfWeek = "Friday"
dayOfWeek: String = Friday
```

```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Sunday"|"Saturday" => "Lazy weekend"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     | }
typeOfDay: String = Other working day
```

It just works!

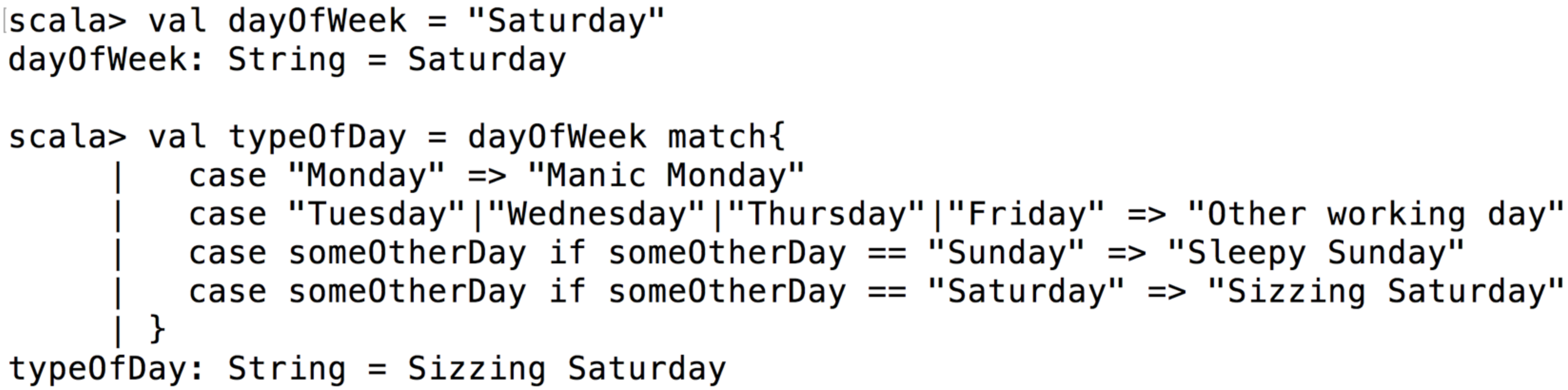
`match` expressions: Pattern guards & OR-ed expressions

There are two ways to add conditions to individual case clauses in a match expression

Pattern guards

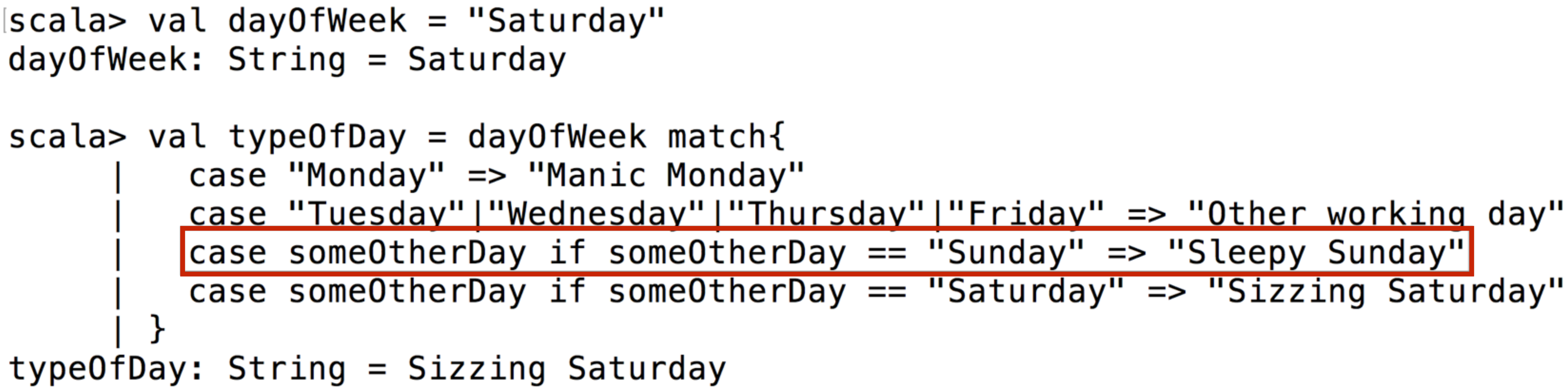
✓ OR-ed expressions

Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |   case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |   case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

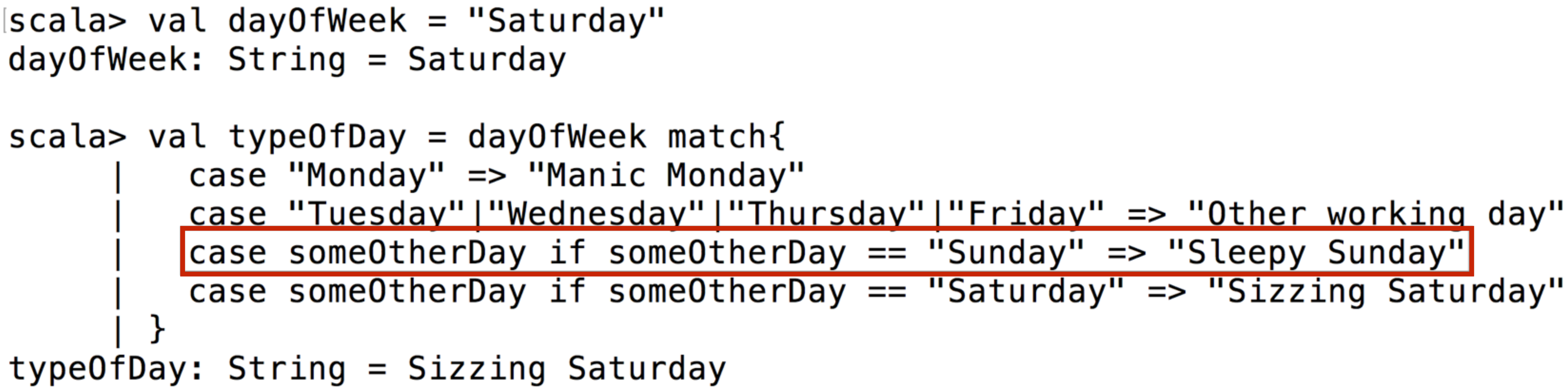
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Pattern guards are a way to add an if expression into a case

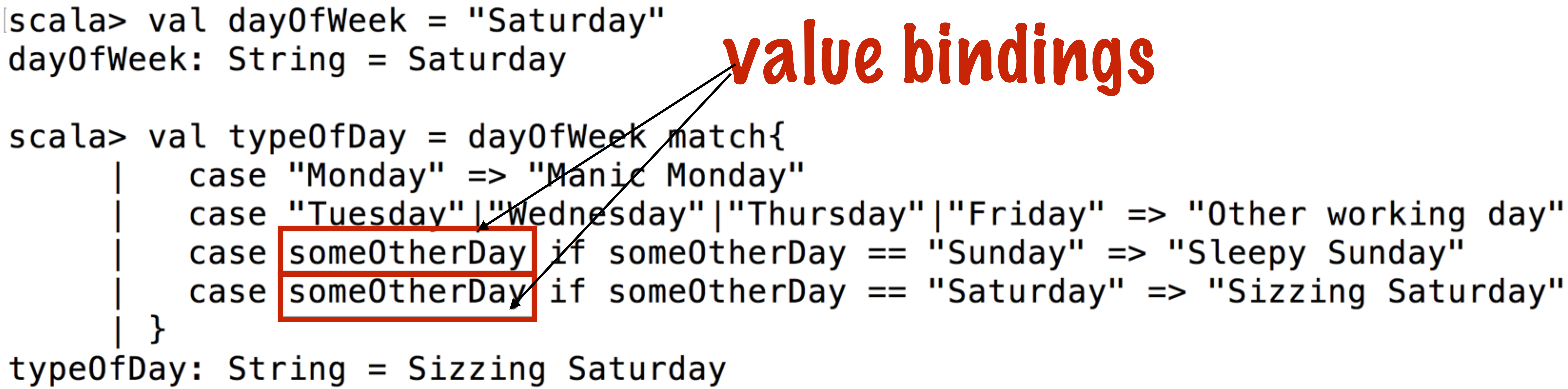
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

A pattern guard starts with an
placeholder variable called a value binding

Pattern guards

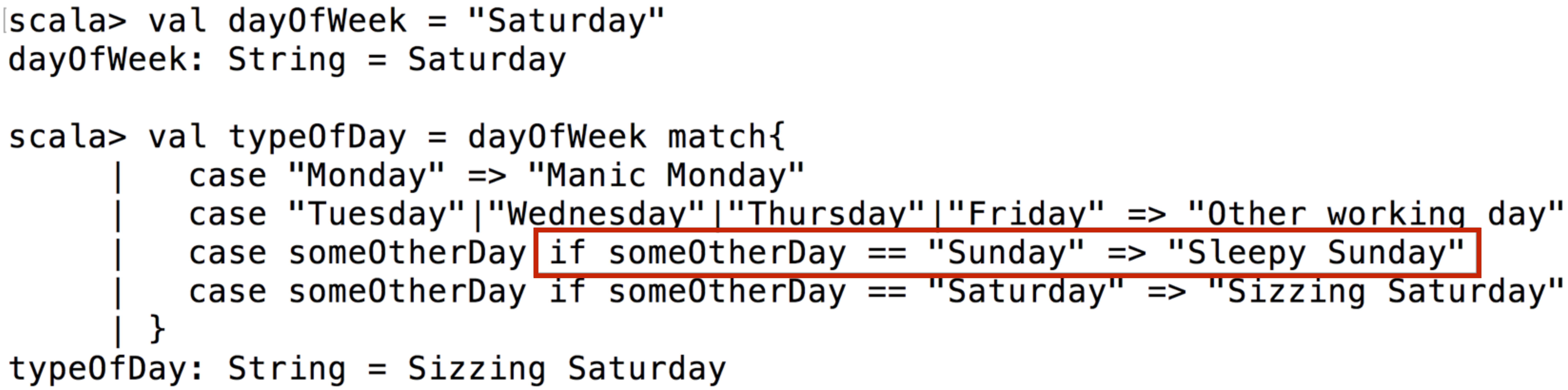


```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

value bindings

A pattern guard starts with an placeholder variable called a value binding

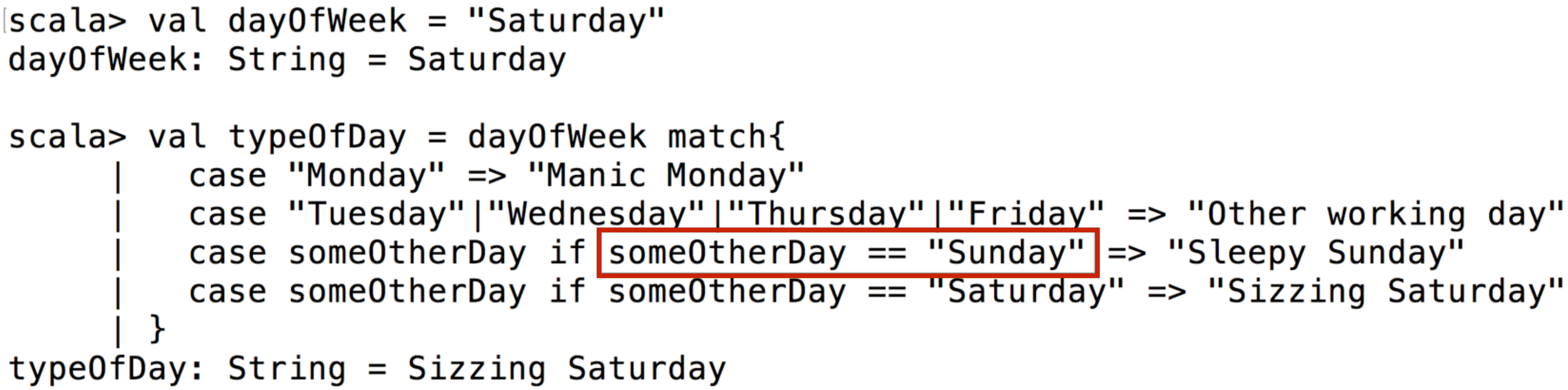
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Next is a rather unusual looking if expression

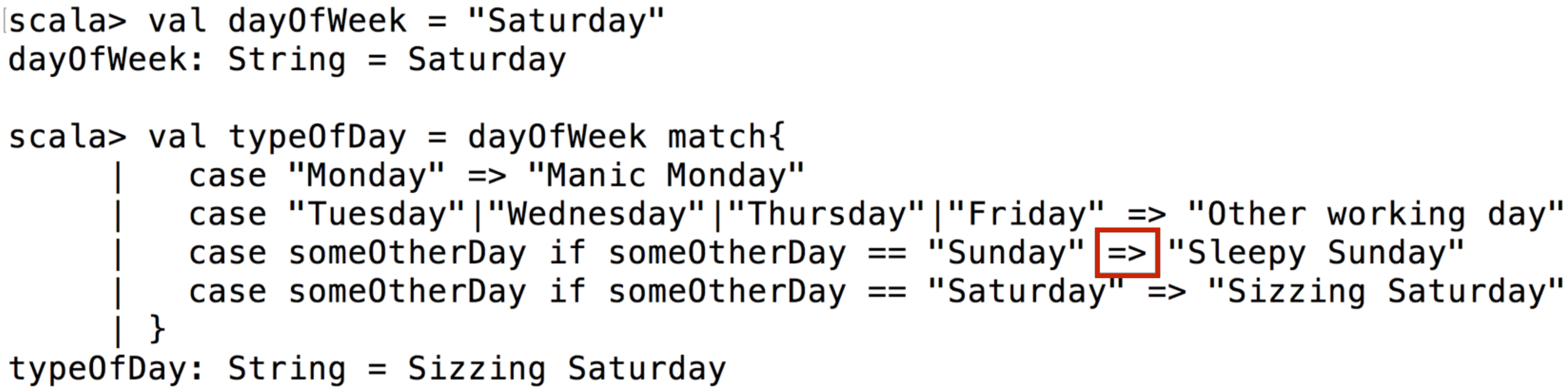
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |   case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |   case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

This if expression has a boolean expression as usual..

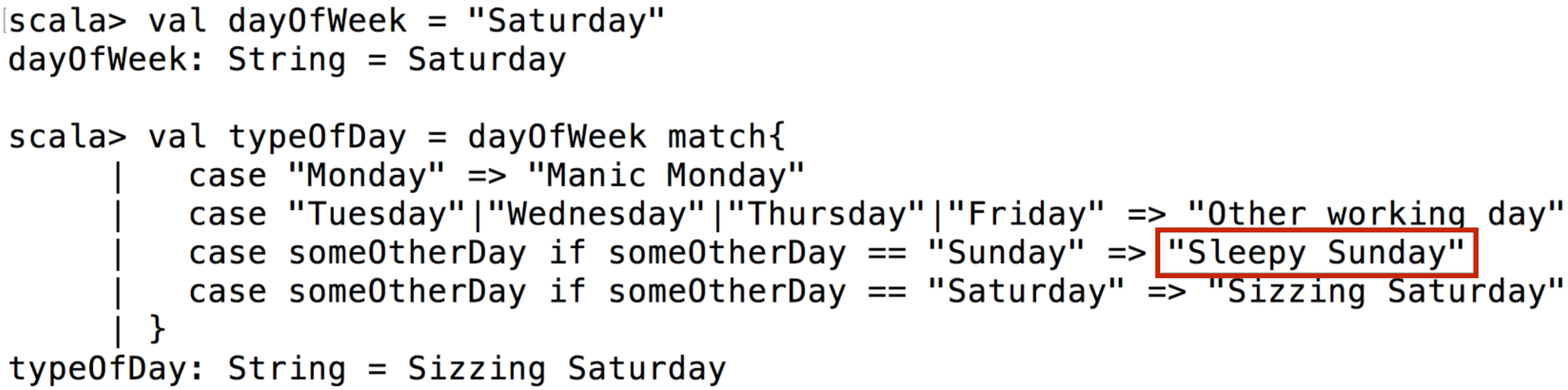
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Followed by the => delimiter..(this is the unusual bit!)

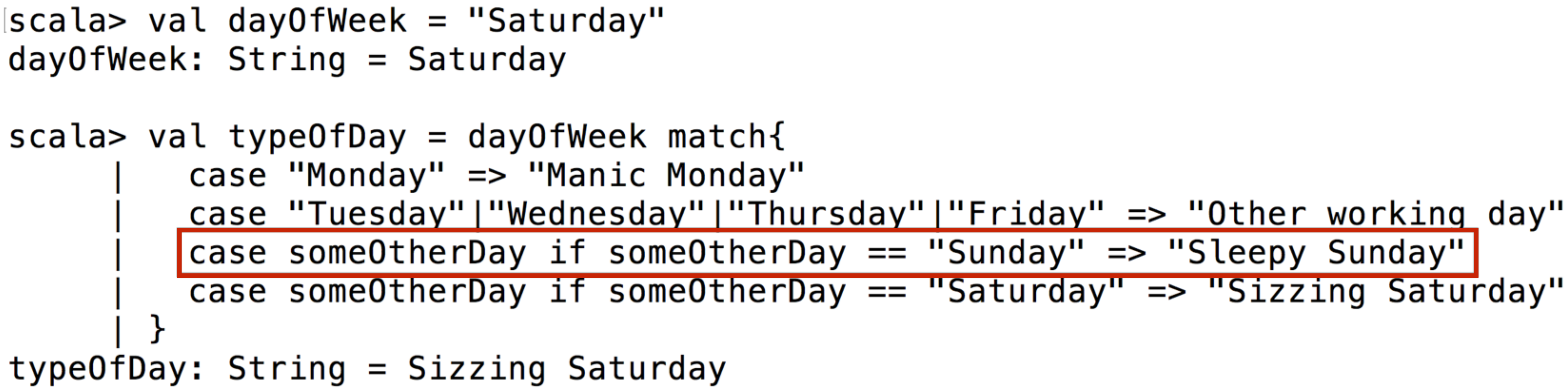
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |   case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |   case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Ending with the return value of the if-expression

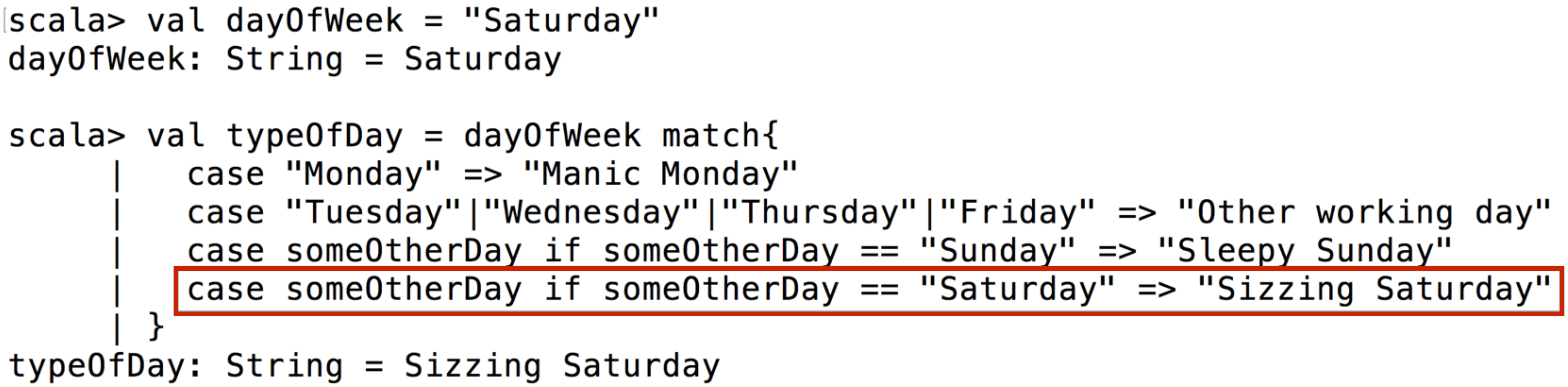
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Pattern guards allow multiple conditions on the same value binding

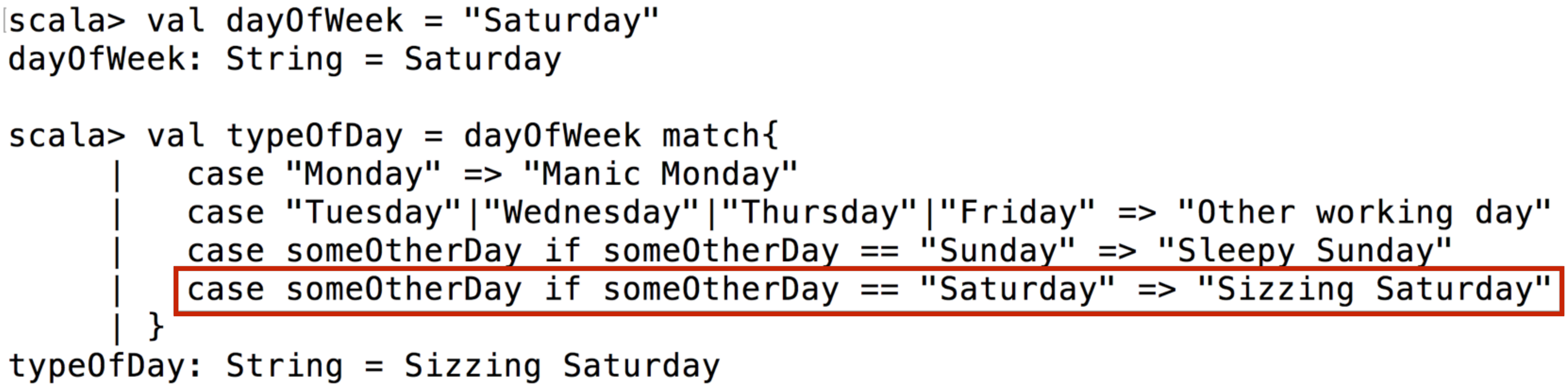
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |   case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |   case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Pattern guards allow multiple
conditions on the same value binding

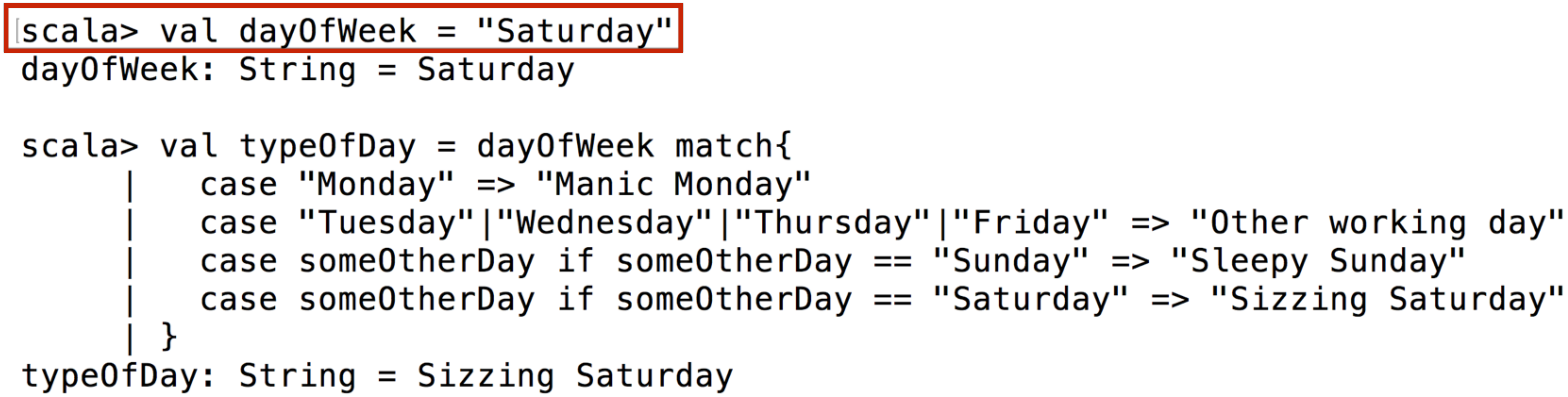
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |   case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |   case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Only the case where the pattern guard evaluates to true will pass

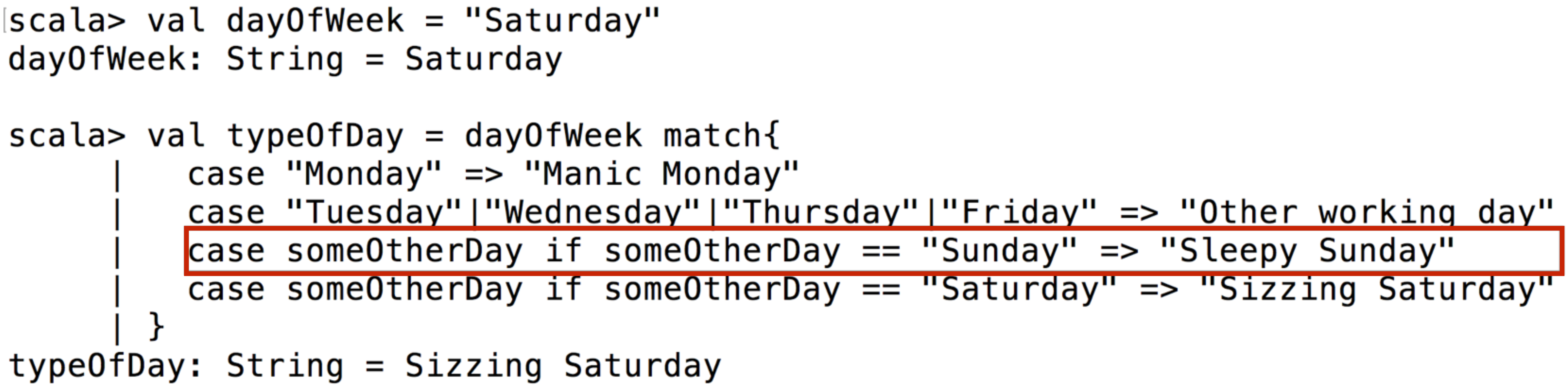
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Only the case where the pattern guard evaluates to true will pass

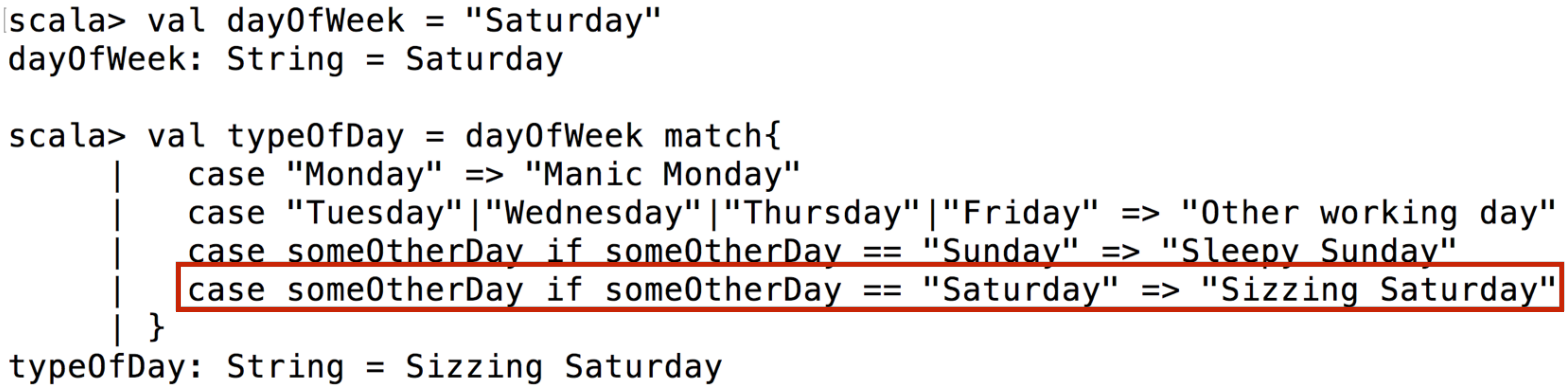
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |   case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |   case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Only the case where the pattern guard evaluates to true will pass

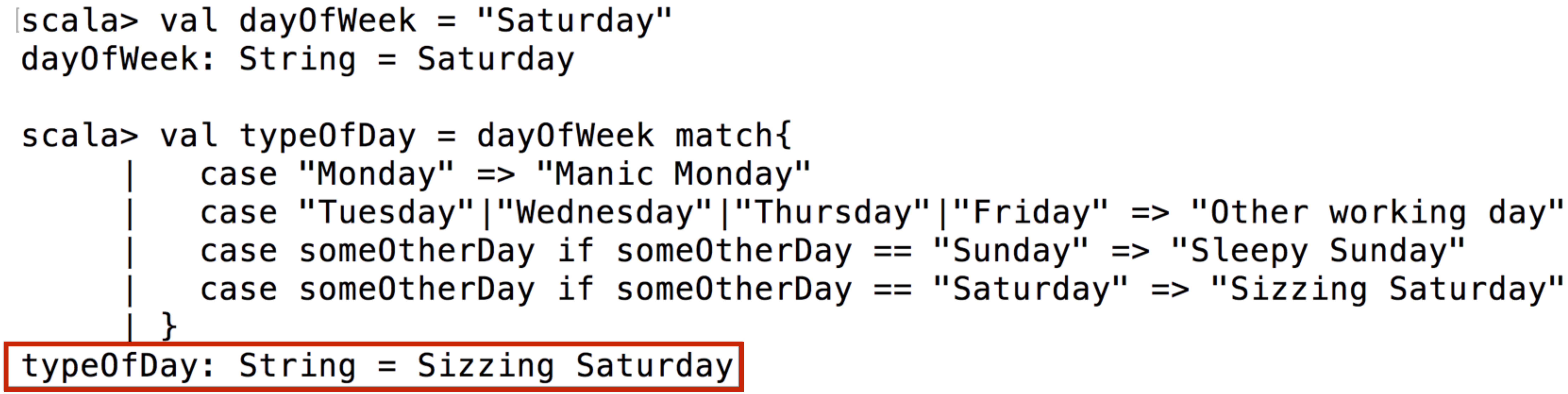
Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Only the case where the pattern guard evaluates to true will pass

Pattern guards



```
scala> val dayOfWeek = "Saturday"
dayOfWeek: String = Saturday

scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Tuesday"|"Wednesday"|"Thursday"|"Friday" => "Other working day"
     |   case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
     |   case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
     | }
typeOfDay: String = Sizzing Saturday
```

Only the case where the pattern guard evaluates to true will pass

match expressions: Pattern guards & OR-ed expressions

There are two ways to add conditions to individual case clauses in a match expression

✓ Pattern guards

✓ OR-ed expressions

`match` expressions: Pattern guards & OR-ed expressions

There are two ways to add conditions to individual case clauses in a match expression

- ✓ Pattern guards
- ✓ OR-ed expressions

Why do these matter?

Why do these matter?

`match` expressions: Pattern guards & OR-ed expressions

`match` expressions are really important in Scala, and these variants help extend their flexibility a great deal

Example 15

`match` expressions: catch-all to match-all

`match` expressions: catch-all to match-all

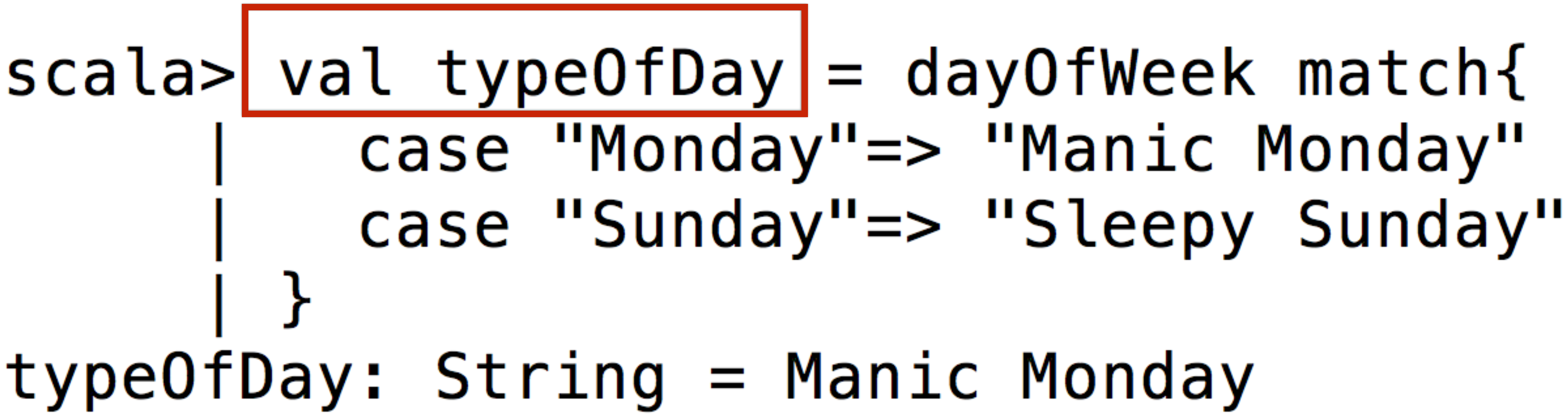
Match expressions are expressions like any others -

use them to initialise values or variables

or "compose" them to create functional chains!

match expressions: catch-all to match-all

use them to initialise values or variables

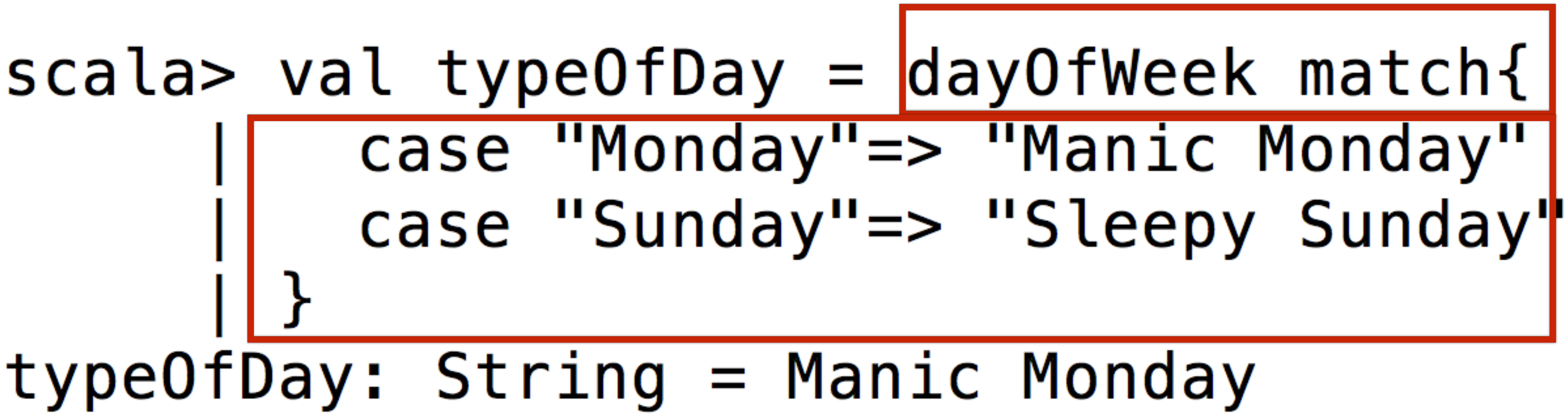


```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

A value, to be initialised via a match expression

match expressions: catch-all to match-all

use them to initialise values or variables



```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The match expression

match expressions: catch-all to match-all

use them to initialise values or variables

```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The identifier to be matched

match expressions: catch-all to match-all

use them to initialise values or variables

The identifier to be matched

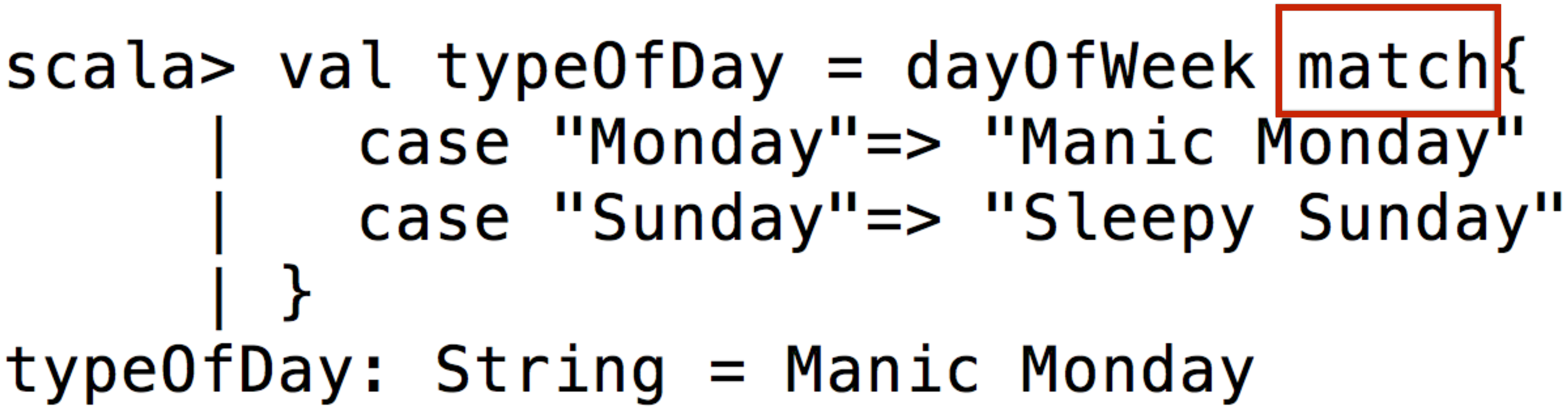
```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

(needs to be previously defined, somewhere)

```
scala> val dayOfWeek = "Monday"
dayOfWeek: String = Monday
```

`match` expressions: catch-all to match-all

use them to initialise values or variables

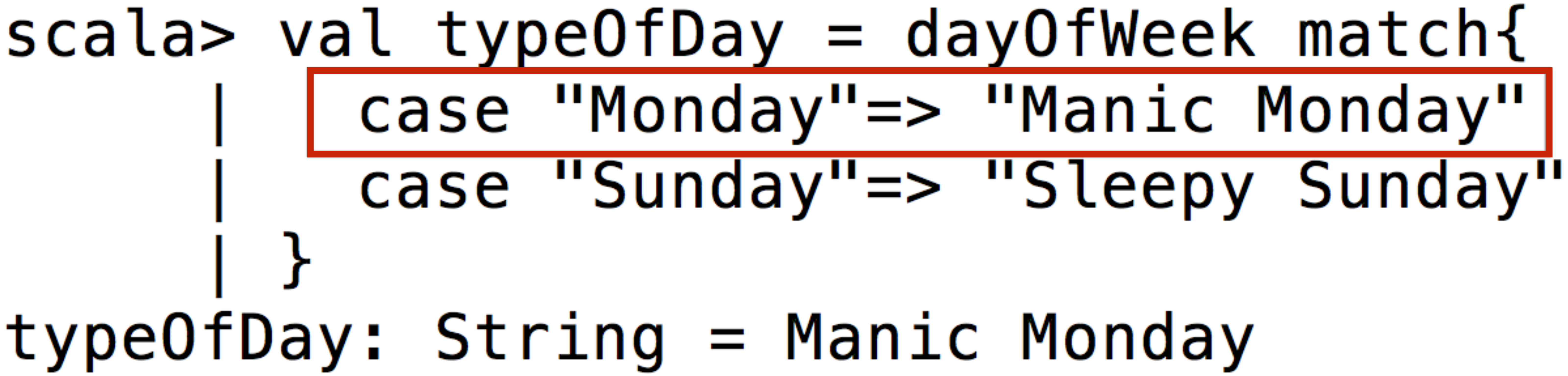


```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The match keyword

match expressions: catch-all to match-all

use them to initialise values or variables

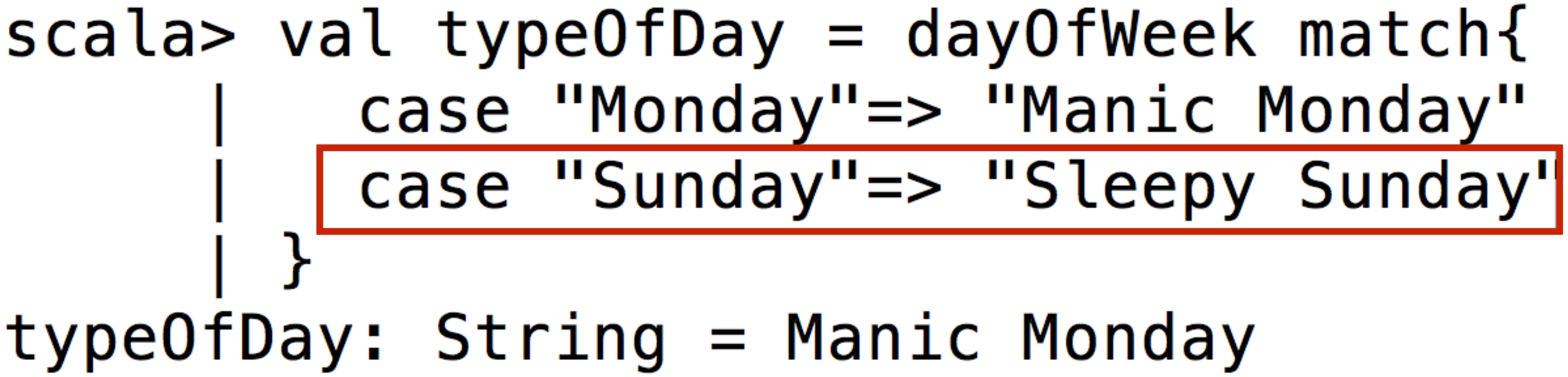


```
scala> val typeOfDay = dayOfWeek match{
     |   case "Monday"=> "Manic Monday"
     |   case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

A set of cases, none or one of which will be matched

match expressions: catch-all to match-all

use them to initialise values or variables

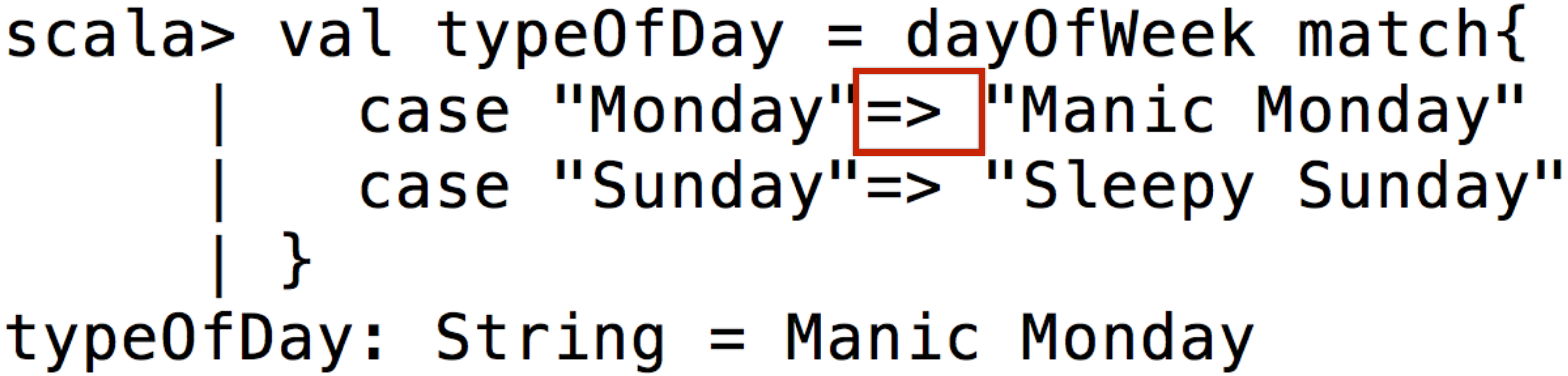


```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

A set of cases, none or one of which will be matched

match expressions: catch-all to match-all

use them to initialise values or variables

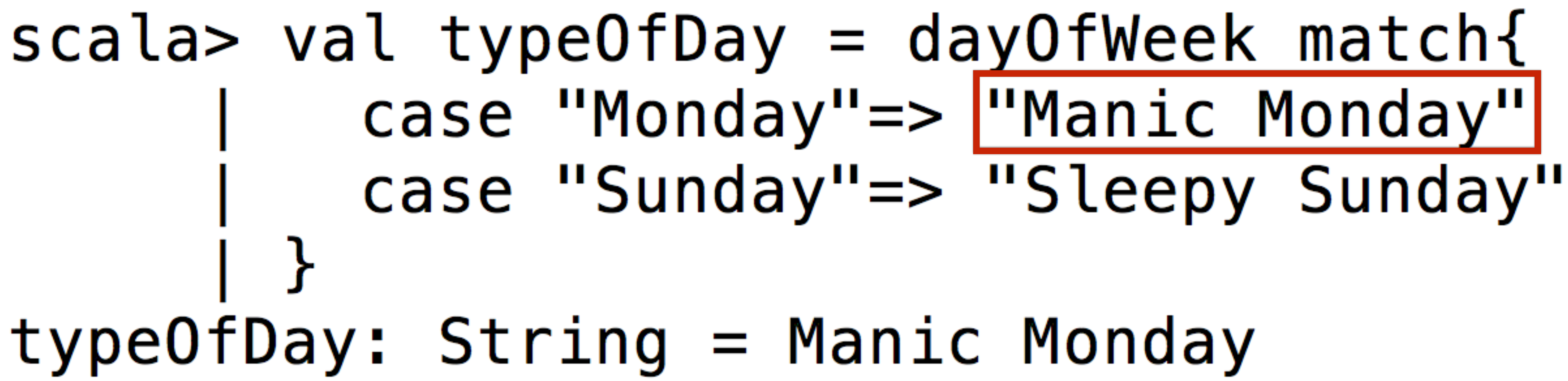


```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The => is followed by an expression that will be returned if that case is satisfied

match expressions: catch-all to match-all

use them to initialise values or variables

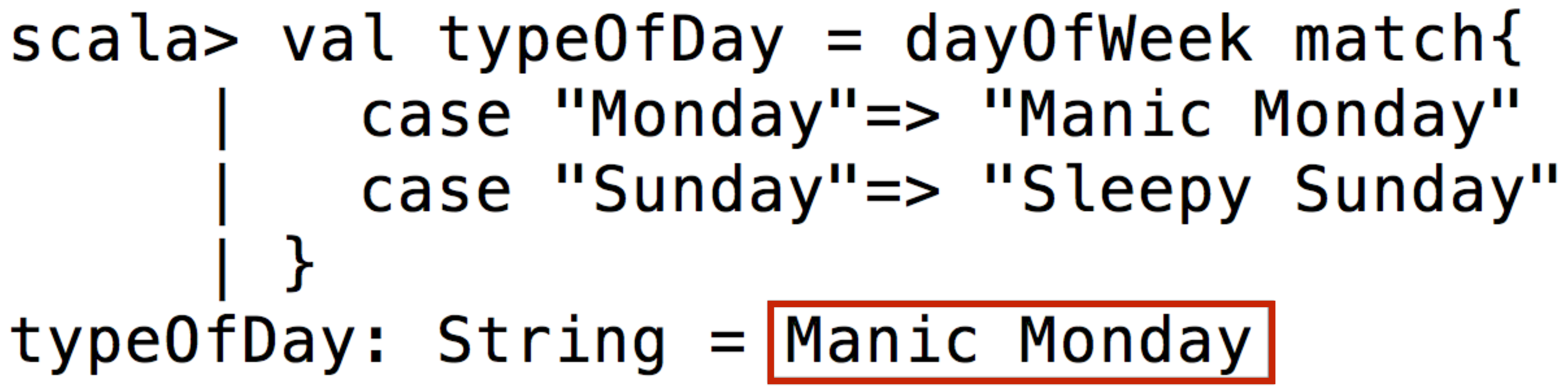


```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     |  }
typeOfDay: String = Manic Monday
```

expression that will be returned if that case is satisfied

match expressions: catch-all to match-all

use them to initialise values or variables



```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
typeOfDay: String = Manic Monday
```

The value returned by the match expression

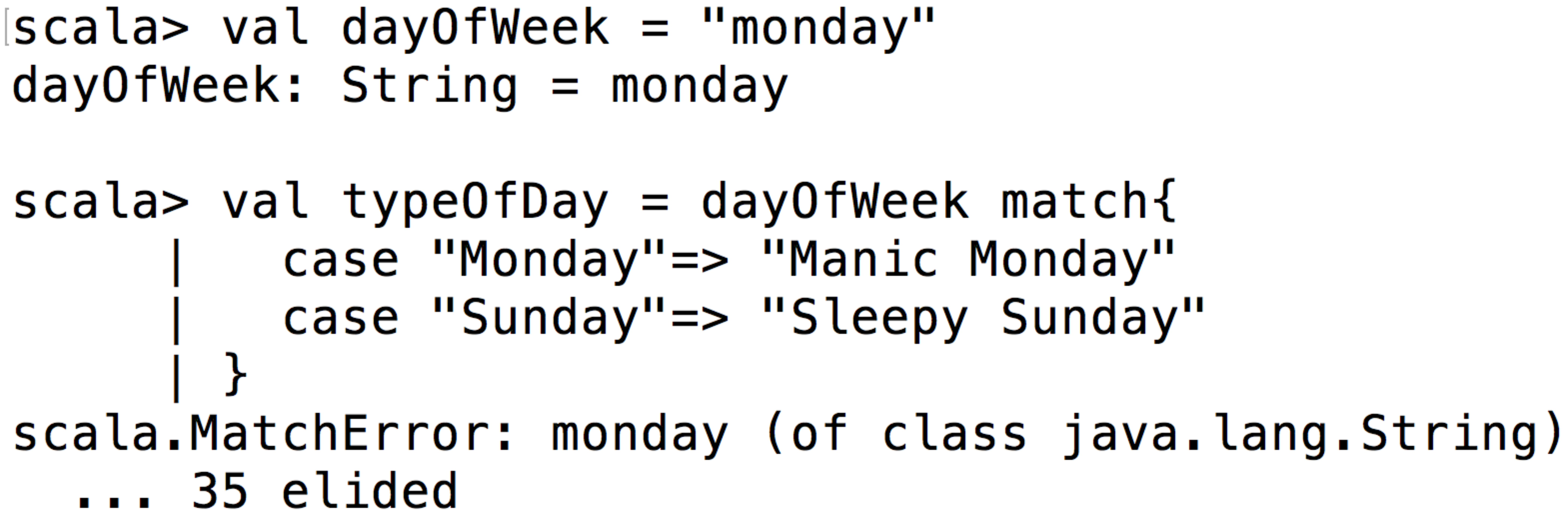
`match` expressions: catch-all to match-all

use them to initialise values or variables

What if no case matches? What value will be assigned?

The result is a Scala.MatchError

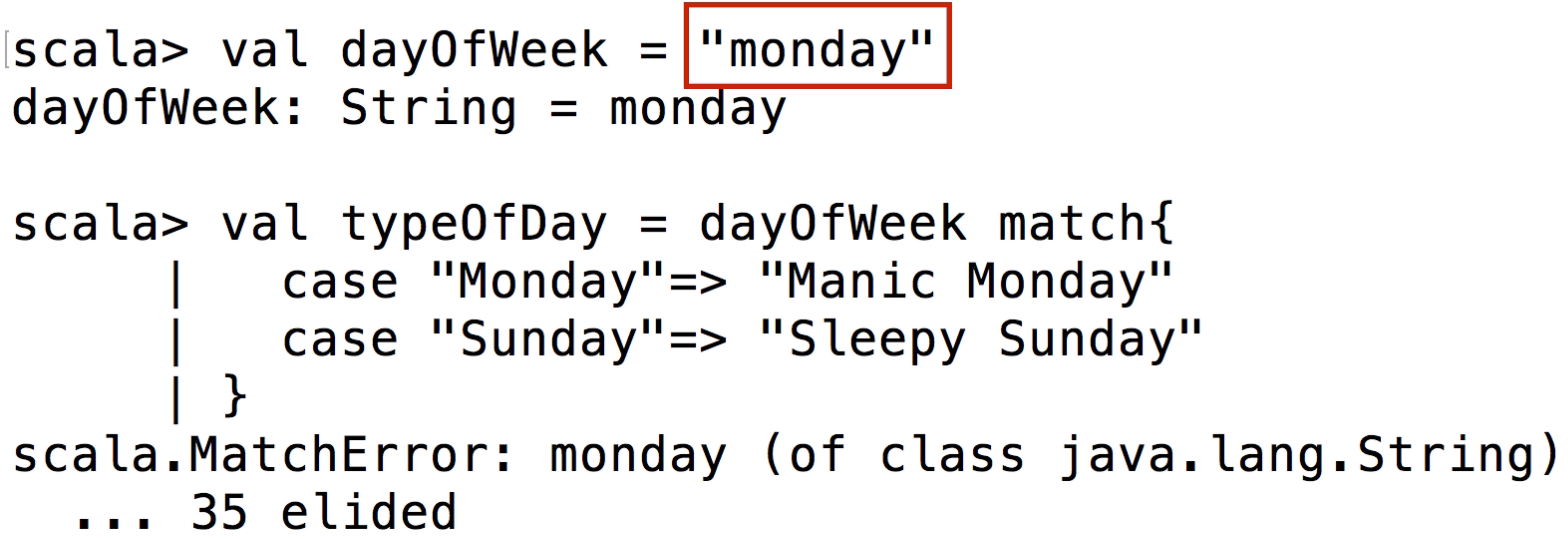
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

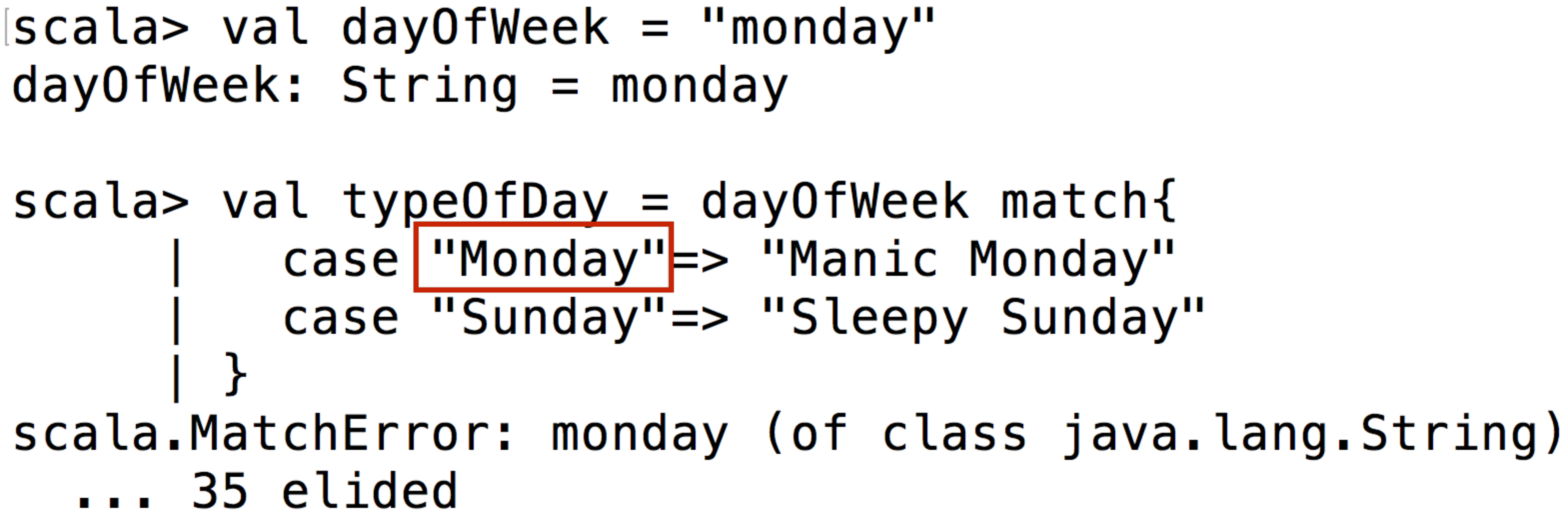
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

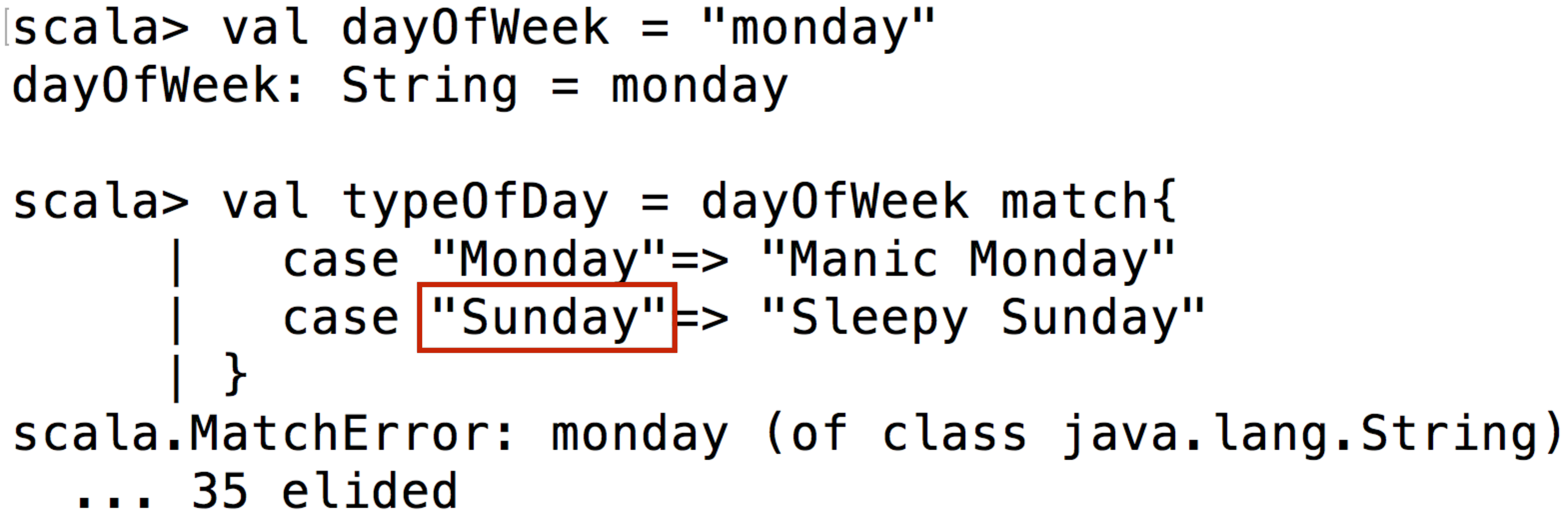
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |     case "Monday"=> "Manic Monday"
     |     case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

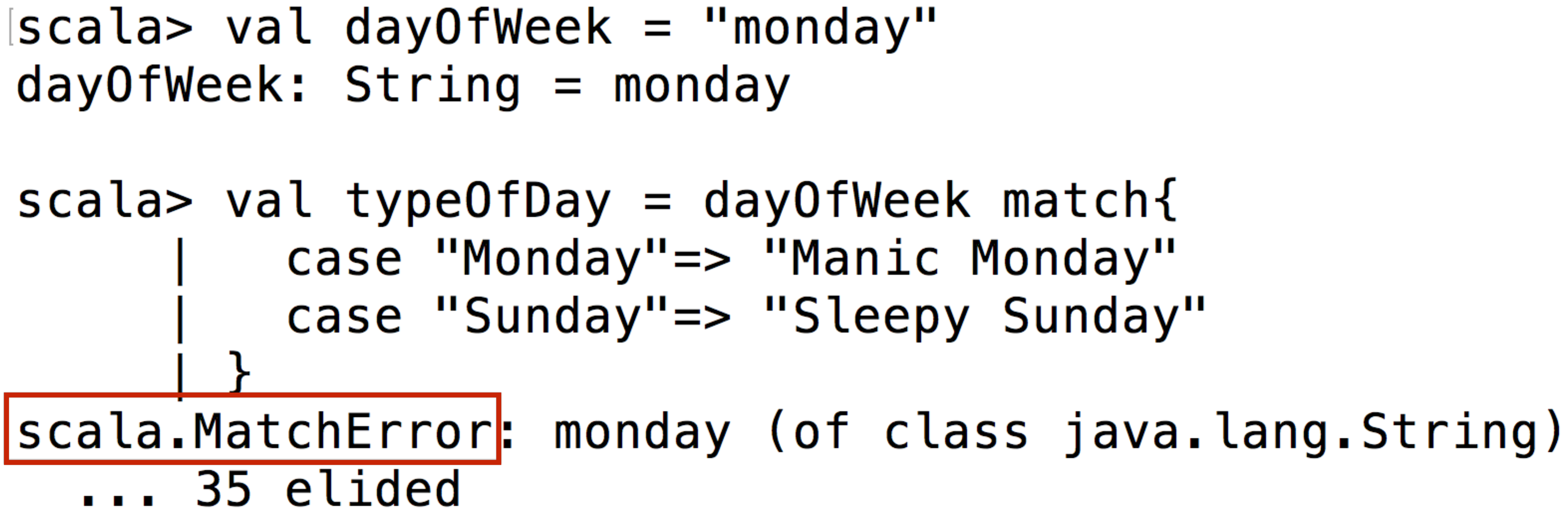
What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

What if no case matches? What value will be assigned?



```
scala> val dayOfWeek = "monday"
dayOfWeek: String = monday

scala> val typeOfDay = dayOfWeek match{
     |    case "Monday"=> "Manic Monday"
     |    case "Sunday"=> "Sleepy Sunday"
     | }
scala.MatchError: monday (of class java.lang.String)
  ... 35 elided
```

The result is a Scala.MatchError

`match` expressions: catch-all to match-all

use them to initialise values or variables

What if no case matches? What value will be assigned?

The result is a Scala.MatchError

`match` expressions: catch-all to match-all

We need something like the default case in a switch statement to prevent such errors

Scala.MatchError

Scala.MatchError

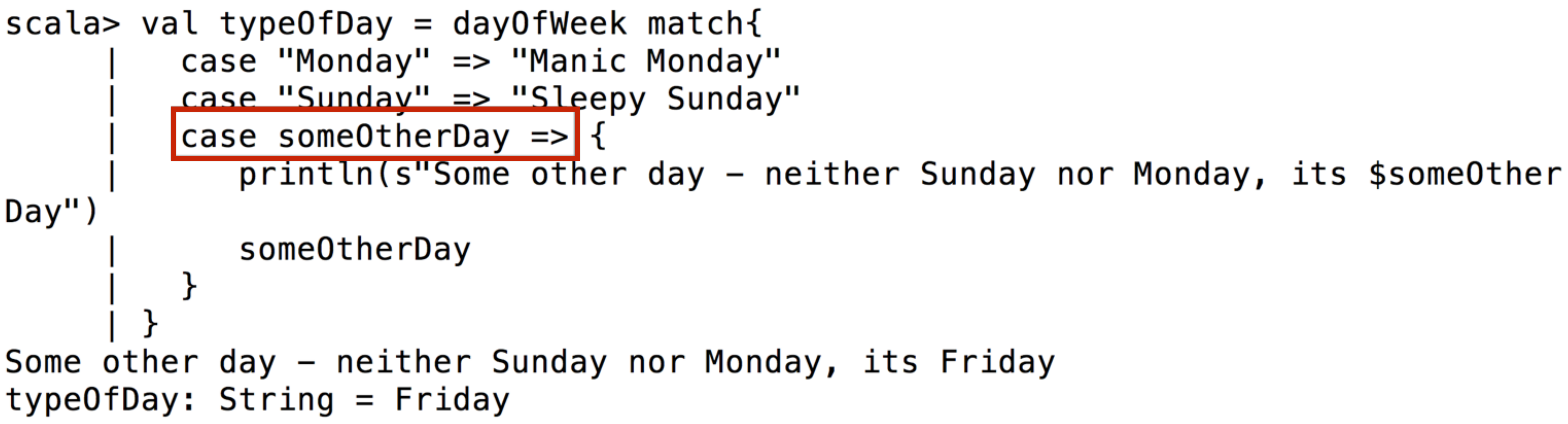
We need something like the default case in a switch statement to prevent such errors

`match` expressions: catch-all to match-all

Value Binding Patterns

_ (Wildcard Operator Patterns)

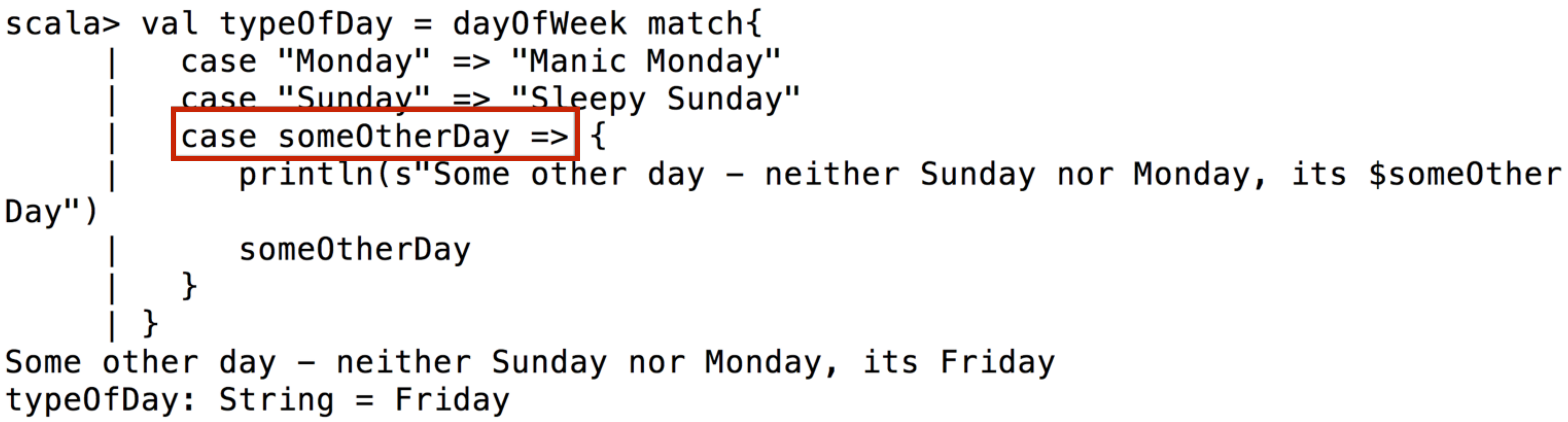
Value Binding Patterns



```
scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Sunday" => "Sleepy Sunday"
     |   case someOtherDay => {
     |      println(s"Some other day - neither Sunday nor Monday, its $someOther
Day")
     |      someOtherDay
     |   }
     | }
Some other day - neither Sunday nor Monday, its Friday
typeOfDay: String = Friday
```

We have encountered value bindings before!

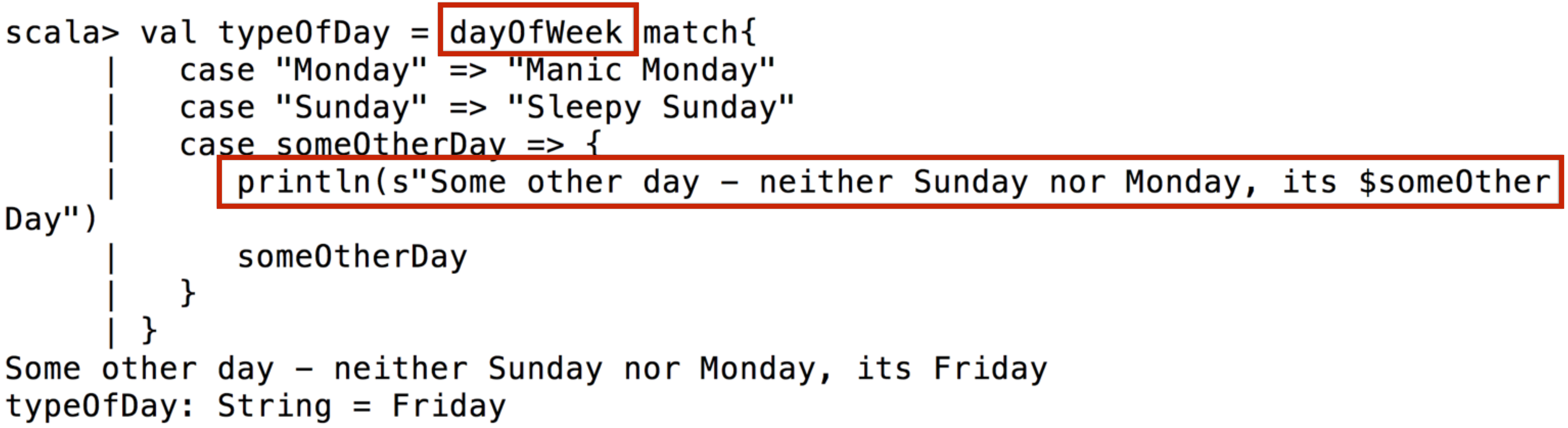
Value Binding Patterns



```
scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Sunday" => "Sleepy Sunday"
     |   case someOtherDay => {
     |      println(s"Some other day - neither Sunday nor Monday, its $someOther
Day")
     |      someOtherDay
     |   }
     | }
Some other day - neither Sunday nor Monday, its Friday
typeOfDay: String = Friday
```

Use a variable to store the value of the match variable

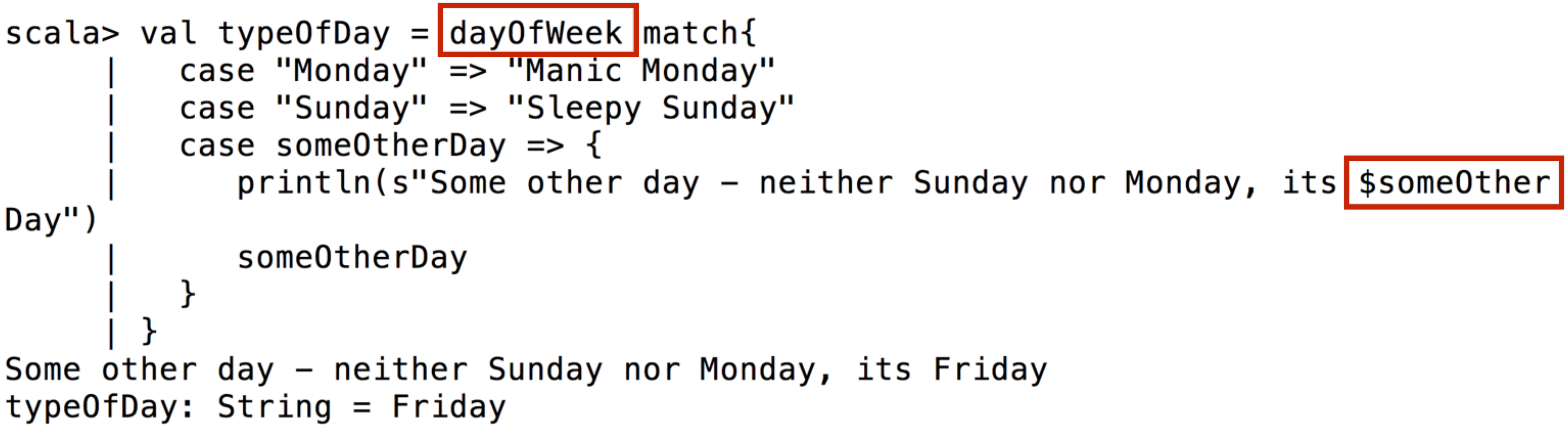
Value Binding Patterns



```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Sunday" => "Sleepy Sunday"
     |    case someOtherDay => {
     |       println(s"Some other day - neither Sunday nor Monday, its $someOther
Day")
     |       someOtherDay
     |    }
     | }
Some other day - neither Sunday nor Monday, its Friday
typeOfDay: String = Friday
```

This variable can then be used in the expression on the right side of the arrow

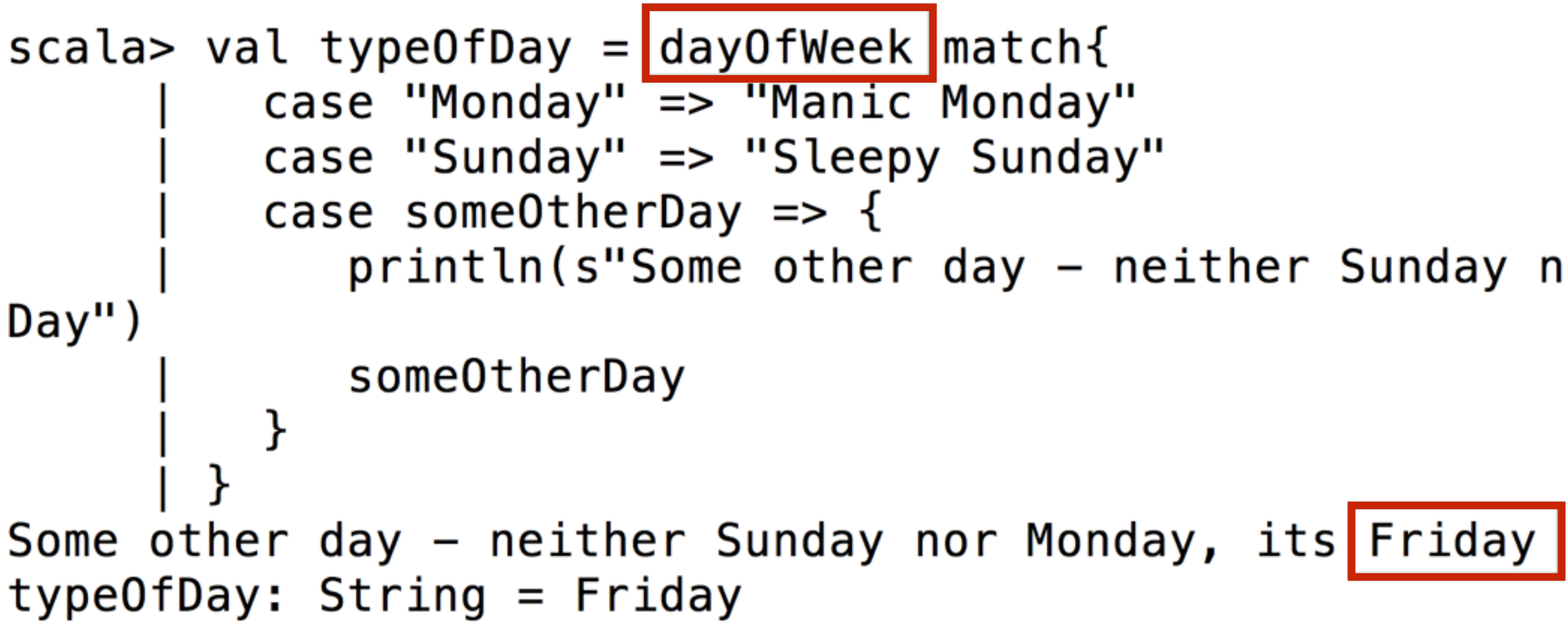
Value Binding Patterns



```
scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Sunday" => "Sleepy Sunday"
     |   case someOtherDay => {
     |      println(s"Some other day - neither Sunday nor Monday, its $someOther
Day")
     |      someOtherDay
     |   }
     | }
Some other day - neither Sunday nor Monday, its Friday
typeOfDay: String = Friday
```

This variable can then be used in the expression on the right side of the arrow

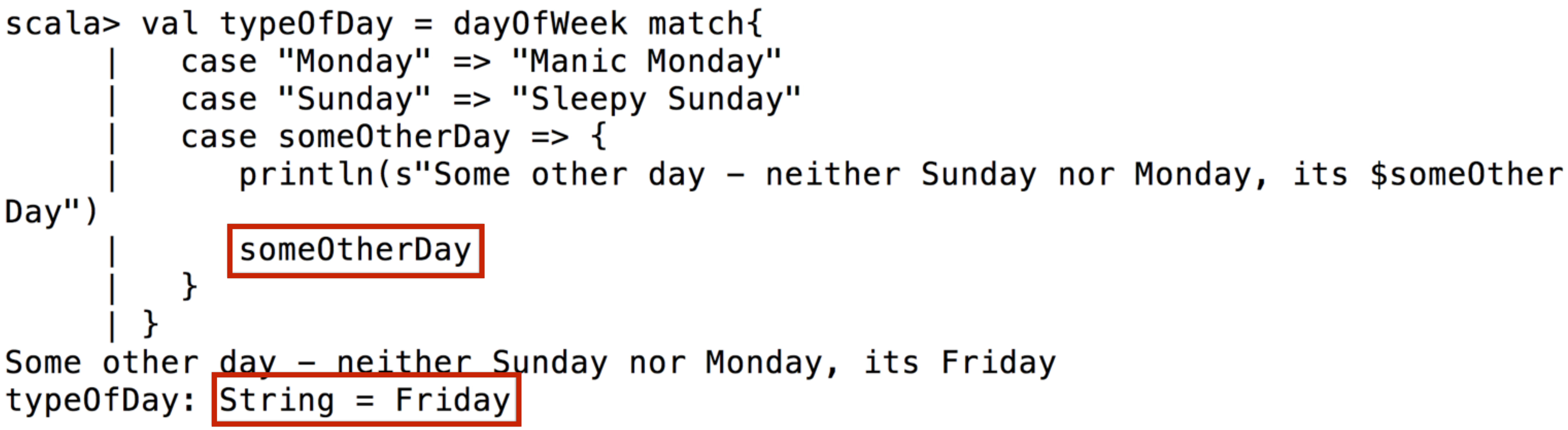
Value Binding Patterns



```
scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Sunday" => "Sleepy Sunday"
     |   case someOtherDay => {
     |      println(s"Some other day - neither Sunday nor Monday, its $someOther
Day")
     |      someOtherDay
     |   }
     | }
Some other day - neither Sunday nor Monday, its Friday
typeOfDay: String = Friday
```

This variable can then be used in the expression on the right side of the arrow

Value Binding Patterns



```
scala> val typeOfDay = dayOfWeek match{
     |   case "Monday" => "Manic Monday"
     |   case "Sunday" => "Sleepy Sunday"
     |   case someOtherDay => {
     |      println(s"Some other day – neither Sunday nor Monday, its $someOther
Day")
     |      someOtherDay
     |   }
     | }
Some other day – neither Sunday nor Monday, its Friday
typeOfDay: String = Friday
```

The result of the expression on the right will be returned - as it should be!

Scala.MatchError

We need something like the default case in a switch statement to prevent such errors

`match` expressions: catch-all to match-all

✓ Value Binding Patterns

_ (Wildcard Operator Patterns)

_ (Wildcard Operator Patterns)

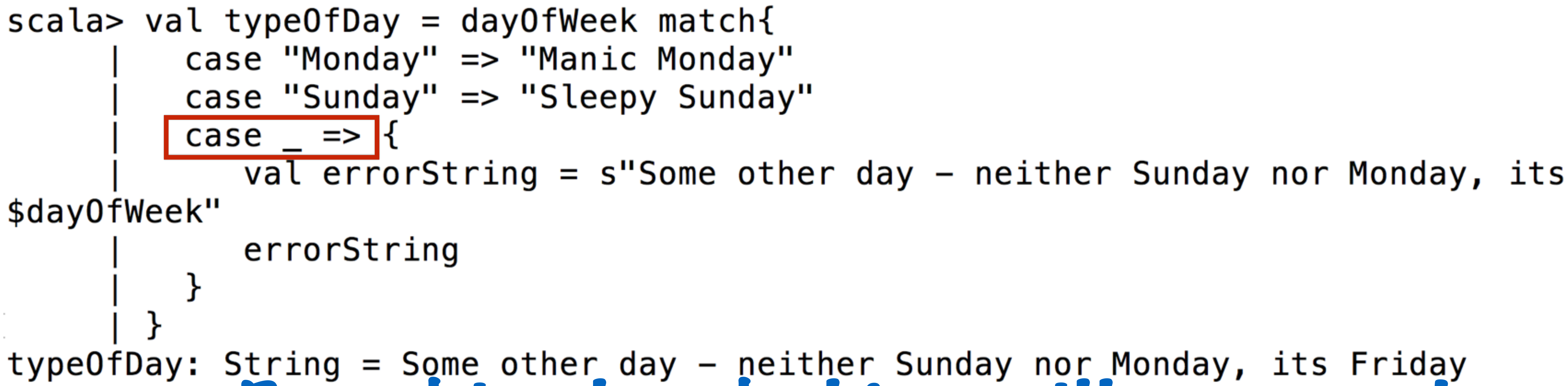
The underscore character _ as a placeholder will be a common theme in Scala

```
case _  => expression
```

Here, _ is an unnamed wildcard for the input value. It will match anything

_ (Wildcard Operator Patterns)

Here, _ is an unnamed wildcard for the input value. It will match anything

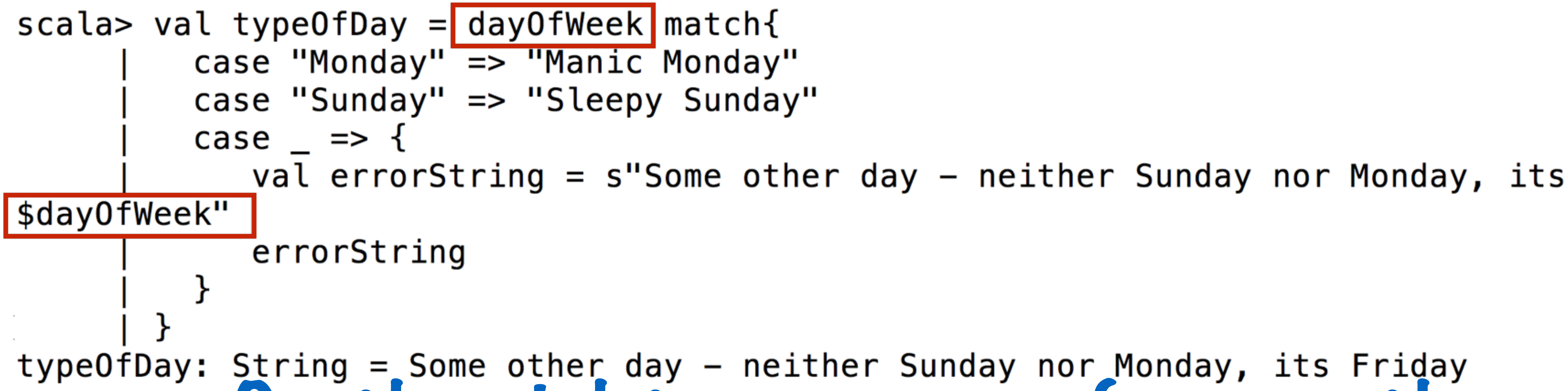


```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday" => "Manic Monday"
     |     case "Sunday" => "Sleepy Sunday"
     |     case _ => {
     |         val errorString = s"Some other day - neither Sunday nor Monday, its
$dayOfWeek"
     |         errorString
     |     }
     | }
typeOfDay: String = Some other day - neither Sunday nor Monday, its Friday
```

But this placeholder will not work on the right of the => sign!

_ (Wildcard Operator Patterns)

But this placeholder will not work on the right of the => sign!

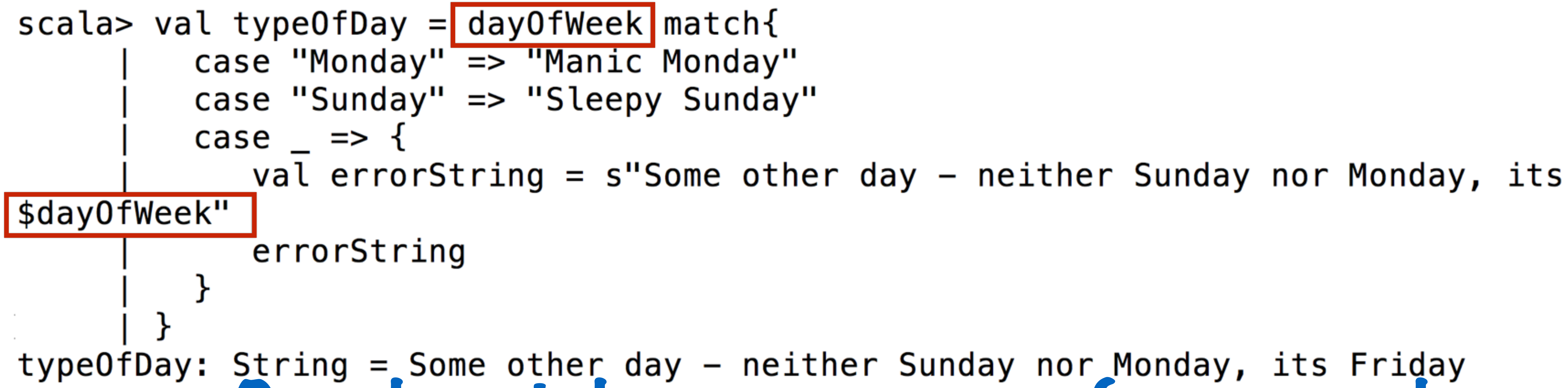


```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday" => "Manic Monday"
     |     case "Sunday" => "Sleepy Sunday"
     |     case _ => {
     |         val errorString = s"Some other day - neither Sunday nor Monday, its
$dayOfWeek"
     |         errorString
     |     }
     | }
typeOfDay: String = Some other day - neither Sunday nor Monday, its Friday
```

On the right, you can reference the original match variable

_ (Wildcard Operator Patterns)

But this placeholder will not work on the right of the => sign!

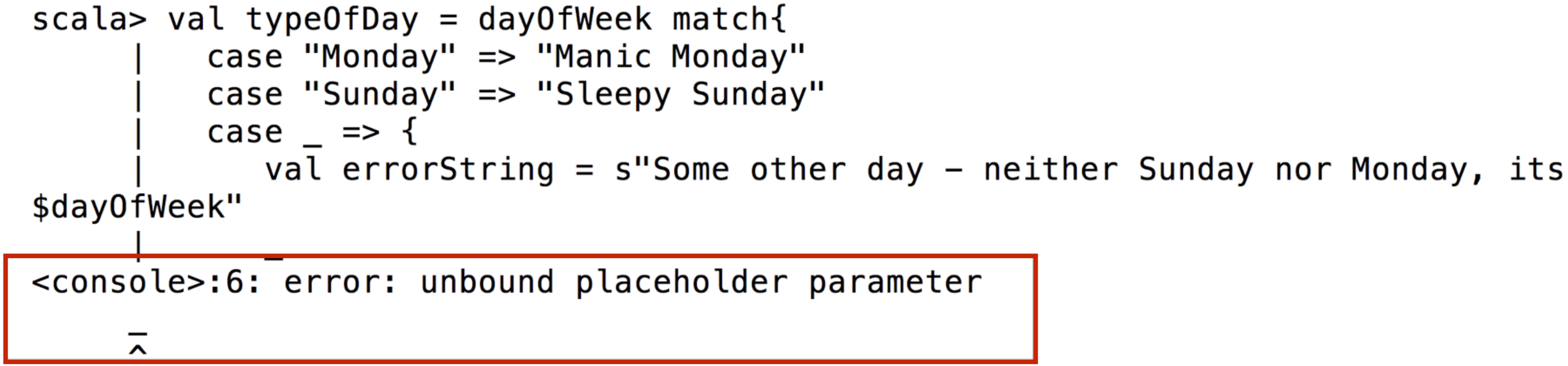


```
scala> val typeOfDay = dayOfWeek match{
     |     case "Monday" => "Manic Monday"
     |     case "Sunday" => "Sleepy Sunday"
     |     case _ => {
     |         val errorString = s"Some other day - neither Sunday nor Monday, its 
$dayOfWeek"
     |         errorString
     |     }
     | }
typeOfDay: String = Some other day - neither Sunday nor Monday, its Friday
```

On the right, you can reference the original match variable

_ (Wildcard Operator Patterns)

But this placeholder will not work on the right of the => sign!



```
scala> val typeOfDay = dayOfWeek match{
     |    case "Monday" => "Manic Monday"
     |    case "Sunday" => "Sleepy Sunday"
     |    case _ => {
     |       val errorString = s"Some other day - neither Sunday nor Monday, its 
$dayOfWeek"
     |
            _
<console>:6: error: unbound placeholder parameter
     _
     ^
```

Attempting to access _ on the right of the => will result in an error

Scala.MatchError

We need something like the default case in a switch statement to prevent such errors

`match` expressions: catch-all to match-all

✓ Value Binding Patterns

✓ _ (Wildcard Operator Patterns)

Example 16

`match` expressions: down casting with Pattern Variables

`match` expressions: down casting with Pattern Variables

In Java, a common use case of nested if statements is to downcast using `instanceof`

Btw, its a key failing of the Java `switch` statement that it can't predicate on type

Scala's `match` is carefully built to test on type of the match variable

match expressions: down casting with Pattern Variables

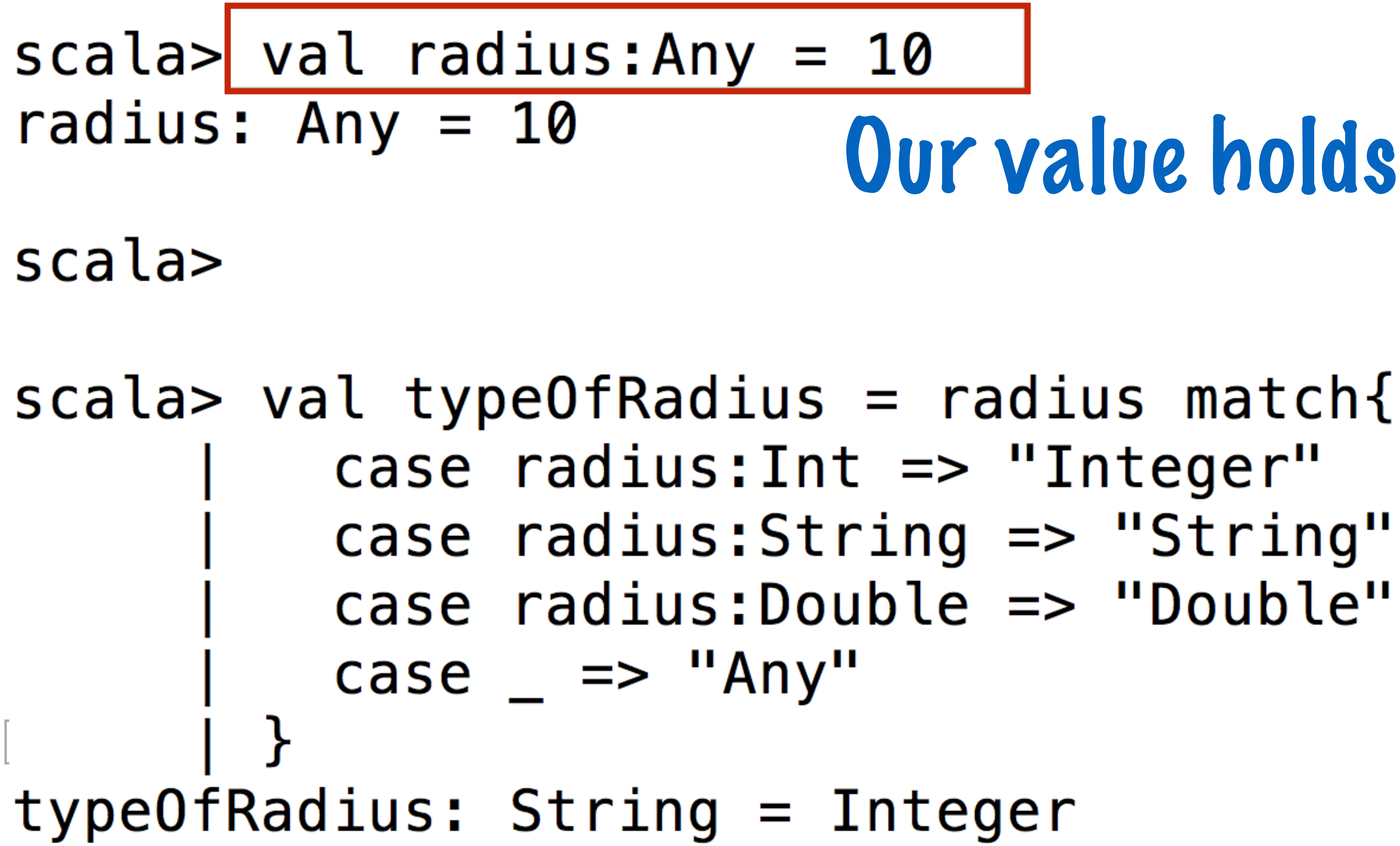
There is a special type of case clause, which tests the case of a variable

```
case <identifier> : <Type> => <expression>
```

Here, someVar is an unnamed wildcard for the input value. It will match anything

Mini-example #1

match expressions: down casting with Pattern Variables

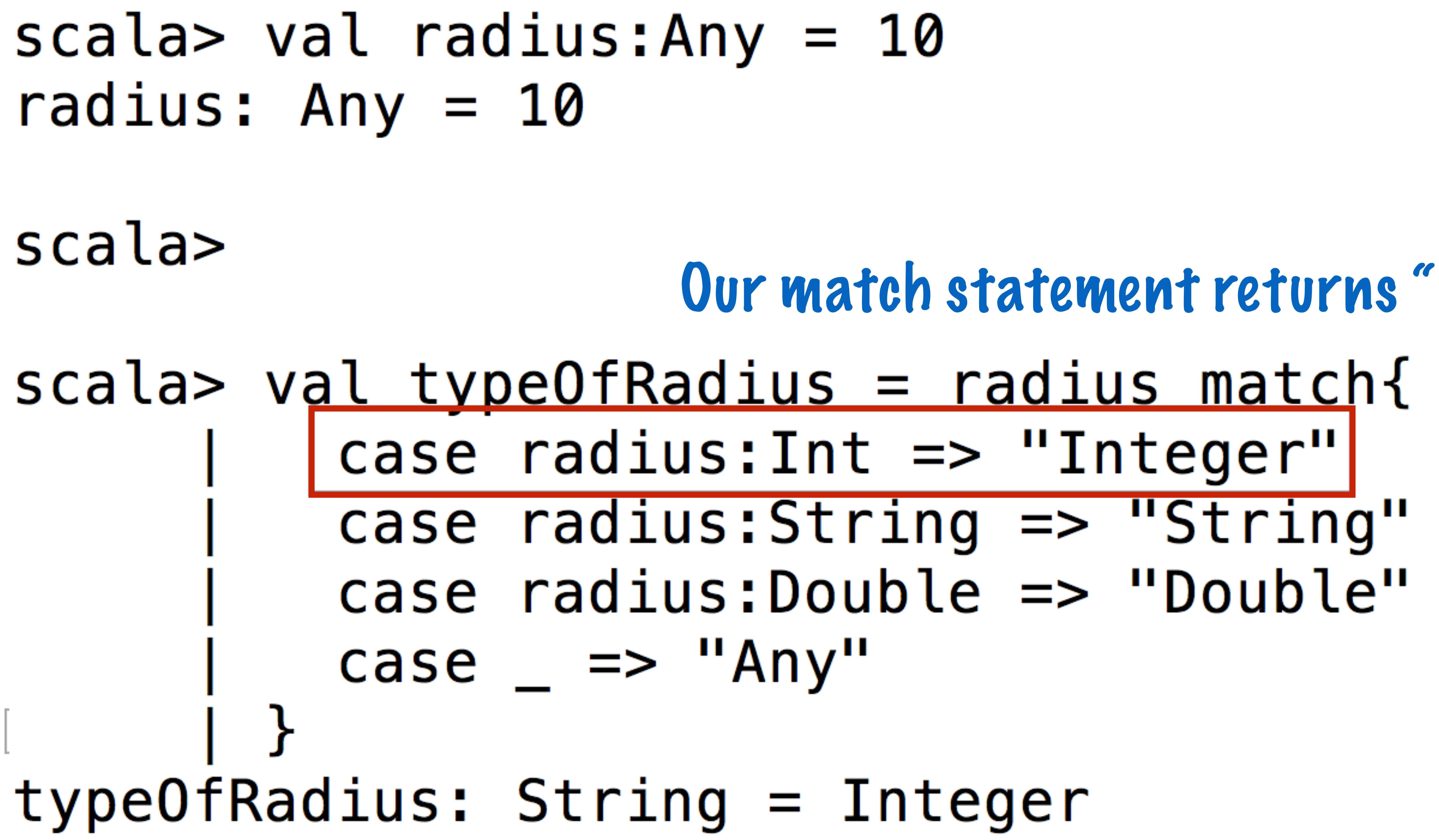


```
scala> val radius:Any = 10
radius: Any = 10

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case radius:Double => "Double"
     |     case _ => "Any"
     | }
typeOfRadius: String = Integer
```

Our value holds an Int

Mini-example #1

`match` expressions: down casting with Pattern Variables

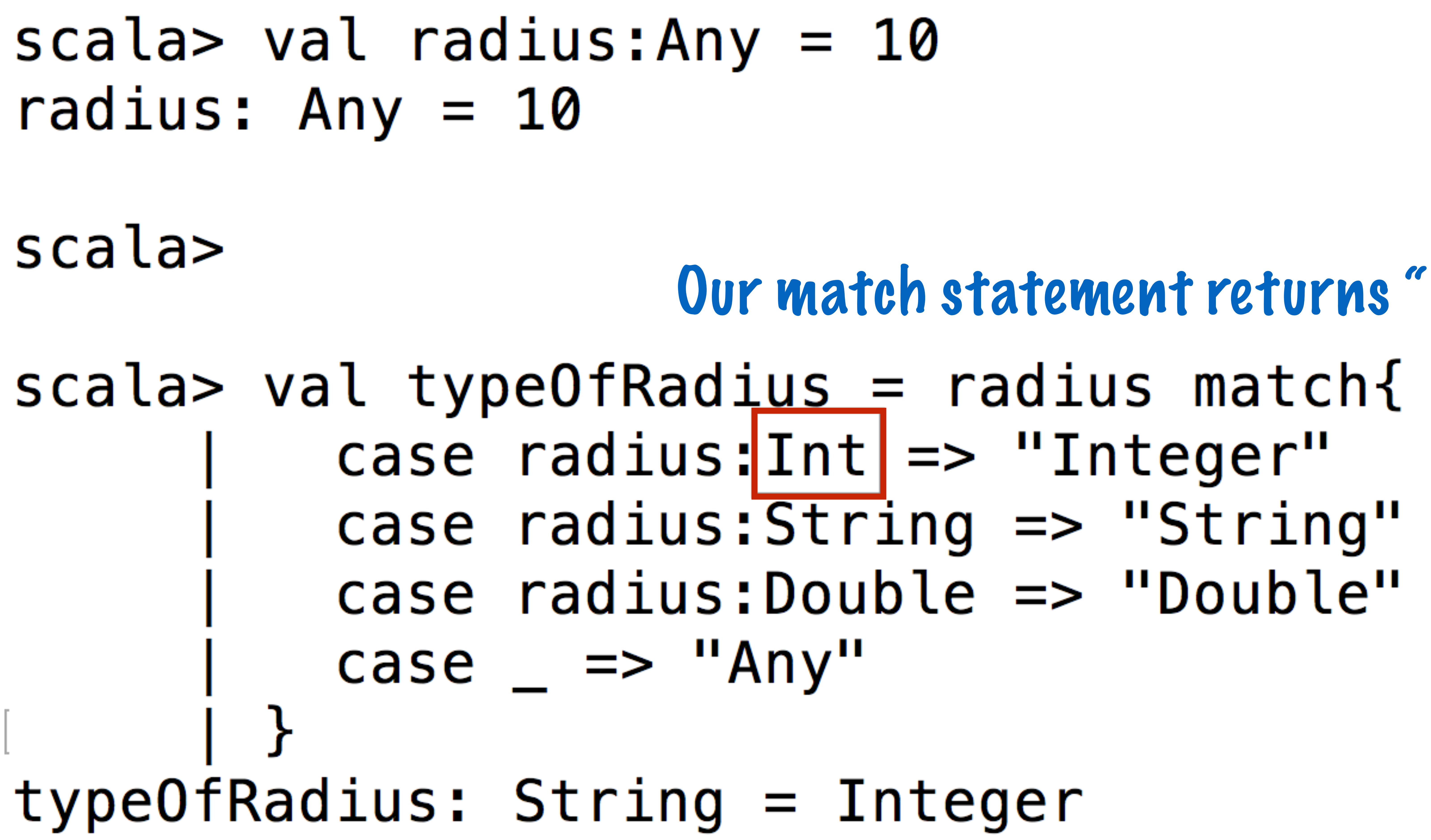


```
scala> val radius:Any = 10
radius: Any = 10

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case radius:String => "String"
     |    case radius:Double => "Double"
     |    case _ => "Any"
     | }
typeOfRadius: String = Integer
```

Our match statement returns "Integer"

Mini-example #1

`match` expressions: down casting with Pattern Variables

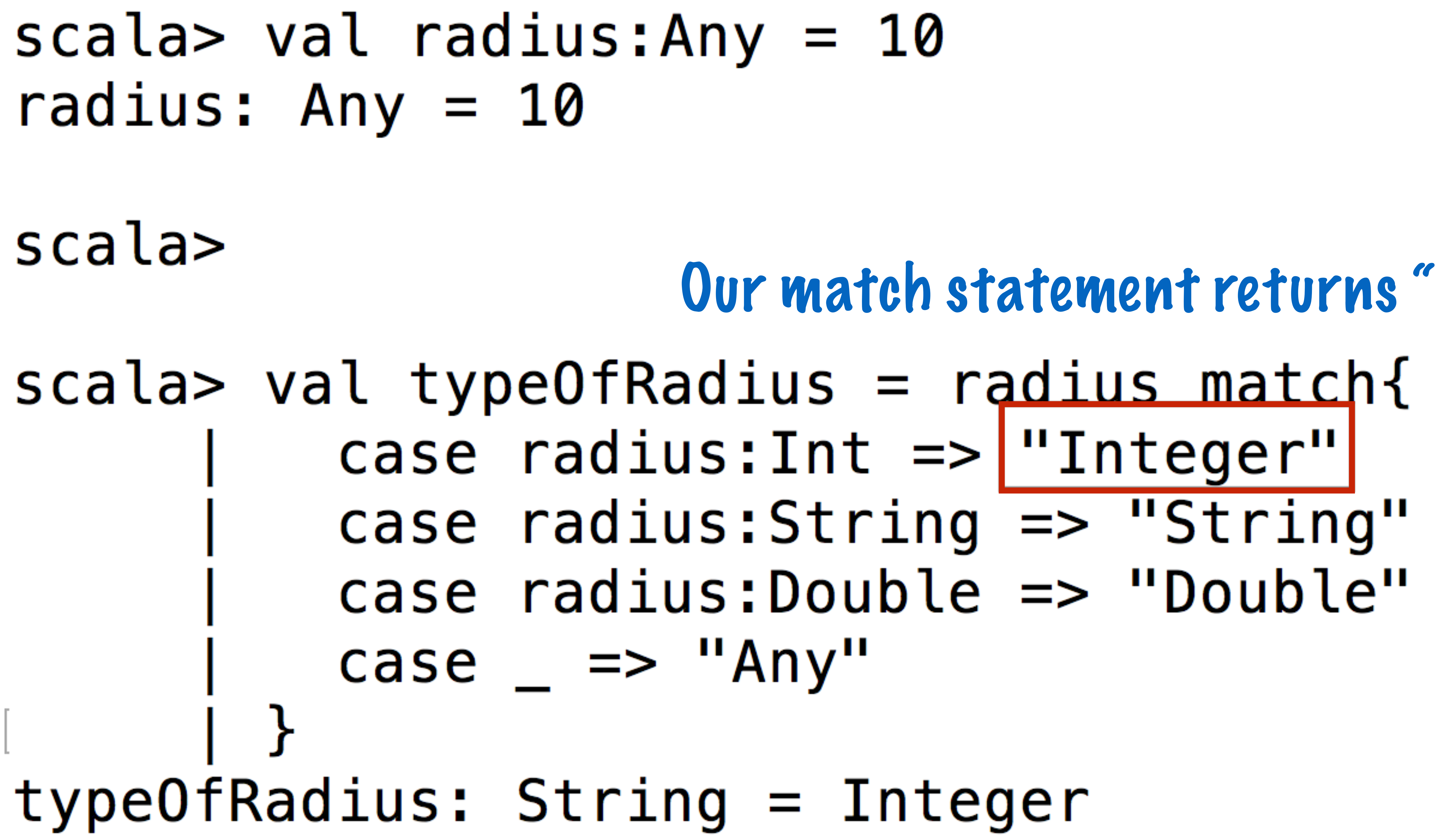


```
scala> val radius:Any = 10
radius: Any = 10

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case radius:String => "String"
     |    case radius:Double => "Double"
     |    case _ => "Any"
     | }
typeOfRadius: String = Integer
```

Our match statement returns "Integer"

Mini-example #1

`match` expressions: down casting with Pattern Variables

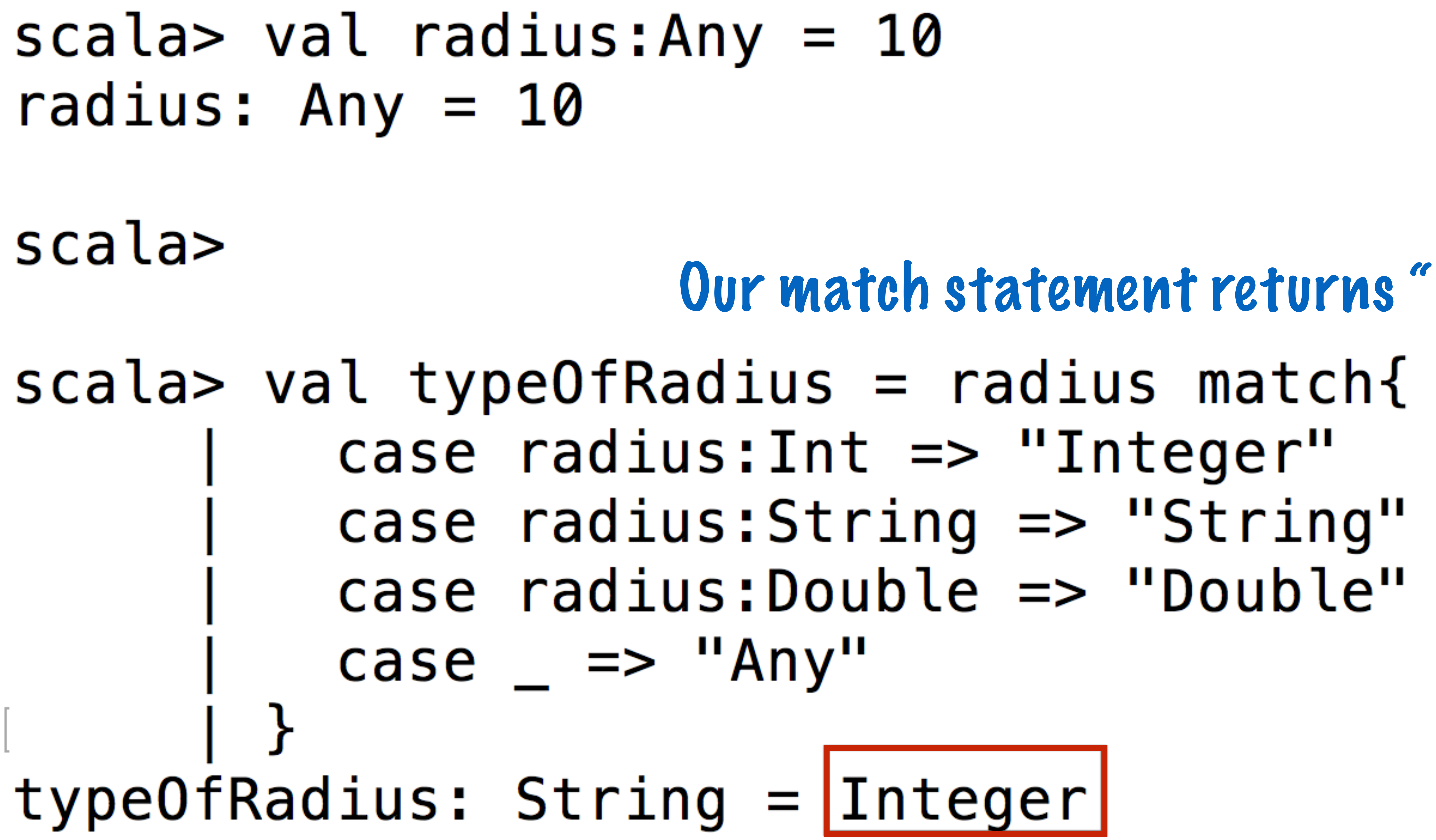


```
scala> val radius:Any = 10
radius: Any = 10

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case radius:String => "String"
     |    case radius:Double => "Double"
     |    case _ => "Any"
     | }
typeOfRadius: String = Integer
```

Our match statement returns "Integer"

Mini-example #1

match expressions: down casting with Pattern Variables



```
scala> val radius:Any = 10
radius: Any = 10

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case radius:Double => "Double"
     |     case _ => "Any"
     | }
typeOfRadius: String = Integer
```

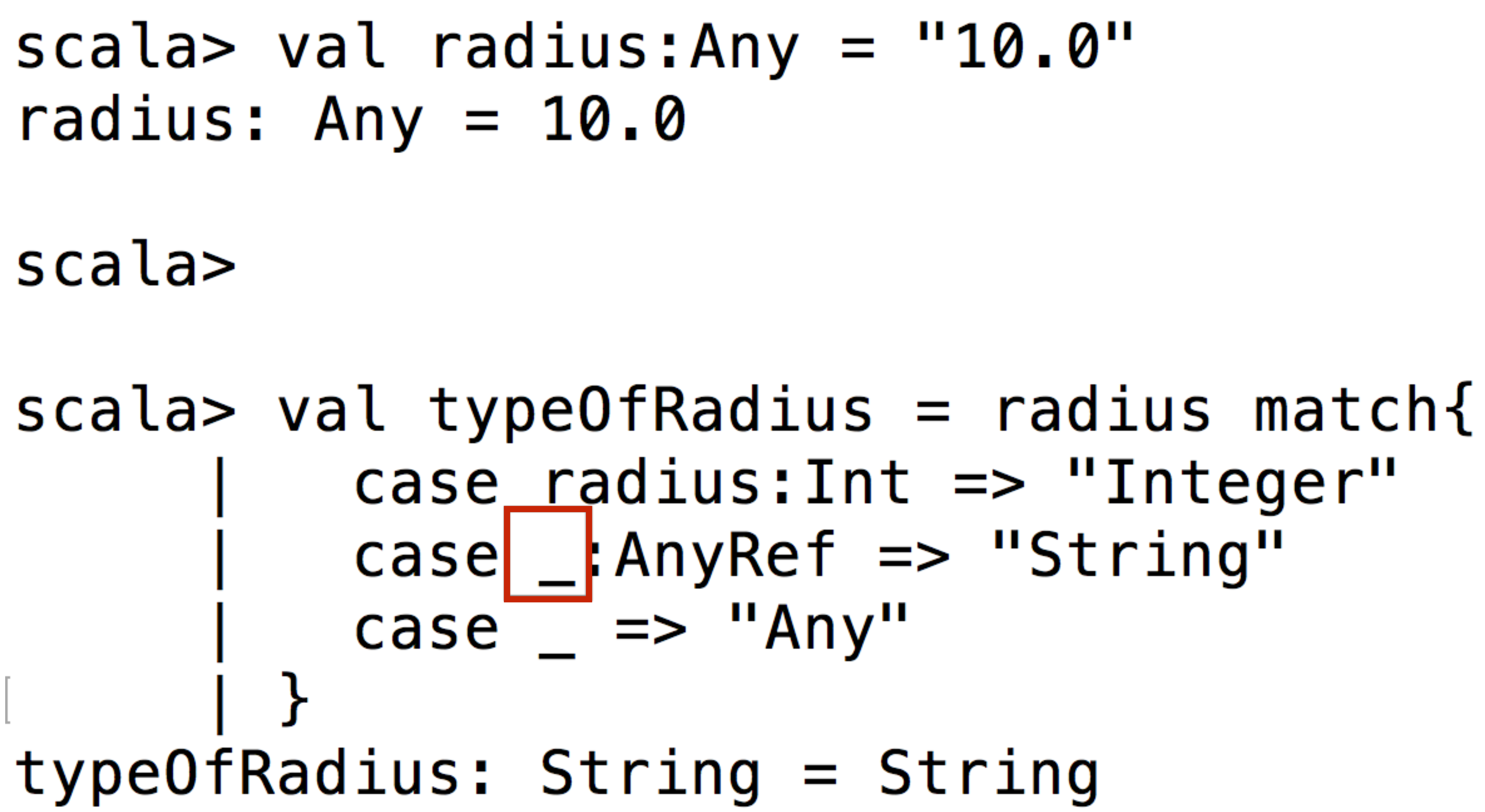
Our match statement returns "Integer"

Mini-example #2

The placeholder _ can be used as the pattern variable

Mini-example #2

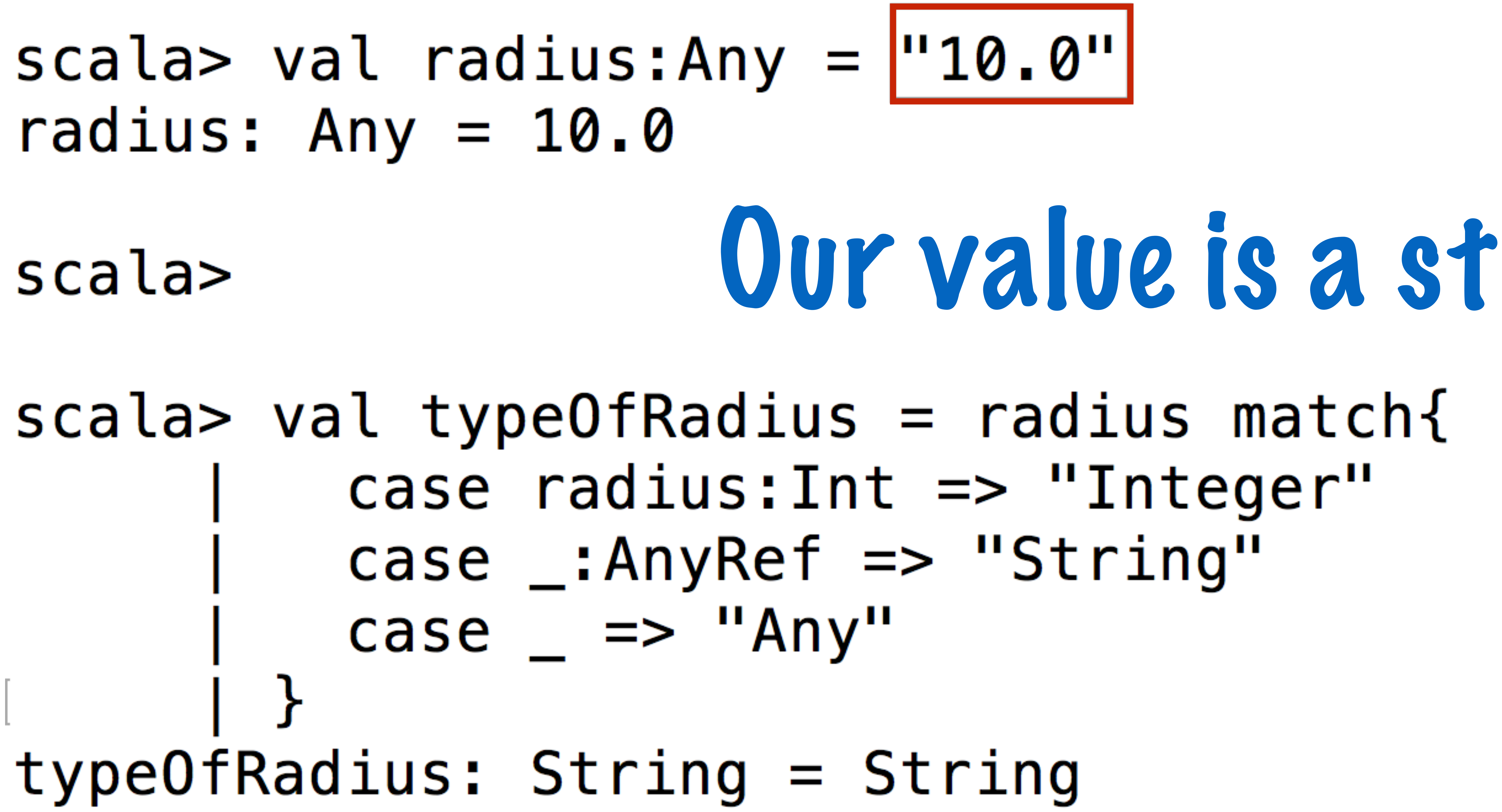
The placeholder _ can be used as the pattern variable



```
scala> val radius:Any = "10.0"
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case _:AnyRef => "String"
     |     case _ => "Any"
     |  }
typeOfRadius: String = String
```

Mini-example #2

The placeholder _ can be used as the pattern variable

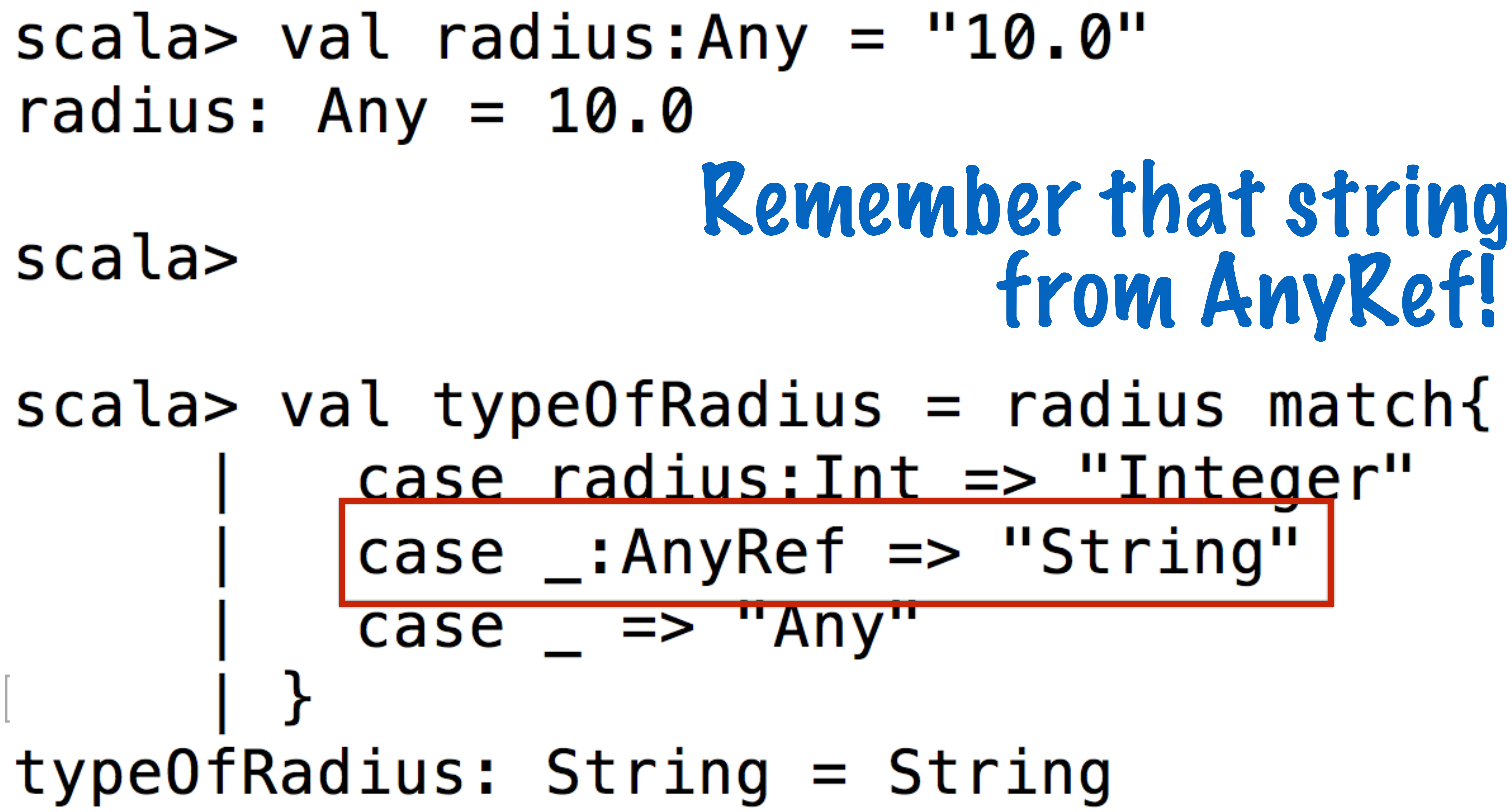


```
scala> val radius:Any = "10.0"
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |   case radius:Int => "Integer"
     |   case _:AnyRef => "String"
     |   case _ => "Any"
     | }
typeOfRadius: String = String
```

Our value is a string

Mini-example #2

The placeholder _ can be used as the pattern variable

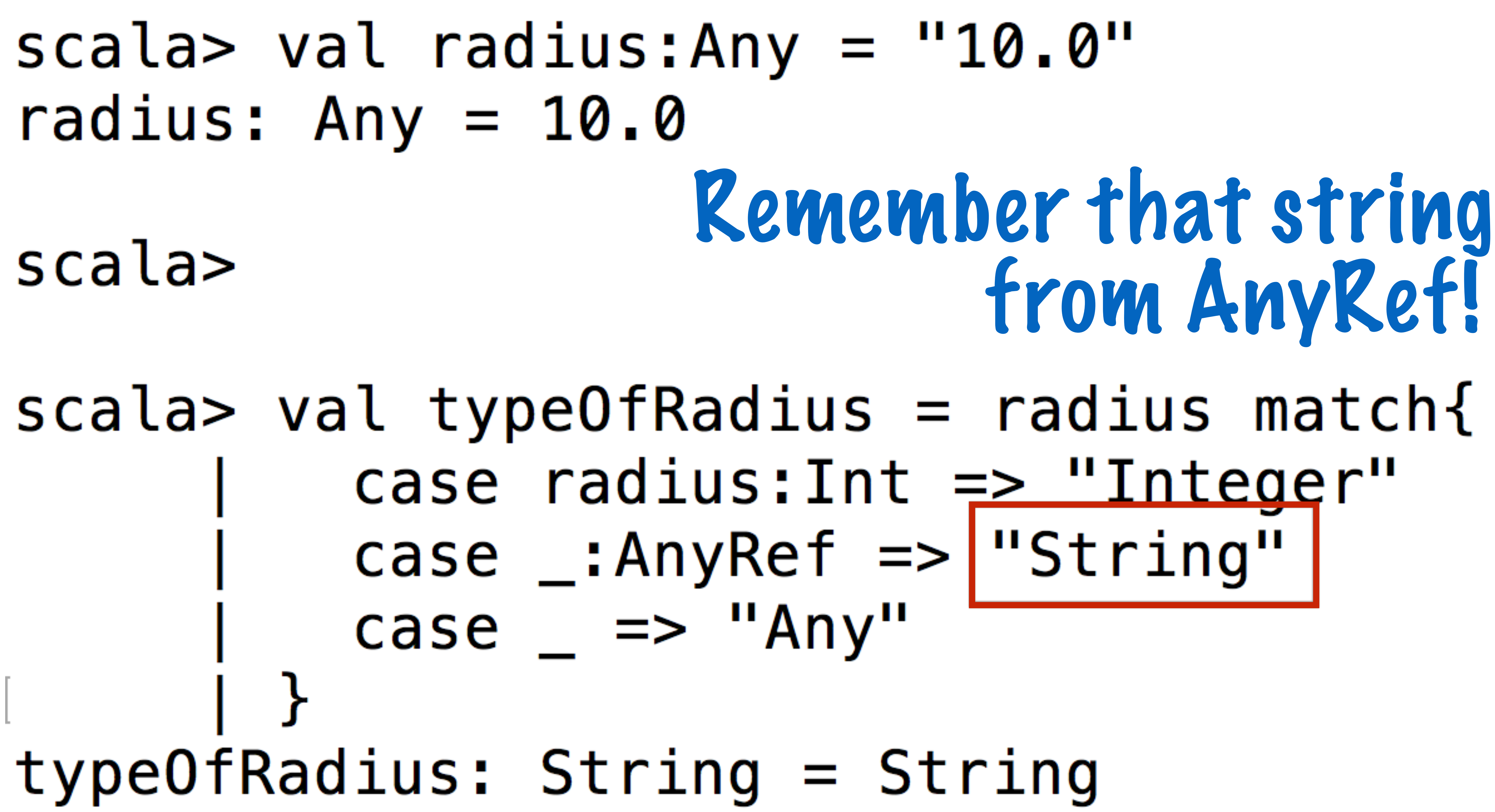


```
scala> val radius:Any = "10.0"
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case _:AnyRef => "String"
     |     case _ => "Any"
     | }
typeOfRadius: String = String
```

Remember that string derives from AnyRef!

Mini-example #2

The placeholder _ can be used as the pattern variable

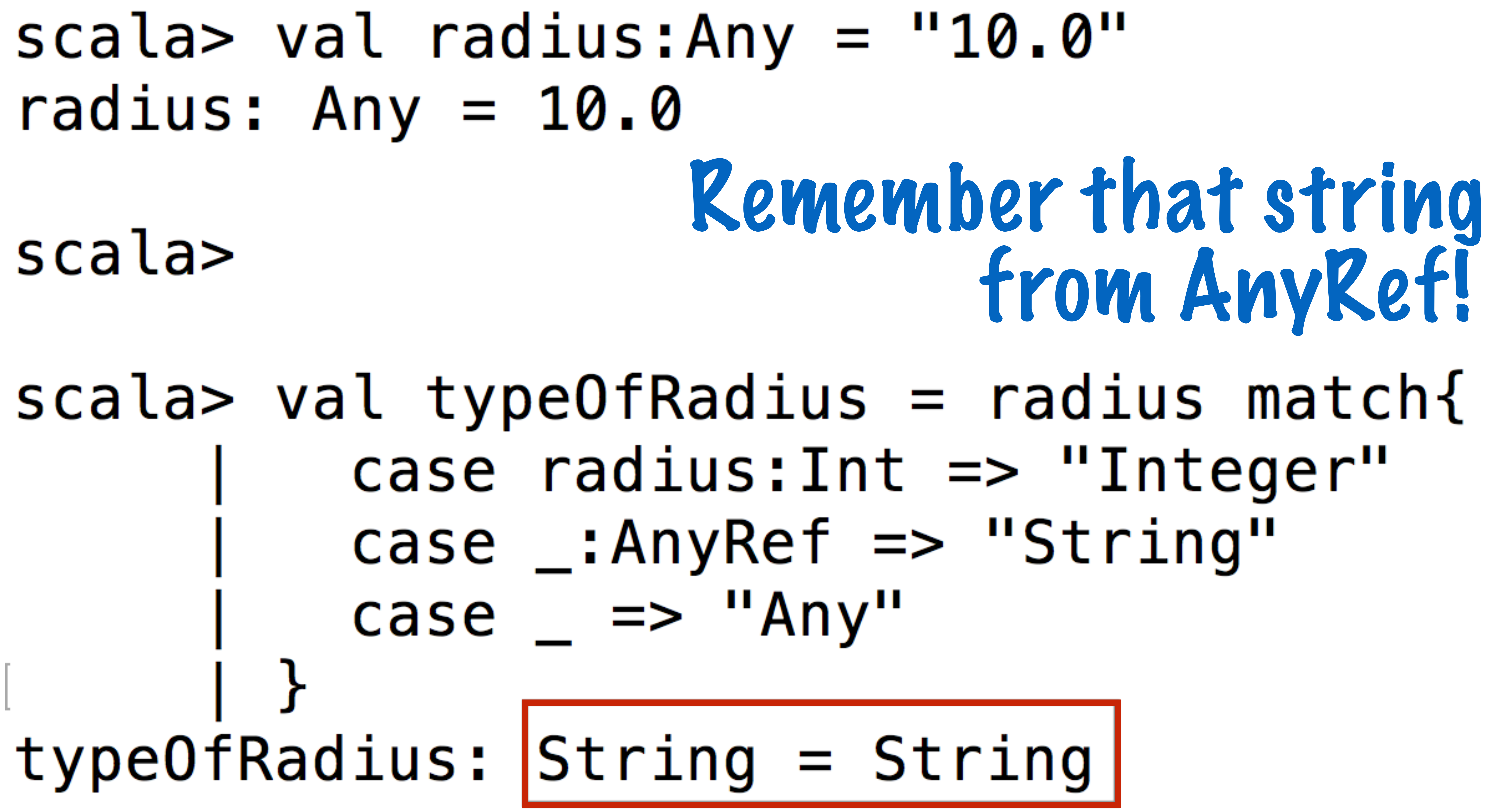


```
scala> val radius:Any = "10.0"
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case _:AnyRef => "String"
     |    case _ => "Any"
     | }
typeOfRadius: String = String
```

Remember that string derives from AnyRef!

Mini-example #2

The placeholder _ can be used as the pattern variable



```
scala> val radius:Any = "10.0"
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |   case radius:Int => "Integer"
     |   case _:AnyRef => "String"
     |   case _ => "Any"
     | }
typeOfRadius: String = String
```

Remember that string derives from AnyRef!

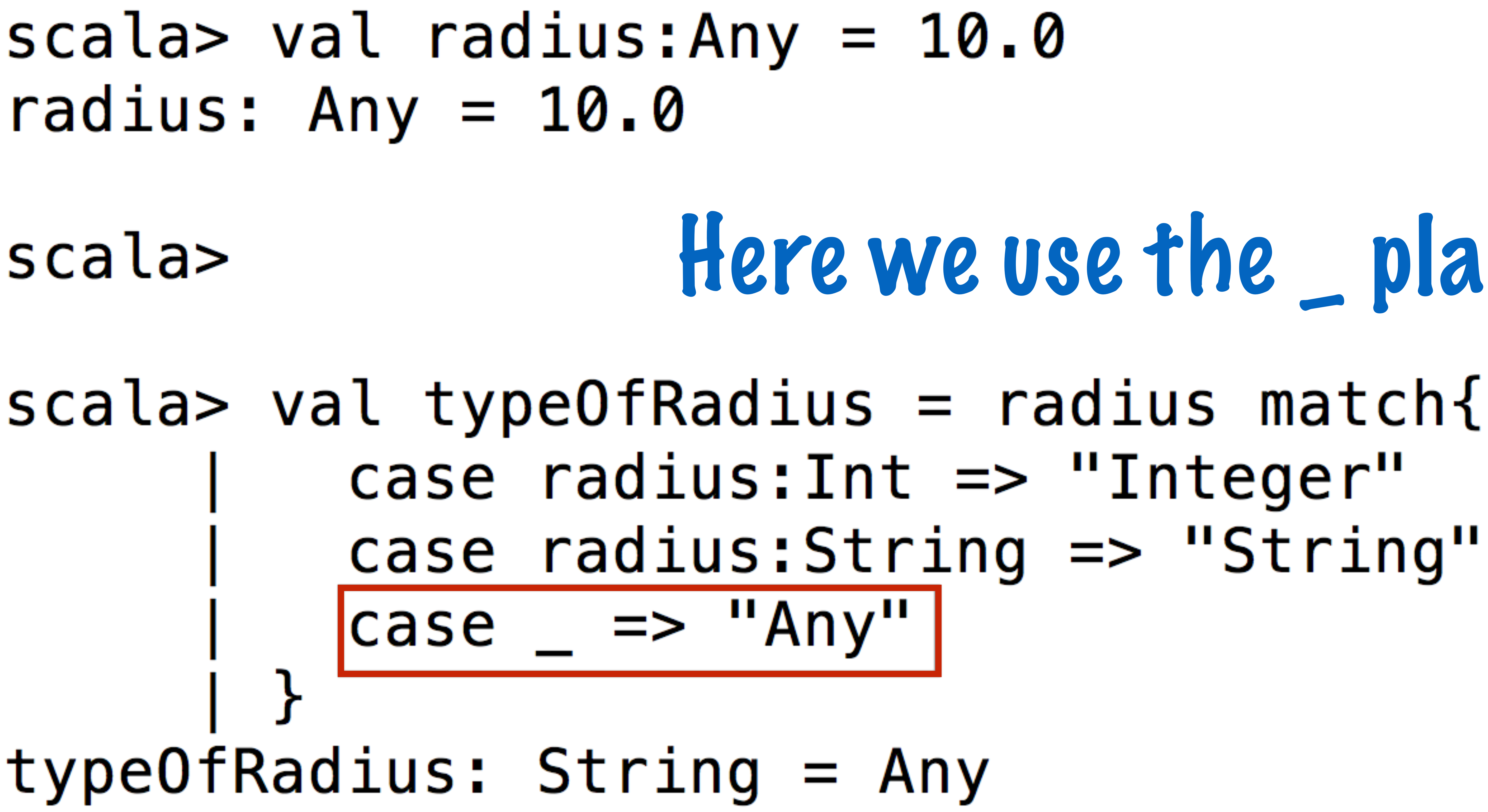
Mini-example #3

A catch-all/match-all will work as usual

You can use either a placeholder or a value binding to make sure that some case is always satisfied

Mini-example #3

A catch-all/match-all will work as usual

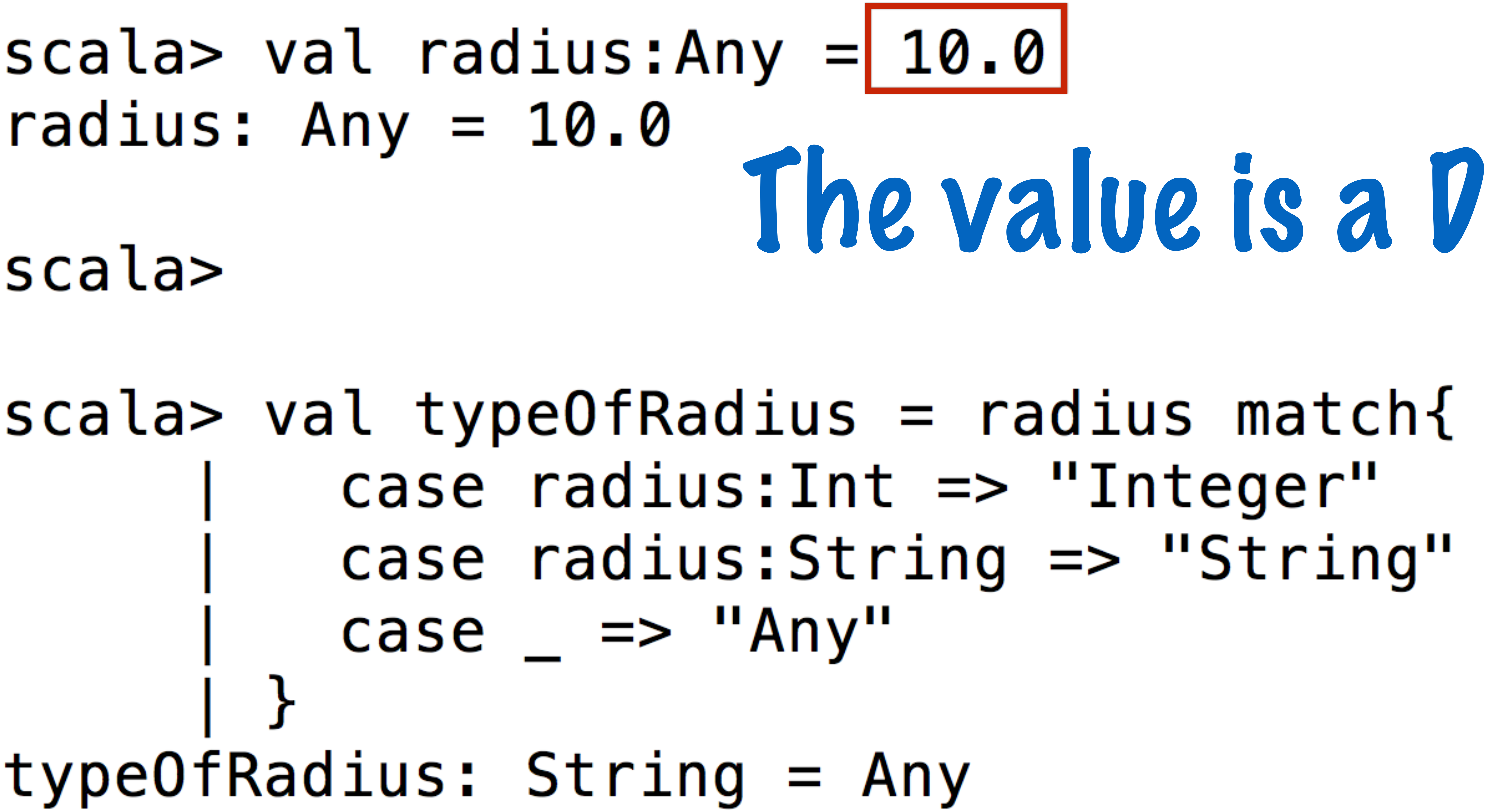


```
scala> val radius:Any = 10.0
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case _ => "Any"
     | }
typeOfRadius: String = Any
```

Here we use the _ placeholder

Mini-example #3

A catch-all/match-all will work as usual

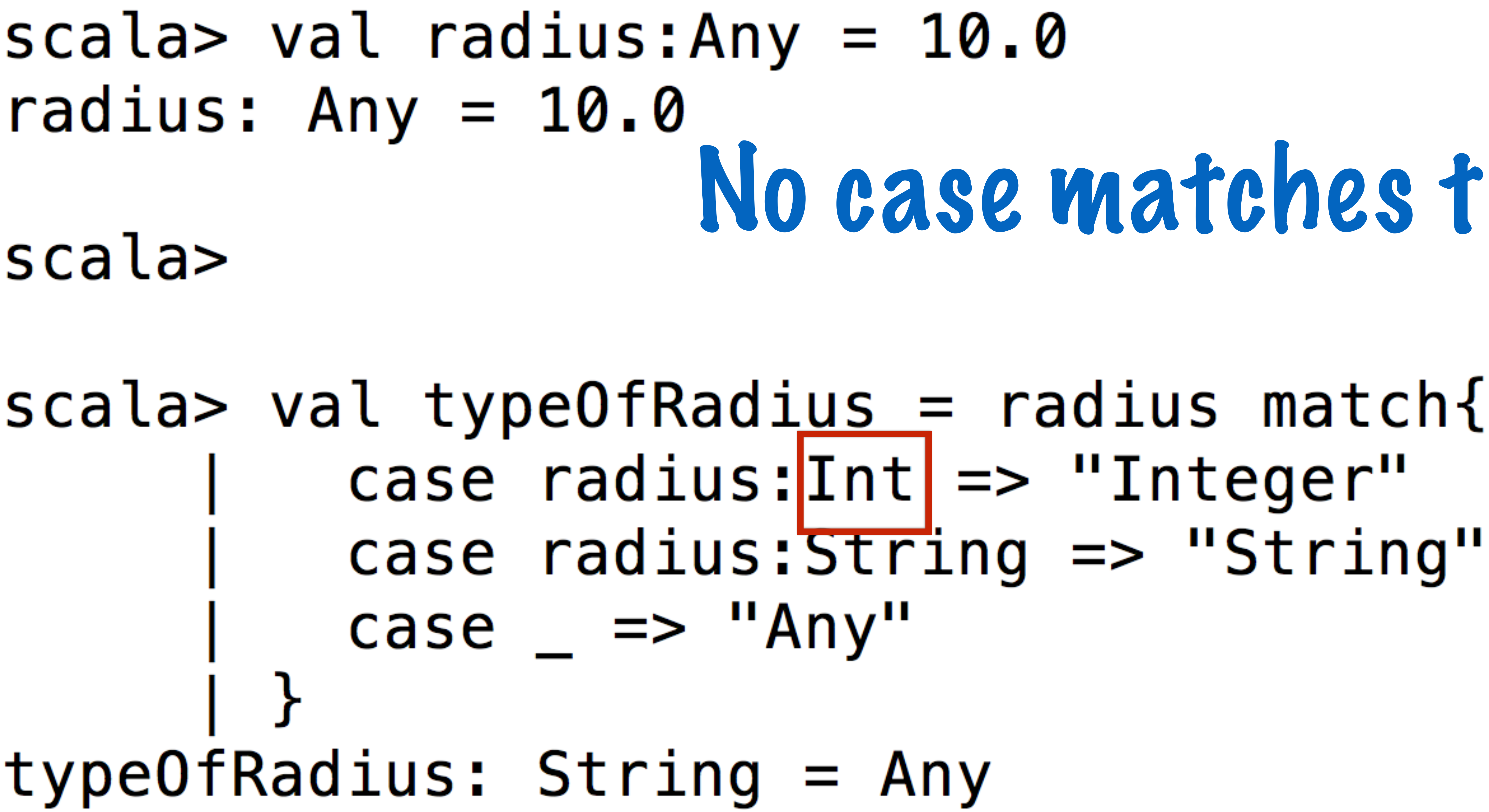


```
scala> val radius:Any = 10.0
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case _ => "Any"
     | }
typeOfRadius: String = Any
```

The value is a Double

Mini-example #3

A catch-all/match-all will work as usual

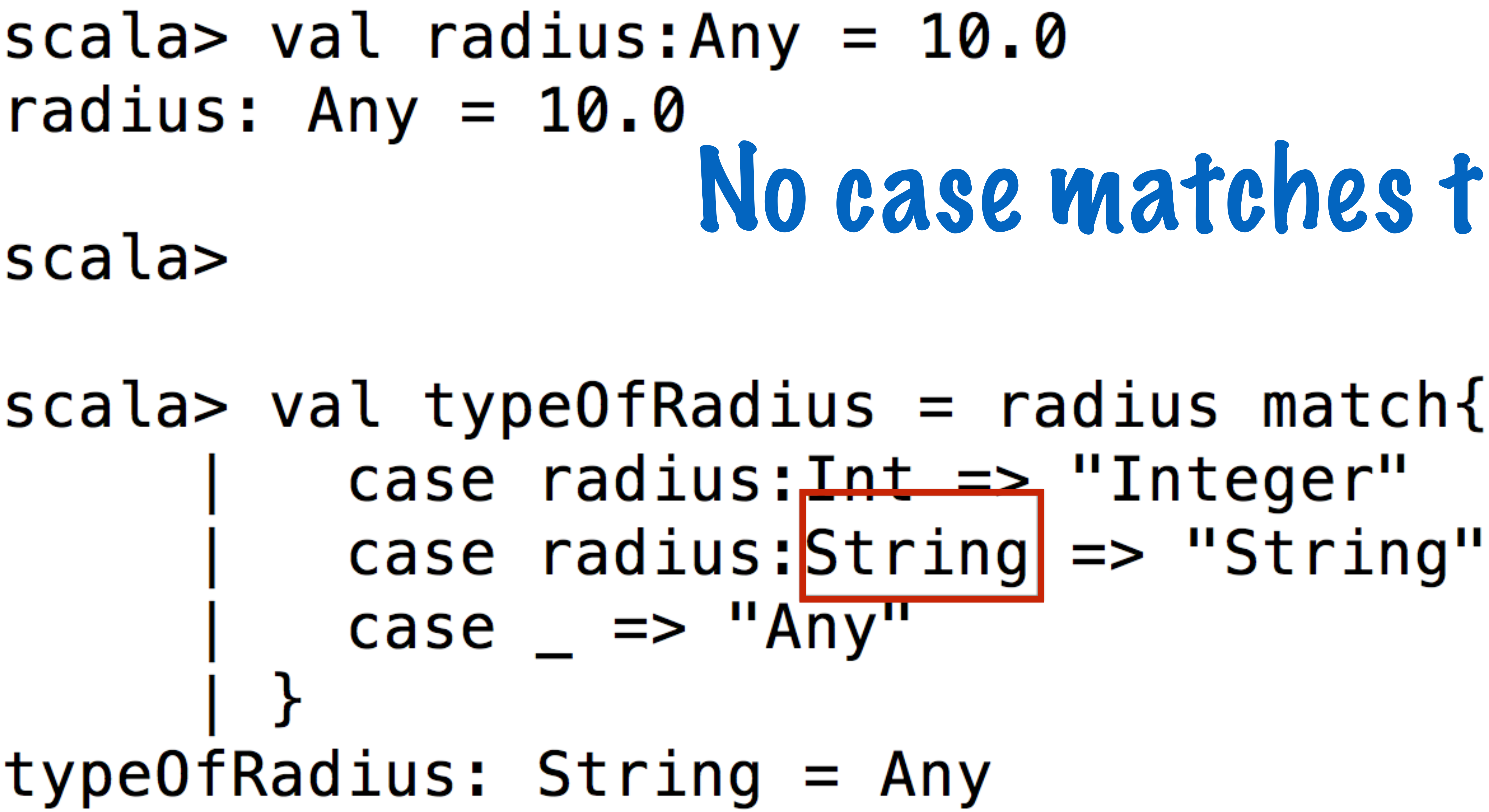


```
scala> val radius:Any = 10.0
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case _ => "Any"
     | }
typeOfRadius: String = Any
```

No case matches this type

Mini-example #3

A catch-all/match-all will work as usual

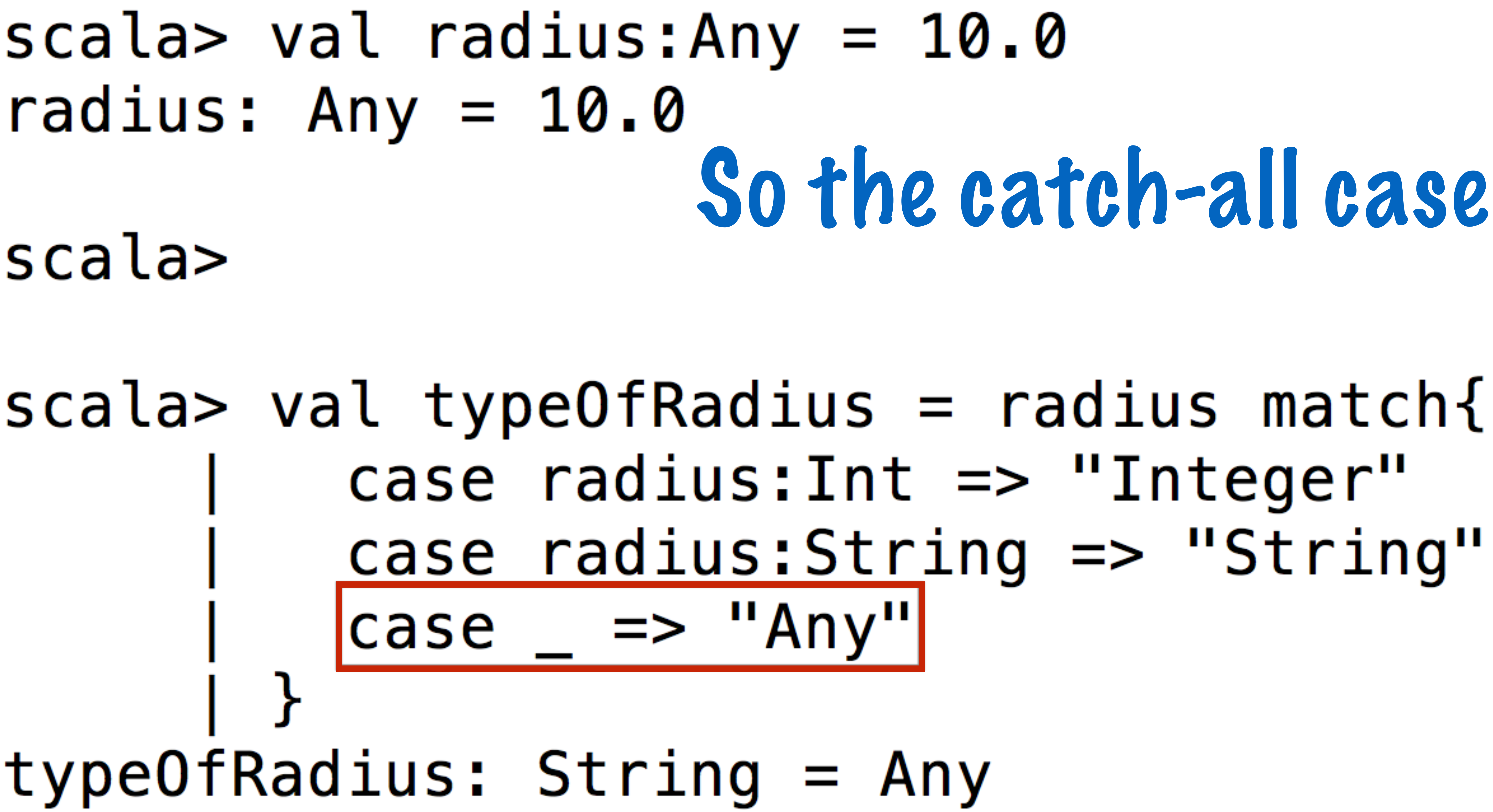


```
scala> val radius:Any = 10.0
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case _ => "Any"
     | }
typeOfRadius: String = Any
```

No case matches this type

Mini-example #3

A catch-all/match-all will work as usual

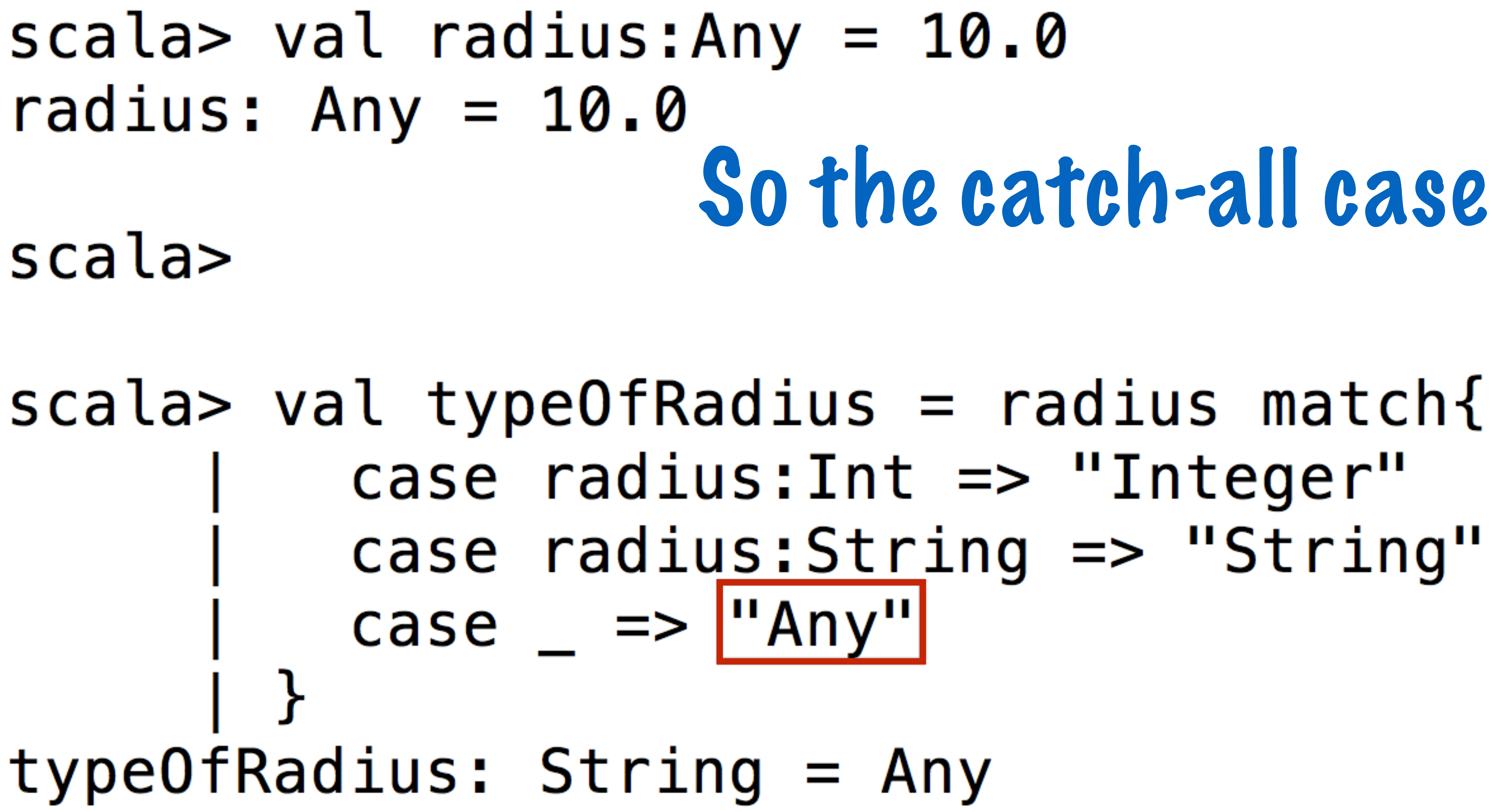


```
scala> val radius:Any = 10.0
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case radius:String => "String"
     |    case _ => "Any"
     | }
typeOfRadius: String = Any
```

So the catch-all case kicks in!

Mini-example #3

A catch-all/match-all will work as usual

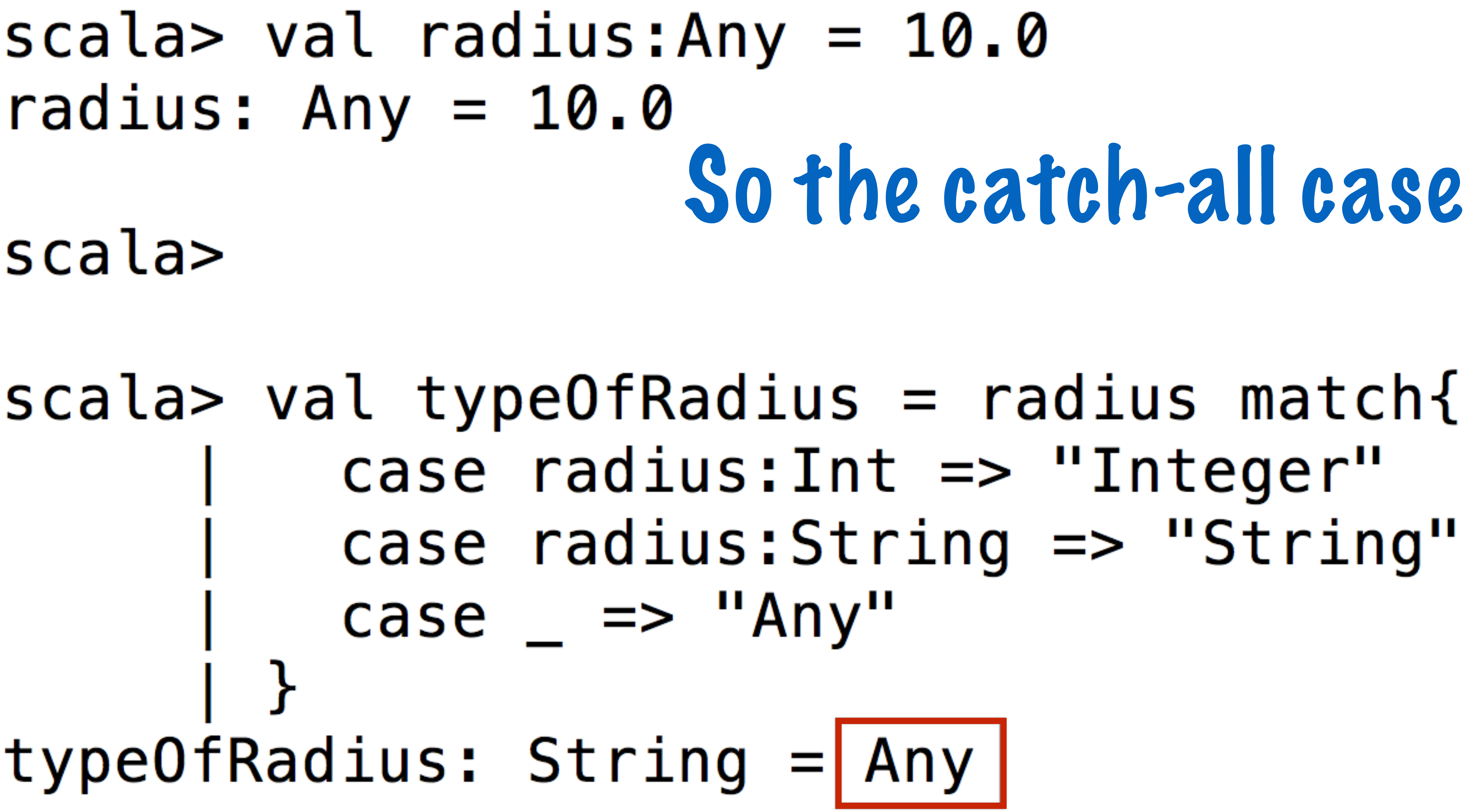


```
scala> val radius:Any = 10.0
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case radius:String => "String"
     |    case _ => "Any"
     | }
typeOfRadius: String = Any
```

So the catch-all case kicks in!

Mini-example #3

A catch-all/match-all will work as usual



```
scala> val radius:Any = 10.0
radius: Any = 10.0

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case _ => "Any"
     | }
typeOfRadius: String = Any
```

So the catch-all case kicks in!

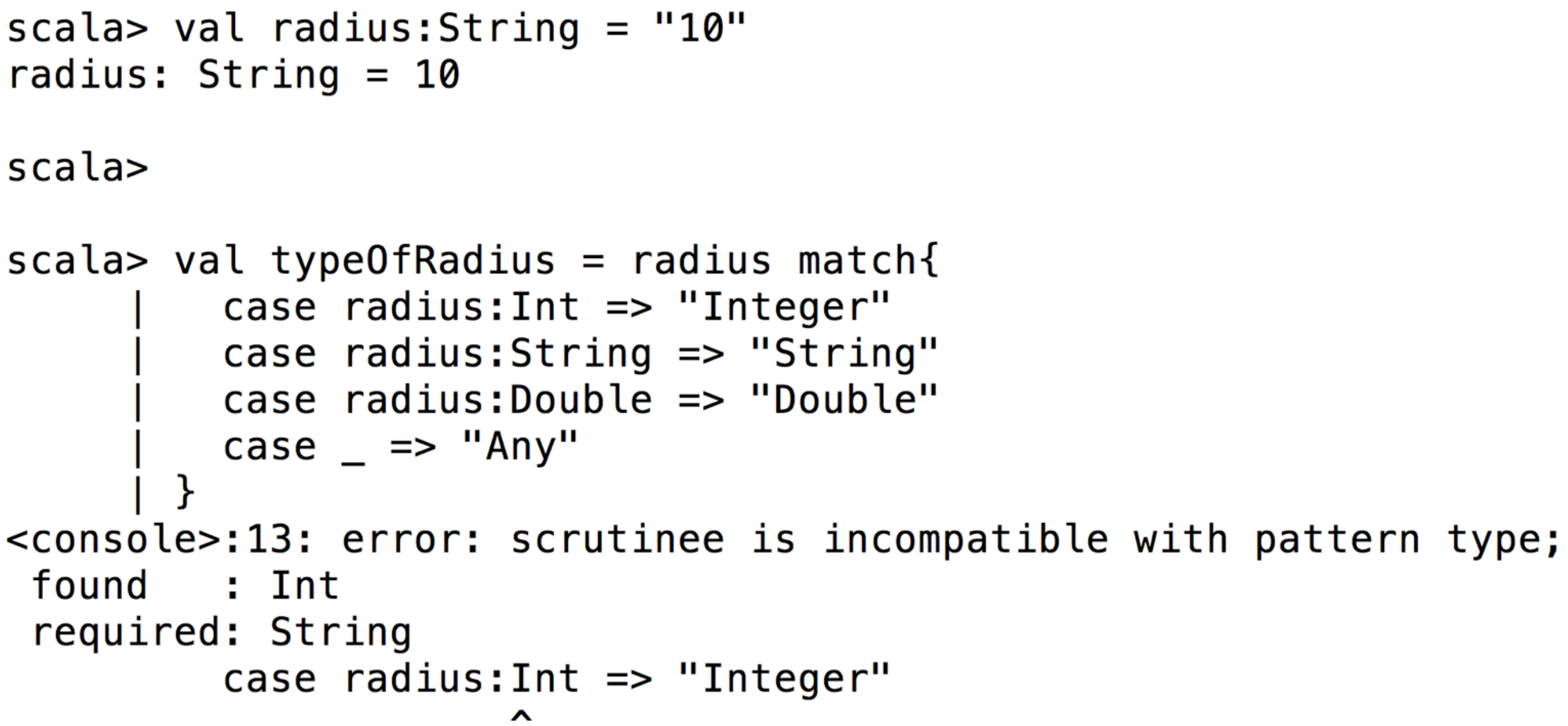
Mini-example #4

The “scrutinee” (variable whose type is matched) must be a base type

Else an error will result

Mini-example #4

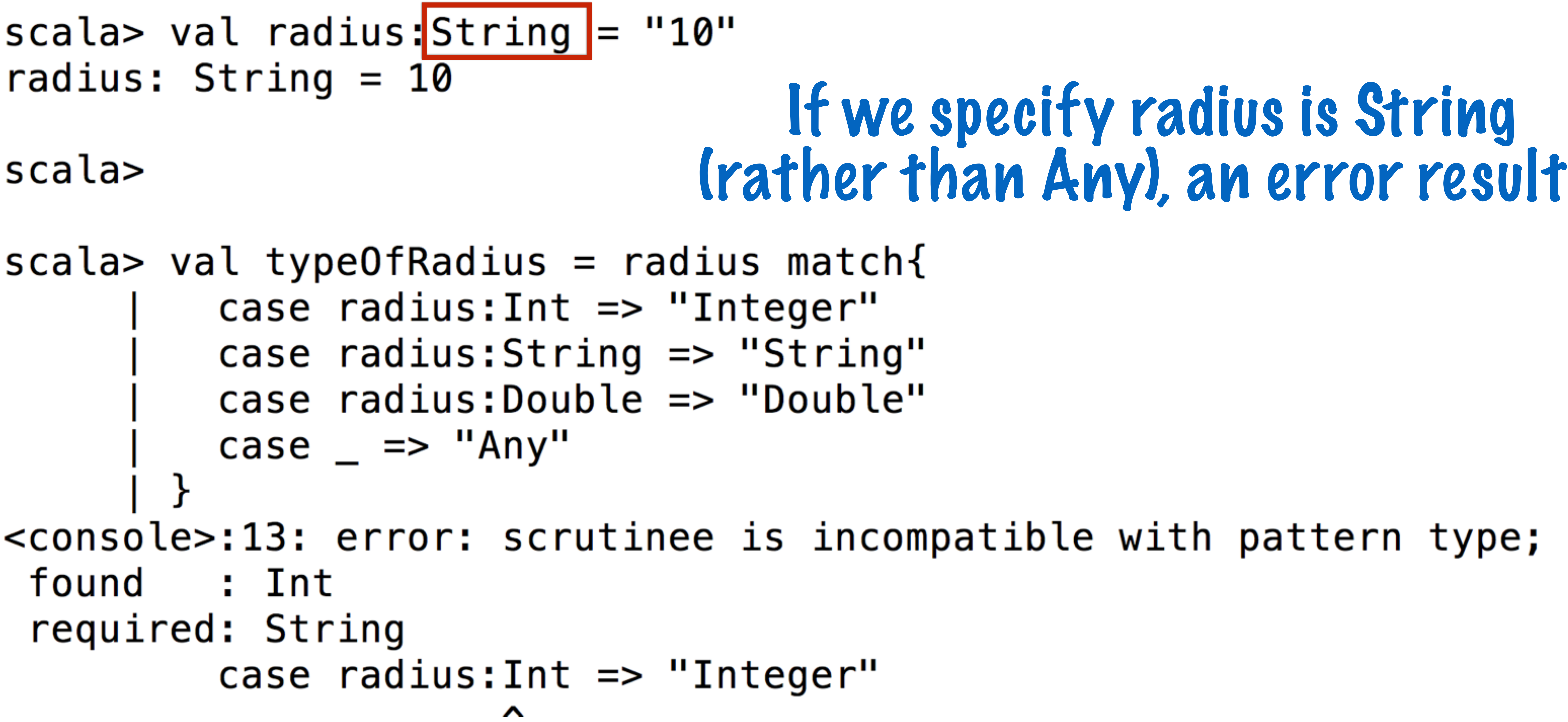
The "scrutinee" (variable whose type is matched) must be a base type



```
scala> val radius:String = "10"
radius: String = 10

scala>


scala> val typeOfRadius = radius match{
     |   case radius:Int => "Integer"
     |   case radius:String => "String"
     |   case radius:Double => "Double"
     |   case _ => "Any"
     | }
<console>:13: error: scrutinee is incompatible with pattern type;
 found   : Int
 required: String
         case radius:Int => "Integer"
                     ^
```

Mini-example #4

The "scrutinee" (variable whose type is matched) must be a base type

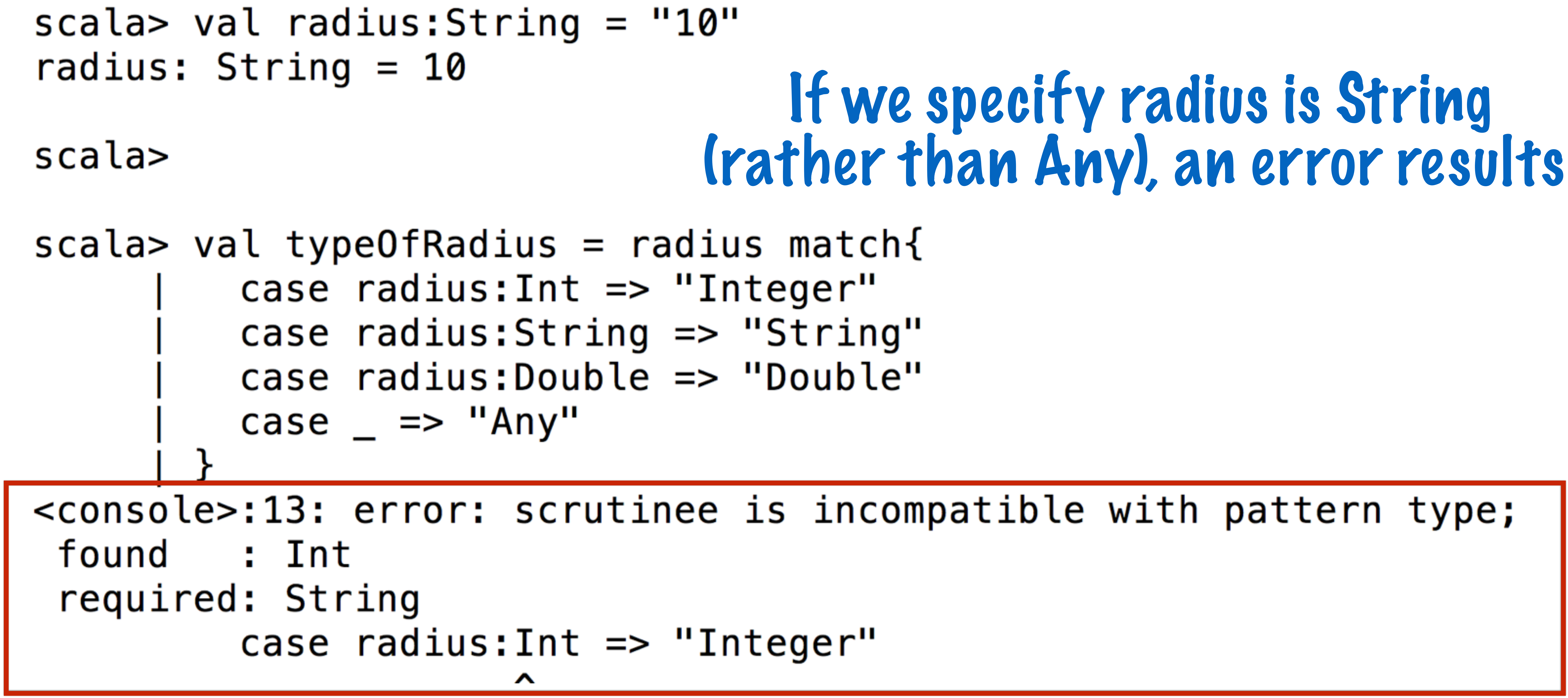


```
scala> val radius:String = "10"
radius: String = 10

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case radius:Double => "Double"
     |     case _ => "Any"
     | }
<console>:13: error: scrutinee is incompatible with pattern type;
 found   : Int
 required: String
           case radius:Int => "Integer"
                       ^
```

If we specify radius is String (rather than Any), an error results!

Mini-example #4

The “scrutinee” (variable whose type is matched) must be a base type

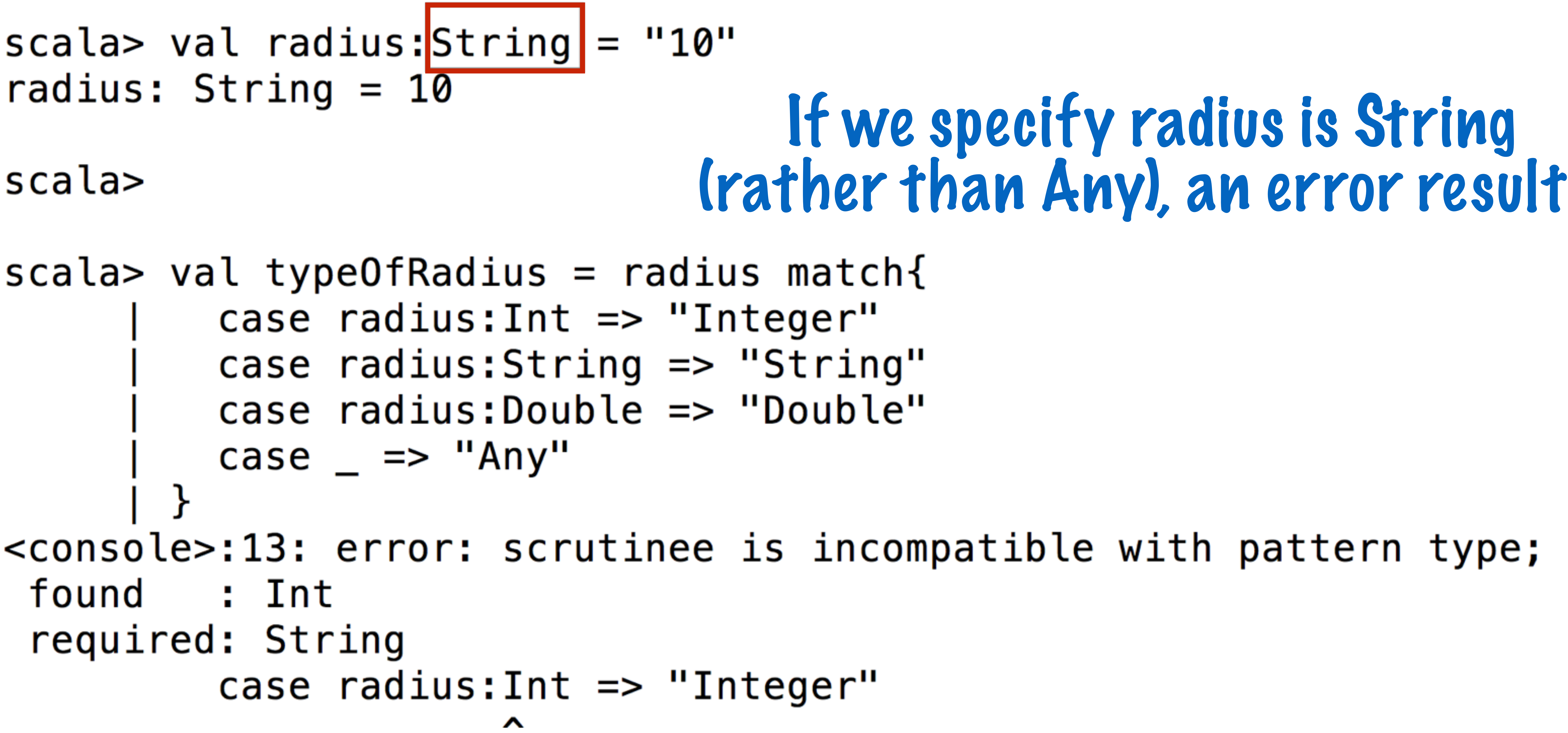


If we specify radius is String (rather than Any), an error results!

```
scala> val radius:String = "10"
radius: String = 10

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case radius:String => "String"
     |    case radius:Double => "Double"
     |    case _ => "Any"
     | }
<console>:13: error: scrutinee is incompatible with pattern type;
 found   : Int
 required: String
          case radius:Int => "Integer"
                      ^
```

Mini-example #4

The "scrutinee" (variable whose type is matched) must be a base type

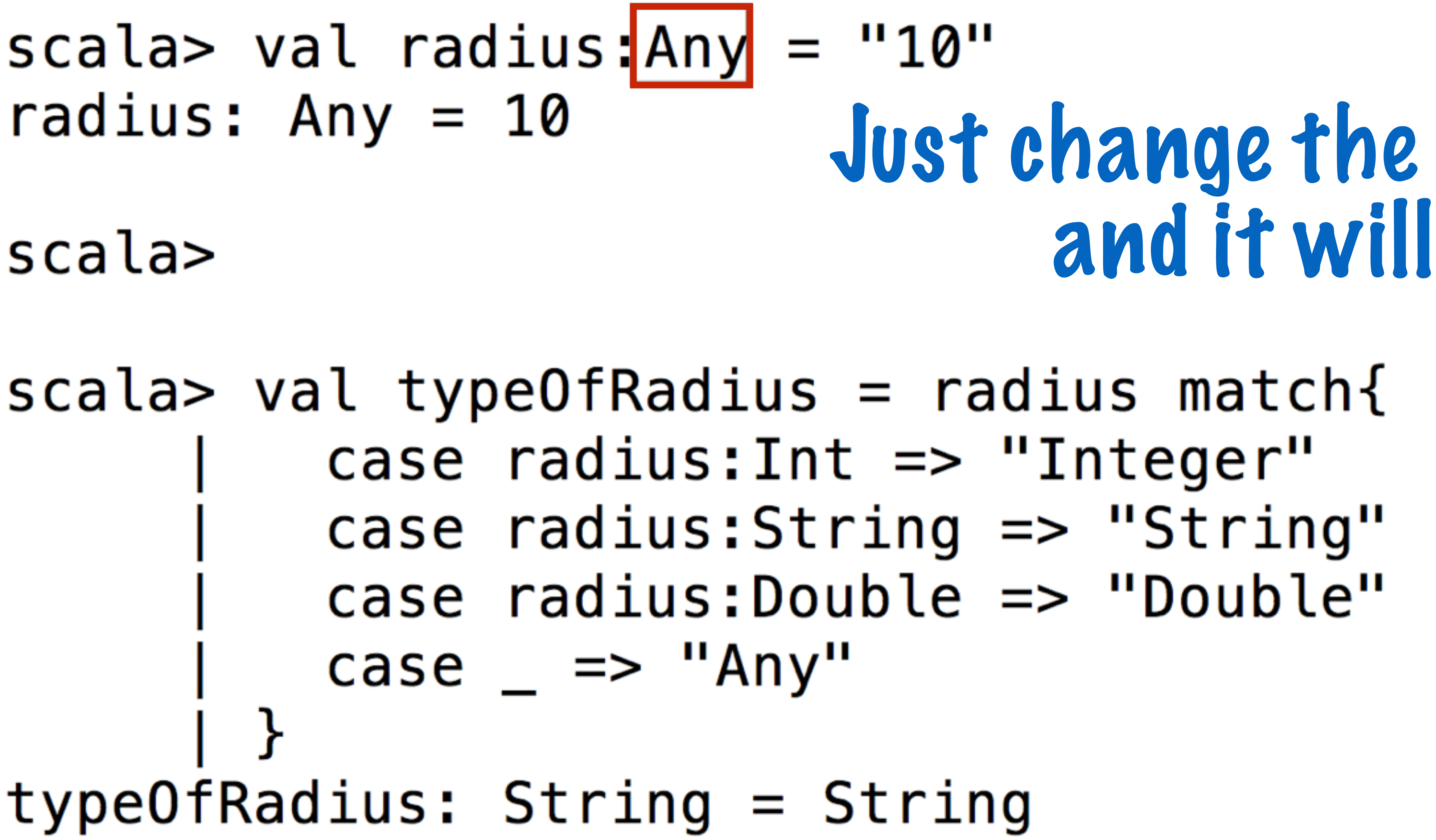


```
scala> val radius:String = "10"
radius: String = 10

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case radius:Double => "Double"
     |     case _ => "Any"
     | }
<console>:13: error: scrutinee is incompatible with pattern type;
 found   : Int
 required: String
           case radius:Int => "Integer"
                       ^
```

If we specify radius is String (rather than Any), an error results!

Mini-example #4

The “scrutinee” (variable whose type is matched) must be a base type

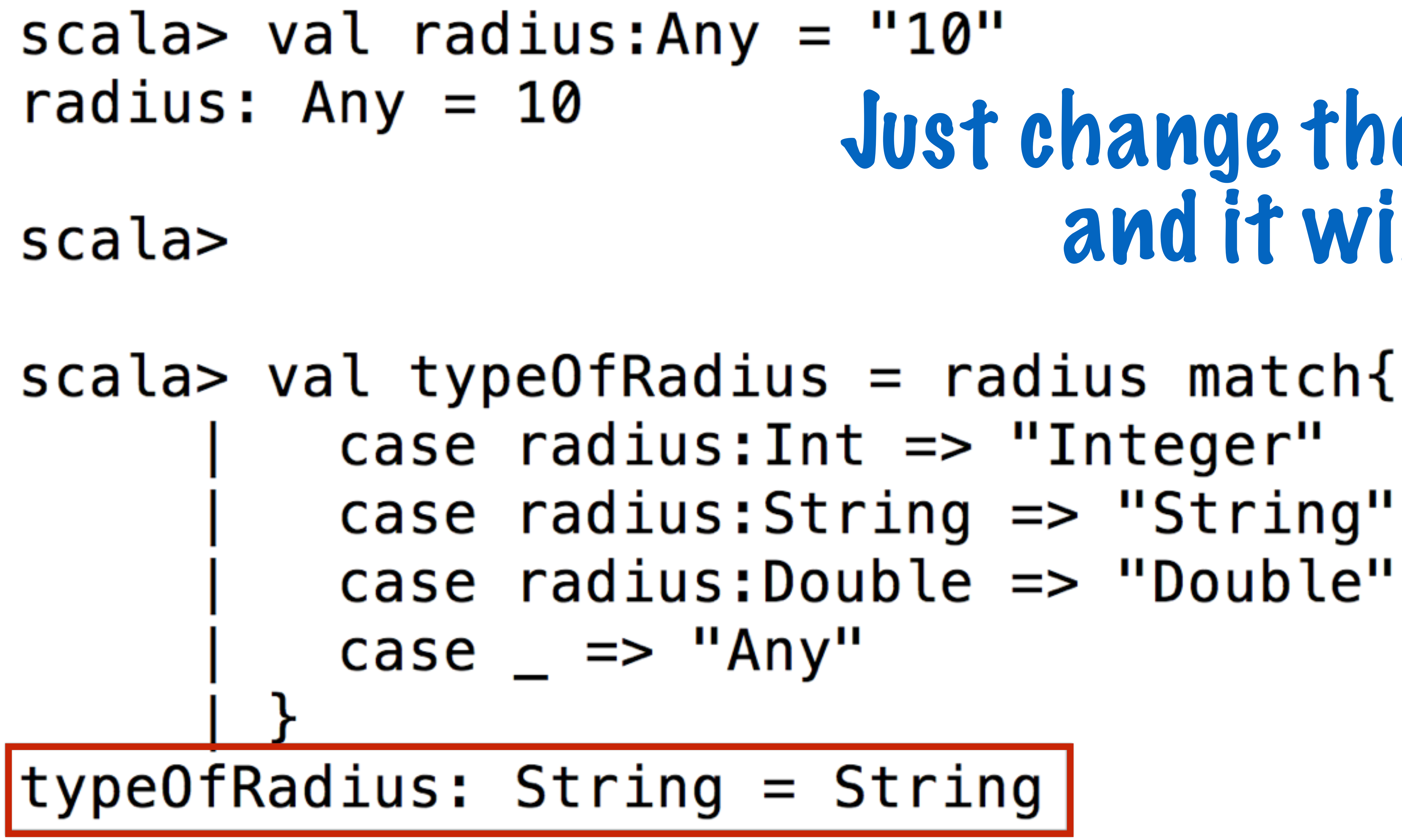


```
scala> val radius:Any = "10"
radius: Any = 10

scala>

scala> val typeOfRadius = radius match{
     |     case radius:Int => "Integer"
     |     case radius:String => "String"
     |     case radius:Double => "Double"
     |     case _ => "Any"
     | }
typeOfRadius: String = String
```

Just change the type to Any, and it will work!

Mini-example #4

The "scrutinee" (variable whose type is matched) must be a base type



```
scala> val radius:Any = "10"
radius: Any = 10

scala>

scala> val typeOfRadius = radius match{
     |    case radius:Int => "Integer"
     |    case radius:String => "String"
     |    case radius:Double => "Double"
     |    case _ => "Any"
     | }
typeOfRadius: String = String
```

Just change the type to Any, and it will work!

`match` expressions: down casting with Pattern Variables

The placeholder _ can be used as the pattern variable

A catch-all/match-all will work as usual

The “scrutinee” (variable whose type is matched) must be a base type

`match` expressions: down casting with Pattern Variables

The placeholder _ can be used as the pattern variable

Why does this matter?

A catch-all/match-all will work as usual

The “scrutinee” (variable whose type is matched) must be a base type

Why does this matter?

`match` expressions: down casting with Pattern Variables

In Java, a common use case of nested if statements is to downcast using `instanceof`

Its a key failing of the Java `switch` statement that it can't predicate on type

Scala's `match` is carefully built to test on type of the match variable

Example 17

`for` loops can be expressions OR statements

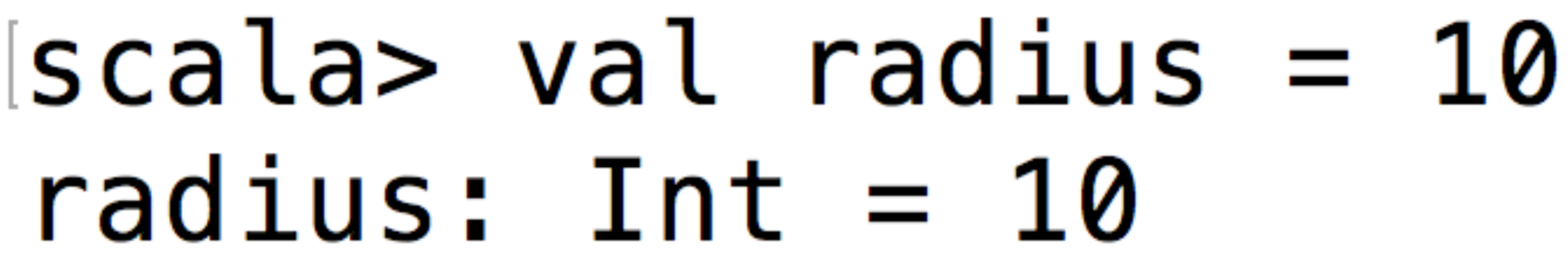
`for` loops can be expressions OR statements

Let's revisit the idea of

Statements v Expressions

Statements

are units of code that do not return a value



```
scala> val radius = 10
radius: Int = 10
```

Statements

are units of code that do not return a value

```
scala> println("hello world")
hello world
```

Statements v Expressions

are units of code that do not return a value

Expressions

are units of code that return a value

Expressions

are units of code that return a value

```
scala> val radius = 10
radius: Int = 10

scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

Statements

are units of code that do not return a value

```
scala> println("hello world")
hello world
```

Statements v Expressions

Why does this matter?

Because many constructs that are statements in Java are expressions in Scala

Statements v Expressions

Many constructs that are statements in Java are expressions in Scala

✓ `if/else`

`for` loops (But not `while` loops)

✓ `match`

`for` loops can be expressions OR statements

For loops can be set up as either statements or expressions by adding or removing just 1 word

`yield`

The presence of this word converts a for-loop into an expression

For loops can be set up as either statements or expressions by adding or removing just 1 word

```
yield
```

The presence of this word converts a for-loop into an expression

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

yield

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

`yield`

A for-loop with yield **will “yield” a collection** of the return values of each iteration of the loop

`yield`

A for-loop with yield will “yield” a collection **of the return values** of each iteration of the loop

`yield`

A for-loop with yield will “yield” a collection of the return values of each iteration of the loop

`yield`

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

A for-loop without yield will simply execute the iterations without saving their return values

(A Java-style, "old-school" for-loop)

A Java-style, “old-school” for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

```
scala> val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
daysOfWeekList: List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Let’s check out a simple example involving a **List**

This is our first encounter with a **Scala collection!**

A Java-style, "old-school" for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

```
scala> val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
daysOfWeekList: List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Define a simple list of the days of the week

A Java-style, “old-school” for-loop

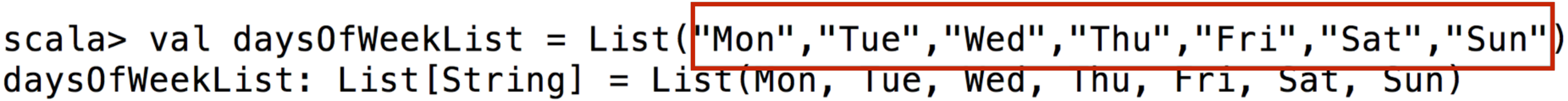
A for-loop without yield will simply execute the iterations without saving their return values

```
scala> val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
daysOfWeekList: List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Define a simple list of the days of the week

A Java-style, "old-school" for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values



```
scala> val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
daysOfWeekList: List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Define a simple list of the days of the week

A Java-style, “old-school” for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

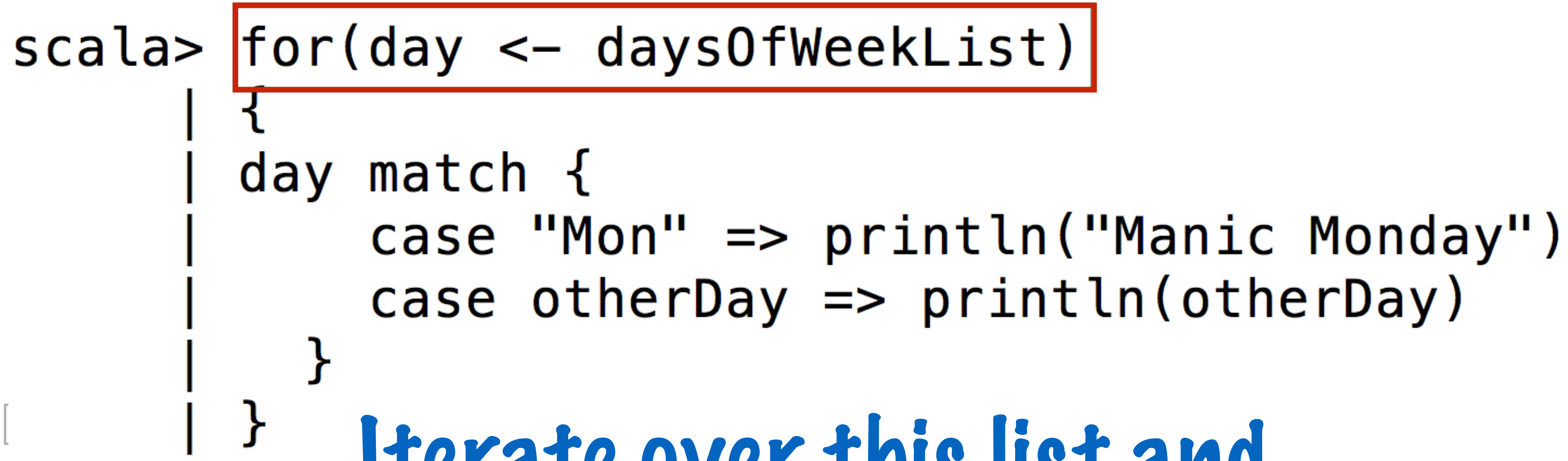
```
scala> val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
daysOfWeekList: List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Define a simple list of the days of the week

`List[String]` in Scala is like **`List<String>`** in Java

A Java-style, "old-school" for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

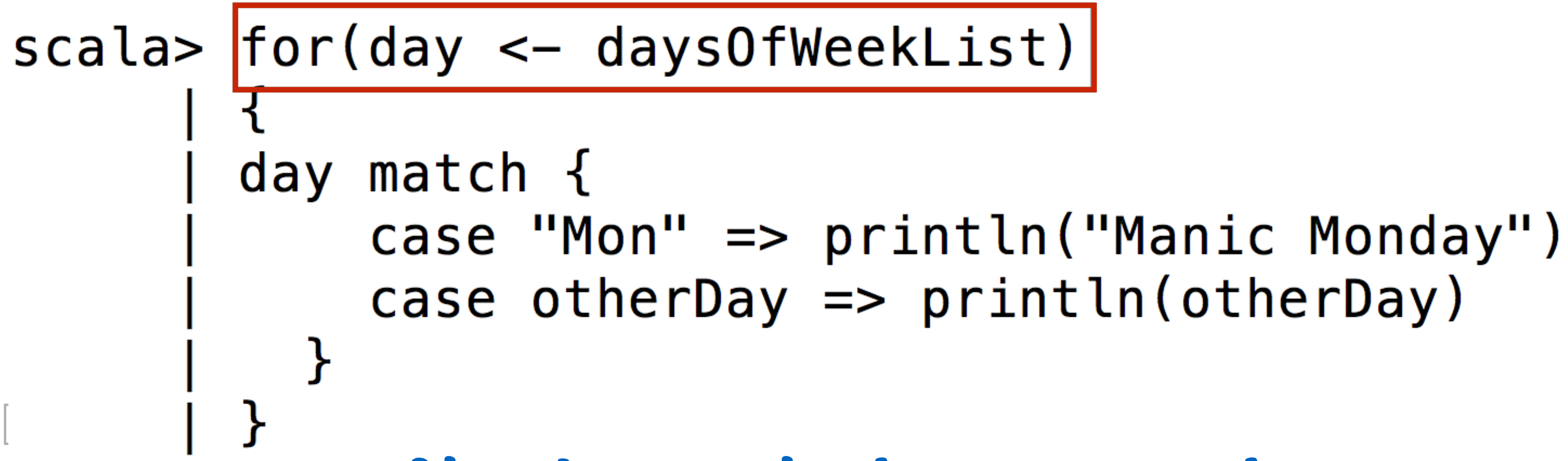


```
scala> for(day <- daysOfWeekList)
     | {
     | day match {
     |      case "Mon" => println("Manic Monday")
     |      case otherDay => println(otherDay)
     |    }
     | }
```

Iterate over this list and print a message

A Java-style, "old-school" for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

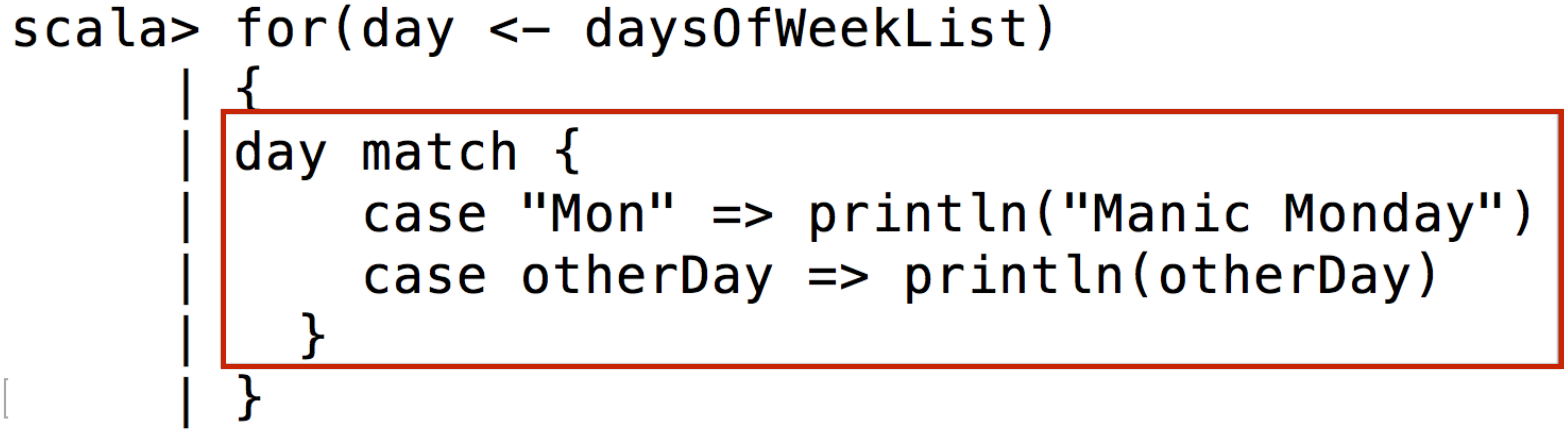


```
scala> for(day <- daysOfWeekList)
     | {
     | day match {
     |       case "Mon" => println("Manic Monday")
     |       case otherDay => println(otherDay)
     |    }
     | }
```

Check out the loop variable - this is like `foreach` in Java

A Java-style, "old-school" for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

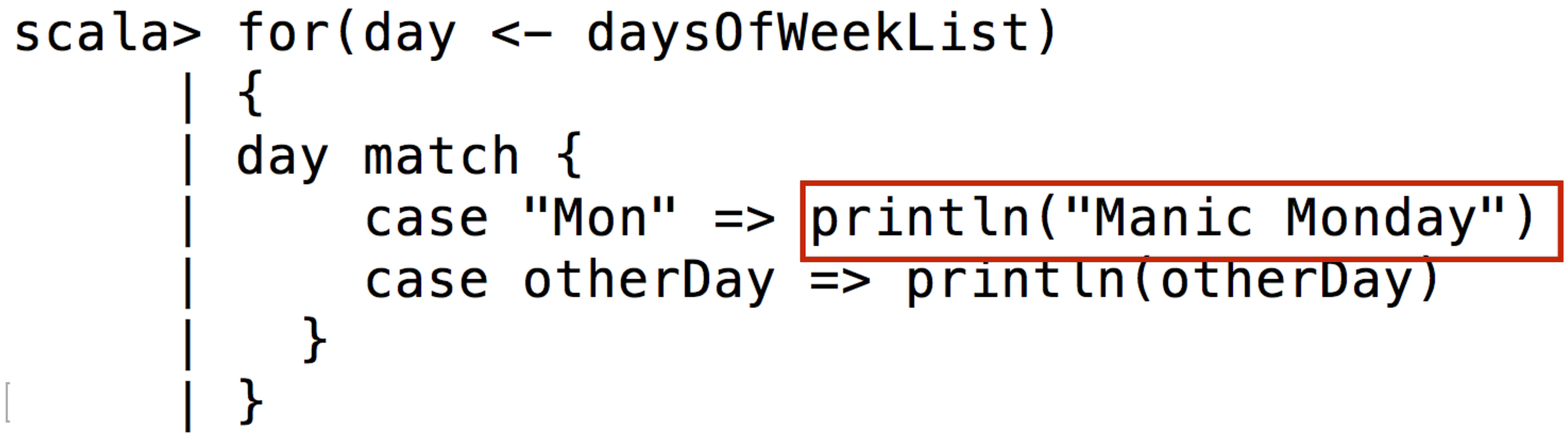


```
scala> for(day <- daysOfWeekList)
     | {
     | day match {
     |     case "Mon" => println("Manic Monday")
     |     case otherDay => println(otherDay)
     |   }
     | }
```

Use a match expression

A Java-style, "old-school" for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

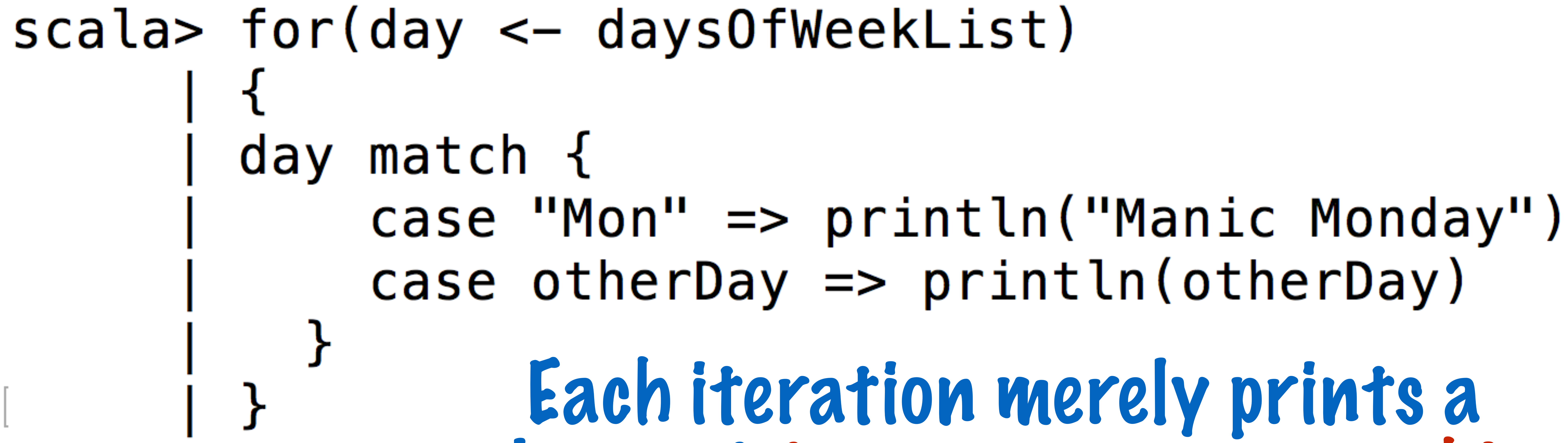


```
scala> for(day <- daysOfWeekList)
     | {
     | day match {
     |     case "Mon" => println("Manic Monday")
     |     case otherDay => println(otherDay)
     |   }
     | }
```

Each iteration merely prints a value, and **does not return anything**

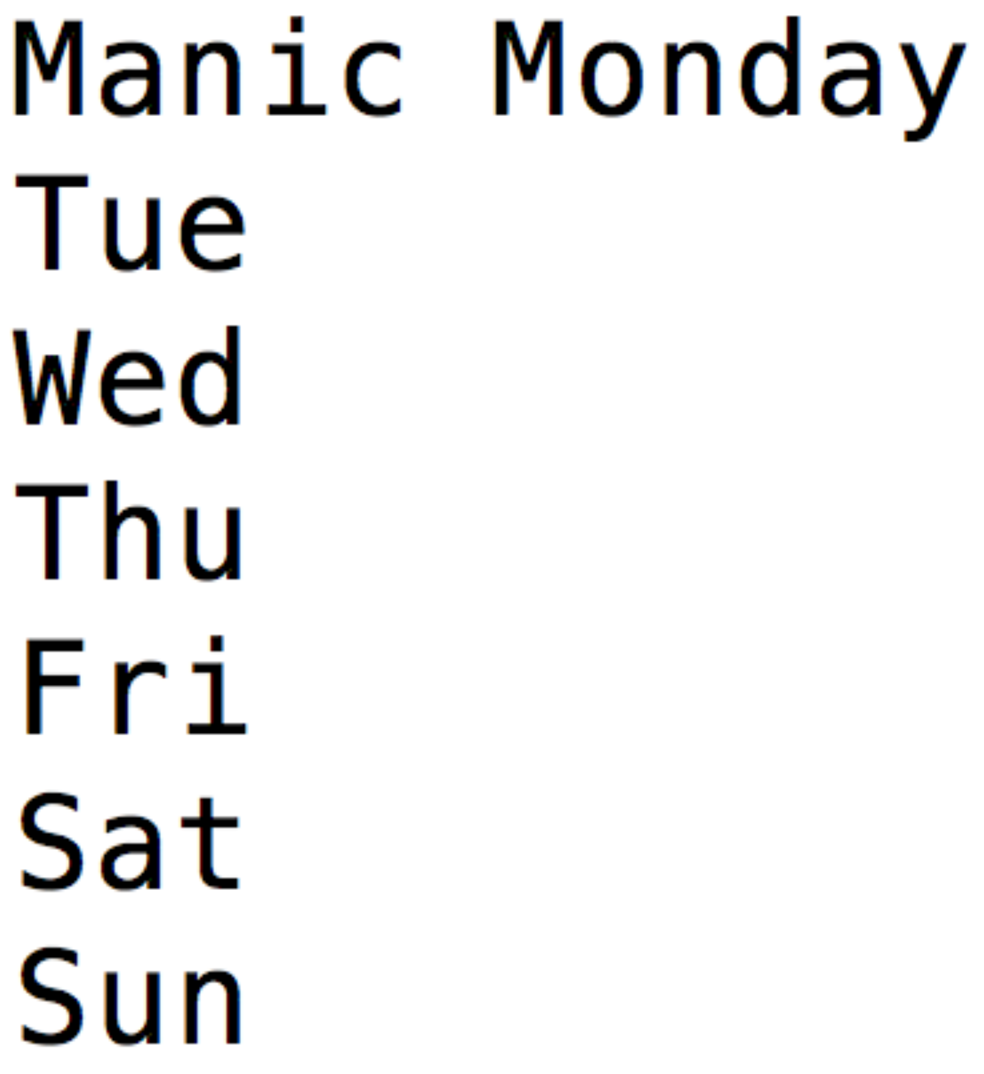
A Java-style, “old-school” for-loop

A for-loop without yield will simply execute the iterations without saving their return values



```
scala> for(day <- daysOfWeekList)
     | {
     | day match {
     |     case "Mon" => println("Manic Monday")
     |     case otherDay => println(otherDay)
     |   }
     | }
```

Each iteration merely prints a value, and does not return anything



```
Manic Monday
Tue
Wed
Thu
Fri
Sat
Sun
```

A Java-style, "old-school" for-loop

A for-loop **without yield** will simply execute the iterations without saving their return values

```
Manic Monday
Tue
Wed
Thu
Fri
Sat
Sun
```

Each iteration merely prints a value, and **does not return anything**

This for-loop was a statement - **nothing was returned**

`yield`

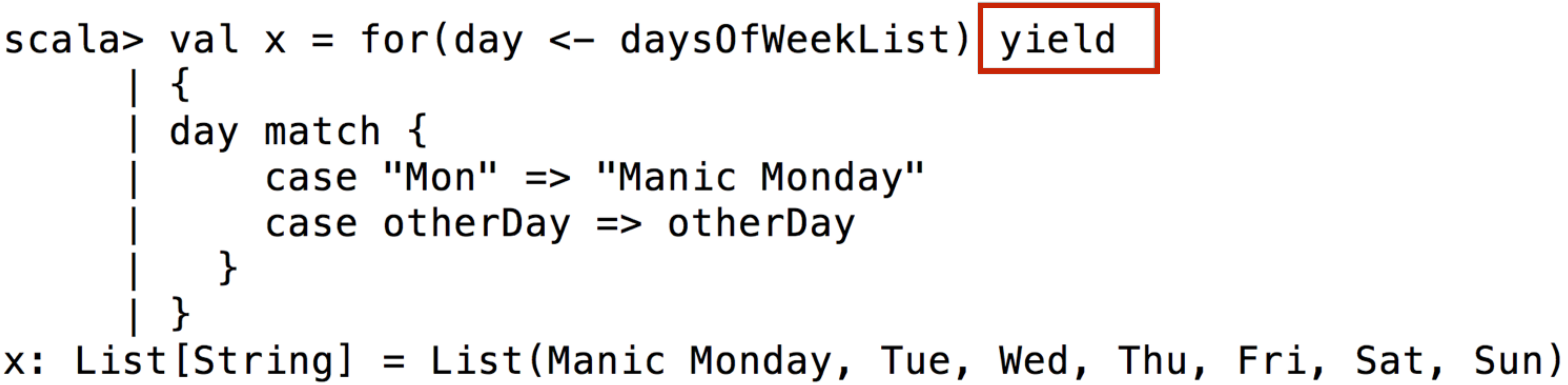
A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

✓ A for-loop without yield will simply execute the iterations without saving their return values

(A Java-style, "old-school" for-loop)

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

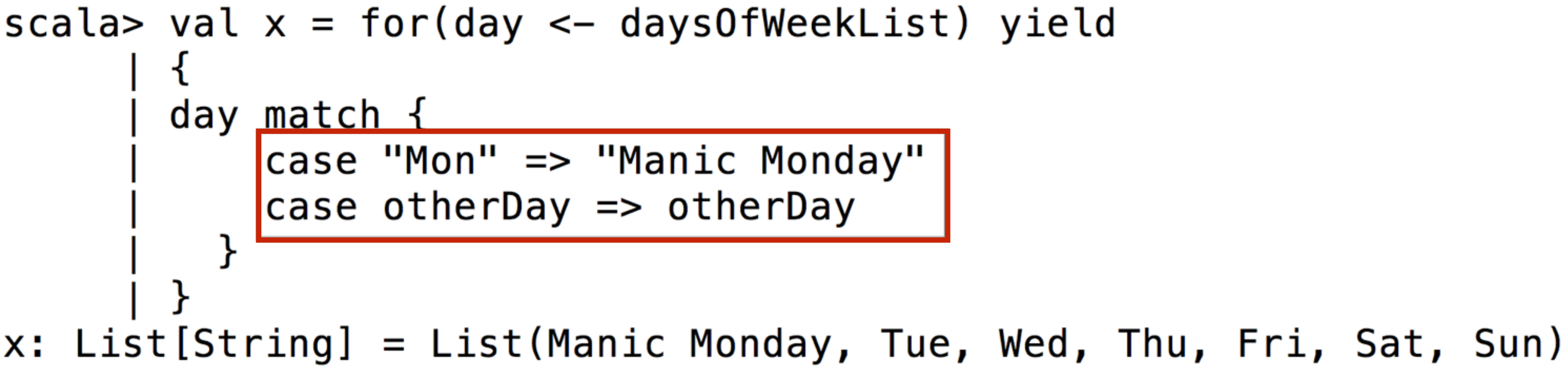
We can convert our for-loop to a statement merely by adding the word **`yield`**



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |      case "Mon" => "Manic Monday"
     |      case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

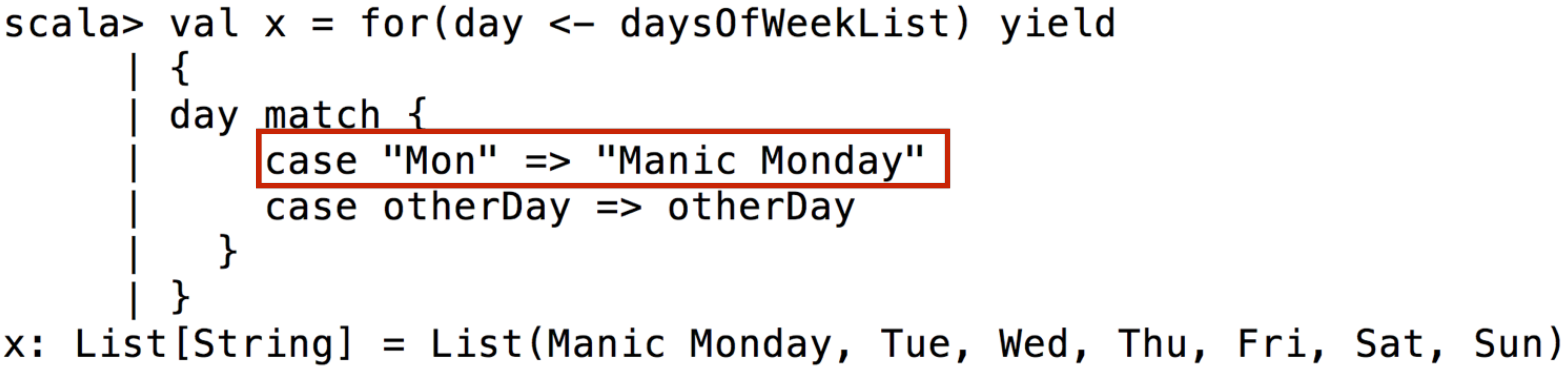
We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |     case "Mon" => "Manic Monday"
     |     case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

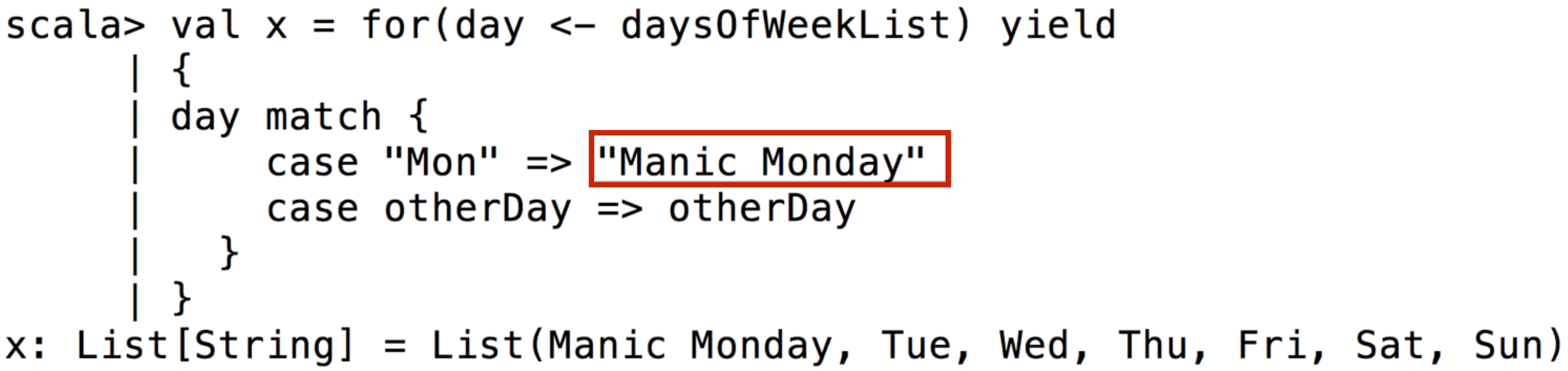
We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |       case "Mon" => "Manic Monday"
     |       case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

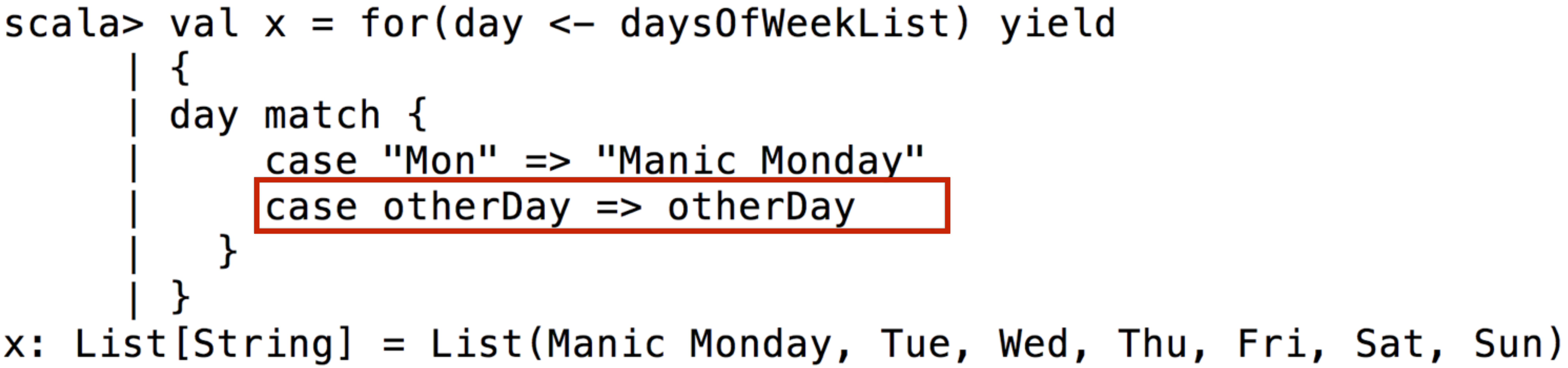
We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |     case "Mon" => "Manic Monday"
     |     case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

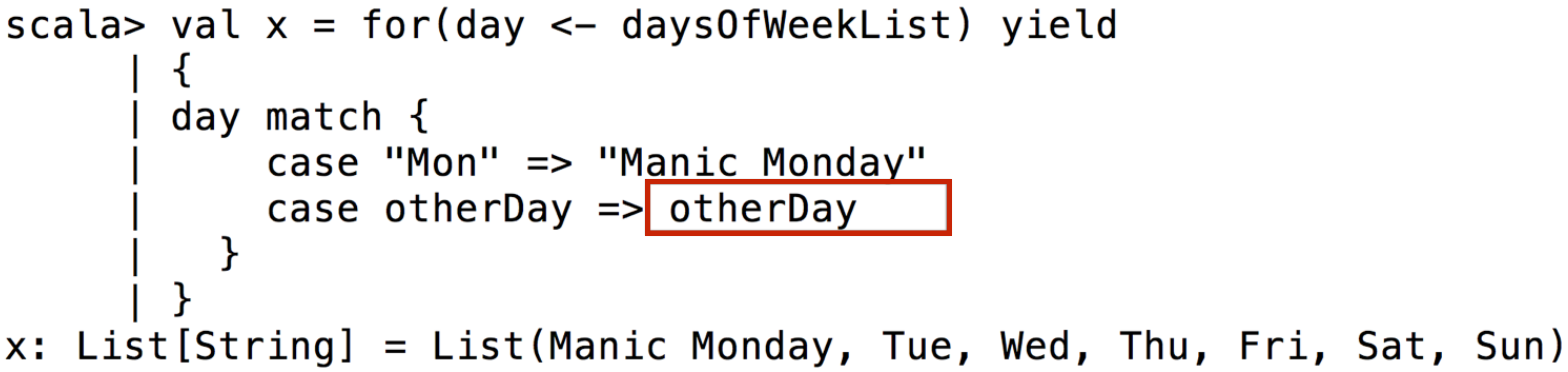
We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |       case "Mon" => "Manic Monday"
     |       case otherDay => otherDay
     |    }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

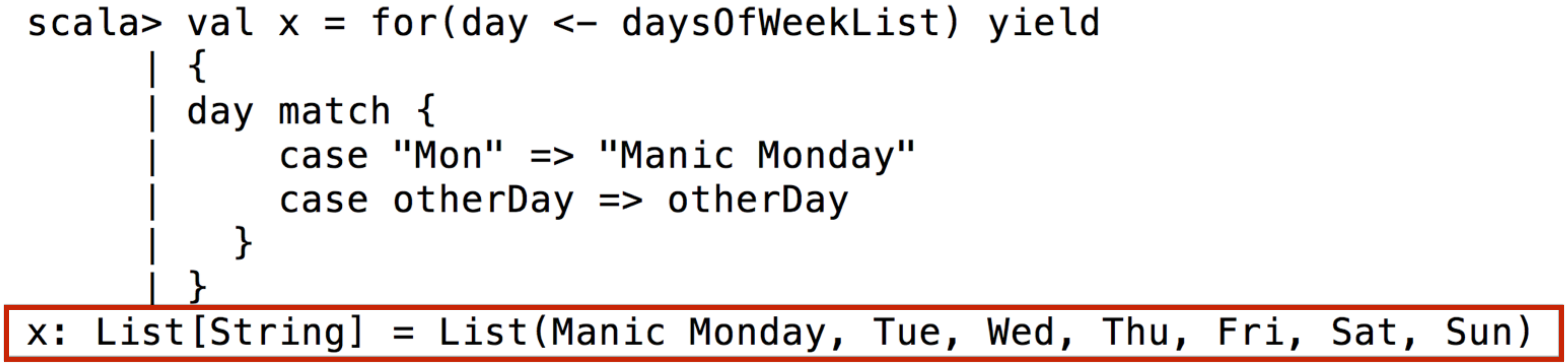
A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |       case "Mon" => "Manic Monday"
     |       case otherDay => otherDay
     |    }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

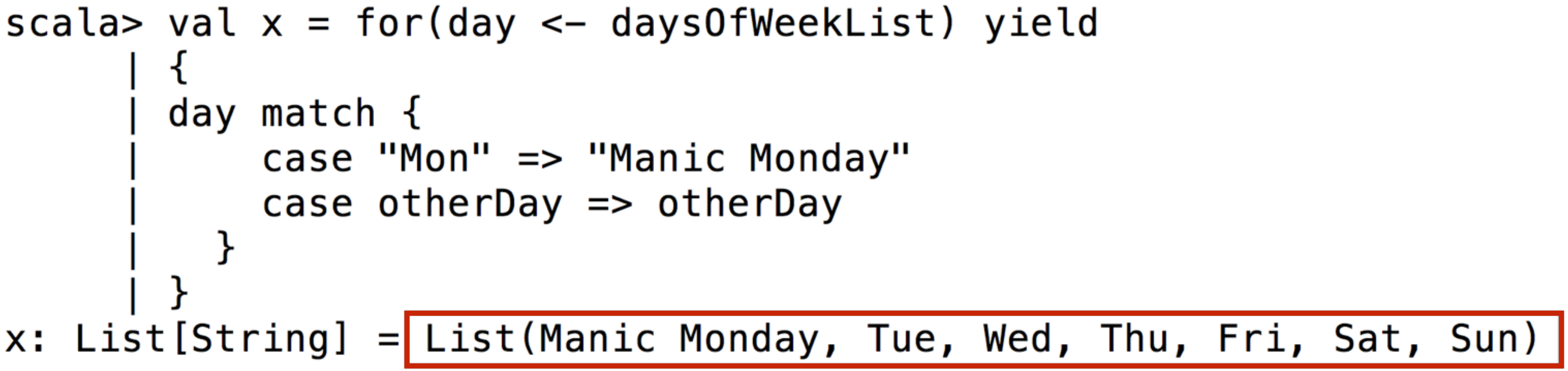
We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |     case "Mon" => "Manic Monday"
     |     case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will “yield” a collection of the return values of each iteration of the loop

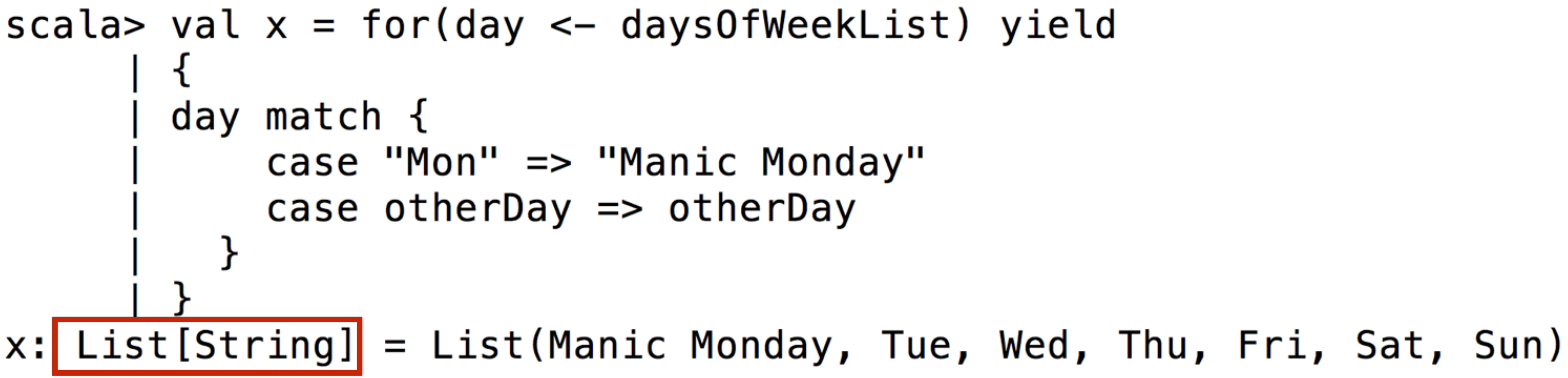
We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |       case "Mon" => "Manic Monday"
     |       case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

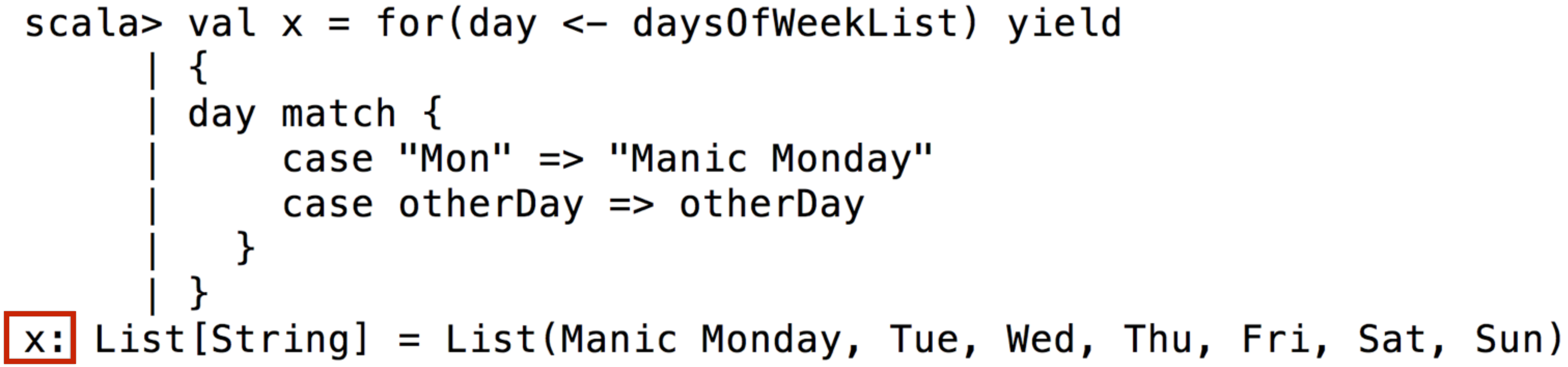
We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |     case "Mon" => "Manic Monday"
     |     case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

We have also changed the match expression to return something now

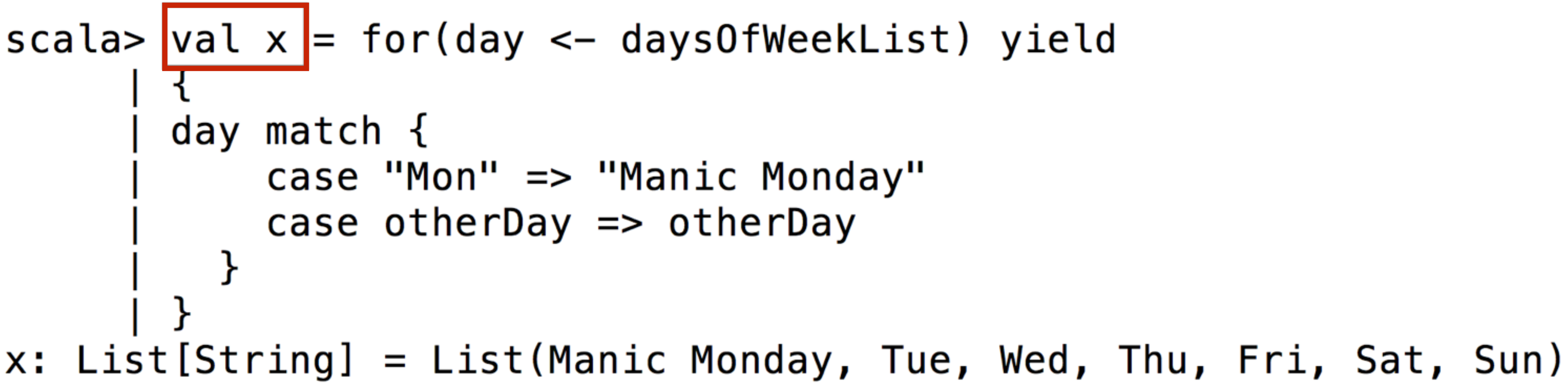


```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |     case "Mon" => "Manic Monday"
     |     case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

We have also changed the match expression to return something now



```
scala> val x = for(day <- daysOfWeekList) yield
     | {
     | day match {
     |      case "Mon" => "Manic Monday"
     |      case otherDay => otherDay
     |   }
     | }
x: List[String] = List(Manic Monday, Tue, Wed, Thu, Fri, Sat, Sun)
```

The results of this for-loop are saved in a value

`yield`

- ✓ A for-loop with yield will “yield” a collection of the return values of each iteration of the loop
- ✓ A for-loop without yield will simply execute the iterations without saving their return values

(A Java-style, “old-school” for-loop)

for loops can be expressions OR statements

For loops can be set up as either statements or expressions by adding or removing just 1 word

```
yield
```

The presence of this word converts a for-loop into an expression

`for` loops can be expressions OR statements

For loops can be set up as either statements or expressions by adding or removing just 1 word

Why does this matter?

`yield`

The presence of this word converts a for-loop into an expression

for loops can be expressions OR statements

Why does this matter?

Think of how often, in Java, you have a bit of boilerplate to collect the results of a for-loop in a list

```
List<String> typesOfDays = new ArrayList<>();
for(String day: daysOfWeek) {
    if(day.equals("Mon")) {
        typesOfDays.add("Manic Monday");
    }
    else {
        typesOfDays.add(day);
    }
```

Why does this matter?

Think of how often, in Java, you have a bit of boilerplate to collect the results of a for-loop in a list

```
List<String> typesOfDays = new ArrayList<>();
for(String day: daysOfWeek) {
    if(day.equals("Mon")) {
        typesOfDays.add("Manic Monday");
    }
    else {
        typesOfDays.add(day);
    }
}
```

Yet another common Java "bloat-case" has been addressed in Scala

for loops can be expressions OR statements

Why does this matter?

Think of how often, in Java, you have a bit of boilerplate to collect the results of a for-loop in a list

Yet another common Java “bloat-case” has been addressed in Scala

Example 18

`for` loops: 2 types of iterators

for loops: 2 types of iterators

```
List<String> daysOfWeek = new ArrayList<String>();
```

In Java, there are 2 ways one could iterate over a list

```
for(String day: daysOfWeek) {
        // Do something
}

for(int i = 0;i < daysOfWeek.size(); i ++) {
    // Do something
}
```

for loops: 2 types of iterators

```
List<String> daysOfWeek = new ArrayList<String>();
```

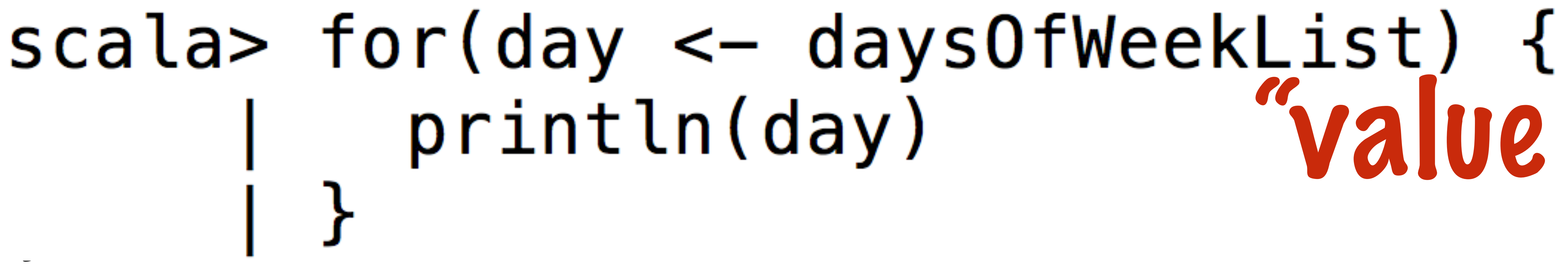
Scala has exact equivalents

```
for(String day: daysOfWeek) {
        // Do something
}

for(int i = 0;i < daysOfWeek.size(); i ++) {
     // Do something
}
```

for loops: 2 types of iterators

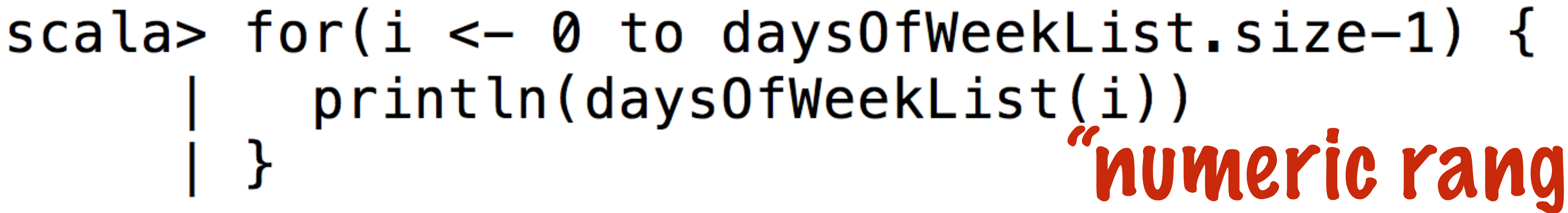
```
val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
```

Scala has exact equivalents



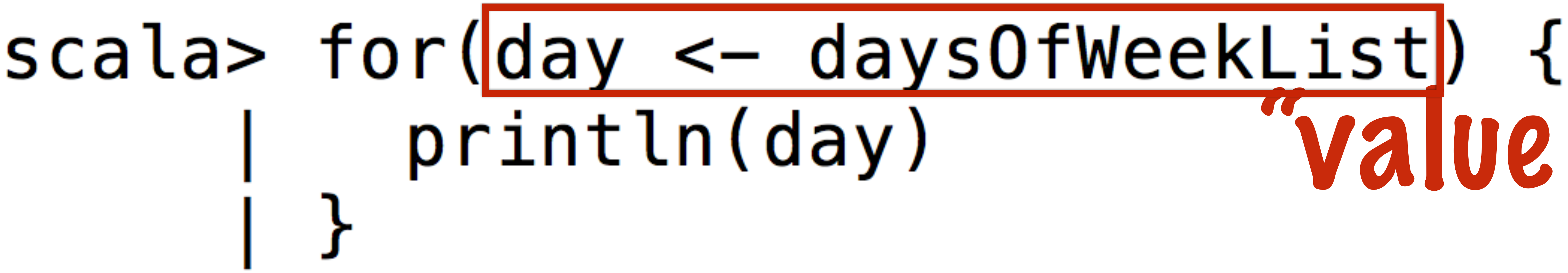
```
scala> for(day <- daysOfWeekList) {
     |   println(day)
     | }
```

"value binding"



```
scala> for(i <- 0 to daysOfWeekList.size-1) {
     |   println(daysOfWeekList(i))
     | }
```

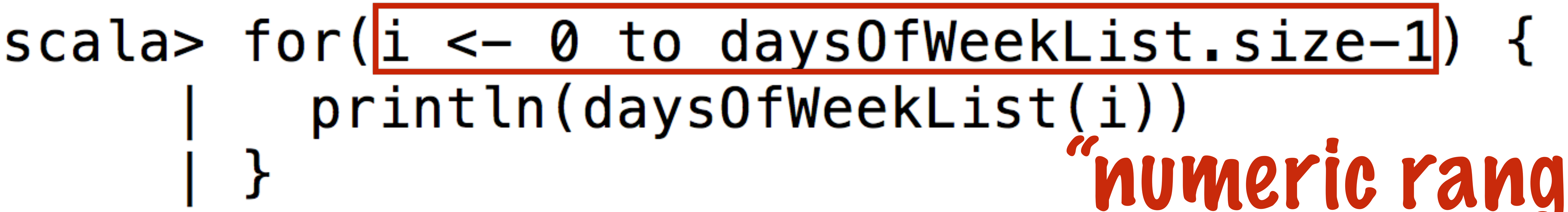
"numeric range"



```
scala> for(day <- daysOfWeekList) {
     |   println(day)
     | }
```

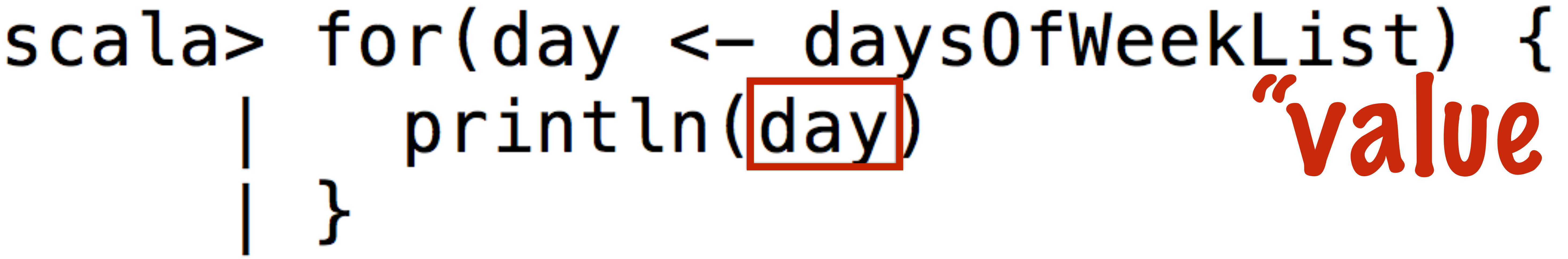
“value binding”

Nothing too complicated here, just notice the way the loops are set up



```
scala> for(i <- 0 to daysOfWeekList.size-1) {
     |   println(daysOfWeekList(i))
     | }
```

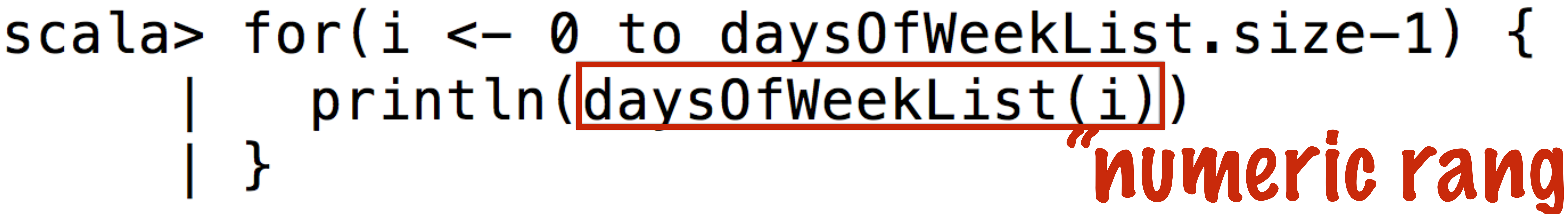
“numeric range”



```
scala> for(day <- daysOfWeekList) {
     |   println(day)
     | }
```

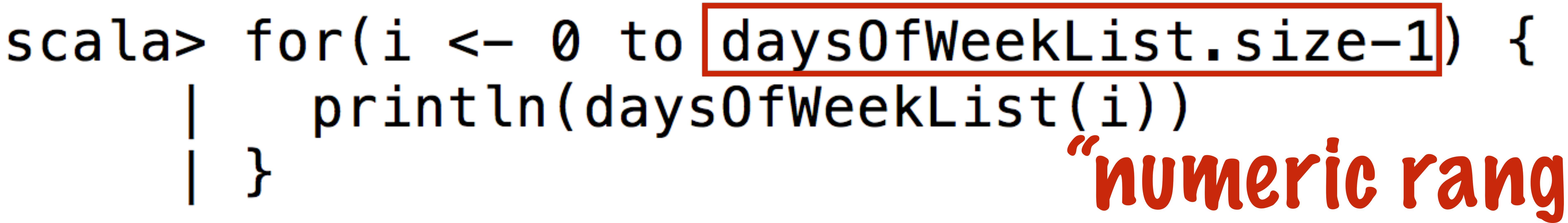
“value binding”

Nothing too complicated here, just notice the way the loops are set up



```
scala> for(i <- 0 to daysOfWeekList.size-1) {
     |   println(daysOfWeekList(i))
     | }
```

“numeric range”



```
scala> for(i <- 0 to daysOfWeekList.size-1) {
     |   println(daysOfWeekList(i))
     | }
```

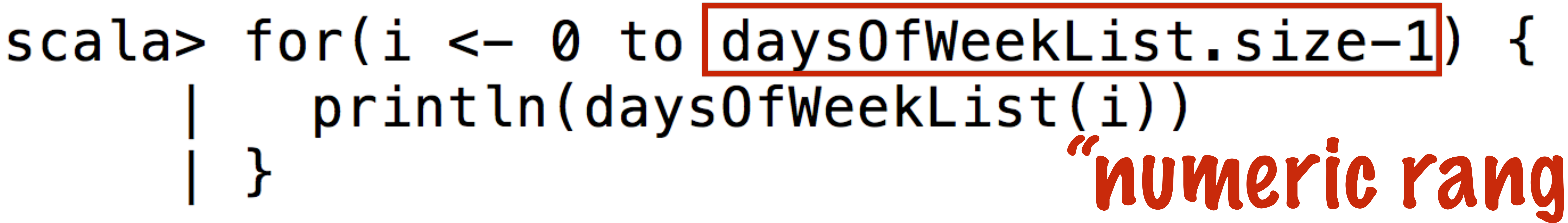
“numeric range”

Scala even has a way to eliminate the clunky “-1”, source of so many off-by-1 errors

```
scala> for(i <- 0 to daysOfWeekList.size-1) {
     |   println(daysOfWeekList(i))
     | }
```

“numeric range”

Scala even has a way to eliminate the clunky “-1”, source of so many off-by-1 errors

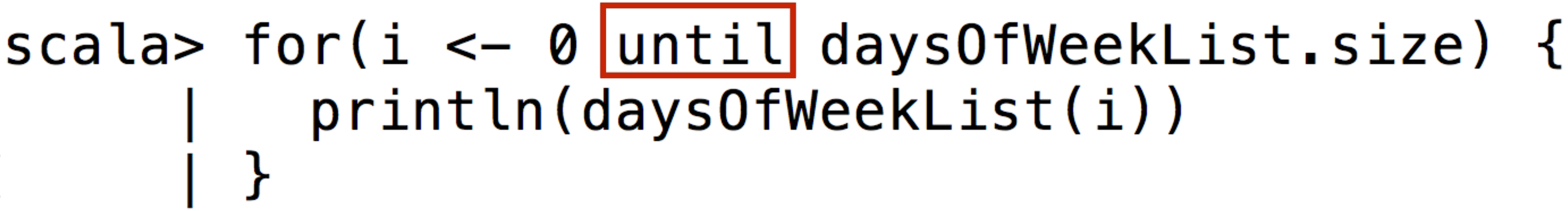


```
scala> for(i <- 0 to daysOfWeekList.size-1) {
     |   println(daysOfWeekList(i))
     | }
```

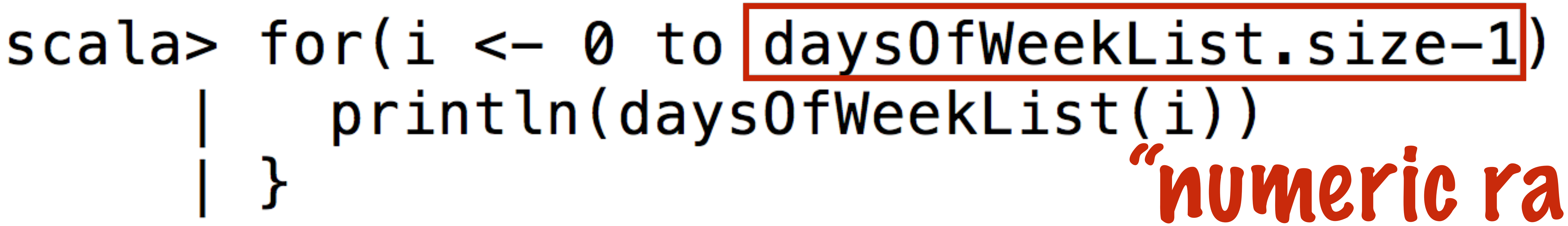
"numeric range"

Scala even has a way to eliminate the clunky "-1", source of so many off-by-1 errors

until



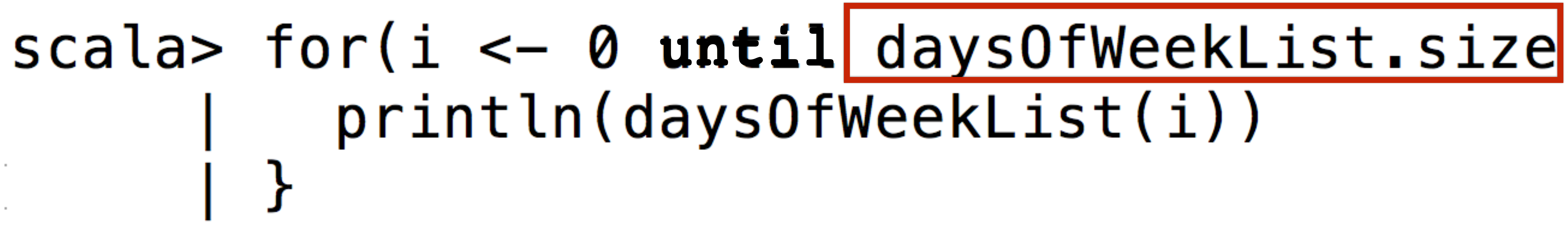
```
scala> for(i <- 0 until daysOfWeekList.size) {
     |   println(daysOfWeekList(i))
     | }
```



```
scala> for(i <- 0 to daysOfWeekList.size-1) {
     |   println(daysOfWeekList(i))
     | }
```

“numeric range”

Scala even has a way to eliminate the clunky “-1”, source of so many off-by-1 errors

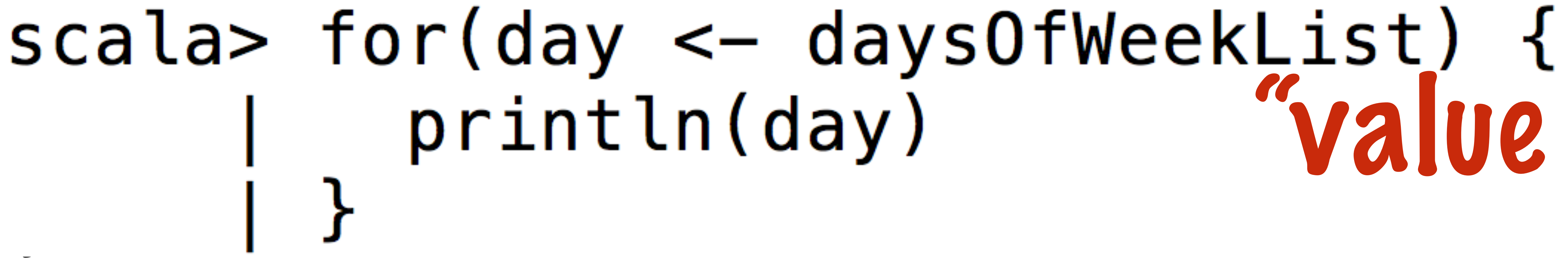


```
scala> for(i <- 0 until daysOfWeekList.size) {
     |   println(daysOfWeekList(i))
     | }
```

for loops: 2 types of iterators

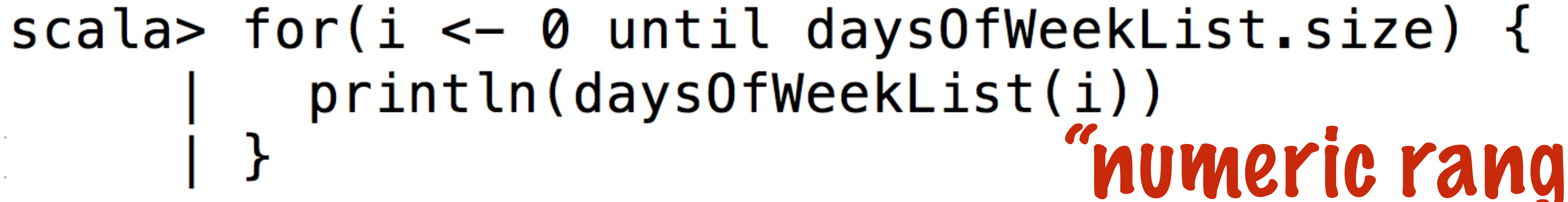
```
val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
```

Scala has exact equivalents



```
scala> for(day <- daysOfWeekList) {
     |   println(day)
     | }
```

“value binding”



```
scala> for(i <- 0 until daysOfWeekList.size) {
     |   println(daysOfWeekList(i))
     | }
```

“numeric range”

for loops: 2 types of iterators

```
val daysOfWeekList = List("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
List<String> daysOfWeek = new ArrayList<String>();
```

So which of these is the best way to iterate over a collection?

```
scala> for(day <- daysOfWeekList) {
     |   println(day)
     | }

for (String day : daysOfWeek) {
    // Do something
}
```

"value binding"

```
scala> for(i <- 0 until daysOfWeekList.size) {
     |   println(daysOfWeekList(i))
     | }
```

"numeric range"

So which of these is the best way to iterate over a collection?

Err..actually neither..

Scala has powerful aggregate functions such as `foreach, map, flatmap`

But you should know how to use `for` loops anyway

Example 19

`for` loops with `if` conditions: Pattern Guards

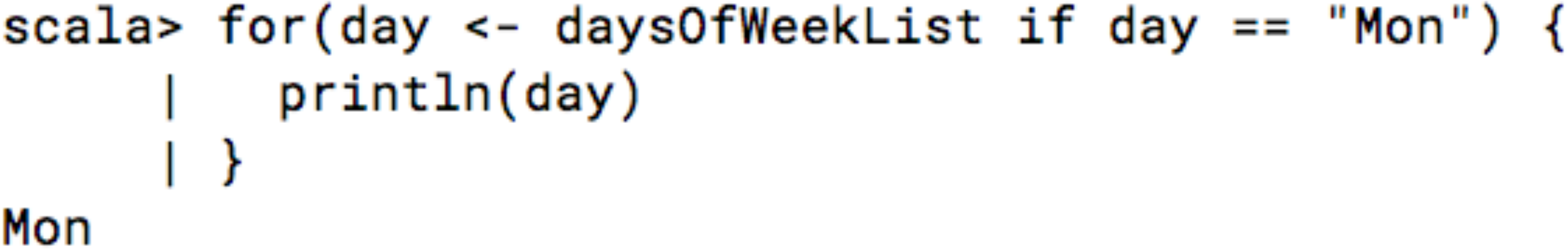
for loops with if conditions: Pattern Guards

Here is how we would combine an if condition with a for loop in Java

```java
for(String day: daysOfWeek) {
    if(day.equals("Monday"))
        System.out.println("Manic Monday!");
}
```

for loops with if conditions: Pattern Guards

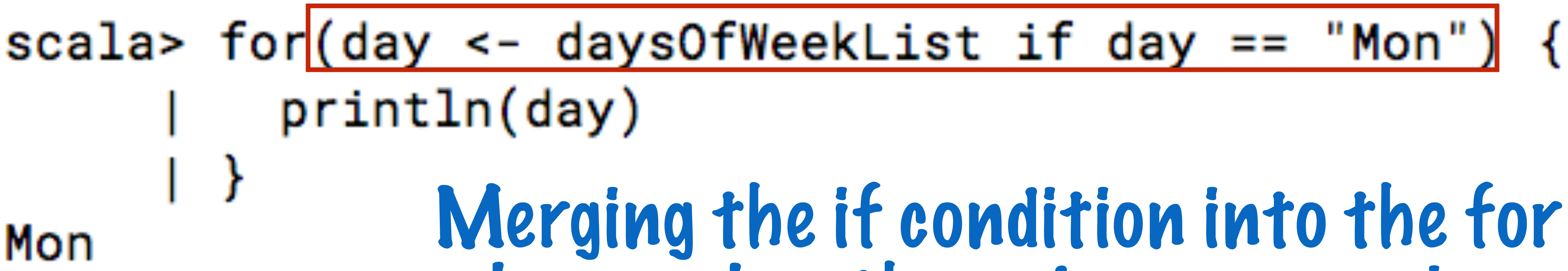
Here is how we would combine an if condition with a for loop in Scala



```
scala> for(day <- daysOfWeekList if day == "Mon") {
     |   println(day)
     | }
Mon
```

for loops with if conditions: Pattern Guards

Here is how we would combine an if condition with a for loop in Scala

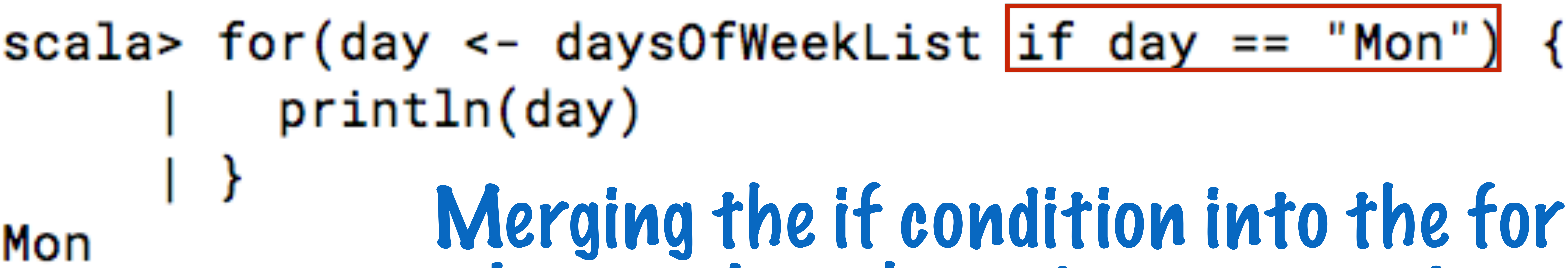


```
scala> for(day <- daysOfWeekList if day == "Mon") {
     |   println(day)
     | }
Mon
```

Merging the if condition into the for loop makes the code more concise

`for` loops with `if` conditions: Pattern Guards

Here is how we would combine an if condition with a for loop in Scala

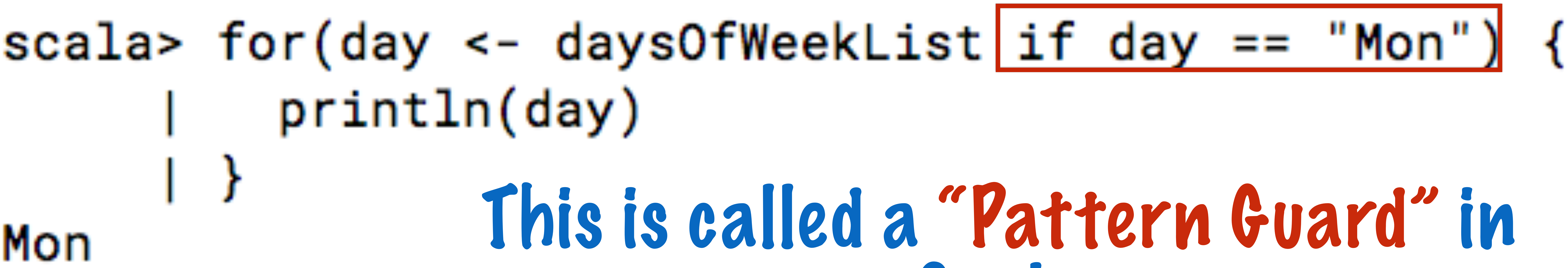


```
scala> for(day <- daysOfWeekList if day == "Mon") {
     |   println(day)
     | }
Mon
```

Merging the if condition into the for loop makes the code more concise

for loops with if conditions: Pattern Guards

Here is how we would combine an if condition with a for loop in Scala



```
scala> for(day <- daysOfWeekList if day == "Mon") {
     |   println(day)
     | }
Mon
```

This is called a "Pattern Guard" in Scala

Example 20

Nested `for` Loops: Nested Iterators

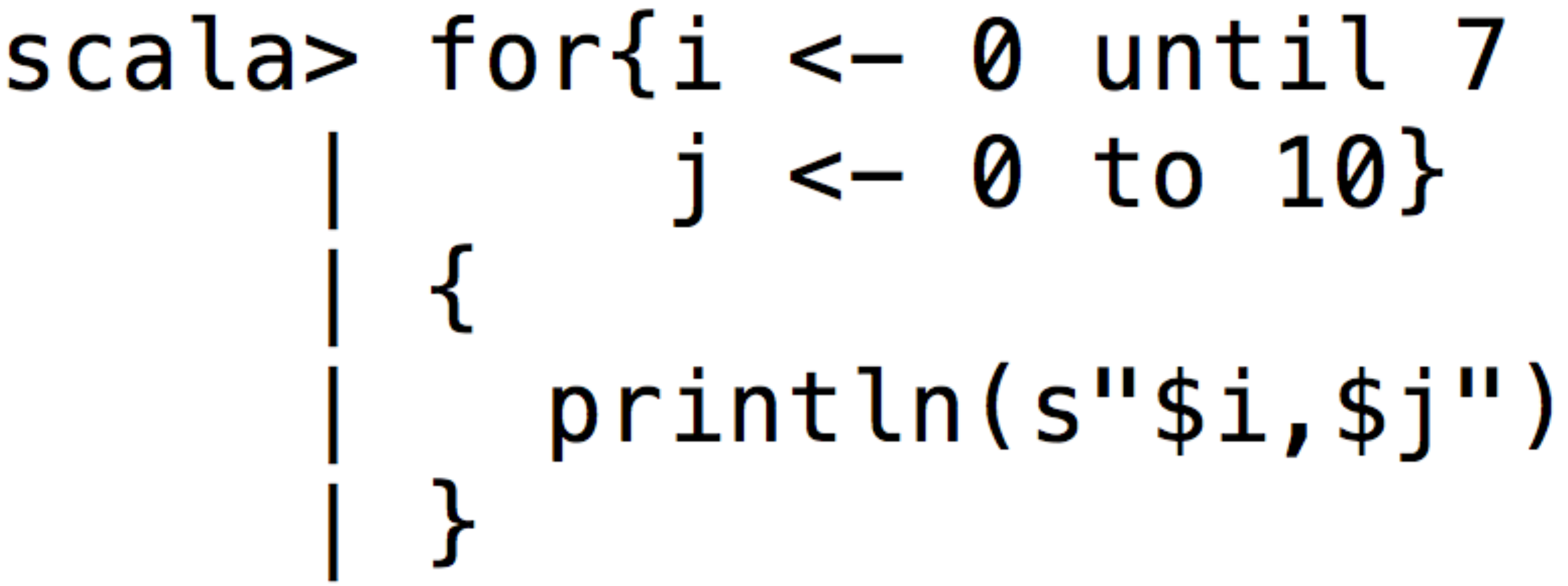
Nested `for` Loops: Nested Iterators

```
for(int i = 0;i < 10; i ++) {
  for(int j = 0;j < 10; j ++) {
    // Do something
  }
}
```

Here is how we would set up a nested loop in Java

Nested for Loops: Nested Iterators

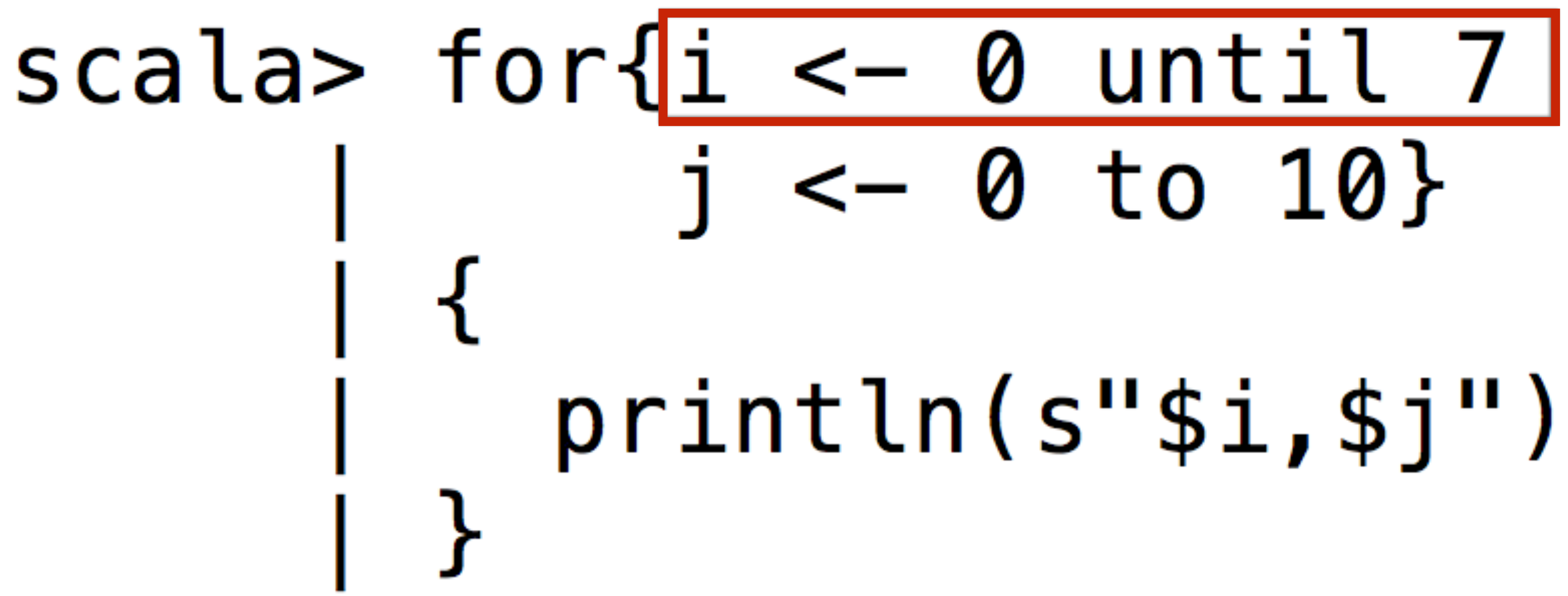
```
fo
{
++) {
}
```



```
scala> for{i <- 0 until 7
     |      j <- 0 to 10}
     | {
     |   println(s"$i,$j")
     | }
```

Here is how we would set up a nested loop in Scala

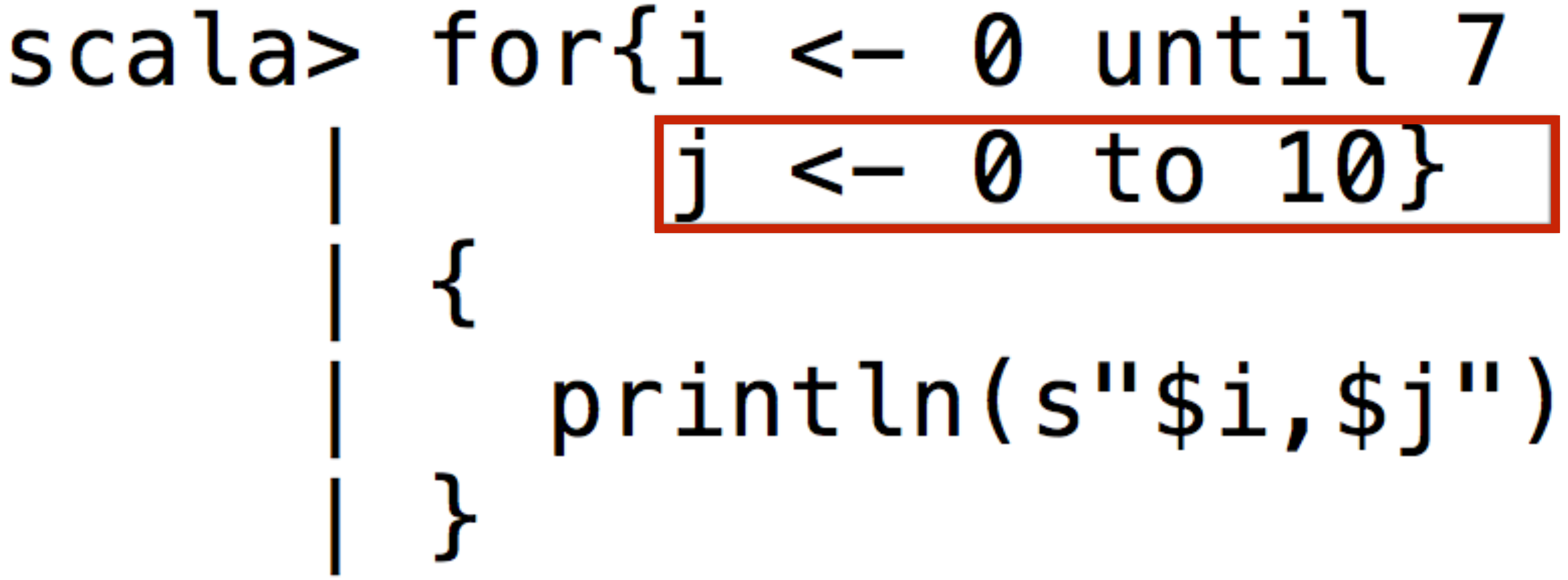
Nested `for` Loops: Nested Iterators



```
scala> for{i <- 0 until 7
     |      j <- 0 to 10}
     | {
     |   println(s"$i,$j")
     | }
```

Notice that there 2 ranges, without a comma separating them

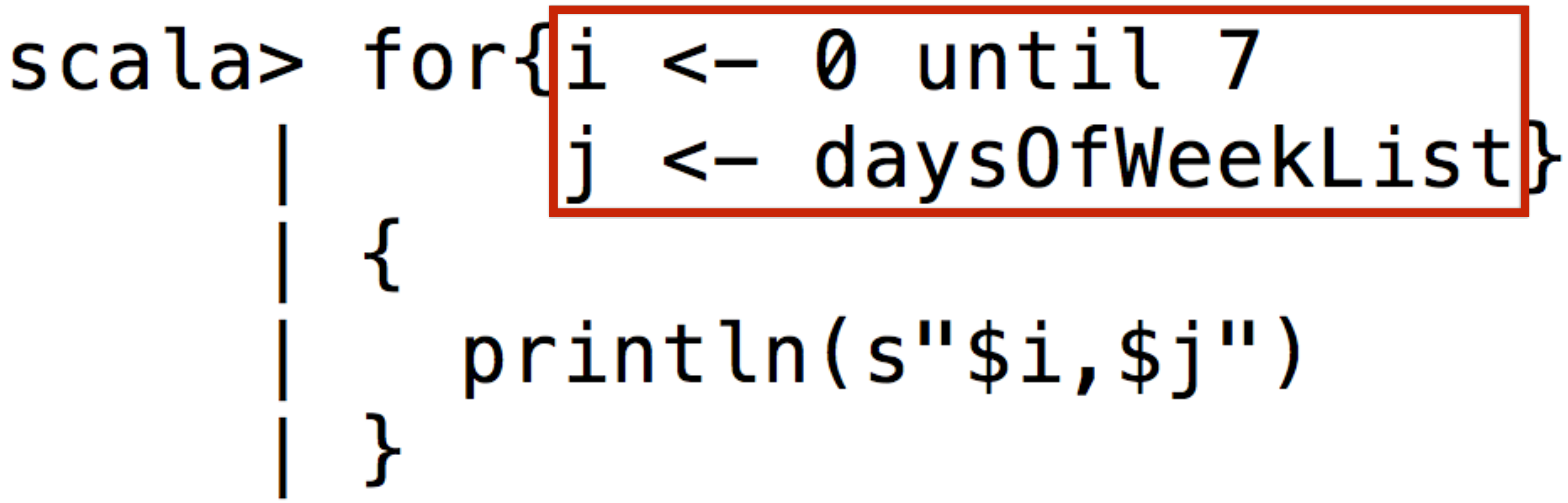
Nested `for` Loops: Nested Iterators



```
scala> for{i <- 0 until 7
     |       j <- 0 to 10}
     | {
     |    println(s"$i,$j")
     | }
```

Notice that there 2 ranges, without a comma separating them

Nested `for` Loops: Nested Iterators



```
scala> for{i <- 0 until 7
     |     j <- daysOfWeekList}
     | {
     |   println(s"$i,$j")
     | }
```

Its perfectly OK to combine different types of iterators

Example 21

`while/do-while`

Loops: Pure Statements

`while`/`do-while` Loops: Pure Statements

Many constructs that are statements in Java are expressions in Scala

✓ `if`/`else`

✓ `for` loops (But not `while` loops)

✓ `match`

while/do-while Loops: Pure Statements

Many constructs that are statements in Java are expressions in Scala

for loops can be expressions OR statements

(But not while loops)

for loops can be expressions OR statements

For loops can be set up as either statements or expressions by adding or removing just 1 word

```
yield
```

The presence of this word converts a for-loop into an expression

For loops can be set up as either statements or expressions by adding or removing just 1 word

`yield`

The presence of this word converts a for-loop into an expression

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

For loops can be set up as either statements or expressions by adding or removing just 1 word

yield

`while/do-while`

Loops: Pure Statements

The presence of this word turns a for-loop into an expression

While loops can't return anything, they don't work with `yield`

A for-loop with yield will "yield" a collection of the return values of each iteration of the loop

while/do-while

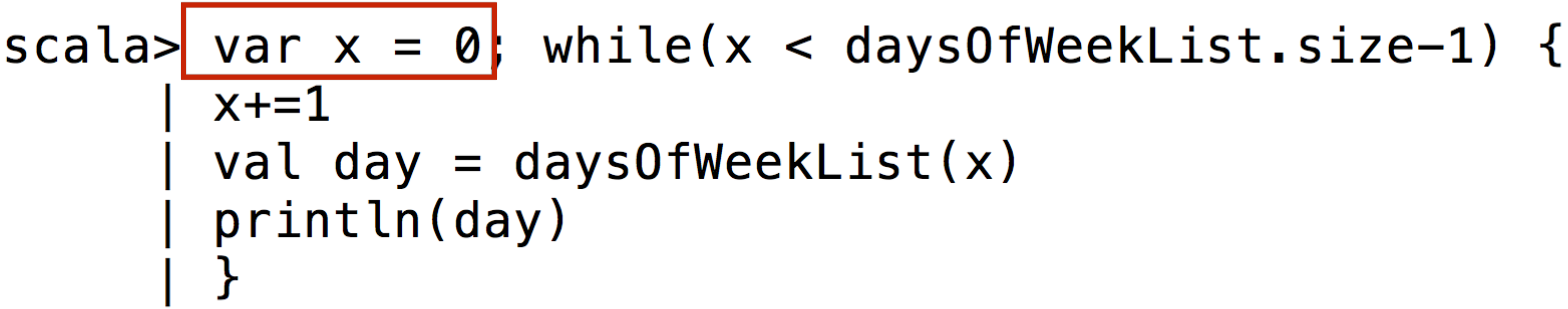
Loops: Pure Statements

```
scala> var x = 0; while(x < daysOfWeekList.size-1) {
     | x+=1
     | val day = daysOfWeekList(x)
     | println(day)
     | }
```

While loops can't return anything, they don't work with `yield`

while/do-while

Loops: Pure Statements

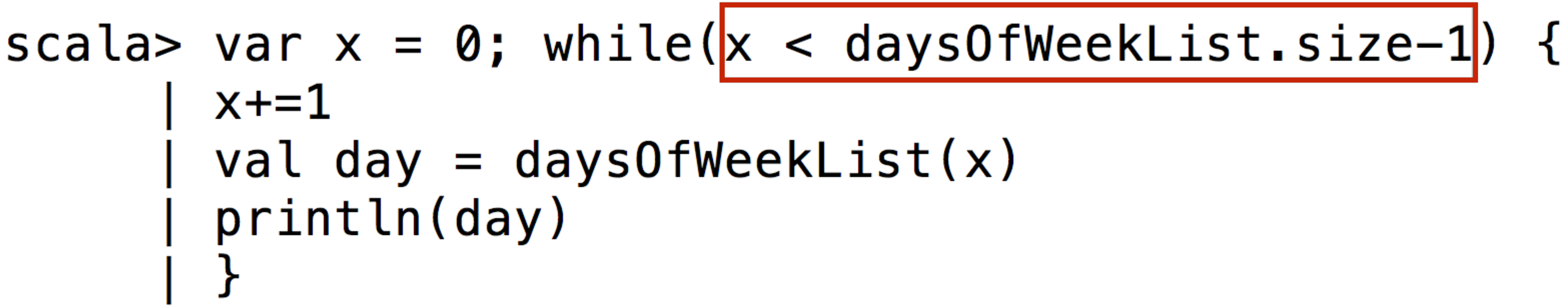


```
scala> var x = 0; while(x < daysOfWeekList.size-1) {
     | x+=1
     | val day = daysOfWeekList(x)
     | println(day)
     | }
```

Notice that we finally have a use for var!

while/do-while

Loops: Pure Statements

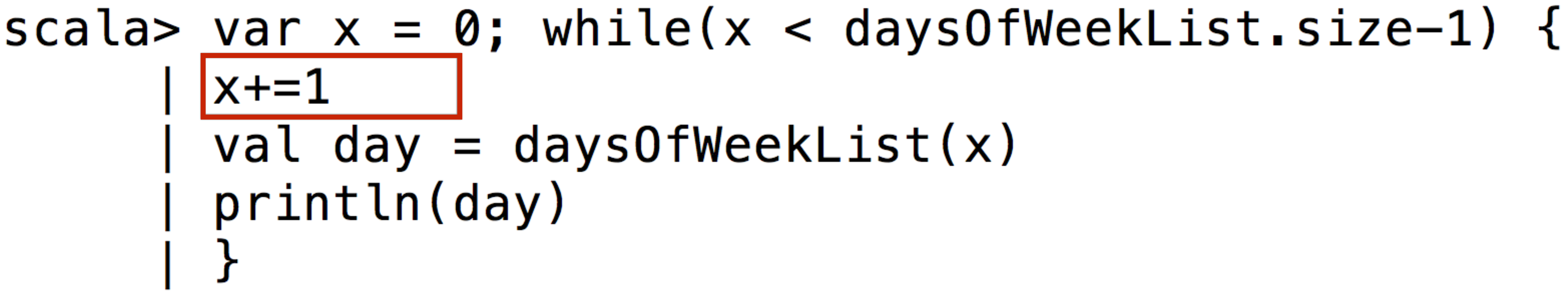


```
scala> var x = 0; while(x < daysOfWeekList.size-1) {
     | x+=1
     | val day = daysOfWeekList(x)
     | println(day)
     | }
```

The while loop condition looks pretty Java-like

while/do-while

Loops: Pure Statements

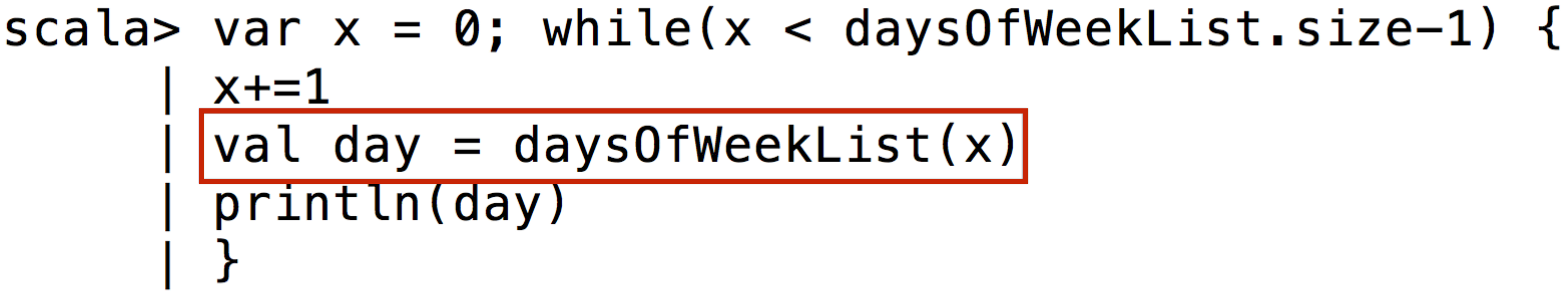


```
scala> var x = 0; while(x < daysOfWeekList.size-1) {
     | x+=1
     | val day = daysOfWeekList(x)
     | println(day)
     | }
```

Clumsy Syntax #1: The body of the while loop has to increment the loop variable

while/do-while

Loops: Pure Statements



```
scala> var x = 0; while(x < daysOfWeekList.size-1) {
     |   x+=1
     |   val day = daysOfWeekList(x)
     |   println(day)
     | }
```

Clumsy Syntax #2: The corresponding value binding needs to be explicit

while/do-while

Loops: Pure Statements

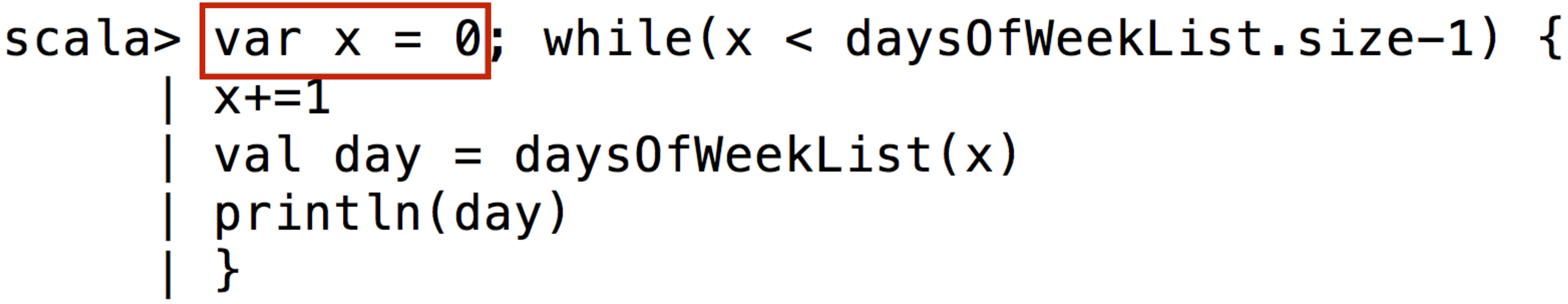
```
scala> var x = 0; while(x < daysOfWeekList.size-1) {
     | x+=1
     | val day = daysOfWeekList(x)
     | println(day)
     | }
```

```
Tue
Wed
Thu
Fri
Sat
Sun
x: Int = 6
```

Clumsy Syntax #3: The output can not be "composed" i.e. passed to a different function

while/do-while

Loops: Pure Statements



```
scala> var x = 0; while(x < daysOfWeekList.size-1) {
     | x+=1
     | val day = daysOfWeekList(x)
     | println(day)
     | }
```

Clumsy Syntax #4: While loops often require mutable variables

Drawbacks: side-effects in code, problems in multithreaded applications etc

`while/do-while`

Loops: Pure Statements

Clumsy Syntax

While loops have a lot of clumsy syntax..they are not used a whole lot in Scala