Introduction to Scala

Ángel J. Rey

AGENDA

1. Introduction & Installation
2. Basic Operations – Type Inference
3. Data: Literals, Values, Variables & Types
4. Expressions & Conditionals
5. Functions
6. First-class functions

AGENDA

7. Common Collections
8. More Collections
9. Classes
10. Objects, Case Classes & Traits
11. Advanced Typing

Scope of the course

- Introduction to Scala with:
  - Lectures
  - Hands-on exercises
  - Virtual Machine
  - And much more …
- Course Materials available in GitHub

https://github.com/chicochica10/spark-scala-developer

Contact me if you need help !!

INTRODUCTION TO SCALA

1. Introduction and Installation

Scalable language

- **Scala** is a **modern multi-paradigm** programming language designed to express **common** programming **patterns** in a **concise**, **elegant**, and **type-safe** way.

- Concise
- Scriptable
- Read-Eval-Print-Loop (REPL)
- Functional (Higher Order Functions)
- Extend existing classes (implicits)
- Duck Typing

- IDE'S Support
- Fewer Tests
- Good Documentation
- Open Source
- Performance
- Runs on JVM, inter-op with Java
- Dynamic Features

Scala good integration

- Can be used as drop-in replacement for Java
- Mixed Scala/Java projects
- Use existing Java libraries
- Use existing Java tools (Ant, Maven, JUnit, etc…)
- Decent IDE Support (NetBeans, IntelliJ, Eclipse)


INTRODUCTION TO SCALA
2. Basic Operations –
Type Inference
Scala

Type Inference

- Implicit types
  - **val** sum = 1 + 2 + 3
  - **val** nums = List(1, 2, 3)
  - **val** map = Map("abc" -> List(1,2,3))
- Explicit types
  - **val** sum**: Int** = 1 + 2 + 3
  - **val** nums**: List[Int]** = List(1, 2, 3)
  - **val** map**: Map[String, List[Int]]** = ...

Exercises

- **Exercise 1 – Basic operations in RPEL / Worksheet**

INTRODUCTION TO SCALA

3. Data: Literals, Values, Variables & Types

Literals, Values, Variables & Types

- A **literal** (or literal data) is data that appears directly in the source code, like the number **5**, the character **A**, and the text **“Hello, World.”**
- A **value** is an immutable, typed storage unit. A value can be assigned data when it is defined, but it can never be reassigned.

```
scala> val x: Int = 20
x: Int = 20
scala> val greeting: String = "Hello, World"
greeting: String = Hello, World
scala> val atSymbol: Char = '@'
atSymbol: Char = @
```

Literals, Values, Variables & Types

- A **variable** is a mutable, typed storage unit. A **variable** can be assigned when it is defined and can also be **reassigned** at any time.

```
scala> var a: Double = 2.72
a: Double = 2.72
scala> a = 355.0 / 113.0
a: Double = 3.141592920353982 5
```

- A **type** is the kind of data you are working with, a definition or classification of data. All data in Scala corresponds to a specific type, and all Scala types are defined as classes with methods that operate on the data. It's optional to type it (**type inference**). Some basic types:

Byte	Short	Int	Long	Float	Double	Boolean	Char	String
1 byte	2 bytes	4 bytes 5, 0x0f	8 bytes 5l, 5L	4 bytes 5f, 5F	8 bytes 5.0, 5d, 5D	true / false	'h'	"hello"

Immutability to avoid side effects

- In **Scala**, **values are preferred over variables** by convention, due to the stability and predictability they bring to source code.
- When you define a **value** you can be assured that it will **retain** the same value regardless of any other code that may access it.
- When working with data that may be available from **concurrent** or **multithreaded** code, an **immutable value** will be more stable and less prone to errors than mutable data that may be modified at unexpected times.
- However, in those places where **variables are more suitable**, such as **local variables that store temporary data** or **accumulate values** in loops, variables will certainly be used.

String Interpolation

- s prefix:

```
scala> val item = "apple"
item: String = apple
scala> s"How do you like them ${item}s?"
res0: String = How do you like them apples?
scala> s"Fish n chips n vinegar, ${"pepper "*3}salt"
res1: String = Fish n chips n vinegar, pepper pepper pepper salt
```

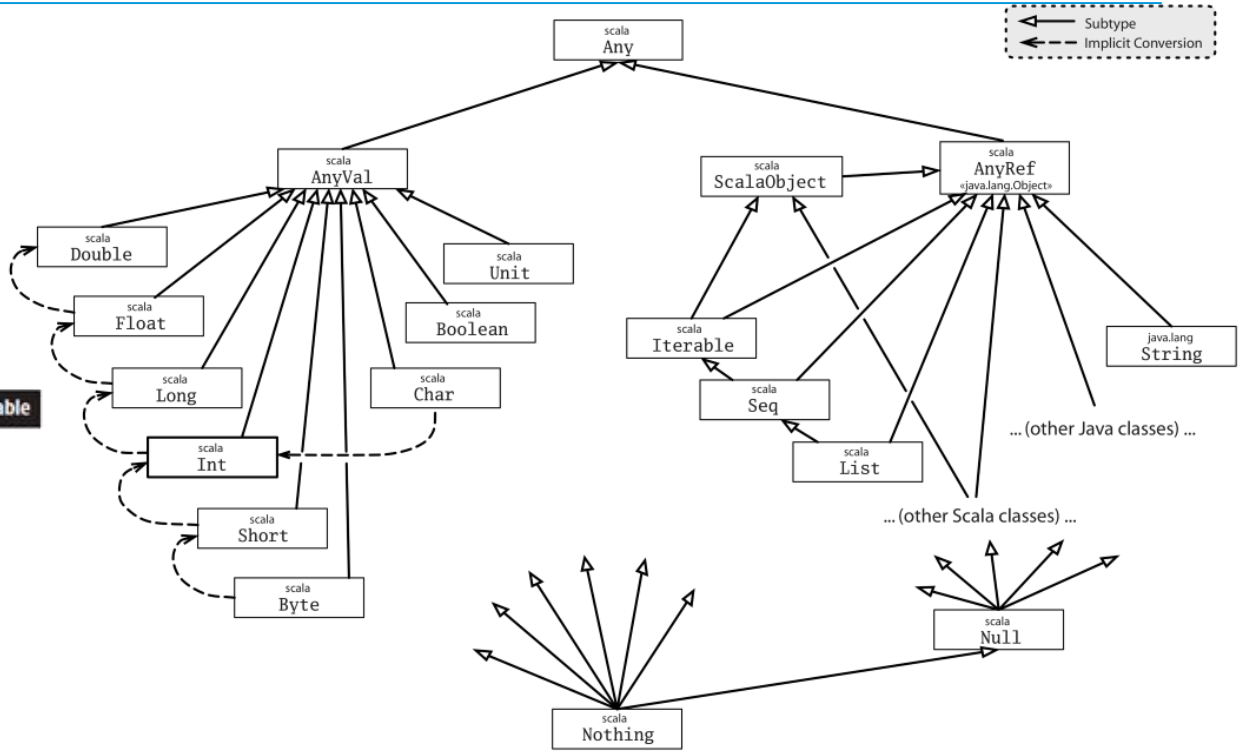
- f prefix (printf notation):

```
scala> val item = "apple"
item: String = apple
scala> f"I wrote a new $item%.3s today"
res2: String = I wrote a new app today
scala> f"Enjoying this $item ${355/113.0}%.5f times today"
res3: String = Enjoying this apple 3.14159 times today
```

Scala class hierarchy

Core non numeric types

Name	Description	Instantiable
`Any`	The root of all types in Scala	No
`AnyVal`	The root of all value types	No
`AnyRef`	The root of all reference (nonvalue) types	No
`Nothing`	The subclass of all types	No
`Null`	The subclass of all AnyRef types signifying a null value	No
`Char`	Unicode character	Yes
`Boolean`	`true` or `false`	Yes
`String`	A string of characters (i.e., text)	Yes
`Unit`	Denotes the lack of a value	No



Type operations

- Common type operations

Name	Example	Description
`asInstanceOf[<type>]`	`5.asInstanceOf[Long]`	Converts the value to a value of the desired type. Causes an error if the value is not compatible with the new type.
`getClass`	`(7.0 / 5).getClass`	Returns the type (i.e., the class) of a value.
`isInstanceOf`	`(5.0).isInstanceOf[Float]`	Returns true if the value has the given type.
`hashCode`	`"A".hashCode`	Returns the hash code of the value, useful for hash-based collections.
`to<type>`	`20.toByte; 47.toFloat`	Conversion functions to convert a value to a compatible value.
`toString`	`(3.0 / 4.0).toString`	Renders the value to a `String`.

Tuples

Creating:

```
scala> val info = (5, "Korben", true)
info: (Int, String, Boolean) = (5,Korben,true)
```

Accessing:

```
scala> val name = info._2
name: String = Korben
```

Using ->

```
scala> val red = "red" -> "0xff0000"
red: (String, String) = (red,0xff0000)
scala> val reversed = red._2 -> red._1
reversed: (String, String) = (0xff0000,red)
```

Exercises

- **Exercise 2 – Literals, Values, Variables & Types**

INTRODUCTION TO SCALA

4. Expressions & Conditionals

Expressions and Conditionals

- Expression: a single unit of code that returns a value

```
scala> "hello"
scala> "hel" + 'l' + "o"
scala> val x = 5 * 20; val amount = x + 10 //2 expr. 1 line
scala> val amount = { val x = 5 * 20; x + 10 } // better
scala> val amount = {
| val x = 5 * 20
| x + 10
| }//the same
//nested expr.
scala> { val a = 1; { val b = a * 2; { val c = b + 4; c } } }
```

- Statements: Expressions that doesn't return a value (return **Unit**)

```
scala> val x = 1
// REPL repeats the definition of x but not data returned that can
// be used to create a new value
```

IF-Else Expressions

- Simple IF-block

```
scala> if ( 47 % 3 > 0 ) println("Not a multiple of 3")
```

- Returns Any type

```
val result = if ( false ) "what does this return?"
```

- if-else

```
val x = 10; val y = 20; val max = if (x > y) x else y
//if-else
```

- **If everything fits on one line**, using a **single expression** without an expression block in **if..else expressions** works well
- But in case if..else expression **doesn't easily fit on a single line**, consider using **expression blocks** to make your code **more readable**.

Match Expressions

- Similar to C and Java but more flexible (types, reg. exp. Numeric range, data structures)
- Preferred over if-else nested blocks

```
scala> val x = 10; val y = 20
x: Int = 10
y: Int = 20
scala> val max = x > y match {
| case true => x
| case false => y
| }
max: Int = 20
```

```
scala> val status = 500
status: Int = 500
scala> val message = status match {
| case 200 => "ok"
| case 400 => {
| println("ERR- not found")
| "error"
| }
| case 500 => {
| println("ERR - internal")
| "error"
| }
| }
```

Match Expressions with patterns

- Value Binding Pattern

```
scala> val message = "Ok"
message: String = Ok
scala> val status = message match {
| case "Ok" => 200
| case other => {
| println(s"Couldn't parse $other")
| -1
| }
|}
```

- Pattern Guard

```
scala> val response: String = null
response: String = null
scala> response match {
| case s if s != null =>
          println(s"Rec.'$s'")
| case s =>
     println("Error! Rec. null res")
| }
```

Match Expressions (cont.)

- Wildcard _ Operator Pattern

```
scala> val message = "Unauthorized"
message: String = Unauthorized
scala> val status = message match {
| case "Ok" => 200
| case _ => {
| println(s"Couldn't parse $message")
| -1
| }
|}
```

- Matching Types

```
scala> val x: Int = 12180
x: Int = 12180
scala> val y: Any = x
y: Any = 12180
scala> y match {
| case x: String => s"'x'"
| case x: Double => f"$x%.2f"
| case x: Float => f"$x%.2f"
| case x: Long => s"${x}l"
| case x: Int => s"${x}i"
| }
```

Loops - For

- For (Simple)

```
scala> for (x <- 1 to 7) { println(s"Day $x:") }
```

- For (with yield)

```
scala> for (x <- 1 to 7) yield { s"Day $x:" }
res10: scala.collection.immutable.IndexedSeq[String] =
Vector(Day 1:,
Day 2:, Day 3:, Day 4:, Day 5:, Day 6:, Day 7:)
scala> for (day <- res0) print(day + ", ")
Day 1:, Day 2:, Day 3:, Day 4:, Day 5:, Day 6:, Day 7:,
```

- For (filter)

```
scala> val threes = for (i <- 1 to 20 if i % 3 == 0) yield i
threes: scala.collection.immutable.IndexedSeq[Int] = Vector(3,
6, 9, 12, 15, 18)
```

Loops - For

- For (Nested)

```
scala> for { x <- 1 to 2
| y <- 1 to 3 }
| { print(s"($x,$y) ") }
(1,1) (1,2) (1,3) (2,1) (2,2) (2,3)
```

- For (functional)

```
scala> val powersOf2 = for (i <- 0 to 8; pow = 1 << i) yield pow
powersOf2: scala.collection.immutable.IndexedSeq[Int] =
Vector(1, 2, 4, 8, 16, 32, 64, 128, 256)
```

Loops – While Do/While

- Do not use them! (preferably with functional programming, but also not a dogma)
- While

```
scala> var x = 10; while (x > 0) x -= 1
x: Int = 0
```

- Do / While

```
scala> val x = 0
x: Int = 0
scala> do println(s"Here I am, x = $x") while (x > 0)
Here I am, x = 0
```

Exercises

- **Exercise 3 – Expressions and conditionals**

INTRODUCTION TO SCALA

5. Functions

Functions (1 / 3)

- *Functions* are **named** and **reusable** expressions.
- They may be parametrized and may return a value
- Scala aims to build **pure functions** (functional programming)
- A **pure function** is one that:
  - Has one or more input parameters
  - Performs calculations using only the input parameters
  - Returns a value
  - Always returns the same value for the same input
  - Does not **use or affect any data outside** the function
  - **Is not affected by any data outside** the function

Functions (2 / 3)

- Defining an Input-less Function – empty () can be used

```
scala> def hi = "hi"
hi: String
```

- Defining a Function with a Return Type (not mandatory except for recursive functions)

```
scala> def hi: String = "hi"
hi: String
```

- General Definition of a function

```
scala> def multiplier(x: Int, y: Int): Int = { x * y }
multiplier: (x: Int, y: Int)Int
scala> multiplier(6, 7)
res0: Int = 42
```

Functions (3 / 3)

- **Procedures** - Functions that don't return a value (returns Unit) and end with a statement

```
scala> def log(d: Double): Unit = println(f"Got value $d%.2f")
log: (d: Double)Unit
scala> log(2.23535)
Got value 2.24
```

- Function Invocation with Expression Blocks

```
scala> def formatEuro(amt: Double) = f"€$amt%.2f"
formatEuro: (amt: Double)String
scala> formatEuro { val rate = 1.32; 0.235 + 0.7123 + rate *
5.32 }
res5: String = €7.97
```

Recursive Functions

- A ***recursive* function** is one that may **invoke itself**, preferably with some type of parameter or external condition that will be checked to avoid an infinite loop of function invocation
- **Functional programming** because they offer a way to iterate over data structures **without mutable data**. Each call has **its own stack**
- Always try to use **tail recursion** (last statement is the recursive invocation) and mark the function with the annotation **@annotation.tailrec**

```
scala> @annotation.tailrec
| def power(x: Int, n: Int, t: Int = 1): Int = {
| if (n < 1) t
| else power(x, n-1, x*t)
| }
power: (x: Int, n: Int, t: Int)Int
scala> power(2,8)
res9: Int = 256
```

Other things about Functions (1/3)

- Nested Functions

```
scala> def max(a: Int, b: Int, c: Int) = {
| def max(x: Int, y: Int) = if (x > y) x else y
| max(a, max(b, c))
| }
max: (a: Int, b: Int, c: Int)Int
scala> max(42, 181, 19)
res10: Int = 181
```

- Named Parameters

```
scala> def greet(prefix: String, name: String) = s"$prefix $name"
greet: (prefix: String, name: String)String
scala> val greeting = greet(name = "Brown", prefix = "Mr")
greeting: String = Mr Brown
```

- Default Values

```
scala> def greet(prefix: String = "", name: String) = s"$prefix$name"
greet: (prefix: String, name: String)String
scala> val greeting1 = greet(name = "Paul")
greeting1: String = Paul
```

Other things about Functions (2/3)

- Var arg Parameters

```
scala> def sum(items: Int*): Int = {
| var total = 0
| for (i <- items) total += i
| total
| }
sum: (items: Int*)Int
scala> sum(10, 20, 30)
res11: Int = 60
scala> sum()
res12: Int = 0
```

- Parameter Groups (more about this later)

```
scala> def max(x: Int)(y: Int) = if (x >
y) x else y
max: (x: Int)(y: Int)Int
scala> val larger = max(20)(39)
larger: Int = 39
```

- Type Parameters (like Generics)

```
scala> def identity[A](a: A): A = a
identity: [A](a: A)A
scala> val s: String =
identity[String]("Hello")
s: String = Hello
scala> val d: Double =
identity[Double](2.717)
d: Double = 2.717
```

Other things about Functions (3/3)

- Methods & Operators

  A *method* is a function defined in a class and available from any instance of the class. (more about classes later)

```
scala> val s = "vacation.jpg"
s: String = vacation.jpg
scala> val isJPEG = s.endsWith(".jpg")
isJPEG: Boolean = true
```

```
scala> val d = 65.642
d: Double = 65.642
scala> d.+(2.721)
res16: Double = 68.363

scala> d compare 18.0
res17: Int = 1
scala> d + 2.721
res18: Double = 68.363
```

Exercises

- **Exercise 4 – Functions**

INTRODUCTION TO SCALA

6. First-class functions

First Order Functions

- Can be created in **literal form** without ever having been assigned an identifier
- Be **stored in a container** such as a value, variable, or data structure
- Be **used as a parameter** to another function or used as the **return value** from another function.
- The final goal is make **declarative programming / functional programing** (what to do) instead imperative programming (how to do)

Function Types and Values (1/2)

- The *type* of a function is a simple grouping of its input types and return value type

```
scala> def double(x: Int): Int = x * 2
double: (x: Int)Int
scala> double(5)
res0: Int = 10
scala> val myDouble: (Int) => Int = double
myDouble: Int => Int = <function1>
scala> myDouble(5)
res1: Int = 10
scala> val myDoubleCopy = myDouble
myDoubleCopy: Int => Int = <function1>
scala> myDoubleCopy(5)
res2: Int = 10
```

- Assigning a Function with _ (to distinguish it from invocation)

```
scala> def double(x: Int): Int = x * 2
double: (x: Int)Int
scala> val myDouble = double _
myDouble: Int => Int = <function1>
scala> val amount = myDouble(20)
amount: Int = 40
```

Function Types and Values (2 / 2)

- Function type with no parameters

```
scala> def logStart() = "=" * 50 + "\nStarting NOW\n"
logStart: ()String
scala> val start: () => String = logStart
start: () => String = <function0>
scala> println( start() )
==================================================
Starting NOW
```

- Function type with multiples parameters

```
scala> def max(a: Int, b: Int) = if (a > b) a else b
max: (a: Int, b: Int)Int
scala> val maximize: (Int, Int) => Int = max
maximize: (Int, Int) => Int = <function2>
scala> maximize(50, 30)
res3: Int = 50
```

Higher-Order Functions

- Functions that have a function type as an input parameter or return value.

```
scala> def safeStringOp(s: String, f: String => String) = {
| if (s != null) f(s) else s
| }
safeStringOp: (s: String, f: String => String)String
scala> def reverser(s: String) = s.reverse
reverser: (s: String)String
scala> safeStringOp(null, reverser)
res4: String = null
scala> safeStringOp("Ready", reverser)
res5: String = ydaeR
```

Functions Literals or Anonymous Functions or Lambdas expressions (1 / 2)

- A function that lacks a name
  - One parameter

```
scala> val doubler = (x: Int) => x * 2
doubler: Int => Int = <function1>
scala> val doubled = doubler(22)
doubled: Int = 44
```

  - Two parameters

```
scala> val maximize = (a: Int, b: Int) => if (a > b) a else b
maximize: (Int, Int) => Int = <function2>
scala> maximize(84, 96)
res6: Int = 96
```

Functions Literals (2 / 2)

- Can be used in higher-order functions

```
scala> def safeStringOp(s: String, f: String => String) = {
| if (s != null) f(s) else s
| }
safeStringOp: (s: String, f: String => String)String
scala> safeStringOp(null, (s: String) => s.reverse)
res7: String = null
scala> safeStringOp("Ready", (s: String) => s.reverse)
res8: String = ydaeR
```

Placeholder _ Syntax (1 / 2)

- Shortened form of functions literals
  - If the function type is outside the literal and the parameter is **used only once**
- **Example 1**

```
scala> val doubler: Int => Int = _ * 2
doubler: Int => Int = <function1>
```

- **Example 2**

```
scala> def safeStringOp(s: String, f: String => String) = {
| if (s != null) f(s) else s
| }
safeStringOp: (s: String, f: String => String)String
scala> safeStringOp(null, _.reverse)
res11: String = null
scala> safeStringOp("Ready", _.reverse)
res12: String = ydaeR
```

Placeholder _ Syntax (2 / 2)

- **Example 3**

```
scala> def tripleOp[A,B](a: A, b: A, c: A, f: (A, A, A) => B) = f(a,b,c)
tripleOp: [A, B](a: A, b: A, c: A, f: (A, A, A) => B)B
scala> tripleOp[Int,Int](1, 1, 1, _ + _ - _)
res15: Int = 1
scala> tripleOp[Int,Double](23, 92, 14, 1.0 * _ / _ / _)
res16: Double = 0.017857142857142856
scala> tripleOp[Int,Boolean](3, 1, 1, _ > _ + _)
res17: Boolean = false
```

Partially Applied Functions and Currying

- def factorOf(x: Int, y: Int) has the function type (Int, Int) => Boolean
- def factorOf(x: Int)(y: Int) has the function type Int => Int => Boolean
- Reuse a function invocation and retain the value of some of the parameters to avoid typing them again

```
scala> def factorOf(x: Int, y: Int) = y % x == 0
factorOf: (x: Int, y: Int)Boolean
scala> val f = factorOf _
f: (Int, Int) => Boolean = <function2>
scala> val x = f(7, 20)
x: Boolean = false
```

Partially Applied Functions and Currying

- For checking multiple of 3:

```
scala> val multipleOf3 = factorOf(3, _: Int)
multipleOf3: Int => Boolean = <function1>
scala> val y = multipleOf3(78)
y: Boolean = true
```

- Better way *Currying* the function:

```
scala> def factorOf(x: Int)(y: Int) = y % x == 0
factorOf: (x: Int)(y: Int)Boolean
scala> val isEven = factorOf(2) _
isEven: Int => Boolean = <function1>
scala> val z = isEven(32)
z: Boolean = true
```

By-Name Parameters

- Each time a **by-name parameter** is used inside a function, it gets **evaluated into a value**.
- If a **value** is passed to the function then there is no effect, but **if a function is passed** then that function **is invoked for every usage**.
- A function passed in a **by-name parameter** will not be invoked if the parameter is not accessed, **so a costly function call can be avoided if necessary**.

By-Name Parameters (Example)

```
scala> def doubles(x: => Int) = {
| println("Now doubling " + x)
| x * 2
| }
doubles: (x: => Int)Int
scala> doubles(5)
Now doubling 5
res18: Int = 10
scala> def f(i: Int) = { println(s"Hello from f($i)"); i }
f: (i: Int)Int
scala> doubles( f(8) )
Hello from f(8)
Now doubling 8
Hello from f(8)
res19: Int = 16
```

The x by-name parameter is accessed here just like a normal by-value parameter

Invoke the doubles method with a regular value and it will operate normally

…but when you invoke it with a function value, that function value will get invoked inside the doubles method

Because the double method refers to the x parameter twice, the “Hello” message gets invoked twice

Partial Functions

- **Function literals** that apply a **series of case patterns** to their input, requiring that the input **match at least one of the given patterns**.
- Invoking one of these partial functions with data that **does not meet at least one case pattern** results in a **Scala error**

```
scala> val statusHandler: Int => String = {
| case 200 => "Okay"
| case 400 => "Your Error"
| case 500 => "Our error"
| }
statusHandler: Int => String = <function1>
525252
scala> statusHandler(200)
res20: String = Okay
scala> statusHandler(401)
scala.MatchError: 401 (of class java.lang.Integer)
    at $anonfun$1.apply(<console>:7)
    at $anonfun$1.apply(<console>:7)
    ... 32 elided
```

Invoking Higher-Order Functions with Function Literal Blocks (1 / 2)

- To invoke utility functions with an expression block. For example, a higher-order function can wrap a given expression block in a single database session or transaction.
- **Example1:**

```
scala> def safeStringOp(s: String)(f: String => String) = {
| if (s != null) f(s) else s
| }
safeStringOp: (s: String)(f: String => String)String
scala> val timedUUID = safeStringOp(uuid) { s =>
| val now = System.currentTimeMillis
| val timed = s.take(24) + now
| timed.toUpperCase
| }
timedUUID: String = BFE1DDDA-92F6-4C7A-8BFC-1394546915011
```

Invoking Higher-Order Functions with Function Literal Blocks (2 / 2)

- **Example 2:**

```
scala> def timer[A](f: => A): A = {
| def now = System.currentTimeMillis
| val start = now; val a = f; val end = now
| println(s"Executed in ${end - start} ms")
| a
| }
timer: [A](f: => A)A
scala> val veryRandomAmount = timer {
| util.Random.setSeed(System.currentTimeMillis)
| for (i <- 1 to 100000) util.Random.nextDouble
| util.Random.nextDouble
| }
```

The type parameter “A” helps the return type of the **“f” by-name** parameter become the return type of the “timer” function, reducing the impact of wrapping code with the “timer” function

This inner, nested function is here for purely aesthetic reasons, enabling us to retrieve the current millisecond amount compactly.

Finally, we have reduced the expression block syntax for higher-order functions to its simplest form: the function name and the block. You can view the code between the braces as being an expression block, or as a function literal block, or as regular code being *wrapped* by the “timer” function

Exercises

- **Exercise 5 – First-Class Functions**

INTRODUCTION TO SCALA

7. Common Collections

Common Collections

- A ***collections* framework** provides data structures for collecting one or more values of a given type such as arrays, lists, maps, sets, and trees.
- **Scala** has a high-performance, **object-oriented**, and **type-parameterized Collections**
- Collections have **higher-order operations** like **map**, filter, and reduce that make it possible to manage and manipulate data with short and expressive expressions
- Scala has **separate mutable versus immutable collection** type hierarchies (always favor immutable version in your code)
- **Root of all iterable collections**: `Iterable`

Lists (1/2)

- List type: An immutable singly linked list.

```
scala> val numbers = List(32, 95, 24, 21, 17)
numbers: List[Int] = List(32, 95, 24, 21, 17)

scala> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)

scala> println(s"I have ${colors.size} colors: $colors")
I have 3 colors: List(red, green, blue)
```

- Head & Tail

```
scala> colors.head
res0: String = red
scala> colors.tail
res1: List[String] = List(green, blue)
scala> colors(1)
res2: String = green
scala> colors(2)
res3: String = blue
```

- Call.isEmpty to check for the end of the list. All lists end with an invisible Nil

List (2 / 2)

- For loops (DON'T use)

```
scala> val numbers = List(32, 95, 24, 21, 17)
numbers: List[Int] = List(32, 95, 24, 21, 17)
scala> var total = 0; for (i <- numbers) { total += i }
total: Int = 189
scala> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)
scala> for (c <- colors) { print(c + " ") }
red green blue
```

- Higher-order functions

```
scala> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)
scala> colors.foreach( (c: String) => print(c + " ") )
red green blue
scala> val sizes = colors.map( (c: String) => c.size )
sizes: List[Int] = List(3, 5, 4)
scala> val total = numbers.reduce( (a: Int, b: Int) => a + b )
total: Int = 189
```

takes a function (a procedure, to be accurate) and invokes it with every item in the list.

map() takes a function that converts a single list element to another value and/or type.

reduce() takes a function that combines two list elements into a single element

List Arithmetic – Operations on lists

Name	Example	Description
::	`1 :: 2 :: Nil`	Appends individual elements to this list. A right-associative operator.
:::	`List(1, 2) ::: List(2, 3)`	Prepends another list to this one. A right-associative operator.
++	`List(1, 2) ++ Set(3, 4, 3)`	Appends another collection to this list.
==	`List(1, 2) == List(1, 2)`	Returns true if the collection types and contents are equal.
distinct	`List(3, 5, 4, 3, 4).distinct`	Returns a version of the list without duplicate elements.
drop	`List('a', 'b', 'c', 'd') drop 2`	Subtracts the first *n* elements from the list.
filter	`List(23, 8, 14, 21) filter (_ > 18)`	Returns elements from the list that pass a true/false function.
flatten	`List(List(1, 2), List(3, 4)).flatten`	Converts a list of lists into a single list of elements.
partition	`List(1, 2, 3, 4, 5) partition (_ < 3)`	Groups elements into a tuple of two lists based on the result of a true/false function.
reverse	`List(1, 2, 3).reverse`	Reverses the list.
slice	`List(2, 3, 5, 7) slice (1, 3)`	Returns a segment of the list from the first index up to but not including the second index.
sortBy	`List("apple", "to") sortBy (_.size)`	Orders the list by the value returned from the given function.
sorted	`List("apple", "to").sorted`	Orders a list of core Scala types by their natural value.
splitAt	`List(2, 3, 5, 7) splitAt 2`	Groups elements into a tuple of two lists based on if they fall before or after the given index.
take	`List(2, 3, 5, 7, 11, 13) take 3`	Extracts the first *n* elements from the list.
zip	`List(1, 2) zip List("a", "b")`	Combines two lists into a list of tuples of elements at each index.

Mapping Lists

- List mapping operations

Name	Example	Description
`collect`	`List(0, 1, 0) collect {case 1 => "ok"}`	Transforms each element using a partial function, retaining applicable elements.
`flatMap`	`List("milk,tea") flatMap (_.split(','))`	Transforms each element using the given function and "flattens" the list of results into this list.
`map`	`List("milk","tea") map (_.toUpperCase)`	Transforms each element using the given function.

Reducing Lists (1 / 2)

- Math reduction operations

Name	Example	Description
max	`List(41, 59, 26).max`	Finds the maximum value in the list.
min	`List(10.9, 32.5, 4.23, 5.67).min`	Finds the minimum value in the list.
product	`List(5, 6, 7).product`	Multiplies the numbers in the list.
sum	`List(11.3, 23.5, 7.2).sum`	Sums up the numbers in the list.

- Boolean reduction operations

Name	Example	Description
contains	`List(34, 29, 18) contains 29`	Checks if the list contains this element.
endsWith	`List(0, 4, 3) endsWith List(4, 3)`	Checks if the list ends with a given list.
exists	`List(24, 17, 32) exists (_ < 18)`	Checks if a predicate holds true for *at least one* element in the list.
forall	`List(24, 17, 32) forall (_ < 18)`	Checks if a predicate holds true for *every* element in the list.
startsWith	`List(0, 4, 3) startsWith List(0)`	Tests whether the list starts with a given list.

Reducing Lists (2 / 2)

- Generic list reduction operations

Name	Example	Description
fold	List(4, 5, 6).fold(0)(_ + _)	Reduces the list given a starting value and a reduction function.reduction function.
foldLeft	List(4, 5, 6).foldLeft(0)(_ + _)	Reduces the list from left to right given a starting value and a reduction function.
foldRight	List(4, 5, 6).foldRight(0)(_ + _)	Reduces the list from right to left given a starting value and a reduction function.
reduce	List(4, 5, 6).reduce(_ + _)	Reduces the list given a reduction function, starting with the first element in the list.
reduceLeft	List(4, 5, 6).reduceLeft(_ + _)	Reduces the list from left to right given a reduction function, starting with the first element in the list.
reduceRight	List(4, 5, 6).reduceRight(_ + _)	Reduces the list from right to left given a reduction function, starting with the first element in the list.
scan	List(4, 5, 6).scan(0)(_ + _)	Takes a starting value and a reduction function and returns a list of each accumulated value.
scanLeft	List(4, 5, 6).scanLeft(0)(_ + _)	Takes a starting value and a reduction function and returns a list of each accumulated value from left to right.
scanRight	List(4, 5, 6).scanRight(0)(_ + _)	Takes a starting value and a reduction function and returns a list of each accumulated value from right to left.

Fold, **reduce**, **scan** are just for distributed parallel, collections

Left is preferred unless right-to-left iteration is needed

Set

- A Set is an **immutable** and **unordered** collection of **unique** elements. It works similarly to a List

```
scala> val unique = Set(10, 20, 30, 20, 20, 10)
unique: scala.collection.immutable.Set[Int] = Set(10, 20, 30)
scala> val sum = unique.reduce( (a: Int, b: Int) => a + b )
sum: Int = 60
```

Map

- A Map is an immutable **key-value** store

```
scala> val colorMap = Map("red"->0xFF0000,"green"->0xFF00,"blue"->0xFF)
colorMap: scala.collection.immutable.Map[String,Int] =
Map(red -> 16711680, green -> 65280, blue -> 255)
scala> val redRGB = colorMap("red")
redRGB: Int = 16711680
scala> val cyanRGB = colorMap("green") | colorMap("blue")
cyanRGB: Int = 65535
scala> val hasWhite = colorMap.contains("white")
hasWhite: Boolean = false
scala> for (pairs <- colorMap) { println(pairs) }
(red,16711680)
(green,65280)
(blue,255)
```

Converting collections (1 / 2)

- It is easy to **convert between types**, so you can create a collection with one type and end up with the other
- Operations to convert collections

Name	Example	Description
mkString	`List(24, 99, 104).mkString(", ")`	Renders a collection to a `Set` using the given delimiters.
toBuffer	`List('f', 't').toBuffer`	Converts an immutable collection to a mutable one.
toList	`Map("a" -> 1, "b" -> 2).toList`	Converts a collection to a `List`.
toMap	`Set(1 -> true, 3 -> true).toMap`	Converts a collection of 2-arity (length) tuples to a `Map`.
toSet	`List(2, 5, 5, 3, 2).toSet`	Converts a collection to a `Set`.
toString	`List(2, 5, 5, 3, 2).toString`	Renders a collection to a `String`, including the collection's type.

Converting collections (2 / 2)

- Java and Scala collection conversions:

```
import collection.JavaConverters._
```

Name	Example	Description
asJava	List(12, 29).asJava	Converts this Scala collection to a corresponding Java collection.
asScala	new java.util.ArrayList(5).asScala	Converts this Java collection to a corresponding Scala collection.

Pattern Matching with Collections (1 / 2)

- Match single value patterns

```
scala> val statuses = List(500, 404)
statuses: List[Int] = List(500, 404)
scala> val msg = statuses.head match {
| case x if x < 500 => "okay"
| case _ => "whoah, an error"
| }
msg: String = whoah, an error
```

- Match a single value inside a collection:

```
scala> val msg = statuses match {
| case x if x contains(500) => "error"
| case _ => "okay"
| }
msg: String = has error
```

- Match entire collections:

```
scala> val msg = statuses match {
| case List(404, 500) => "not found"
| case List(500, 404) => "error"
| case List(200, 200) => "okay"
| case _ => "not sure what happened"
| }
msg: String = error & not found
```

Pattern Matching with Collections (2 / 2)

- Bind values to some or all elements of a collection

```
scala> val msg = statuses match {
| case List(500, x) => s"Error by $x"
| case List(e, x) => s"$e by $x"
| }
msg: String = Error followed by 404
```

- Lists are decomposable in head & tail

```
scala> val head = List('r','g','b')
match {
| case x :: xs => x
| case Nil => ' '
| }
head: Char = r
```

- Tuples also support patter matching

```
scala> val code = ('h', 204, true)
match {
| case (_, _, false) => 501
| case ('c', _, true) => 302
| case ('h', x, true) => x
| case (c, x, true) => {
| println(s"not expect code $c")
| x
| }
| }
code: Int = 204
```

Exercises

- **Exercise 6 – Common Collections**

INTRODUCTION TO SCALA

8. More Collections

Mutable / Immutable (1/2)

- Until now immutable collections

```
scala> val m = Map("AAPL" -> 597, "MSFT" -> 40)
m: scala.collection.immutable.Map[String,Int] =
Map(AAPL -> 597, MSFT -> 40)
scala> val n = m - "AAPL" + ("GOOG" -> 521)
n: scala.collection.immutable.Map[String,Int] =
Map(MSFT -> 40, GOOG -> 521)
scala> println(m)
Map(AAPL -> 597, MSFT -> 40)
```

A new map with “AAPL” and “MSFT” keys.

Removing “APPL” and **adding** “GOOG” gives us a **different collection**…

… while the **original collection** in “m” remains **the same**

Mutable / Immutable (2/2)

- We can work with mutable collections when **it will be safe** to use:
  - **Mutable data** structure within a function
  - Data converted to **immutable before being returned** (*E.g. .toList*)

```
scala> val nums = collection.mutable.Buffer[Int]()
nums: scala.collection.mutable.Buffer[Int] = ArrayBuffer()
scala> for (i <- 1 to 10) nums += i
scala> println(nums)
Buffer(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

- Mutable collection types

Immutable type	Mutable counterpart
collection.immutable.List	collection.mutable.Buffer
collection.immutable.Set	collection.mutable.Set
collection.immutable.Map	collection.mutable.Map

Arrays

- **Fixed-size**, **mutable**, **indexed collection**.
- It's not officially a collection, because it isn't in the "scala.collections" package and **doesn't extend from the root Iterable type**
  - Although it has all of the **Iterable operations** like map and filter)
- Wrapper around Java's array type
- Can be used like a sequence used to deal with Java code

```
scala> val colors = Array("red", "green", "blue")
colors: Array[String] = Array(red, green, blue)
scala> colors(0) = "purple"
scala> colors
res0: Array[String] = Array(purple, green, blue)
scala> println("very purple: " + colors)
very purple: [Ljava.lang.String;@70cf32e3

scala> val files = new java.io.File(".").listFiles
files: Array[java.io.File] = Array(./Build.scala, ./Dep.scala)
scala> val scala = files map (_.getName) filter(_ endsWith "scala")
scala: Array[String] = Array(Build.scala, Dep.scala)
```

Use a zero-based index to replace any item in an Array

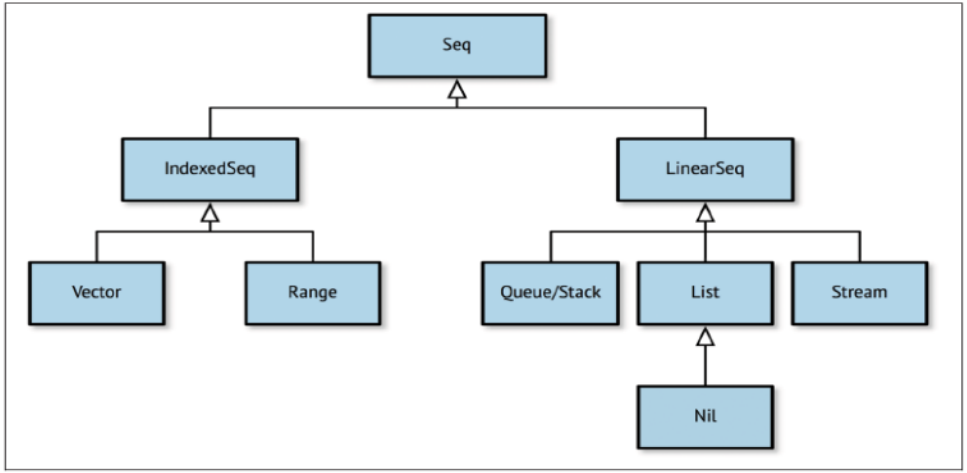
The Scala REPL knows how to print an Array …

but not println(), which can only call a type's toString() method

The **listFiles** method in java.io.File, a JDK class, returns an array that we can easily map and filter

Seq & Sequences

- **Seq** the root type of all sequences
- Not **instantiable** but **shortcut** for creating List
- The sequence collections hierarchy:



Name	Description
Seq	The root of all sequences. Shortcut for `List()`.
IndexedSeq	The root of indexed sequences. Shortcut for `Vector()`.
Vector	A list backed by an `Array` instance for indexed access.
Range	A range of integers. Generates its data on-the-fly.
LinearSeq	The root of linear (linked-list) sequences.
List	A singly linked list of elements.
Queue	A first-in-last-out (FIFO) list.
Stack	A last-in-first-out (LIFO) list.
Stream	A lazy list. Elements are added as they are accessed.
String	A collection of characters.

Streams

- Popular collection in functional programming Languages
- Builds **itself** as its elements are accessed
- **Lazy collection**. Generated from one or more starting elements and a recursive function
- They are **theoretically infinite collections**
- Streams **end** with Stream.Empty

Streams – Example

```
scala> def inc(i: Int): Stream[Int] = Stream.cons(i, inc(i+1))
//or using cons operator #:: (via implicit conversion)
scala> def inc(head: Int): Stream[Int] = head #:: inc(head+1)

inc: (i: Int)Stream[Int]
scala> val s = inc(1)
s: Stream[Int] = Stream(1, ?) //one element and a promise of futures
vals (?)
scala> val l = s.take(5).toList //forcing to build next four elements
l: List[Int] = List(1, 2, 3, 4, 5)
scala> s
res1: Stream[Int] = Stream(1, 2, 3, 4, 5, ?)
```

Streams – Bounded Stream

```
scala> def to(head: Char, end: Char): Stream[Char] = (head > end)
match {
| case true => Stream.empty
| case false => head #:: to((head+1).toChar, end)
| }
to: (head: Char, end: Char)Stream[Char]
scala> val hexChars = to('A', 'F').take(20).toList
hexChars: List[Char] = List(A, B, C, D, E, F)
```

Monadic Collections

- Scale only for a single element
- Support transformative operations like the ones in **Iterable** (that support “map” for example)

- Most important:
  - Option Collections
  - Try Collections
  - Future Collections

Option Collections (1/2)

- Will **never** be larger **than one**.
- Option represents the presence or **absence of a single value**
- Safe **replacement** for **null values**
- **Safe** way to build chains of operations with only valid values
- **Option** has two subtypes:
  - **Some**: Type-parametrized collection of one element
  - **None**: Empty collection

```
scala> var x: String = "Indeed"
x: String = Indeed
scala> var a = Option(x)
a: Option[String] = Some(Indeed)
scala> x = null
x: String = null
scala> var b = Option(x)
b: Option[String] = None
scala>//checks if Some a is defined?
scala>println(s"a is? ${a.isDefined}")
a is? true
scala> println(s"b is not? ${b.isEmpty}")
//checks if is None
b is not ? true
```

Option Collections (2/2)

- Prevent division by zero

```
scala> def divide(amt: Double, divisor: Double): Option[Double] = {
| if (divisor == 0) None
| else Option(amt / divisor)
| }
divide: (amt: Double, divisor: Double)Option[Double]
scala> val legit = divide(5, 2)
legit: Option[Double] = Some(2.5)
scala> val illegit = divide(3, 0)
illegit: Option[Double] = None
```

Option Collections (2/2)

- Extracting values from Options:
  - **Option.get()**: Avoid it If you call get on an None instance “no such element” error will be trigger
- Safe Option extractions:

Name	Example	Description
getOrElse	`nextOption getOrElse 5` or `nextOption getOrElse { println("error!"); -1 }`	Returns the value for Some or else the result of a by-name parameter (see “By-Name Parameters” on page 75) for None.
orElse	`nextOption orElse nextOption`	Doesn’t actually extract the value, but attempts to fill in a value for None. Returns this Option if it is nonempty, otherwise returns an Option from the
Match expressions	`nextOption match { case Some(x) => x; case None => -1 }`	Use a match expression to handle the value if present. The Some(x) expression extracts its data into the named value “x”, which can be used as the return value of the match expression or reused for further transformation.

Try Collections (1/2)

- The **util.Try** collection turns error handling into collection management.
- It provides a mechanism to catch errors that occur in a given function parameter, returning either the error or the result of the function if successful.
- Scala supports try {} .. catch {} blocks but util.Try () is the functional way.
- To throw an exception:

```
scala> def loopAndFail(end: Int, failAt: Int): Int = {
| for (i <- 1 to end) {
| println(s"$i) ")
| if (i == failAt) throw new Exception("Too many iter")
| }
| end
| }
scala> loopAndFail(10, 3)
1)
2)
3)
java.lang.Exception: Too many iter
```

Try Collections (2/2)

- **Util.Try** is unimplemented but has **two implemented subtypes**
- **Success type** contains the return value of the attempted expression if no exception was thrown
- **Failure type** contains the thrown Exception

```
scala> val t1 = util.Try( loopAndFail(2, 3) )
1)
2)
t1: scala.util.Try[Int] = Success(2)
scala> val t2 = util.Try{ loopAndFail(4, 2) }
1)
2)
t2: scala.util.Try[Int] = Failure(java.lang.Exception: Too many iter)
```

Try Collections – Handling Errors

```
scala> def nextError = util.Try{ 1 / util.Random.nextInt(2) }
nextError: scala.util.Try[Int]
scala> val x = nextError
x: scala.util.Try[Int] = Failure(
java.lang.ArithmeticException: / by zero)
scala> val y = nextError
y: scala.util.Try[Int] = Success(1)
```

Try Collections – Other Examples

```
scala> val input = " 123 "
input: String = " 123 "
scala> val result = util.Try(input.toInt) orElse
 util.Try (input.trim.toInt)
result: scala.util.Try[Int] = Success(123)
scala> result foreach { r => println(s"Parsed '$input' to $r!") }
Parsed ' 123 ' to 123!
scala> val x = result match {
| case util.Success(x) => Some(x)
| case util.Failure(ex) => {
| println(s"Couldn't parse input '$input'")
| None
| }
| }
x: Option[Int] = Some(123)
```

Try Collections – Table

Name	Example	Description
`foreach`	`nextError foreach(x => println("success!" + x))`	Executes the given function once in case of `Success`, or not at all in case of a `Failure`.
`getOrElse`	`nextError getOrElse 0`	Returns the embedded value in the `Success` or the result of a by-name parameter in case of a `Failure`.
`orElse`	`nextError orElse nextError`	The opposite of `flatMap`. In case of `Failure`, invokes a function that also returns a `util.Try`. With `orElse` you can potentially turn a `Failure` into a `Success`.
`toOption`	`nextError.toOption`	Convert your `util.Try` to `Option`, where a `Success` becomes `Some` and a `Failure` becomes `None`. Useful if you are more comfortable working with options, but the downside is you may lose the embedded `Exception`.
`map`	`nextError map (_ * 2)`	In case of `Success`, invokes a function that maps the embedded value to a new value.
Match expressions	`nextError match { case util.Success(x) => x; case util.Failure(error) => -1 }`	Use a match expression to handle a `Success` with a return value (stored in "x") or a `Failure` with an exception (stored in "error"). Not shown: logging the error with a good logging framework, ensuring it gets noticed and tracked.
Do nothing	`nextError`	This is the easiest error-handling method of all and a personal favorite of mine. To use this method, simply allow the exception to propagate up the call stack until it gets caught or causes the current application to exit. This method may be too disruptive for certain sensitive cases, but ensures that thrown exceptions will never be ignored.

Future Collections (1/2)

- Concurrent.Future **Initiates a background task**
- Like Option & Try **safe way to chain** additional operations or extract value
- But may **be not immediately available**. The background task launched could still be working
- It is necessary **to specify the "context"** in the current session or application for running functions concurrently. We'll use the default "global" context, which makes use of Java's thread library

```
scala> val f = concurrent.Future { Thread.sleep(5000); println("hi") }
f: scala.concurrent.Future[Unit] =
scala.concurrent.impl.Promise$DefaultPromise@4aa3d36
scala> println("waiting")
waiting
scala> hi
```

- You can **chain a function or another future** to be executed following the completion of a future and eventually will return util.Try with the result or an exception
- **BETTER USE AKKA FRAMEWORK**

Future Collections (2/3)

- Operations for chaining futures and setting callback functions.

```
scala> import concurrent.ExecutionContext.Implicits.global
import concurrent.ExecutionContext.Implicits.global
scala> import concurrent.Future
import concurrent.Future
scala> def nextFtr(i: Int = 0) = Future {
| def rand(x: Int) = util.Random.nextInt(x)
|
| Thread.sleep(rand(5000))
| if (rand(3) > 0) (i + 1) else throw new Excep
| }
nextFtr: (i: Int)scala.concurrent.Future[Int]
```

Name	Example	Description
fallbackTo	nextFtr(1) fallbackTo nextFtr(2)	Chains the second future to the first and returns a new overall future. If the first is unsuccessful, the second is invoked.
flatMap	nextFtr(1) flatMap nextFtr()	Chains the second future to the first and returns a new overall future. If the first is successful, its return value will be used to invoke the second.
map	nextFtr(1) map (_ * 2)	Chains the given function to the future and returns a new overall future. If the future is successful, its return value will be used to invoke the function.
onComplete	nextFtr() onComplete { _ getOrElse 0 }	After the future's task completes, the given function will be invoked with a `util.Try` containing a value (if success) or an exception (if failure).
onFailure	nextFtr() onFailure { case _ => "Error!" }	If the future's task throws an exception, the given function will be invoked with that exception.
onSuccess	nextFtr() onSuccess { case x => s"Got $x" }	If the future's task completes successfully, the given function will be invoked with the return value.
Future.sequence	concurrent.Future se quence List(nextFtr(1), nextFtr(5))	Runs the futures in the given sequence concurrently, returning a new future. If all futures in the sequence are successful, a list of their return values will be returned. Otherwise the first exception that occurs across the futures will be returned.

Future Collections (3/3)

Name	Example	Description
`fallbackTo`	`nextFtr(1) fallbackTo nextFtr(2)`	Chains the second future to the first and returns a new overall future. If the first is unsuccessful, the second is invoked.
`flatMap`	`nextFtr(1) flatMap nextFtr()`	Chains the second future to the first and returns a new overall future. If the first is successful, its return value will be used to invoke the second.
`map`	`nextFtr(1) map (_ * 2)`	Chains the given function to the future and returns a new overall future. If the future is successful, its return value will be used to invoke the function.
`onComplete`	`nextFtr() onComplete { _ getOrElse 0 }`	After the future's task completes, the given function will be invoked with a `util.Try` containing a value (if success) or an exception (if failure).
`onFailure`	`nextFtr() onFailure { case _ => "Error!" }`	If the future's task throws an exception, the given function will be invoked with that exception.
`onSuccess`	`nextFtr() onSuccess { case x => s"Got $x" }`	If the future's task completes successfully, the given function will be invoked with the return value.
`Future.sequence`	`concurrent.Future sequence List(nextFtr(1), nextFtr(5))`	Runs the futures in the given sequence concurrently, returning a new future. If all futures in the sequence are successful, a list of their return values will be returned. Otherwise the first exception that occurs across the futures will be returned.

Exercises

- **Exercise 7 – More Collections**

INTRODUCTION TO SCALA

9. Classes

OOScala - Classes

- *Classes* are the core building block of object-oriented languages, a combination of data structures with functions (“methods”)

Field of a class

n parameter class just for initialization fields, not can be used in methods

Method of a class

Override default JVM toString

```
scala> class User(n: String) {
| val name: String = n
| def greet: String = s"Hello from $name"
| override def toString = s"User($name)"
| }
defined class User
scala> val u = new User("Zeniba")
u: User = User(Zeniba)
scala> println(u.greet)
Hello from Zeniba
```

OOScala - Classes

```
scala> class User(val name: String) {
| def greet: String = s"Hello from $name"
| override def toString = s"User($name)"
| }
defined class User
scala> val users = List(new User("Shoto"), new User("Art3mis"),
new User("Aesch"))
users: List[User] = List(User(Shoto), User(Art3mis), User(Aesch))
scala> val sizes = users map (_.name.size)
sizes: List[Int] = List(8, 7, 5)
scala> val third = users find (_.name contains "3")
third: Option[User] = Some(User(Art3mis))
scala> val greet = third map (_.greet) getOrElse "hi"
greet: String = Hello from Art3mis
```

By adding the keywords **val** or **var** before a class **param** becomes a field in the class

Classes Inheritance & Polymorphism

- **Inheritance**: Like in Java we have extends, this & super

```
scala> class A {
| def hi = "Hello from A"
| override def toString = getClass.getName
| }
defined class A
scala> class B extends A
defined class B
scala> class C extends B { override def hi = "hi C -> " + super.hi }
defined class C
scala> val hiA = new A().hi
hiA: String = Hello from A
scala> val hiB = new B().hi
hiB: String = Hello from A
scala> val hiC = new C().hi
hiC: String = hi C -> Hello from A
```

Classes Inheritance & Polymorphism

- Polymorphism

```
scala> val a: A = new A
a: A = A
scala> val a: A = new B
a: A = B
scala> val b: B = new A
<console>:9: error: type mismatch;
found : A
required: B
val b: B = new A
^
scala> val b: B = new B
b: B = B
```

- An instance of a subclass can be used in place of an instance of its parent class, but not the inverse

- Inference of common type:

```
scala> val misc = List(new C, new A, new B)
misc: List[A] = List(C, A, B)
scala> val messages = misc.map(_.hi).distinct.sorted
messages: List[String] = List(Hello from A, hi C -> Hello from A)
```

Class Definition (1 / 3)

- With **val** and **var** fields as parameters

```
scala> class Car(val make: String, var reserved: Boolean) {
| def reserve(r: Boolean): Unit = { reserved = r }
| }
defined class Car
scala> val t = new Car("Toyota", false)
t: Car = Car@4eb48298
scala> t.reserve(true)
scala> println(s"My ${t.make} is now reserved?
${t.reserved}")
My Toyota is now reserved? True
```

Class Definition (2/3)

- Subclass definition

```
scala> class Lotus(val color: String, reserved: Boolean) extends
Car("Lotus", reserved)
defined class Lotus
scala> val l = new Lotus("Silver", false)
l: Lotus = Lotus@52c46334
scala> println(s"Requested a ${l.color} ${l.make}")
Requested a Silver Lotus
```

Class Definition (3/3)

- Default Values

```
scala> class Car(val make: String, var reserved: Boolean = true,
| val year: Int = 2015) {
| override def toString = s"$year $make, reserved = $reserved"
| }
defined class Car
scala> val a = new Car("Acura")
a: Car = 2015 Acura, reserved = true
scala> val l = new Car("Lexus", year = 2010)
l: Car = 2010 Lexus, reserved = true
scala> val p = new Car(reserved = false, make = "Porsche")
p: Car = 2015 Porsche, reserved = false
```

Classes with Type Parameters

- Let's create our own collection and use a type parameter to ensure type safety. The new collection will extend Traversable[A], the parent class of Iterable

```
scala> class Singular[A](element: A) extends Traversable[A] {
| def foreach[B](f: A => B) = f(element)
| }
defined class Singular
scala> val p = new Singular("Planes")
p: Singular[String] = (Planes)
scala> p foreach println
Planes
scala> val name: String = p.head
name: String = Planes
```

passing a type parameter to the parent class in the class definition

By defining a foreach() operation, Traversable will ensure our class is a real collection

Here is a validation of our type-parameterized class

An example usage of the foreach method we defined

Another example usage of foreach, indirectly this time, as we access Traversable.head, which invokes foreach for us. By extending Traversable we can access head and a range of other standard collection operations

Abstract classes

- An *abstract* class is a class designed to be extended by other classes but not instantiated itself

```
scala> abstract class Car {
| val year: Int
| val automatic: Boolean = true
| def color: String
| }
defined class Car
```

- First Implementation

```
scala> class RedMini(val year: Int) extends Car {
| def color = "Red"
| }
defined class RedMini
scala> val m: Car = new RedMini(2005)
```

- Best implementation

```
scala> class Mini(val year: Int, val color: String) extends Car
defined class Mini
scala> val redMini: Car = new Mini(2005, "Red")
redMini: Car = Mini@1f4dd016
scala> println(s"Got a ${redMini.color} Mini")
Got a Red Mini
```

you can implement a required method using a
Value if the method is parentheses and parameter-free

Anonymous Classes

- Javascript style:

```
scala> abstract class Listener { def trigger }
defined class Listener
scala> class Listening {
| var listener: Listener = null
| def register(l: Listener) { listener = l }
| def sendNotification() { listener.trigger }
| }
defined class Listening
scala> val notification = new Listening()
notification: Listening = Listening@66596c4c
scala> notification.register(new Listener {
| def trigger { println(s"Trigger at ${new java.util.Date}") }
| })
scala> notification.sendNotification
Trigger at Fri Jan 24 13:15:32 PDT 2014
```

Field and Method Types

- Overloaded Methods

```
scala> class Printer(msg: String) {
| def print(s: String): Unit = println(s"$msg: $s")
| def print(l: Seq[String]): Unit = print(l.mkString(", "))
| }
defined class Printer
scala> new Printer("Today's Report").print("Foggy" :: "Rainy" :: Nil)
Today's Report: Foggy, Rainy, Hot
```

Field and Method Types - Apply Method

- **Apply Method** can be invoked without the method name.
- It's the default method of a class (invoked with ())

```
scala> class Multiplier(factor: Int) {
| def apply(input: Int) = input * factor
| }
defined class Multiplier
scala> val tripleMe = new Multiplier(3)
tripleMe: Multiplier = Multiplier@339cde4b
scala> val tripled = tripleMe.apply(10)
tripled: Int = 30
scala> val tripled2 = tripleMe(10)
tripled2: Int = 30
```

- **In lists the syntax for retrieving an element by its index uses the List.apply method**

```
scala> val l = List('a', 'b', 'c')
l: List[Char] = List(a, b, c)
scala> val character = l(1)
character: Char = b
```

Field and Method Types – Lazy Values

- **Lazy value** expression is executed when the value is invoked, but **only the very first time**

```
scala> class RandomPoint {
| val x = { println("creating x"); util.Random.nextInt }
| lazy val y = { println("now y"); util.Random.nextInt }
| }
defined class RandomPoint
scala> val p = new RandomPoint()
creating x
p: RandomPoint = RandomPoint@6c225adb
scala> println(s"Location is ${p.x}, ${p.y}")
now y
Location is 2019268581, -806862774
scala> println(s"Location is ${p.x}, ${p.y}")
Location is 2019268581, -806862774
```

Packaging (1/2)

- System for code organization like Java

```
package com.netflix.utilities

// stores classes under com/netflis/utilities directory
```

- Accessing Packaged classes:
  - Full qualified name:

```
scala> val d = new java.util.Date
```

  - Import them (from anywhere of code)

```
scala> import java.util.Date
import java.util.Date
scala> val d = new Date
```

Packaging (2/2)

- Accessing Packaged classes:
  - Import the entire contents of a package with _

```
scala> import collection.mutable._
import collection.mutable._
scala> val b = new ArrayBuffer[String]
b: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer()
```

  - To avoid importing full package you can use groups

```
scala> import collection.mutable.{Queue,ArrayBuffer}
import collection.mutable.{Queue, ArrayBuffer}
scala> val q = new Queue[Int]
```

  - Import Alias (to avoid name conflicts)

```
scala> import collection.mutable.{Map=>MutMap}
import collection.mutable.{Map=>MutMap}
scala> val m1 = Map(1 -> 2)
m1: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2)
scala> val m2 = MutMap(2 -> 3)
m2: scala.collection.mutable.Map[Int,Int] = Map(2 -> 3)
```

Privacy Controls

- By default, Scala **does not add privacy controls**. Any class you write will be instantiable and its fields and methods accessible by any other code
- To **add privacy controls**, such **as mutable state** that should only be handled inside the class:
  - **protected**: limits the access of fields and methods to the same class or its subclasses

```
scala> class User { protected val passwd = util.Random.nextString(10) }
defined class User
scala> class ValidUser extends User { def isValid = ! passwd.isEmpty }
defined class ValidUser
scala> val isValid = new ValidUser().isValid
isValid: Boolean = true
```

  - **private**: limits the access of fields and methods to the same class which they are defined

```
scala> class User(private var password: String) {
| def update(p: String) {
| println("Modifying the password!")
| password = p
| }
| def validate(p: String) = p == password
| }
defined class User
```

Final and Sealed Classes

- The protected and private access controls and their modifiers can limit access to a class or its members overall or based on location. However, they lack the ability to restrict creating subclasses.
- ***Final* class** members can never be overridden in subclasses. Marking a value, variable, or method with the **final** keyword ensures that the implementation is the one that all.
- Entire classes can be marked as **final** as well, preventing any possible subclasses of that class
- **Sealed** classes restrict the subclasses of a class **to being located in the same file** as the parent class. By sealing a class, you can write code that makes safe assumptions about its hierarchy. Classes are sealed by prefixing the class definition and class keyword with the **sealed** keyword.

Exercises

- **Exercise 8 – Classes**

INTRODUCTION TO SCALA

10. Objects, Case Classes & Traits

Objects (1/4)

- A type of class with just one instance (singleton)
- Until it is accessed the first time it won't get instantiated
- An object can extend another class, making its fields and methods available in a global instance. The reverse is not true
- Objects can be used to provide pure functions (only depending on the input) as utilities

--object

```
scala>object Hello {println("Hello");def hi = "hi" }
defined object Hello
scala> println(Hello.hi)
Hello
hi
scala> println(Hello.hi)
Hi
```

– pure functions

```
scala> object HtmlUtils {
| def removeMarkup(input: String) = {
| input
| .replaceAll("""</?\w[^>]*>""","")
| .replaceAll("<.*>","")
| }
| }
scala> val html = "<html>Intro</html>"
scala> val text = HtmlUtils.removeMarkup(html)
text: String = Introduction
```

Objects (2/4)

- An Object can not take class parameters but it can use:
  - **apply()** method: Makes it posible to invoke an object by name like List (1,2,3) wich is really an object.
- List object has an apply() that takes arguments and returns a new collection from them.
- This is known a **factory pattern**. A popular way to generate new instances of class from its **companion object**
- A **companion object** is an object that shares the same name as a class and is defined together in the same file as the class
- **Companion objects** and classes are considered a **single unit in terms of access controls**, so they can access each other's private and protected fields and methods.

Objects (3/4)

```
scala> class Multiplier(val x: Int) {
| def product(y: Int) = x * y
|}
scala> object Multiplier { def apply(x: Int)
| = new Multiplier(x) }
// Exiting paste mode, now interpreting.
defined class Multiplier
defined object Multiplier
scala> val tripler = Multiplier(3)
tripler: Multiplier = Multiplier@5af28b27
scala> val result = tripler.product(13)
result: Int = 39
```

Same class without companion object

```
scala> class Multiplier(factor: Int) {
| def apply(input: Int) = input * factor
| }
defined class Multiplier
scala> val tripleMe = new Multiplier(3)
tripleMe: Multiplier = Multiplier@339cde4b
scala> val tripled = tripleMe.apply(10)
tripled: Int = 30
scala> val tripled2 = tripleMe(10)
tripled2: Int = 30
```

Objects (4/4)

- Special access controls that companion object shares with its companion class

```
object DBConnection {
   private val db_url = "jdbc://localhost"
   private val db_user = "franken"
   private val db_pass = "berry"
   def apply() = new DBConnection
}
class DBConnection {
   private val props = Map(
         "url" -> DBConnection.db_url,
         "user" -> DBConnection.db_user,
         "pass" -> DBConnection.db_pass
   )
   println(s"Created new connection for " + props("url"))
}
scala> val conn = DBConnection()
Created new connection for jdbc://localhost
conn: DBConnection = DBConnection@4d27d9d
```

Command-Line Applications with Objects

- Special **main** method in Object as the entry point for the application:

```
$ cat Cat.scala
object Cat {
def main(args: Array[String]) {
for (arg <- args) {
println( io.Source.fromFile(arg).mkString )
}
}
}
$ scalac Cat.scala
$ scala Cat Cat.scala
```

Case Classes (1/3)

- A **case class** is an instantiable class that includes **several automatically generated methods**.
- It also includes an automatically generated companion object with its own automatically generated methods.
- Include **equals** implementation based on the fields
- A pretty print **toString**
- Used for storing data (pojos)
- By default, case classes convert parameters to value fields so it isn't necessary to prefix them with the val keyword. You can still use the var keyword if you need a variable field.

Case Classes (2/3)

- Automatically generated case class methods

Name	Location	Description
`apply`	Object	A factory method for instantiating the case class.
`copy`	Class	Returns a copy of the instance with any requested changes. The parameters are the class's fields with the default values set to the current field values.
`equals`	Class	Returns true if every field in another instance match every field in this instance. Also invocable by the operator ==.
`hashCode`	Class	Returns a hash code of the instance's fields, useful for hash-based collections.
`toString`	Class	Renders the class's name and fields to a `String`.
`unapply`	Object	Extracts the instance into a tuple of its fields, making it possible to use case class instances for pattern matching.

Case Classes (3/3)

- Use **case classes** over classes for data storage
- Use **object and traits** (see next) for writing functions over classes

Case Classes – Example

```
scala> case class Character(name: String, isThief: Boolean)
defined class Character
scala> val h = Character("Hadrian", true)
h: Character = Character(Hadrian,true)
scala> val r = h.copy(name = "Royce")
r: Character = Character(Royce,true)
scala> h == r
res0: Boolean = false
scala> h match {
| case Character(x, true) => s"$x is a thief"
| case Character(x, false) => s"$x is not a thief"
| }
res1: String = Hadrian is a thief
```

Here the **object's factory method** Character.apply()

The second instance shares the same value for the second field, so we only specify a new value for the first field

If both are non-null, the == operator triggers an instance's equals

The companion object's unapply method allows us to decompose the instance into its parts, binding the first field and using a literal value to match

Traits (1/2)

- Trait is a kind of class for multiple inheritance
- Classes, case classes, and objects can extend just for one class but **traits** can extend multiple traits at the same time
- **Traits** cannot be instantiated and take class parameters
- **Traits** can take type parameters to make them very reusable

```
scala> trait HtmlUtils {
| def removeMarkup(input: String) = {
| input
| .replaceAll("""</?\w[^>]*>""","")
| .replaceAll("<.*>","")
| }
| }
defined trait HtmlUtils
scala> class Page(val s: String) extends
HtmlUtils {
| def asPlainText = removeMarkup(s)
| }
defined class Page
scala> new
Page("<html>Intro</html>").asPlainText
res2: String = Introduction
```

Traits (2/2)

- If you are extending a class and one or more traits, you will need to extend the class before you can add the traits using the **with** keyword. A parent class, if specified, must always come before any parent traits

```
scala> trait SafeStringUtils {
| // Returns a trimmed version of the string wrapped in an Option,
| // or None if the trimmed string is empty.
| def trimToNone(s: String): Option[String] = {
| Option(s) map(_.trim) filterNot(_.isEmpty)
| }
| }
defined trait SafeStringUtils
scala> class Page(val s: String) extends SafeStringUtils with HtmlUtils {
| def asPlainText: String = {
| trimToNone(s) map removeMarkup getOrElse "n/a"
| }
| }
defined class Page
scala> new Page("<html><body><h1>Introduction</h1></body></html>").asPlainText
res3: String = Introduction
scala> new Page(" ").asPlainText
res4: String = n/a
scala> new Page(null).asPlainText
res5: String = n/a
```

Linearization - Traits

- Linearization:

```
scala> trait Base { override def toString = "Base" }
defined trait Base
scala> class A extends Base { override def toString = "A->" + super.toString }
defined class A
scala> trait B extends Base { override def toString = "B->" + super.toString }
defined trait B
scala> trait C extends Base { override def toString = "C->" + super.toString }
defined trait C
scala> class D extends A with B with C { override def toString = "D->" +
super.toString }
defined class D
scala> new D()
res50: D = D->C->B->A->Base
```

Self Types in Traits (1 / 2)

- **self type** is a trait annotation that asserts that the trait must be mixed in with a specific type, or its subtype, when it is added to a class
- A popular use of self types is to add functionality with traits to classes that require input parameters

```
scala> class A { def hi = "hi" }
defined class A
scala> trait B { self: A =>
| override def toString = "B: " + hi
| }
defined trait B
scala> class C extends B
<console>:9: error: illegal inheritance;
self-type C does not conform to B's selftype B with A
class C extends B
scala> class C extends A with B
defined class C
scala> new C()
res1: C = B: hi
```

Our trait B has a self type, adding the requirement that the trait can only ever be mixed into a subtype of the specified type, the A class

… but just to prove it, let's try defining a class with trait B but without the requested class. No luck

This time, trait B is directly extending its requested type, A, so its self type requirement has been met

When our C class is instantiated, B.toString is invoked, which then invokes A.hi. The B trait is indeed used as a subtype of A here and can invoke one of its methods

Self Types in Traits (2 / 2)

- Benefits of **self types** (extend a class without specifying its input parameters):

```
scala> class TestSuite(suiteName: String) { def start() {} }
defined class TestSuite
scala> trait RandomSeeded { self: TestSuite =>
| def randomStart() {
| util.Random.setSeed(System.currentTimeMillis)
| self.start()
| }
| }
defined trait RandomSeeded
scala> class IdSpec extends TestSuite("ID Tests") with RandomSeeded {
| def testId() { println(util.Random.nextInt != 1) }
| override def start() { testId() }
|
| println("Starting...")
| randomStart()
| }
defined class IdSpec
```

Here is the base class, TestSuite, which takes an input parameter

Our trait needs to invoke TestSuite.start() but cannot extend TestSuite because it would require hardcoding the input parameter. By using a self type, the trait can expect to be **a subtype of TestSuite** without explicitly being declared as one

The test class IdSpec defines our self-typed trait as a subclass, allowing its randomStart() to be invocable

Instantiation with Traits (1 / 2)

- **Instantiation with Traits**: add the traits when the class is instantiated
- The traits must use **with** and no **extends.** Are the traits which are been extended with the classs

```
scala> class A
defined class A
scala> trait B { self: A => }
defined trait B
scala> val a = new A with B
a: A with B = $anon$1@26a7b76d
```

The instance's class is indeed anonymous, because it contains a combination of a class and trait that are not formally included in any named class definition

Instantiation with Traits (2 / 2)

- The real value in instantiating with traits is in adding new functionality or configurations to existing classes.
- This feature is commonly known as **dependency injection:**

```
scala> class User(val name: String) {
| def suffix = ""
| override def toString = s"$name$suffix"
| }
defined class User
scala> trait Attorney { self: User => override def suffix = “with attorney" }
defined trait Attorney

scala> val h = new User(“John")
h: User =  John
scala> val m = new User(“Mary") with Attorney
m: User with Attorney = Mary with Attorney
```

Import Instance Members

- The import keyword can also be used to import *members* of classes and objects into the current namespace

```
scala> case class Receipt(id: Int, amount: Double, who: String, title: String)
defined class Receipt
scala>
 val latteReceipt = Receipt(123, 4.12, "fred", "Medium Latte")
 import latteReceipt._
 println(s"Sold a $title for $amount to $who")
Sold a Medium Latte for 4.12 to fred
```

- Use of members of util.Random object

```
scala> import util.Random._
import util.Random._
scala> val letters = alphanumeric.take(20).toList.mkString
letters: String = MwDR3EyHa1cr0JqsP9Tf
scala> val numbers = shuffle(1 to 20)
numbers: scala.collection.immutable.IndexedSeq[Int] = Vector(5, 10, 18, 1,
16, 8, 20, 14, 19, 11, 17, 3, 15, 7, 4, 9, 6, 12, 13, 2)
```

Exercises

- **Exercise 9 - Object, Case Classes and Traits**

INTRODUCTION TO SCALA

11. Advanced Typing

Implicit parameters

- Invoke a function without specifying all of the parameters (not partially applied functions, not default parameters)

```
scala> object Doubly {
| def print(num: Double)(implicit fmt: String) = {
| println(fmt format num)
| }
| }
defined object Doubly
scala> Doubly.print(3.724)
<console>:9: error: could not find implicit value for parameter fmt: String
Doubly.print(3.724)
scala> Doubly.print(3.724)("%.1f") //using all parameters
3.7
scala> case class USD(amount: Double) {
| implicit val printFmt = "%.2f"  // define an implicit value in the caller
| def print = Doubly.print(amount)
| }
defined class USD
scala> new USD(81.924).print
81.92
```

Excessive use can make your code hard to read and understand

Implicit Classes

- Add new methods to a Class without modification
- Example add a method “fishes” to Class Int

```
object IntUtils {
  implicit class Fishies(val x: Int) {
      def fishes = "Fish" * x
  }
}

import IntUtils._
println(3.fishes)
```

Fishies, defined inside an object, implicitly converts integers to itself …

… so that the fishes() method will be defined for all integers.

Before using it, the implicit class must be added to the namespace

… and then the fishes() method can be invoked on any integer.

Types - Type alias

- Type alias: creates a new named type for a specific, existing type (or class).

```
scala> object TypeFun {
| type Whole = Int
| val x: Whole = 5
| type UserInfo = Tuple2[Int,String]
| val u: UserInfo = new UserInfo(123, "George")
| type T3[A,B,C] = Tuple3[A,B,C]
| val things = new T3(1, 'a', true)
| }
defined object TypeFun
scala> val x = TypeFun.x
x: TypeFun.Whole = 5
scala> val u = TypeFun.u
u: TypeFun.UserInfo = (123,George)
scala> val things = TypeFun.things
things: (Int, Char, Boolean) = (1,a,true)
```

Types - Abstract Types

- Used to create **type declarations** in abstract classes, which declare types that concrete (non abstract) subclasses must implement.

```
scala> class User(val name: String)
defined class User
scala> trait Factory { type A; def create: A }
defined trait Factory
scala> trait UserFactory extends Factory {
| type A = User
| def create = new User("")
| }
defined trait UserFactory
```

- The same using type paramenters

```
scala> trait Factory[A] { def create: A }
defined trait Factory
scala> trait UserFactory extends Factory[User] { def create = new User("") }
defined trait UserFactory
```

Types- Bounded Types (1 / 2)

- A **bounded type** is restricted to being either a specific class or else its subtype or base type
- An **upper bound** restricts a type to only that type or one of its subtypes
- A **lower bound** restricts a type to only that type or else one of the base types it extends.

Types- Bounded Types (2 / 2)

```
scala> class BaseUser(val name: String)
defined class BaseUser
scala> class Admin(name: String, val level: String) extends BaseUser(name)
defined class Admin
scala> class Customer(name: String) extends BaseUser(name)
defined class Customer
scala> class PreferredCustomer(name: String) extends Customer(name)
defined class PreferredCustomer
scala> def check[A <: BaseUser](u: A) { if (u.name.isEmpty) println("Fail!") }
check: [A <: BaseUser](u: A)Unit
scala> check(new Customer("Fred"))
scala> check(new Admin("", "strict"))
Fail!
scala> def recruit[A >: Customer](u: Customer): A = u match {
| case p: PreferredCustomer => new PreferredCustomer(u.name)
| case c: Customer => new Customer(u.name)
| }
recruit: [A >: Customer](u: Customer)A
scala> val customer = recruit(new Customer("Fred"))
customer: Customer = Customer@4746fb8c
scala> val preferred = recruit(new PreferredCustomer("George"))
preferred: Customer = PreferredCustomer@4cd8db31
```

Now we'll define a function that takes a parameter with an upper bound

And a function that takes a parameter with an lower bound

Types - Type Variance (1 / 2)

- Ading **upper** or **lower** bounds will make **type parameters** more restrictive,
- Ading **type variance** makes type parameters *less* restrictive
- **Type variance** specifies how a type parameter may adapt to meet a base type or subtype
- Type parameters are **invariant** by default, cannot adapt to alternate types even if they are compatible

Types - Type Variance

- Scala's polymorphism, allows lower types to be stored in values with higher types

```
scala> class Car { override def toString = "Car()" }
defined class Car
scala> class Volvo extends Car { override def toString = "Volvo()" }
defined class Volvo
scala> val c: Car = new Volvo()
c: Car = Volvo()
```

- The same polymorphic adaptation doesn't hold for type parameters

```
scala> case class Item[A](a: A) { def get: A = a }
defined class Item
scala> val c: Item[Car] = new Item[Volvo](new Volvo)
<console>:12: error: type mismatch;
found : Item[Volvo]
required: Item[Car]
Note: Volvo <: Car, but class Item is invariant in type A.
You may wish to define A as +A instead. (SLS 4.5)
val c: Item[Car] = new Item[Volvo](new Volvo)
```

Types – Covariant type

- **Covariant** type parameters can automatically morph into one of their base types when necessary

```
scala> case class Item[+A](a: A) { def get: A = a }
defined class Item
scala> val c: Item[Car] = new Item[Volvo](new Volvo)
c: Item[Car] = Item(Volvo())
scala> val auto = c.get
auto: Car = Volvo()
```

Types – Contravariant type

- **Covariant** is not always applicable. An input parameter to a method cannot be covariant. it would be bound to a subtype but be invokable with a base type

```
scala> class Check[+A] { def check(a: A) = {} }
<console>:7: error: covariant type A occurs in contravariant
position in
type A of value a
class Check[+A] { def check(a: A) = {} }
```

- ***Contravariance*** is where a type parameter may morph into a subtype
- They can be used for input parameters to methods but not as their return types. Return types are covariant, because their result may be a subtype that is polymorphically converted to a base type

```
scala> class Check[-A] { def check(a: A) = {} }
defined class Check
```

Types – covariant / contravariant

- Example

```
scala> class Car; class Volvo extends Car; class VolvoWagon extends Volvo
defined class Car
defined class Volvo
defined class VolvoWagon
scala> class Item[+A](a: A) { def get: A = a }
defined class Item
scala> class Check[-A] { def check(a: A) = {} }
defined class Check
scala> def item(v: Item[Volvo]) { val c: Car = v.get }
item: (v: Item[Volvo])Unit
scala> def check(v: Check[Volvo]) { v.check(new VolvoWagon()) }
check: (v: Check[Volvo])Unit
```

Thank you

LinkedIn
https://es.linkedin.com/in/chicochica10