Intro to Scala: Fear No More

Sujan Kapadia

Philly Tech Week 2016

Workshop Goals

- Answer the question: What is Scala?
- Introduce you to basic programming constructs
- View **functions** as basic building blocks of behavior
- Hopefully dispel ***some*** of the fear around Scala
- Prepare you to continue further study of Scala (if you so desire)

What this won't cover

- In-depth functional programming
- Advanced type system (generics, variance, bounds)
- OO aspects of Scala (Classes, Traits, etc.)
- A **lot** of other stuff! Scala is a rich language...

Workshop Tasks

- Small exercises to explore the basics of Scala
- Parsing a CSV file (**comma separated values**)
- Representing the data from the CSV records
- Performing simple operations on this collection of data
- The source of data will be my Amazon purchase history since 2006
- If you have an Amazon account, you can also run a CSV report from here: http://www.amazon.com/gp/b2b/reports

What?

- **Statically typed** language that runs on the **JVM**
- Provides both ***functional*** and ***object-oriented*** constructs

Huh?

- **Type:** Something that takes on a specific range of values
- Examples:
  - Color: Red, Blue, Green
  - Date (some representation of time)
  - Integer (has a minimum and maximum value)
  - Types have **names**

Static Vs Dynamic Typing

- **Static Typing:**
  - Types must be specified before your program is compiled.
  - **You** provide the types *or* the **compiler** must be able to figure them out.
  - Types are **enforced**: You cannot assign a name to an integer or pass in "Monday" where a date is expected
  - If you try to assign a value of a different type, the compiler will return an error.

Static Vs Dynamic Typing

- **Dynamic Typing:**
  - Types may not be known until the program is actually running.
  - You don't have to specify types ahead of time.
  - A variable could be assigned a number, and later on be changed to a Date or a Color
  - Allows for a lot of flexibility, but can be easy to introduce errors and not know until you run your program.

JVM

- **Java Virtual Machine**
- Write Java, Scala, Groovy, Clojure, etc.
- Compilation produces ***platform independent bytecode***
- Bytecode is an intermediate representation that can be interpreted by the JVM
- The JVM will interpret and optimize and compile this to the target platform it runs on

Who, When, Where, Why?

- **Who:** Martin Odersky
- **When:** Publicly released in 2004
- **Where**: EPFL
- **Why:** Provide a JVM based language that supports functional programming and a rich, expressive type system

How?

- **Scala Compiler:** scalac
- **Scala REPL:** Read Eval Print Loop
- **SBT**: Build Tool / Dependency Management
- **IDE:** IntelliJ, Eclipse, etc.

> **IntelliJ Worksheet**: Built-in interactive environment to explore Scala expressions

The Basics

Interactive Mode

- Make sure “Interactive Mode” is checked.
- This causes any code on the left side to be immediately compiled whenever changes are made.
- Output is shown on the right side.

Type

$$\text{res0: Int} = 42$$

- **Int:** Integer type, 32 bit value
- The number 42 is **evaluated**
- Since we are not assigning this a name, the result is *automatically stored* in **res0**
  - In a REPL "session", one could use **res0** afterwards. Since the worksheet is recompiled every time, we can't here

Type

$$\text{res0: Int} = 42$$

- The identifier is followed by a colon and a type name
- Types are specified ***after an identifier!***
- In this case, the type was automatically **inferred**

Exercise 1: Fahrenheit to Celsius

- Does anyone know the F to C conversion off the top of their head?

Exercise 1: Fahrenheit to Celsius

- Does anyone know the F to C conversion off the top of their head?
- Formula:

$$C = (F - 32) * 5/9$$

Exercise 1: Fahrenheit to Celsius

- Does anyone know the F to C conversion off the top of their head?
- Formula:

$$C = (F - 32) * 5/9$$

- Let's enter the right hand side in the worksheet!
- My phone tells me the current temp is...

Exercise 1: Fahrenheit to Celsius

- The compiler evaluated an **expression**
- Without the parentheses, 32 * 5/9 would have been evaluated first (precedence)
- Scala is an ***expression-oriented language***. All things evaluate to a value

Exercise 1: Fahrenheit to Celsius

- But hey that shouldn't be an integer! My calculator tells me **it should be 11.111111**
- What gives?
- In this expression, every part is an **Int**, so Scala assumes you want an integer and happily complies.
- We'll revisit this in more detail shortly.

How do we deal with decimals?

- Let's type in a number like **5.9**

How do we deal with decimals?

- Let's type in a number like **5.9**
- **res2: Double = 5.9**
- This time, Scala evaluated the expression as a Double
- **Double** = 64-bit Floating Point Number
- There's also another type, **Float**: 32-bit Floating Point Number

Numeric Literals

- Temperature could easily fit in a Float, but a Double is automatically chosen
- A *literal* just means you're directly including the value
- In literal expressions, we can tell Scala what kind of number we want:
  - Float: append **f** or **F**
  - Double: append **d** or **D**
  - Long (64 bit integer): append **l** or **L**

Exercise 1 Revisited

```
(52f - 32) * 5/9
(52 -32) * 5f/9
```

- res5: Float = 11.111111
- *Why does that work?*
- It turns out that these numbers have **operations** attached to them.
- The numbers are **objects** themselves.
- Type an Int in, then press Ctrl + Space
- Scroll down to the arithmetic operators

Exercise 1 Revisited

```
(52f - 32) * 5/9
(52 -32) * 5f/9
```

- Do the same for Float
- For **Int**, + and / take **Int** arguments and *return* **Int**
- For **Float**, + and / take **Float** arguments and *return* **Float**

> Numbers are objects and you can invoke useful functions on them!

Text

- Type in “Hello Philadelphia”
- The String type represents a sequence of (Unicode) characters
- Strings are surrounded by double quotes
- A single character can be represented by using single quotes: ‘c’
- Strings have many operations: startsWith, endsWith, length, toUppercase, etc.

Exercise 2

- What is the length of “Hello Philadelphia” ?
- In two steps, convert “Hello Philadelphia” to “Yo Philly”.
- Then add a “!” to the end of it
- Then upper case it.

Exercise 2

```
"Hello Philadelphia".replaceAll("Hello", "Yo")
  .replaceAll("Philadelphia", "Philly")
  .concat("!") // Shouting
  .toUpperCase
```

- Operations can be invoked after a .
- Invocations can be ***chained*** together
- Strings also implement the + operator.
- Can you explain the difference between + and concat?
- Single line comments start with //

Values

- It would be painful to have to repeat literals or not refer to them by name.
- Scala allows us to assign an expression to a **val**

```
val productTitle: String = "Code: The
Hidden Language"

val productQuantity: Int = 1
```

val keyword + identifier + colon + Type = expression

Exercise 3a: Values

- Assign an Int to a value defined as String
- Assign a String to a value defined as Int
- What happens? Why?

Exercise 3a: Values

- Assign an Int to a value defined as String
- Assign a String to a value defined as Int
- What happens? Why?

Types are enforced.

Exercise 3b: Values

- Convert the Int value to a String
- Convert the String to an Int
- **Hint: Look at the available operations!**
- What happens if you try to convert `"42abc"` to an Int?

Exercise 3b: Values

- Convert the Int value to a String: `5.toString`
- Convert the String to an Int: `"42".toInt`
- **Hint: Look at the available operations!**
- What happens if you try to convert "42abc" to an Int?
  - `java.lang.NumberFormatException`

Exercise 3c: Reassign the value

```
val priceOfItem1 = 5.75f
val priceOfItem2 = 12.50f
val totalPrice = priceOfItem1
```

- Add priceOfItem2 to totalPrice and reassign to totalPrice
- What happens?

Exercise 3c: Reassign the value

```
val priceOfItem1 = 5.75f
val priceOfItem2 = 12.50f
val totalPrice = priceOfItem1
```

- Add priceOfItem2 to totalPrice and reassign to totalPrice
- What happens?

Values are immutable!

Exercise 3c: Reassign the value

```
val priceOfItem1 = 5.75f
val priceOfItem2 = 12.50f
val totalPrice = priceOfItem1
```

- Add priceOfItem2 to totalPrice and reassign to totalPrice
- What happens?

Aside: By convention, constants are capitalized in Scala:

```
val EarthGravity = 9.8f
```

Variables

- What if we want to capture a changing value?

```
var productQuantity = 1
productQuantity = productQuantity + 5
```

- **var** can be reassigned.

var keyword + identifier + colon + Type = expression

Why Values then?

- You want to return a value that cannot be changed
- You want to ensure that a value cannot get accidentally changed or corrupted by multiple users
- It's easier to reason about and test code when the system guarantees that a value cannot change.
- There are some cases where a **var** is useful - we will show an example later.

String Interpolation

- With **val**s, we can build more complex expressions now
- A common operation is to build a message to display to the user, or log information to a file.

```
val productCost = 12.50f
val message = s"This item costs ${productCost} US dollars"
```

- An *interpolated* string can reference expressions
- ${ **expression** }

String Interpolation

- More complex expressions can be used as well.

```
val priceOfItem1 = 5.75f
val priceOfItem2 = 12.50f
val message2 = s"The total cost of your
shopping cart is ${priceOfItem1 +
priceOfItem2}"
```

- How does it convert the expression to a String?
- All **objects** in Scala provide a **toString** operation
- We saw this earlier when converting an Int

Exercise 4

- Given a temperature in Fahrenheit, convert to Celsius and provide a friendly message indicating the current temperature.
- Given a price in US dollars and an exchange rate to Mexican pesos, provide a message indicating the cost in pesos.
  - Don't worry about rounding!

Conditionals and Booleans

- How do we make decisions based on expressions?

```
if (expression) {
  expressions
}
else if (expression) {
  expressions
}
else {
  expressions
}
```

Conditionals and Booleans

- We can perform comparisons with operators like the ones below:

> greater than	== equals
< less than	!= does not equal
>= greater than or equal to	&& logical AND
<= less than or equal to	|| logical OR

Exercise 5a: Branches

- For inventory:
  - Greater than 50: "In Stock"
  - Less than or equal to 50: "Less than 50 remaining"
  - Less than or equal to 10: "Only a few left!"
  - Equal to 0: "Out of stock"

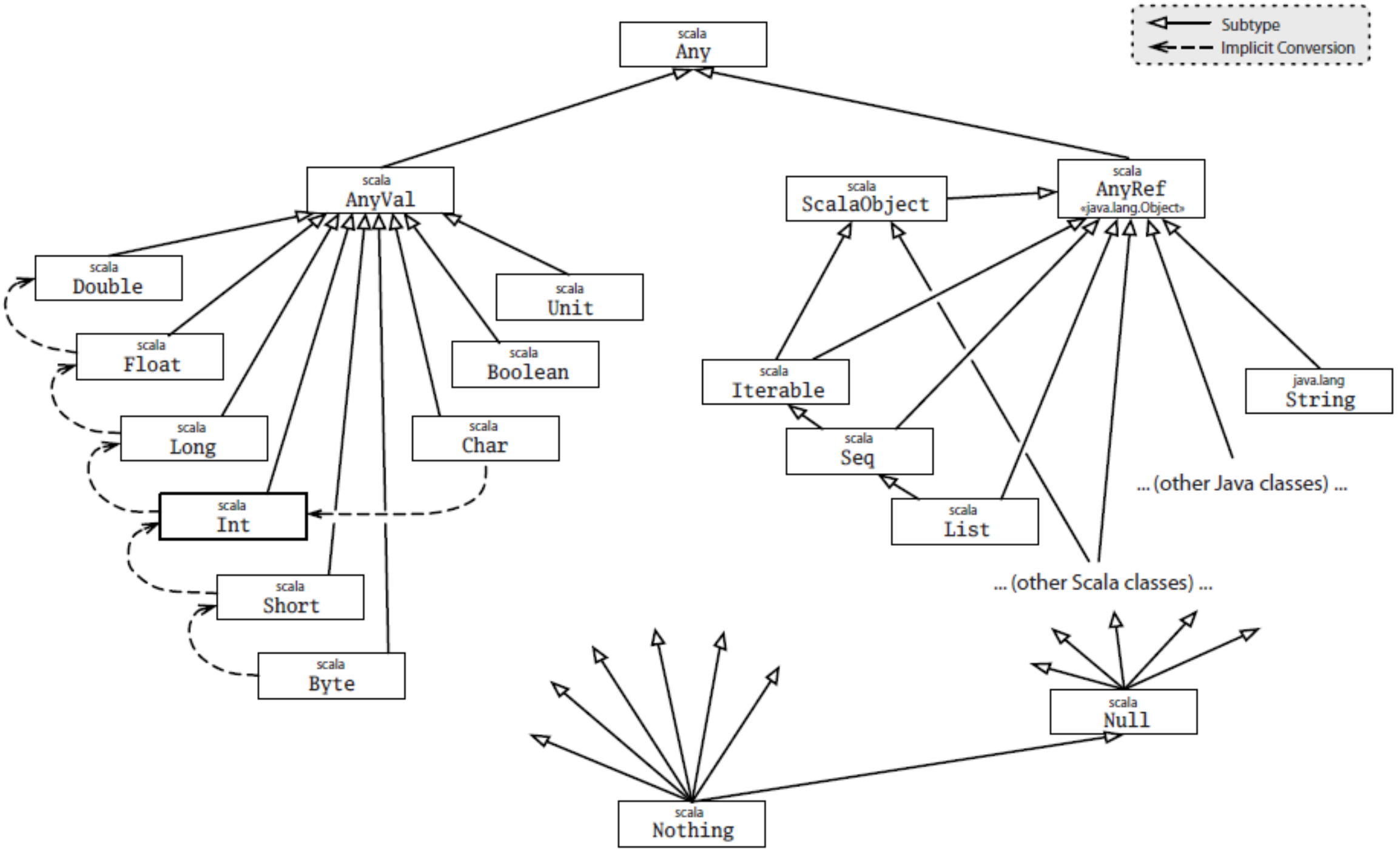
Exercise 5a: Branches

- What was the type of the result? Why?
- Wait, an **if** construct returns a result?
- An if construct is an **expression**

Exercise 5a: Branches

- What was the type of the result? **Any**
- Why?
  - **Any** is at the top of the *type hierarchy*
  - Since we did not provide an **else** branch, the compiler cannot determine the type in the case none of the branches are selected.
  - The most common type it can resort to is **Any**
- **Provide an else branch so String type can be chosen**

Aside: Type Hierarchy



Boolean expressions

- Boolean is a **true** or **false** value
- Example boolean expressions

```
val needMoreCoffee: Boolean = true
val currentTemp = 76
val isHot = (currentTemp >= 76 &&
currentTemp <= 100)
```

- The expression within an if statement must evaluate to a Boolean

Exercise 5b: String comparisons

- Example string: "Code: The Hidden Language of Computer Hardware and Software"
- Determine if the title has the word "Software" in it
- Determine if the title equals "code: the hidden Language of Computer Hardware and software"
- **Hint: Look up the operations on String**

Exercise 5b: String comparisons

- Example string: "Code: The Hidden Language of Computer Hardware and Software"
- Determine if the title has the word "Software" in it
  - **contains**
- Determine if the title equals "code: the hidden Language of Computer Hardware and software"
  - ==
  - **equals**
  - **equalsIgnoreCase**

Aside: Object Equality

- **equals**
  - Implemented by all objects
  - Can be customized *(overridden)*
- ==
  - Internally calls **equals**
  - Cannot be customized
- **Null Safe**
  - If a value is not assigned, it is *null*
  - Trying to perform operations on *null* result in a **NullPointerException**

Blocks

- The body of the if is called a **block**
- Blocks are surrounded by curly braces
- Blocks evaluate to a value
- They can contain one or more expressions
- The **last expression** in a block is what the block evaluates to

```
if (expression) {
  expressions
}
```

```
val result = {
  val x = 5
  "I'm a block"
}
```

Exercise 6a: CSV Parsing: Naive Version

- Goal: Split a string into its column values, separated by commas
- **Hint: Look through the operations on String**

Exercise 6a: CSV Parsing: Naive Version

- Goal: Split a string into its column values, separated by commas
- **Hint: Look through the operations on String**
- After you invoke the operation, what do you notice?
  - What is returned? What is the type?
  - Is the String split up as you would expect?

Exercise 6a: CSV Parsing: Naive Version

- After you invoke the operation, what do you notice?
  - What is returned? **Array[String]**
  - Is the String split up as you would expect? **No, one of the column values itself contains commas!**

Exercise 6b: CSV Parsing: Less Naive Version

- How can we solve the previous problem?
- This is a bit more complicated, so we'll work through this step by step.

Exercise 6b: CSV Parsing: Less Naive Version

Basic idea: Process the String one character at a time

1. If we are within double quotes:
    1. Process commas like any other character
    2. If we encounter a ““ (a quote followed by a quote), process that as one quote
    3. If we encounter a ”, we are now outside of double quotes

Exercise 6b: CSV Parsing: Less Naive Version

Basic idea:

Else:

1. If we encounter a “, we are within double quotes.
2. If we encounter a comma, take the characters for this column and build a String.
3. Otherwise, this character is part of the current column being processed.

Exercise 6b: CSV Parsing: While Loop

Process the String one character at a time:

```
while (boolean expression) {
  block
}
```

- When should the loop end?
- How do we express that boolean condition?
- How do we get the characters from a String?

Exercise 6b: CSV Parsing: Arrays

An **array** is a *mutable* collection of objects with a fixed length

Creation:

```
val temps: Array[Int] = Array(45, 32, 71)
val chars: Array[Char] = Array('s','c','a','l','a')

val length: Int = chars.length
val thirdChar: Char = chars[2]
```

Exercise 6b: CSV Parsing: Arrays

- Assign the character array to a value
- Get the length of the array
- For each iteration, get the character at that position
- Advance the position - how do we keep track of it? **val** or **var ?**

Exercise 6b: CSV Parsing: State

- Determine if the character is a quote and store the result in a **val**

Exercise 6b: CSV Parsing: State

- If the current character is a quote, we're in double quotes - use a Boolean to mark this.
  - Else keep track of the current character
- Check if we're already within double quotes:
  - If we're processing a quote, we're outside of quotes now, set the Boolean to false
- Else keep track of the current character

Exercise 6b: CSV Parsing: StringBuilder

How do we keep track of the current character?
What do we add it to?

StringBuilder allows you to build strings character by character, and then return one String

- append(x: Char): StringBuilder
- toString

Exercise 6b: CSV Parsing: State

- If the current character is a quote, we're in double quotes - use a Boolean to mark this.
  - Else **if the current character is a comma**, turn the StringBuilder into a String and add the column to a *collection*.
  - Else keep track of the current character
- Check if we're already within double quotes:
  - If we're processing a quote, we're outside of quotes now, set the Boolean to false
  - Else keep track of the current character

Exercise 6b: CSV Parsing: ListBuffer

- **ListBuffer** is our second collection! Our first was an **Array**
- This is also a *mutable* collection
- How can we add elements to it?

Exercise 6b: CSV Parsing: ListBuffer

- **ListBuffer** is our second collection! Our first was an **Array**
- This is also a *mutable* collection
- How can we add elements to it?
  - += operator
  - append

Exercise 6b: CSV Parsing: State

- If the current character is a quote, we're in double quotes - use a Boolean to mark this.
  - Else **if the current character is a comma**, turn the StringBuilder into a String and add the column to a *collection*.
- Let's get rid of *leading* and *trailing* spaces
- Let's reuse the StringBuilder by **clearing** it

Exercise 6b: CSV Parsing: What's missing?

- Hey we're not getting the last column!
  - **Hint**: We still have characters in our StringBuilder that didn't get stored because we never hit a final comma
- We don't have time to cover the escaped quote case, but I'll show you the code.
  - It requires performing a *lookahead* by one character.

Functions

- It becomes very tedious to repeat code over and over again
- It would be great to wrap up our operations into ***reusable*** units of code
- **What are functions?**

Functions

- **What are functions?** They are named operations that can take zero or more parameters and return a result.
  - A ***pure*** function is like a mathematical function.
  - For a given input, it will **always** return the same output
  - It will produce no **side effects**

Functions

- **What are functions?**

```
def add(x: Int, y: Int): Int = x + y
```

```
def generateUniqueId: String = {
  // Code to generate id
}
```

Functions: Side Effects

- **What are functions that produce side effects?**

```
def writeToFile(line: String): Unit
```

```
println(“Hello World”)
```

- Functions that only produce side effects return the **`Unit`** type
- They do not return a useful value

Functions

- **How do you invoke a function?**

```
add(2, 2)
```

```
val result = add(2, 2)
```

Functions

- You can create *anonymous* functions (aka **closures**) and assign them to a **val**

```
val add = (x: Int, y: Int) => x + y
```

- Functions themselves are types in Scala
  - add: (Int, Int) => Int = <function2>

Exercise 8a: Celsius to Fahrenheit

- Write a function to perform this conversion

$$\boxed{\mathsf{F = C * 9/5 + 32}}$$

Exercise 8b: Function Composition

- Imagine we had a function that mapped temperature to an English adjective:

```
def tempToDescription(currentTemp: Float): String = {
  if (currentTemp >= 0 && currentTemp <= 32) "freezing"
  else if (currentTemp > 32 && currentTemp < 60) "cold"
  else if (currentTemp >= 60 && currentTemp < 75) "warm"
  else if (currentTemp >= 76 && currentTemp < 100) "hot"
  else if (currentTemp == 75) "perfect"
  else "yikes"
}
```

Exercise 8b: Function Composition

- Perform a C to F conversion and then pass the result to this function:

```
def tempToDescription(currentTemp: Float): String = {
  if (currentTemp >= 0 && currentTemp <= 32) "freezing"
  else if (currentTemp > 32 && currentTemp < 60) "cold"
  else if (currentTemp >= 60 && currentTemp < 75) "warm"
  else if (currentTemp >= 76 && currentTemp < 100) "hot"
  else if (currentTemp == 75) "perfect"
  else "yikes"
}
```

Exercise 8b: Function Composition

- Perform a C to F conversion and then pass the result to this function:

```
val tempF = celsiusToFahrenheit(30)
tempToDescription(tempF)
tempToDescription(celsiusToFahrenheit(30))
```

Exercise 8b: Function Composition and Partially Applied

- You can build values out of functions!

```
val c2f = celsiusToFahrenheit _
val tempToDesc = tempToDescription _
```

- The underscore syntax means the argument will be provided later. It produces a function value!

```
val add = (x: Int, y: Int) => x + y
val inc = add(_: Int, 1)
```

Exercise 8b: Function Composition and Partially Applied

- Now you can compose functions in different ways

```
c2f andThen tempToDesc
```

```
tempToDesc compose c2f
```

Exercise 9: Parser Function

- Let's refactor the line parser into a function!

Exercise 9: Parser Function

- Let's refactor the line parser into a function!
- This is a pure function!
- Wait it has *mutating* state! But that state is never exposed outside the function - they are **local variables**
- **Why are we returning a mutable collection? The caller shouldn't be able to change the parsed result!**

Exercise 9: Parser Function

- **Why are we returning a mutable collection? The caller shouldn't be able to change the parsed result!**
- We can convert the `ListBuffer` into an immutable collection via the `toList` operation

Exercise 9: Parser Function

- **Immutable collections cannot be changed - each change creates a new collection!**
- For performance reasons, within the while loop we use mutable state (since its not exposed):
  - We avoid creating new values and collections repeatedly
- BUT Do not return mutable state from a function!

Aside: When possible avoid vars and while loops

- **In fact, we can even avoid the while loop!**
- While loops encourage a mutable style due to their in place looping.
- We could create a recursive function that processes one character at a time, and takes in all of the current state (position, within double quotes, etc.?). This would also be a pure function!
- If the final call is a call back to the same function, it can be turned into a loop!

BREAK

Collections and Structured Data

Immutable Collections

- We've already seen some mutable collections: `Array` and `ListBuffer`
- In our last exercise, we modified our parser function to a return a `List`
- What is a `List`?

List

- An immutable, ordered collection that can grow in size (backed by a linked list) and ***cannot randomly access***
- CSV processing:
  - We need to process an unknown number of records
  - After processing, we don't need to directly access any specific element

List

- Let's create a few `List` instances
- Temperatures: `List(41, 32, 75, 100)`
- Names: `List("Bob", "Alice", "John", "Judy")`
- Notice the types: `List[Int]`, `List[String]`

List

- Computing the length requires traversing
- Getting the last element requires traversing
- Conversion to and from:
  - `Array(1,2,3,4,5).toList`

Higher Order Functions: `map`

- We can perform operations on collections by passing functions.
- A function that takes in and/or produces functions is called a **higher order function**

- If we want to transform every element in a List

```
map[B](f: (A) => B): List[B]
```

Examples

```
val temps:List[Float] = List(41,32,75,100)
temps.map(fahrenheitToCelsius)
temps.map { temp => fahrenheitToCelsius(temp) }
temps.map { fahrenheitToCelsius(_) }
```

```
map[B](f: (A) => B): List[B]
```

Exercise 10

- Given a List of lines from the CSV data, return a list of parsed records
- `List[String] => List[List[String]]`

Higher Order Functions: `filter`

- Returns a new collection with only the elements that pass the filter *predicate.*
- The return type remains the same.

```
filter(p: (A) => Boolean): List[A]
```

Exercise 11

- Before parsing, filter out empty lines
- After parsing, filter out records with all columns empty
  - **Hint**: Look at `forall` and `exists` (either can be used)
- Combine these steps and parsing into one function

Structuring Data: Case Classes

- Dealing with **String**s is a pain. We don't know what the data is and can't manipulate it.
- How can we build up ***meaningful types*** from the parsed records?
- **Case Classes!**

Structuring Data: Case Classes

```
case class FullName(firstName: String,
lastName: String)
```

```
case class PurchaseOrder(productId: String,
orderDate: LocalDate, price: Double,
quantity: Int = 1)
```

Case classes are immutable.

Structuring Data: Case Classes

```
FullName("Sujan", "Kapadia")

PurchaseOrder(productId = "SomeProductID",
orderDate = LocalDate.of(2016, 5, 4), price
= 23.50d)
```

- Enter these into the worksheet and see the output

Data Modeling

- AmazonProduct
  - **Product ID**
  - **Title**
  - **Category**
  - **Condition**

Data Modeling

- AmazonProduct
  - **Product ID:** Has an ASIN Code
  - **Title:** String
  - **Category:** Could have things like parent category, etc.
  - **Condition:** Limited set of values - *enumerated*

Data Modeling

AmazonProduct

```
case class ProductId(asinCode: String)
case class Category(name: String)
case class AmazonProduct(id: ProductId, title:
String, category: Category)
```

Case Class Hierarchy

Condition (we won't be using this today)

```
sealed trait Condition
case object UsedCondition extends Condition
case object GoodCondition extends Condition
case object NewCondition extends Condition
```

> Case objects have no properties. They are *singleton* instances

Data Modeling

- PurchaseOrder
  - **Order ID**
  - **AmazonProduct**
  - **Order Date**
  - **Unit Price**
  - **Quantity**
  - **Total Price**

Data Modeling

- PurchaseOrder
  - **Order ID**: String with a special format
  - **AmazonProduct**
  - **Order Date:** LocalDate
  - **Unit Price:** Double
  - **Quantity:** Int
  - **Total Price:** Double

Data Modeling

PurchaseOrder

```
case class OrderId(value: String)
```

```
case class PurchaseOrder(id: OrderId, product:
AmazonProduct, orderDate: LocalDate,
unitPrice: Double, quantity: Int, totalPrice:
Double)
```

Exercise 12a: Convert record to PurchaseOrder

- Write a function that converts a List[String] to a PurchaseOrder

Exercise 12a: Hints

- Date conversion:

```
val df = DateTimeFormatter.ofPattern("MM/dd/yy")
val dateString = "06/21/2007"
LocalDate.parse(dateString, df)
```

- Converting a dollar amount to a Double:

```
val price = "$19.99"
price.drop(1).toDouble
```

- Accessing an element by index

```
val x = cols(1)
```

Data Modeling

- What List operation allows us to transform a List[A] (elements of type A) to List[B] (elements of type B)?

Exercise 12b: Convert a List of lines

- What List operation allows us to transform a List[A] (elements of type A) to List[B] (elements of type B)?
- `map`
- Convert lines - `List[String]` to a `List[PurchaseOrder]`

Optional data: `Option[T]` type

- If you look at the data, you'll notice things like Condition and Category are sometimes empty
- An empty String or a `null` to represent this is a poor choice.
  - It's error prone - have to always check for empty or null
  - It's not clear to the user of the API that the data is optional

Optional data: `Option[T]` type

- Scala provides an Option type
- An Option can take on one of two values
  - `Some(value)`: this means its defined
  - `None`: this means its empty
- Let's see some examples

Exercise 13: Analyzing the data

- Determine the unique ***set*** of categories
- Return all items greater than or equal to $100
- Determine the total spent per category
  - Return the top 2
- Determine the total spent per year

What do Case Classes give us?

- equals method
- toString method
- copy method to create new, modified instances
- immutability
- pattern matching: very powerful and expressive, but we won't have time to review it today :(

Takeaways

- Values and immutability
- Expression-oriented language
- Static typing: Types are enforced
- Everything is an object; objects have operations
  - Primitives extend from AnyVal and all others from AnyRef
  - Any is the root type

Takeaways

- Immutable vs mutable collections
- Functions and higher order functions are the basic building blocks
- Functions themselves are objects
- Do not expose mutable state outside of a function!
- Choose the right data structure based on the constraints.

Takeaways

- Model data with case classes
- Use Option instead of relying on null
- Look up the operations, use ScalaDoc, even look at the actual Scala implementation!

SO MUCH more to cover

- Things not covered that are typically covered in a 1 - 2 day Scala course:
  - Case Class Pattern Matching
  - For Comprehensions
  - Traits, Classes, Objects, Packaging
  - Functional processing of Lists
  - Type system
  - Unit testing (Scalatest)

Consider Lightbend's Fast Track to Scala Course

THANK YOU!

Sujan Kapadia
Philly Tech Week 2016

CHARIOT SOLUTIONS