CS162 Week 2

Kyle Dewey

Overview

- More on Scala
- Loops in functional languages

Classes

- Created with the `class` reserved word
- Defaults to `public` access
- Constructors are **not** typical

Traits

- Created with the `trait` reserved word
- Like a mixin in Ruby
- Think Java interfaces, but they can have methods defined on them
  - More powerful than that, but not relevant to this course

`object`

- Used in much the same way as `static` is in Java
- Defines both a class and a single instance of that class (and **only** a single instance)
- Automated implementation of the Singleton design pattern
- Keeps everything consistently an object

`equals`, `==`, **and** `eq`

- As with Java, if you want to compare **value** equality, you must extend `equals`
  - Case classes automatically do this for you
- However, instead of saying `x.equals(y)`, **merely say** `x == y`
- If you want **reference** equality, say: `x eq y`

Case Classes

- Behave just like classes, but a number of things are automatically generated for you
  - Including `hashCode`, `equals`, and getters
- Typically used for pattern matching

Pattern Matching

- Used extensively in Scala
- Like a super-powerful `if`
- Used with the `match` reserved word, followed by a series of `case`s

`null`

- In general, `null` is an excellent wonderful/ terrible feature
- Often poorly documented whether or not `null` is possible
  - Checking for impossible cases
  - Not checking for possible cases

Option

- A solution: encode `null` as part of a type
- For some type, say `Object`, if `null` is possible say we have a `NullPossible<Object>`
- Scala has this, known as `Option`
- In general, if `null` is possible, use `Option`

Tuples

- For when you want to return more than one thing
- Can be created by putting datums in parenthesis
- Can pattern match on them

Sequence Processing Functions

AKA: Why `while` is rare and `for` isn't `for`

Looping

- Scala has a `while` loop, but its use is highly discouraged (again, point loss)
  - It's not actually needed
- General functional programming style is recursion, but this is usually overkill

Taking a Step Back...

- When do we write loops?
  - Transform data
  - Scan data
  - Aggregate data
- Higher-order functions allow us to abstract away much of this

foreach

- Applies a given function to each element of a `Seq`

`map`

- Like `foreach`, in that it applies a given function to each element of a sequence
- However, it also returns a new sequence that holds the return values of each of the function calls

`filter`

- Takes a predicate, i.e. a function that returns true or false
- Applies the predicate to each item in a list
- A new list is returned that contains all the items for which the predicate was true

`foldLeft`

- Extremely flexible, but sometimes unwieldy
- Takes a base element
- Takes a function that takes a current result and a current list element
- The function will manipulate result with respect to the current element

`flatMap`

- Like `map`, but made especially for functions that return `Seq`s
- Will internally "flatten" all of the inner `Seq`s into a single `Seq`
- More on this later in the course

`for` Comprehensions

- Much like Python's list comprehensions
- Internally translated into a series of `foreach`, `flatMap`, `map`, **and** `filter` operations