Day 3

Dark Syntax Sugars

So Sweet!

Single arg method calls allow { }

```
val result = singleParamMethod {
  // ...
  42
}
```

Single abstract method

```
trait Action {
  def act(x: Int): Unit
}

val action: Action = (x: Int) => println(x)
```

Methods with ':' are special

- right-associative if ending in ':'
- left-associative otherwise

```
class MyStream[T] {
  def -->:(value: T): MyStream[T] = ???
}

val stream = 1 -->: 2 -->: new MyStream[Int]
```

So Sweet!

Method naming ++

```
class Teen {
  def `and then said`(message: String) = ???
}

mike `and then said` "funny name!"
```

Infix types

```
trait <[A, B]
val lessThan: A < B = ???
```

Mutation on arrays: update()

```
val anArray = Array(1,2,3)
anArray(2) = 7 // <=> anArray.update(2, 7)
```

Mutation on classes: setters

```
trait MutableContainer {
  def member: Int
  def member_=(value: Int): Unit
}

mContainer.member = 42
```

Advanced Pattern Matching

Advanced Pattern Matching

We can define our own patterns!

```
class Person(val name: String, val age: Int)
object Person {
  def unapply(person: Person): Option[(String, Int)] =
    Some((person.name, person.age))
}
```

the object we want to decompose

results as an Option or Option(tuple)

```
person match {
  case Person(name, _) => println(s"Hi, I'm $name.")
}
```

the compiler searches the appropriate unapply

Patterns are independent of classes we decompose.

Advanced Pattern Matching

Infix patterns

```
numbers match {
  case head :: Nil => println("single element" + head)
  // equivalent:
  case ::(head, Nil) => println("single element" + head)
}
```

Unapply for sequences

```
object MyList {
  def unapplySeq[A](list: MyList[A]): Option[Seq[A]] = // ...
}

myList match {
  case MyList(1,2, _*) => // ...
}
```

Implicits

Implicits

Implicit values & arguments

```
def methodWithImplicitArg(normalArg: Int)(implicit specialArg: Int) = ???
implicit val meaningOfLife: Int  = 45
methodWithImplicitArg(67) // the implicit one passed automatically
```

Implicit classes & extension methods

```
implicit class MyRichInt(number: Int) {
  def times(string: String): String = ...
}

3.times("Hello world") // new MyRichInt(3).times("Hello world")
```

Implicit methods & conversions

```
implicit def string2Human(string: String): Human = ...
"Daniel".greet // string2Human("Daniel").greet
```

Organizing Implicits

The compiler looks for implicits (vals, classes, defs) in

- the local scope
- imported scope
- the companion object of the type whose method we're calling
- the companion objects of all the types involved in the method call (if generic method)

Best practice:

1. If there is a single possible value for an implicit val, define it in the companion object of its type.
2. If there are multiple possible values for an implicit val, define the "best"/most common one in the companion object of its type.
3. Define all other implicits (defs, classes, uncommon vals) in objects, and import selectively, only what you need.

Type Classes

Selectively add/restrict functionality for some types and not for others

Part 1: the type class definition

```
trait MyTypeClassTemplate[T] {
  def action(value: T): String
}
```

Part 2: type class instances

```
implicit object MyTypeClassInstance extends MyTypeClassTemplate[Int] { ... }
```

Part 3: extension methods

```
implicit class ConversionClass[T](value: T) {
  def action(implicit instance: MyTypeClassTemplate[T]) = instance.action(value)
}
```

```
2.action
```

only available for the types you support
(with existing TC instances)

You Rock