Currying

Apply some arguments now, and others later

```
def multiply(m: Int)(n: Int): Int = m * n
```

Scala is expression-oriented.

Everything (almost) is an expression

Abstract class defines methods but does not implement them

Traits are collections of fields and behaviours that you can extend/mixin to your classes

A class can extend several traits by using 'with'

Trait vs abstract classes. If you want to define an interface-like type, it might be hard to choose. Favor using Traits! Handy. If you need a constructor use abstract class

types [F] generic.

apply methods ⇒ gives you a nice syntactic sugar for when a class or object has one main use

—

Pattern matching

- on values (_ wildcard)

```
case i if i == 1 => "one"
case i if i == 2 => "two"
case _ => "some other number"
```

if's can be used

```
case 1 => "one"
case 2 => "two"
case _ => "some other number"
```

you can match on types

```
def bigger(o: Any): Any =
  o match {
  case i: Int if i < 0 => i - 1
  case i: Int => i + 1
  case d: Double if d < 0 => d - 0.1
  case d: Double => d + 0.1
  case text: String => text + "s"
```

CASE CLASSES!

Conveniently store and match on the contents of a class. Constructed without new

Automatically have equality and nice toString methods based on the constructor arguments

Can have methods like normal classes

Case classes w/ pattern matching!

They are designed to be used with pattern matching

```
def calcType(calc: Calculator) = calc match {
  case Calculator("HP", "20B") => "financial"
  case Calculator("HP", "48G") => "scientific"
  ...
  case Calculator(brand, model) => "calc: %s %s is of unknown type".format(brand, model)
  case Calculator(_, _) => "Calculator of unknown type"
```

Instead of accessing the fields and using getters!

↓ can be done as written

Exception

```
try { ... }
catch { case e: IllegalArgEx => log.error(e, "illegal par") }
finally { ... }
```

BASIC DATA STRUCTURES

Arrays preserve order, and are mutable!

val numbers = Array(1,2,3,4)

numbers(0) → 1

update!

numbers(3) = 10

Lists

preserve order, and are immutable!

val numbers = List(1,2,3)

numbers(2) = 10

↑ error: update value is not a member of list

sets do not preserve order and have duplicates!

val number = Set(1,1,2,3,3,3,)

> numbers : Set = (1,2,3)

tuple groups simple logical collections of items without using a class

val hostPort ("localhost", 80)

hostPort : (String, Int) = ("localhost", 80)

- Unlike classes they don't have named accessors use ._1 ._2 here
- fit pattern matching nicely
- 1 → 2 creates (1,2) tuple of type (Int, Int)

hostport match {
("localhost", port) =>
case (host, port) =>

Maps can hold basic datatypes

Map(1 → 2) Map("FOO" → "BAR")

→ is used to create tuples (no special syntax)

Map(1 → "one", 2 → "two", 3 → "three")

expands to

Map((1, "one"), (2, "two"), (3, "three"))

with first elements being keys and followed by values

Map(1 → Map("FOO" → "BAR"))

Maps can themselves contain Maps

Map("timesTwo" → {timesTwo(_)})

... or functions

Option is a container that may or may not hold something

subclasses are Some[T] and None

Map.get uses option for its return type. Option tells that the method might not return what you're asking for

```
val numbers = Map("one" → 1, "two" → 2)
numbers.get("two")   → Some(2)
numbers.get("three") → None
```

isDefined checks whether it is Some

getOrElse usually is better, which lets you define a default value

Better matching fits naturally with Option

```
val result = test match {
  case Some(n) => n*2
  case None => 0 }
```

FUNCTIONAL COMBINATORS!

List(1,2,3) map squared applies squared to the elements of the list, returning a new list (1,4,9). We call operations like map combinators

map evaluates a function over each element in the list, returning a new list

```
numbers.map((i: Int) => i*2)  or
numbers.map(timesTwo)
```

Foreach is like map but returns nothing. It is intended for side effects only

filter removes any element

```
numbers.filter((i: Int) => i%2 == 0)
numbers.filter(isEven)
```

zip aggregates contents of two lists into a list of pairs

partition splits a list based on where it falls with respect to a predicate function

Find returns the first element of a collection that matches a predicate function

drop and dropWhile

drop drops the first i elements

dropWhile removes the first elements that match a predicate function

FoldLeft. val numbers = List(1,2,3,4,5,6,7,8,9,10)

numbers.foldLeft(0)(m: Int, n: Int) => m + n)

↑ Initial value ← acts as an accumulator

FoldLeft → 55

m	n
0	1
1	2
3	3
6	4
10	5
15	6
21	7
28	8
36	9
45	10

res0 → 55

FoldRight

m	n
10	0
9	n10
8	numbers
7	n19
6	n27
5	n34
4	n40
3	n45
2	n49
1	n52
	n54
	n55

↓ .foldRight(0)

(m: Int, n: Int)

⇒ m + n)

FoldRight runs in opposite direction

Flatten collapses one level of nested structure

List(List(1,2), List(3,4)) →
List(1,2,3,4)

FlatMap frequently used! combines mapping and flattening.
It takes a function that works on the nested lists and then concatenates the results back together

List(List(1,2), List(3,4)).FlatMap(_*2)
→ List(2,4,6,8)

same as List(List(1,2), List(3,4)).map(
(x:List[Int]) ⇒ x.map(_*2)).
flatten

Generalized functional combinators
All of the functional combinators shown work on maps too

Maps can be seen as a list of pairs, so the functions work on a pair of keys of values

val extensions = Map("Steve" → 100, "bob" → 101,
extensions.Filter((namephone: (String, Int)) ⇒ namephone._2 < 200)

same as → Filter({case (name, extension) ⇒ extension < 200

Function composition compose

```
def f(s: String) = "f(" + s + ")"
def g(s: String) = "g(" + s + ")"

val fComposeG = f _ compose g _
fComposeG("yay") → f(g(yay))
```

Compute g and pass result to f

andThen is like compose but calls the first function first and then the other

```
val fAndThenG = f _ andThen g _
fAndThenG("yay") → g(f(yay))
```

What are case statements?
It's a subclass of Function called a PartialFunction

What is a collection of multiple case statements?
They are multiple PartialFunctions composed together

Understanding PartialFunctions

A Function works for every argument of the defined type

(Int) ⇒ String only takes Ints and returns a String

A PartialFunction is only defined for certain values of the defined type.

A PartialFunction (Int) ⇒ String might not accept every Int

isDefinedAt is a method that can be used to determine if the partialfunction will accept the given argument

```
val one: PartialFunction[Int, String] = {
    case 1 ⇒ "One" }
one.isDefinedAt(1)  → true
one(1)  → "one"
```

—

PartialFunctions can be combined with orElse

```
val two: PartialFunction[Int, String] =
    { case 2 ⇒ "Two" }
val three: PartialFunction[Int, String] =
    { case 3 ⇒ "three" }
val wildcard: PartialFunction[Int, String] =
    { case _ ⇒ "something else"
val partial = one orElse two orElse three orElse wildcard
partial(5)  → something else
partial(3)  → three
```

Types in Scala ~

Scala's powerful type system allows for very rich expression. Some of its chief features are parametric polymorphism, [illegible] inference, cast (local) type inference, existential quantification and views

Parametric polymorphism

polymorphism is used in order to write generic code (for values of different types) without compromising static type richness

```
2 :: 1 :: "bar" :: "foo" :: Nil
res0.head  →  res1: Any = 2
```

(↑ we don't get the type)

Polymorphism is achieved by specifying type variables

```
def drop1[A](l: List[A]) = l.tail
```

Rank-1 polym!

There are some type that are too generic for the compiler to understand

```
def toList[A](a: A) = List(a)
```

which you wished to use generically

```
def Foo[A,B](f: A ⇒ List[A], b: B) = f(b)
```

Doesn't compile because all type variables must be fixed at the invokation site

Type inference (H)

The first classic type inference method in functional prog languages is Hindley-Milner

```
val x = 5        val x = "Hello"        val x = Array()
```

In Scala it is just local.

Variance!

Scala type system has to account for class hierarchies with polymorphism.

Variance annotations allows to express the following relationships between class hierarchies and polymorphic types

covariant	C[T'] is subclass of C[T]	[+T]
contravariant	C[T²] is subclass of C[T']	[-T]
invariant	C[T] and C[T'] are not related	[T]

```
class Covariant[+A]
val cv: Covariant[AnyRef] =
    new Covariant[String]

val cv: Covariant[String] =
    new Covariant[AnyRef]
ERROR!
```

```
class Covariant[-A]
val cv: Covariant[String] =
    new Covariant[AnyRef]

val cv: Covariant[AnyRef] =
    new Covariant[String]
ERROR!
```

Contravariance, when is it used?

```
class Animal { val sound = "rustle" }
class Bird extends Animal { val sound = "call" }
```

Bounds

Scala allows to restrict polymorphic variables using bounds. Those bounds express subtype relationships

```
def cacophony[T](things: Seq[T]) = things map (_.sound)
                 ERROR: sound not member of T

def biophony[T <: Animal](things: Seq[T]) =
    Things map (_.sound)
```

Quantification

Sometimes you do not care of naming a type variable

```
def count[A](l: List[A]) = l.size
```

you can use wildcards instead

```
def count(l: List[_]) = l.size
```

shorthand for

```
def count[A](l: List[A forSome {type A}]) = l.size
```

We can't say anything about T, but we can apply bounds like before

Simple build tool

project - project definition files

- project/build/yourproj.scala, main project definition file
- project/build.properties - project, sbt, and scala versions definitions
- src/main - your app code goes here, in a subdirectory indicating the code language
- src/main/resources - static files you want added to your jar
- src/test - like src/main but for tests
- lib_managed - the jar files your project depends on. Populated by sbt update
- target - the destination for generated stuff (generated thrift, code, class files, jars)

sbt allows you to start a scala REPL with all your project dependencies loaded. It compiles your project source before launching the console, providing us a quick way to bench test

Adding dependencies!

SBT considers Scala files in the project/build directory to be project definitions

```
import sbt._
class SampleProject(info: ProjectInfo) extends
      DefaultProject(info) {
  val jackson = "org.codehaus.jackson" %
      "jackson-core-asl" % "1.6.1"
  val specs = "org.scala-tools.testing" %"specs_2.8.0"
      % "1.6.5" % "test"
```

A project definition is an SBT class. Here we extend SBT's DefaultProject

You declare dependencies by specifying a val that is a dependency

Now we can pull down dependencies for our project. Run sbt update

Adding tests

Now that we have a test library [redacted] inside src/test/scala/com/twitter/sample/SimpleParserSpec.scala

```
package com.twitter.sample
import org.specs._
object SimpleParserSpec extends Specification {
  "SimpleParser" should {
      val parser = new SimpleParser()
    "work with basic tweet" in {
      val tweet = ...
    }
  }
}
```

... in sbt console run test

Prefixing an action with a tilde starts up a loop that runs the action any time source files change

~test

Packaging and publishing