Case Classes

Three invisible features

case class Foo(name: String, rating: Int) {
    val test = "example"
    def isGood = rating > 3
}

val x = Foo("cat", 2)
println(x)
println(x.name)

Output
Foo(cat,2)
cat
case class Foo(name: String, rating: Int) {
  val test = "example"
  def isGood = rating > 3
}

val x = Foo("cat", 2)

x match {
  case Foo("Cat", rating) => println(rating)
  case Foo(name, rating) if rating < 3 => println("poor")
  case Foo(name, 2) => println(name)
  case _ => println("Default")
}

Output
poor
Case Class & Match

case class Bar(name: String, rating: Int) {
    def isGood = rating < 3
}

val x = Bar("cat", 2)

x match {
    case Foo("Cat", rating) => println(rating)
    case Foo(name, rating) if rating < 3 => println("poor")
    case Foo(name, 2) => println(name)
    case _ => println("Default")
}

Output
<console>:12: error: constructor cannot be instantiated to expected type;
found   : Foo
required: Bar
    case Foo("Cat", rating) => println(rating)
abstract class FooBar

case class Foo(name: String, rating: Int) extends FooBar{
    val test = "example"
    def isGood = rating > 3
}

case class Bar(name: String, rating: Int) extends FooBar{
    def isGood = rating < 3
}
val x: FooBar = Bar("cat", 2)

x match {
  case Foo("Cat", rating) => println(rating)
  case Foo(name, rating) if rating < 3 => println("poor")
  case Foo(name, 2) => println(name)
  case _ => println("Default")
}

Output
Default
Beware of Implicit Types

val x = Bar("cat", 2)

x match {
    case Foo("Cat",rating) => println(rating)
    case Foo(name, rating) if rating < 3 => println("poor")
    case Foo(name, 2) => println(name)
    case _ => println("Default")
}

Output
<console>:12: error: constructor cannot be instantiated to expected type;
found  : Foo
required: Bar
    case Foo("Cat",rating) => println(rating)
abstract class Expression

case class Number(number: Double) extends Expression

case class UnaryOperator(operator: String, argument: Expression) extends Expression

case class BinaryOperator(operator: String, left: Expression, right: Expression) extends Expression
def evaluate(value: Expression): Double = value match {
    case Number(number) => number
    case BinaryOperator("+", left, right) => evaluate(left) + evaluate(right)
    case BinaryOperator("-", left, right) => evaluate(left) - evaluate(right)
    case UnaryOperator("+", value) => evaluate(value)
    case UnaryOperator("-", value) => -evaluate(value)
    case _ => throw new Exception("Invalid Expression")
}
Sample Use

\[-10 + 3\]

Parse to get expression tree

```scala
val x:Expression = BinaryOperator("+", UnaryOperator("-", Number(10)), Number(3))
evaluate(x)
```
class Bar {
    def apply(a: String, b: Int) =
        println("Apply with two args")

    def apply(a: String) =
        println("Apply " + a)

    def apply() = println("no args")
}

val x = new Bar
x("cat", 5)
x("sam")
x()

Output
Apply with two args
Apply sam
no args
object EMail {
  def apply(user: String, domain:String) = user +"@" + domain

  def unapply(email:String): Option[(String, String)] = {
    val parts = email split "@
    if (parts.length == 2)
      Some(parts(0), parts(1))
    else
      None
  }
}

val example: String = EMail("whitney", "cs.sdsu.edu")
Using UnApply - Extractor

"whitney@cs.sdsu.edu" match {
  case EMail(user, domain) => println( user + " " + domain)
  case _ => println("not email")
}

Output

whitney cs.sdsu.edu
Regular Expressions

val atTest = "\\wat".r
val withTripleQuotes = """"\\wat"""".r

for (r <- atTest findAllIn "cat rat mat")
    println(r)

Output
    cat
    rat
    mat
Regular Expressions & Parts

val email = """"([\w]+)@([^@]+)"""".r

val email(user, domain) = "whitney@cs.sdsu.edu"
println(user)
println(domain)

val decimal = """"(-)?(\d+)\.(\d*)?"""".r

val decimal(sign, integer, decimalPart) = "-1.234"
Concurrency & Actors
The Problem

Client-side programming using highly multi-core processors

How to do it?
Actor Model

Mathematical model of concurrent computation

Actor is the primitive of concurrent operations

Actors
- Respond to messages
- Create other actors
- Send messages to other actors
Actor History

Theoretical Work

1973 Hewitt, Bishop & Steiger

1975 Hewitt & Baker
1975 Irene Greif Ph.D. Dissertation
1981 Clinger Ph.D. Dissertation
1985 Gul Agha Ph.D. Dissertation
Actor Programming Languages

Pure
Everything is an Actor

Actor
SALSA
IO
ACT1
galsC
with many not listed

Recent Popular Hybrids
Erlang
Scala
import scala.actors.Actor

class Example(name: String) extends Actor {
  def act = {
    for (k <- 1 to 10) {
      println(name + " " + k)
    }
  }
}

val a = new Example("a")
a.start

Output
a 1
a 2
a 3
a 4
a 5
a 6
a 7
a 8
a 9
a 10
Showing Concurrency

scala> new Example("a").start; new Example("b").start
a 1
a 2
a 3
a 4
a 5
a 6
a 7
a 8

scala> b 1
b 2
a 9
b 3
a 10
b 4
b 5
e tc
import scala.actors.Actor

class Example(name: String) extends Actor {
  def act = {
    for (k <- 1 to 10) {
      println(name + " " + k)
    }
  }
}

object Main extends Application {
  val a = new Example("a")
  val b = new Example("b")
  a.start
  b.start
}

Run as Application
scalac example.scala
scala Main
Output is interleaved

Run in interpreter
scala
scala> :load example.scala
scala> Main
Output is interleaved first time
Output is not interleave on second load & run
Singleton Object Actor

```scala
import scala.actors.Actor

object SampleActor extends Actor {
  def act = {
    for (k <- 1 to 10) {
      println("A " + k)
    }
  }
}
```

SampleActor.start
import scala.actors.Actor._

val x = actor {
  for (k <- 1 to 10) {
    println("Hello " + k)
  }
}

Output
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10
import scala.actors.Actor

class Example(name: String) extends Actor {
  def act = {
    while (true) {
      receive {
        case "bye" => {println("bye"); exit}
        case x: Int => println("Int "+x)
        case message =>
          println(name + " received "+message)
      }
    }
  }
}

val a = new Example("a")
a.start
a ! "hi"
a ! 12
a ! "bye"

Output
a received hi
Int 12
bye
Some Details

! Messages are send asynchronously

Each actor with a receive runs in a separate thread
import scala.actors.Actor.

class Adder extends Actor {
  def act = {
    loop {
      receive {
        case x: Int => sender ! x + 1
      }
    }
  }
}

class Requester(adder: Actor) extends Actor {
  def act = {
    adder ! 3
    receive {
      case x: Int => println("Answer " + x)
    }
  }
}

val a = new Adder
a.start
val sender = new Requester(a)
sender.start
import scala.actors.Actor._

class Adder extends Actor {
  def act = {
    loop {
      receive {
        case (x: Int, receiver: Actor) =>
          receiver ! x + 1
      }
    }
  }
}

class Receiver extends Actor {
  def act = {
    loop {
      receive {
        case x: Int =>
          println("Answer "+ x)
      }
    }
  }
}
Using the Example

val a = new Adder
a.start
val sender = new Receiver
sender.start

\text{a ! (12, sender)}
\text{a ! 12}
\text{a ! (3, sender)}
\text{a ! "cat"}

\text{Output}
\text{Answer 13}
\text{Answer 4}