Imperative Programming 2:
Pattern matching

Hongseok Yang
University of Oxford

- Pattern matching is widely used in Scala.

- Pattern matching is widely used in Scala.
- Very similar to pattern matching in Haskell.

```haskell
data Tree = Node Tree Tree | Leaf Int

count :: Tree -> Int
count (Leaf _) = 1
count (Node l r) = count l + count r
```

```scala
trait Tree
case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

def count(t: Tree): Int =
  t match {
    case Leaf(_) => 1
    case Node(t1,t2) => count(t1) + count(t2)
  }
```

- Pattern matching is widely used in Scala.
- Very similar to pattern matching in Haskell.

```
data Tree = Node Tree Tree | Leaf Int

count :: Tree -> Int
count (Leaf _) = 1
count (Node l r) = count l + count r
```

Case class & inheri. instead of data types.

```
trait Tree
case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

def count(t: Tree): Int =
  t match {
    case Leaf(_) => 1
    case Node(t1,t2) => count(t1) + count(t2)
  }
```

- Pattern matching is widely used in Scala.
- Very similar to pattern matching in Haskell.

```
data Tree = Node Tree Tree | Leaf Int

count :: Tree -> Int
count (Leaf _) = 1
count (Node l r) = count l + count r
```

Case class & inheri. instead of data types. Match & case syntax.

```
trait Tree
case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

def count(t: Tree): Int =
  t match {
    case Leaf(_) => 1
    case Node(t1,t2) => count(t1) + count(t2)
  }
```

- Pattern matching is widely used in Scala.
- Very similar to pattern matching in Haskell.

```
data Tree = Node Tree Tree | Leaf Int

count :: Tree -> Int
count (Leaf _) = 1
count (Node l r) = count l + count r
```

Case class & inheri. instead of data types. Match & case syntax.

```
trait Tree
case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

def count(t: Tree): Int =
  t match {
    case Leaf(_) => 1
    case Node(t1,t2) => count(t1) + count(t2)
  }
```

- Of course, a few interesting differences exist.

- Pattern matching is widely used in Scala.
- Very similar to pattern matching in Haskell.

```
data Tree = Node Tree Tree | Leaf Int

count :: Tree -> Int
count (Leaf _) = 1
count (Node l r) = count l + count r
```

Case class & inheri. instead of data types. Match & case syntax.

```
sealed trait Tree
case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

def count(t: Tree): Int =
  t match {
    case Leaf(_) => 1
    case Node(t1,t2) => count(t1) + count(t2)
  }
```

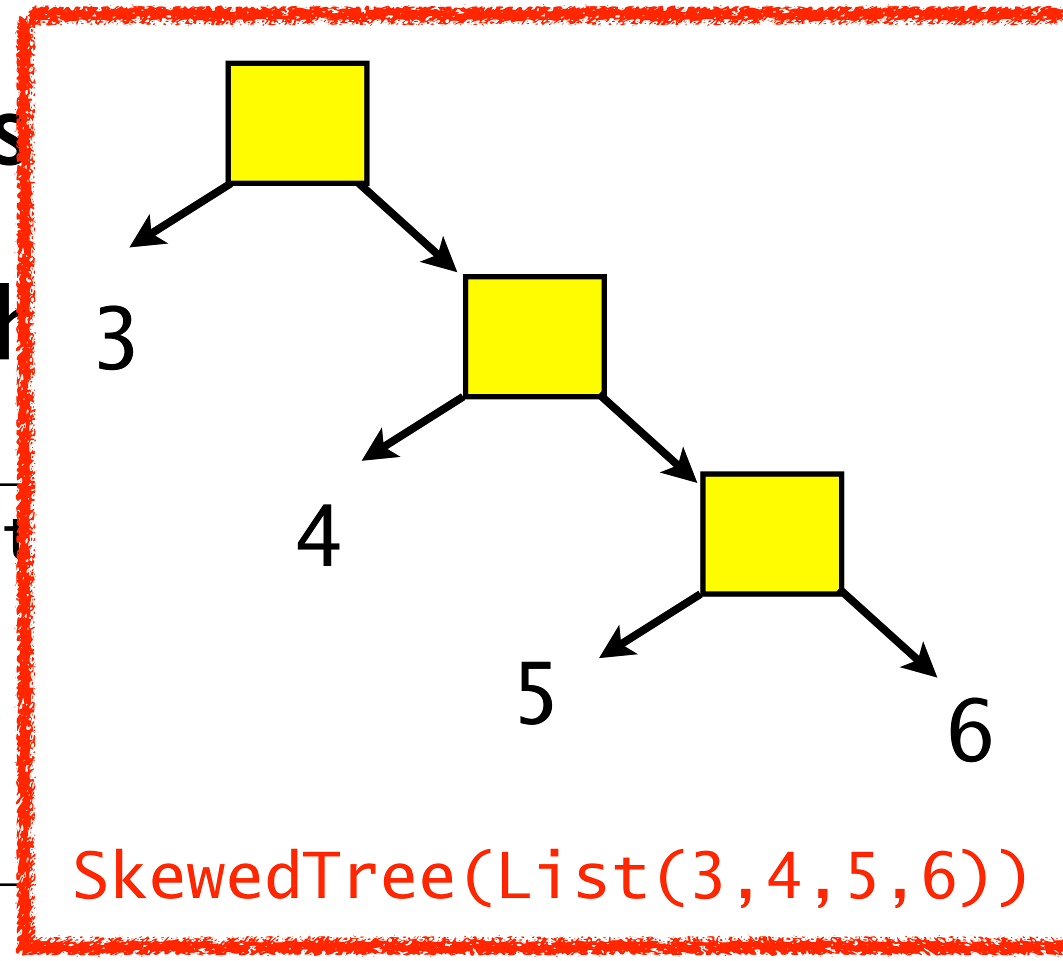
Sealed.

- Of course, a few interesting differences exist.

- Pattern matching is widely us
- Very similar to pattern match

```
data Tree = Node Tree Tree | Leaf Int

count :: Tree -> Int
count (Leaf _) = 1
count (Node l r) = count l + count r
```




```
sealed trait Tree
case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

def count(t: Tree): Int =
  t match {
    case Leaf(_) => 1
    case SkewedTree(_) => 1
    case Node(t1,t2) => count(t1) + count(t2)
  }
```

Sealed.

Can define a custom pattern.

- Of course, a few interesting differences exist.

- Pattern matching is widely used in Scala.
- Very similar to pattern matching in Haskell.

```
data Tree = Node Tree Tree | Leaf Int

count :: Tree -> Int
count (Leaf _) = 1
count (Node l r) = count l + count r
```

Case classes instead of data types.
Match & case syntax.

```
trait Tree
case class Node(l:Tree, r:Tree) extends Tree
case class Leaf(v:Int) extends Tree

def count(t: Any): Int =
  t match {
    case Leaf(_) => 1
    case SkewedTree(_) => 1
    case Node(t1,t2) => count(t1) + count(t2)
    case _ : Int => 1
    case _ => -1
  }
```

Sealed.
Can define a custom pattern.
Can match over types.

- Of course, a few interesting differences exist.

Learning outcome

- Can explain the following features of Scala:
  1. Case class.
  2. Pattern matching.
  3. Extractor.
- Can write interesting programs using these features.

Challenge 1 : Arithmetic expression

- Represent arithmetic expressions in Scala:

$$(7.0 * x^3) + (3.8 * -y), \qquad (x + y) / 5.0$$

- Implement a function that simplifies an expression using the following equalities:

$$-(-e) = e \qquad e * 1 = e \qquad e + 0 = e$$

Plan

1. Represent different types of expressions using case classes and inheritance.
    - A common idiom in Scala for implementing an algebraic data type (such as tree and list etc).
2. Implement simplification by pattern matching.

1. Represent different types of expressions using case classes and inheritance.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = ...
```

1. Represent different types of expressions using case classes and inheritance.

Subtyping allows us to treat objects of all four classes as expressions.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = ...
```

1. Represent different types of expressions using case classes and inheritance.

Objects can be constructed without new.
Class names can be used as patterns.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = ...









val e = new BOp("*", new UOp("-", new UOp("-", new
Num(0.0))), new Num(1.0))
```

1. Represent different types of expressions using case classes and inheritance.

Objects can be constructed without new.
Class names can be used as patterns.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = ...









val e = BOp(“*“, UOp(“-“, UOp(“-“, Num(0.0))), Num(1.0))
```

1. Represent different types of expressions using case classes and inheritance.

Objects can be constructed without new.
Class names can be used as patterns.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => ...
  ...
  ...
  ...
  case BOp(op, l, r) => ...
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

1. Represent different types of expressions using case classes and inheritance.

2. Implement simplification by pattern matching.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)

  ...

  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

1. Represent different types of expressions using case classes and inheritance.

2. Implement simplification by pattern matching.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)

  ...

  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

[Q1] Implement simplification for e*1=e and e+0=e.

1. Represent different types of expressions using case classes and inheritance.

2. Implement simplification by pattern matching.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = expr match {
  case UOp(“-“, UOp(“-“, e)) => simplify(e)
  case BOp(“+“, e, Num(0.0)) => simplify(e)
  case BOp(“*“, e, Num(1.0)) => simplify(e)
  case UOp(op, e) => UOp(op, simplify(e))
  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp(“*“, UOp(“-“, UOp(“-“, Num(0.0))), Num(1.0))
```

[Q1] Implement simplification for e*1=e and e+0=e.

1. Represent different types of expressions using case classes and inheritance.

2. Implement simplification by pattern matching.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)
  case BOp("+", e, Num(0.0)) => simplify(e)
  case BOp("*", e, Num(1.0)) => simplify(e)
  case UOp(op, e) => UOp(op, simplify(e))
  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

[Q2] Add a new simplification case for e+e = 2*e.

1. Represent different types of expressions using case classes and inheritance.

2. Implement simplification by pattern matching.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp

def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)
  case BOp("+", e, Num(0.0)) => simplify(e)
  case BOp("+", x, y) if x==y => BOp("*", Num(2), simplify(x))
  case BOp("*", e, Num(1.0)) => simplify(e)
  case UOp(op, e) => UOp(op, simplify(e))
  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

[Q2] Add a new simplification case for e+e = 2*e

match statement

```
selector match {
  case pattern1 => instruction1
  case pattern2 => instruction2
  ...
  case patternk => instructionk
}
```

Some possible patterns:

```
2                    UOp("-", UOp("-", x))
(x:Var)              BOp("+",x,y) if x==y
```

More patterns are described in the textbook (Chap 15).

Challenge 2 : Hide representation of Exp

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp


def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)
  case BOp("+", e, Num(0.0)) => simplify(e)
  case BOp("*", e, Num(1.0)) => simplify(e)
  case UOp(op, e) => UOp(op, simplify(e))
  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

Challenge 2 :

Alice Hide representation of Exp

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp


def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)
  case BOp("+", e, Num(0.0)) => simplify(e)
  case BOp("*", e, Num(1.0)) => simplify(e)
  case UOp(op, e) => UOp(op, simplify(e))
  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

Bob

Challenge 2 :

Alice **Hide representation of Exp**

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp, prec: Int)
  extends Exp

def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)
  case BOp("+", e, Num(0.0)) => simplify(e)
  case BOp("*", e, Num(1.0)) => simplify(e)
  case UOp(op, e) => UOp(op, simplify(e))
  case BOp(op, l, r) => BOp(op, simplify(l), simplify(r))
  case _ => expr
}

val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

Bob's code will not work after Alice's change. Bob

Idea

Bob is using names of case classes only in two ways:

1. Factory method for creating an object.
2. Pattern in pattern matching.

```
def simplify(expr: Exp): Exp = expr match {
  case UOp("-", UOp("-", e)) => simplify(e)
  ...
}
val e = BOp("*", UOp("-", UOp("-", Num(0.0))), Num(1.0))
```

Idea : Write custom factory methods and patterns, and export them, but not classes.

Option type

- Option[C] is a new type that includes all values of type C and None.

```
val x: Option[Double] = Some(3.2)
val y: Option[Double] = None
```

- Values of an option type are typically constructed by a function that works for only some, not all inputs.

```
def mySqrt(x: Double): Option[Double] =
  if (x >= 0.0) Some(math.sqrt(x)) else None
```

- Normally, they get destructed by pattern matching.

```
def show(x: Option[Double]) = x match {
  case Some(v) => println(v)
  case None => println(“?“)
}
```

Understanding case class

- The compiler generates factory method and pattern for each case class.
- They are apply and unapply methods in the companion object, which is also generated by the compiler.

```
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

```
object BOp ... {
  def apply(op: String, left: Exp, right: Exp): BOp = ...
  def unapply(bop: Any): Option[(String, Exp, Exp)] = ...
}
```

Understanding case class

- The compiler generates factory method and pattern for each case class.
- They are apply and unapply methods in the companion object, which is also generated by the compiler.

```
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

↓

```
object BOp ... {
  def apply(op: String, left: Exp, right: Exp): BOp = ...
  def unapply(bop: Any): Option[(String, Exp, Exp)] = ...
}
```

```
val e=BOp("*", Num(1.0), Num(1.0))
```

Understanding case class

- The compiler generates constructor (factory method) and destructor for each case class.

- They are apply and unapply methods in the companion object, which is also generated by the compiler.

```
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

```
object BOp ... {
  def apply(op: String, left: Exp, right: Exp): BOp = ...
  def unapply(bop: Any): Option[(String, Exp, Exp)] = ...
}
```

```
val e=BOp(“*“, Num(1.0), Num(1.0))
```

```
expr match {
  case BOp(“+“, _, _) => ...
  case _ => ...
}
```

Recipe for building custom factory method and pattern

- Declare a singleton object T.
- Define apply & unapply methods in the object T.
- Now we have a factory method T(...) and a pattern T(...).
- Good practice : unapply is the inverse of apply.
- A singleton object with unapply is called extractor.

1. Declare a singleton object T.

2. Define apply & unapply methods in the object T.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

original representation

1. Declare a singleton object T.

2. Define apply & unapply methods to the object T.

```
sealed trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

original representation

```
trait Exp
object Var { // extractor
  private class Var(val name: String) extends Exp







}
```

new representation

1. Declare a singleton object T.

2. Define apply & unapply methods to the object T.

```
trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

original representation

```
trait Exp
object Var { // extractor
  private class Var(val name: String) extends Exp
  def apply(n: String): Exp = new Var(n)







}
```

new representation

1. Declare a singleton object T.

2. Define apply & unapply methods to the object T.

```
trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

original representation

```
trait Exp
object Var { // extractor
  private class Var(val name: String) extends Exp
  def apply(n: String): Exp = new Var(n)
  def unapply(e: Any): Option[String] =
    e match {
      case e1: Var => Some(e1.name)
      case _ => None
    }
}
```

new representation

1. Declare a singleton object T.

2. Define apply & unapply methods to the object T.

```
trait Exp
case class Var(name: String) extends Exp
```

Compiler-generated code for the case class Var:

```
object Var ... { ...
  def apply(n: String): Var = new Var(n)
  def unapply(e: Any): Option[String] = ...
}
```

```
trait Exp
object Var { // extractor
  private class Var(val name: String) extends Exp
  def apply(n: String): Exp = new Var(n)
  def unapply(e: Any): Option[String] =
    e match {
      case e1: Var => Some(e1.name)
      case _ => None
    }
}
```

new representation

1. Declare a singleton object T.

2. Define apply & unapply methods to the object T.

```
trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

original representation

```
trait Exp
object Var { // extractor
  private class Var(val name: String) extends Exp
  def apply(n: String): Exp = new Var(n)
  def unapply(e: Any): Option[String] =
    e match {
      case e1: Var => Some(e1.name)
      case _ => None
    }
}
```

new representation

[Q] Do the same thing for BOp, and hide its implementation

1. Declare a singleton object T.

2. Define apply & unapply methods to the object T.

```
trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

original representation

```
trait Exp
```

```
object BOp { // extractor
  private class BOp(val op:String, val l:Exp, val r:Exp)
    extends Exp
  def apply(op:String, l:Exp, r:Exp): Exp = new BOp(op,l,r)
  def unapply(e: Any): Option[(String, Exp, Exp)] =
    e match {
      case e1: BOp => Some((e1.op, e1.l, e1.r))
      case _ => None
    }
}
```

1. Declare a singleton object T.

2. Define apply & unapply methods to the object T.

```
trait Exp
case class Var(name: String) extends Exp
case class Num(num: Double) extends Exp
case class UOp(op: String, arg: Exp) extends Exp
case class BOp(op: String, left: Exp, right: Exp) extends Exp
```

original representation

```
trait Exp
```

```
object BOp { // extractor
  private class BOp(val op:String, val l:Exp, val r:Exp,
    val prec:Int) extends Exp
  def apply(op:String, l:Exp, r:Exp): Exp = new BOp(op,l,r,0)
  def unapply(e: Any): Option[(String, Exp, Exp)] =
    e match {
      case e1: BOp => Some((e1.op, e1.l, e1.r))
      case _ => None
    }
}
```

Challenge 3: Email address

- Email addresses are strings of the form:

  string@string

- We would like to use patterns for email addresses:

```
def printEMail(s: String) {
  s match {
    case EMail(user, domain) => println(user +“ AT “+ domain)
    case _ => println(“not an email address“)
  }}
```

Challenge 3: Email address

- Email addresses are strings of the form:

  string@string

- We would like to use patterns for email addresses:

```
def printEMail(s: String) {
  s match {
    case EMail(user, domain) => println(user +“ AT “+ domain)
    case _ => println(“not an email address“)
  }}
```

- Complete the definition of the extractor EMail:

```
object EMail {
  def apply(user: String, domain: String) = user +“@“+ domain
  def unapply(s: String): Option[(String, String)] = ...



}
```

Hint: `“a@b@c“.split(“@“) = List(“a“,“b“,“c“)`

Challenge 3: Email address

- Email addresses are strings of the form:

string@string

- We would like to use patterns for email addresses:

```
def printEMail(s: String) {
  s match {
    case EMail(user, d
    case _ => println(
  }}
```

EMail implements an alternative view on certain strings.

- Complete the definition of the extractor EMail:

```
object EMail {
  def apply(user: String, domain: String) = user +“@“+ domain
  def unapply(s: String): Option[(String, String)] = {
    val parts = s.split(“@“)
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
```

Extractor

- A singleton object that contains the unapply method.
- It can be used as a pattern in pattern matching.
- Two usages seen today:
    - Information hiding.
    - An alternative view on data.

Summary

- Key concepts covered:
  - Pattern matching.
  - Case class.
  - apply (factory method) and unapply (pattern).
  - Extractor.
- Read Chap 15 and Chap 26.1 - 26.3