Pattern Matching

CIS-3030, Vermont Technical College
Peter C. Chapin

Pattern Matching

- What is it?
  - Match a complex data structure against a pattern
  - Common feature of functional languages
- Example

```
def sumAndDifference(x: Int, y: Int) = (x + y, x – y)

val result = sumAndDifference(1, 2)
val (resultSum, resultDifference) = result
  // Match result against a “tuple pattern”
```

Details

- `val (resultSum, resultDifference) = result`
  - Names bound to components of `result`
  - Names are vals here (could also be vars)
  - *Names have types inferred*
    - `result` has type (Int, Int) so `resultSum` must be Int
  - Names can be used like any other val (or var)
    - `val x = resultSum + 1`

Compare Approaches

- Contrast:
  - Without pattern matching
    - ```
      val result = sumAndDifference(a, b)
      val x = result._1 + 1
      val y = result._1 * result._2
      ```
  - With pattern matching
    - ```
      val (sum, difference) =
          sumAndDifference(a, b)
      val x = sum + 1
      val y = sum * difference
      ```

Usefulness of Tuples

- Why Tuples?
  - Can (easily) return multiple values from a method
  - Caller pattern matches to extract values
    - … and give them suitable names
    - Tuple value returned often not manipulated directly
- Pattern Matching Called *Deconstruction*
  -
```
val myArray = Array( (1, “Hello”), (2, “World”) )
  // myArray has type Array[(Int, String)]

val (key, message) = myArray(1)
  // Deconstruct tuple in array element #1
```

List Patterns

- You Can Pattern Match Lists
  - `val myList = List(1, 2, 3)`
    `val x :: xs = myList`
  - The `::` symbol separates the “head” and “tail.”
    - *Defn: The head of a list is the first element*
    - *Defn: The tail of a list is everything else (a list)*
  - After the above code…
    - `x == 1`
    - `xs == List(2, 3)`

Nil

- The Symbol `Nil` is the Empty List
  - ```
    val myList = List(1)
    val x :: xs = myList
    ```
  - After this code executes
    - `x == 1`
    - `xs == Nil`
  - The empty list can also be represented as `List()`
    - The distinction between `List()` and `Nil` does not concern us now
    ```

Impossible Matches?

- Consider
  - `val myList: List[Int] = List()`
    `val x :: xs = myList  // What happens?`
  - When this code executes…
    - `scala.MatchError` exception is thrown!
    - If the match executes successfully, the names *are* bound to something.
  - `val myPair = (1, 2)`
    `val x :: xs = myPair  // Huh?`
    - Compiler says: “error: constructor can’t be instantiated to expected type.”

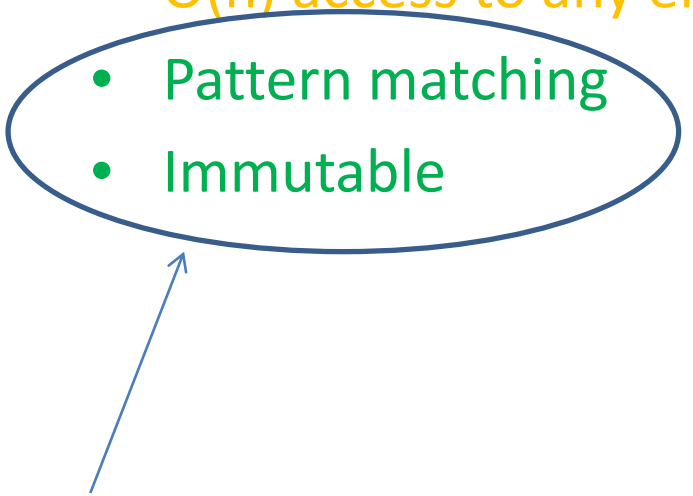
Arrays vs. Lists

Arrays

- Fast access to first element
- Fast access to any element
- No pattern matching
- Mutable

Lists

- Fast access to first element
- O(n) access to any element
- Pattern matching
- Immutable



Prefer List unless you need fast random access or mutability

What Else?

- Arbitrary Pattern Matching
  - For your own classes define method `unapply`
    - Beyond the scope of these slides
  - For many uses define a *case class*
    - Compiler creates `unapply` for you...
    - … and also some other services.

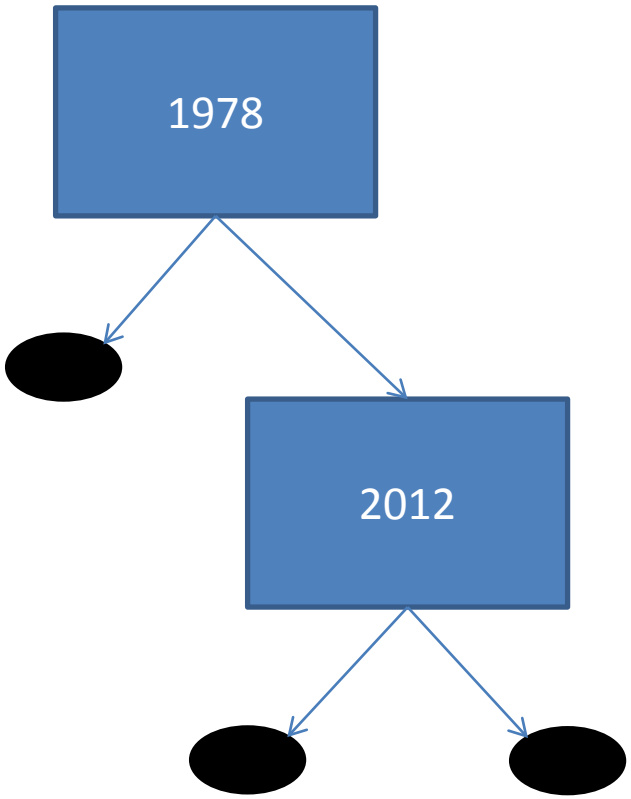
Case Classes

- Simple Example
  - ```
    case class Student(
      ID      : Int,
      name    : String,
      balance: Double)
    ```
  - Example use
    - ```
      val studentList = getAllStudents(2012)
      for (Student(ID, name, balance) <- studentList) {
        // ID, name, and balance for “current” student
      }
      ```
    - Pattern match in blue above.
    - Pattern matching allowed inside for bindings also!

Case Classes and Inheritance

- Case Classes can be related
  - Useful for creating complex data structures
    - ```
      sealed abstract class Tree
      case object Leaf extends Tree
      case class Node(
        data: Int, left: Tree, right: Tree) extends Tree
      ```
  - Both Left and Node are trees. Thus:
    - ```
      val myTree =
        Node( 1978, Leaf, Node(2012, Leaf, Leaf) )
      displayTree(myTree)
      ```
  - Can create instances without new

Picture



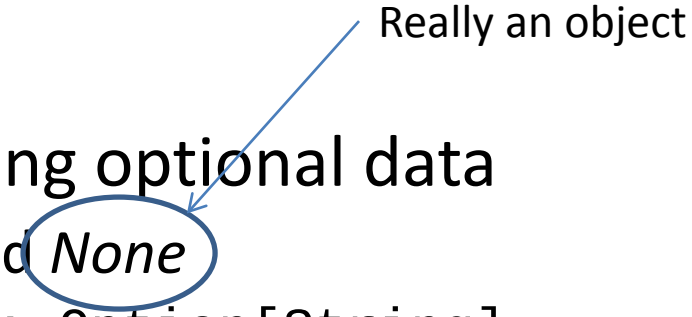

```
val myTree =
  Node( 1978, Leaf, Node(2012, Leaf, Leaf) )
```

Use Pattern Matching

- Deconstruct Trees
  - `val Node(_, _, Node(value, _, _)) = myTree`
  - The _ symbol means:
    - "match anything"
    - "I don't care what it is"
  - The pattern above…
    - Matches `myTree` to a tree with a certain shape
    - Binds `value` to the data item in the right child
    - Throws an exception if the match fails
    - Infers the type of `value` as Int.

Option

- Scala Library Option
  - Case class for representing optional data
    - Two subclasses: *Some* and *None*
    - `def getName(ID: Int): Option[String] = …`
      …
      `val Some(name) = getName(1234)`
    - Throws MatchError if `getName` returns *None*
  - Option used instead of null (as in Java)
    - Better type safety
    - More flexible. Option has methods to allow processing of optional data safely even if it's not really there.



Match Expressions

- Roughly Similar to switch in C/Java.
  - ```
    val x = someInt match {
      case 1 => 3.14
      case 2 => 2.78
      case _ => 0.0
    }
    ```
  - Expression evaluates to a value depending on match taken.
  - Matches checked in order (top to bottom)
  - The _ symbol means “anything else.”
    ```

Conditional Not Necessary

- Conditional Expressions are Redundant
  - `val x = if (condition) e1 else e2`
  - ```
    val x = condition match {
      case true  => e1
      case false => e2
    }
    ```
  - Conditionals provided as convenience. Potentially easier to optimize.
  - Compiler infers type of match as with conditional (least upper bound type of the branches)
    ```

Match Cases Are Patterns

- Pattern Matching Applies
  - ```
    val myPair = (1, 2)
    val result = myPair match {
      case (1, b) => b + 1
      case (a, 1) => a + 1
      case (_, b) => b
    }
    ```
  - Last pattern above matches everything. Must be last.
  - *Can deconstruct complex data in different ways and do different things in each case*.
    ```

List Matches

- Computing List Length Without Looping
  - ```
    def length[A](myList: List[A]): Int =
      myList match {
        case Nil => 0
        case _ :: tail => 1 + length(tail)
      }
    ```
  - This is idiomatic function style.
    - Note the use of recursion instead of (explicit) looping
    - No vars
    - Recursive observation: “Length of a list is one plus the length of the tail”
    ```

Handling Option

- Pattern Matching Style
  - ```
    getName(ID) match {
      case None =>
        println(“Invalid ID: ” + ID)
      case Some(name) =>
        println(“Processing ” + name + “…”)
    }
    ```
  - `getName` method returns `Option[String]`
  - This is still not the most idiomatic style.
    - Will show another way once we have higher order methods.
    ```

Tree Matches

- Matching Complex Data Structures
  - ```
    val result = myTree match {
      case Leaf => 0
      case Node(1, _, _) => 1
      case Node(_, Node(v1, _, _), Node(v2, _, _)
        => v1 + v1
      case _ => throw new InvalidTreeShapeException
    }
    ```
  - ```
    val newTree = myTree match {
      case Leaf => Leaf
      case Node(0, left, Node(x, _, right))
        => Node(x, left, right)
      case _ => throw new InvalidTreeShapeException
    }
    ```

Guarded Patterns

- Patterns in a match can be qualified
  - ```
    someValue match {
      case 1 => println(“It’s one”)
      case a if (a < 0) =>
        println(“The value ” + a + “ is negative”)
      case _ => println(“It’s something else”)
    }
    ```
  - Cases tried in order…
    - … but if a guard is false that case is skipped.
    ```

Compare

- As a guarded pattern
  - ```
    x match {
      case a if (a < 0) => …
      case _ => …
    }
    ```
- With a conditional in the branch
  - ```
    x match {
      case a => if (a < 0) …
      case _ => …
    }
    ```

More Interesting Example

- Guarded patterns and more complex matching
  - ```
    val myPair: (Int, Int) = …
    myPair match {
      case (a, b) if (isPrime(a)) => …
      case (a, b) if (a == 2*b) => …
      case (a, b) => …
    }
    ```
  - Consider: complex tree patterns with elaborate guard conditions on subtrees, etc.
    ```

Regular Expressions

Must use upper case first letter!

- Pattern matching and regular expressions
  - ```
    val Name = """^\s*(\w+)\s+(\w+)\s*$""".r
    val FirstName = """^\s*(\w+)\s*$""".r
    "Jill Jones" match {
      case FirstName(first) => println(s"$first")
      case Name(first, last) =>
        println(s"$last, $first")
      case _ => println("Invalid name format")
    }
    ```
  - Triple quoted strings disable escape sequences.
  - Note use of r method on string. This converts string to regular expression object.
  - Matching extracts parenthesized fields
    ```