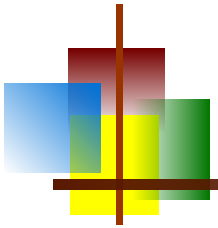
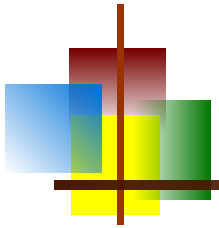
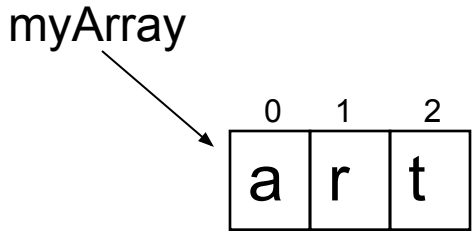


Lists

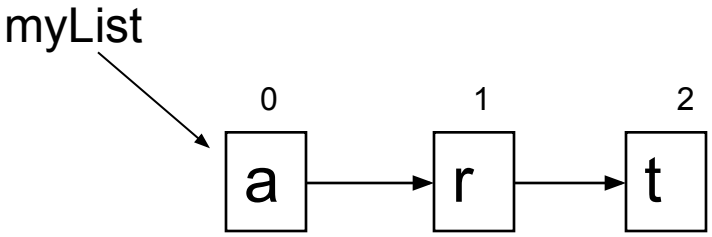


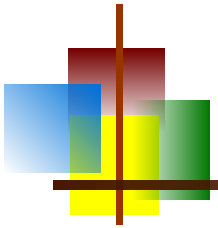
Arrays and Lists

- Arrays are a fixed length and occupy sequential locations in memory
  - This makes random access (for example, getting the 37th element) very fast--O(1)



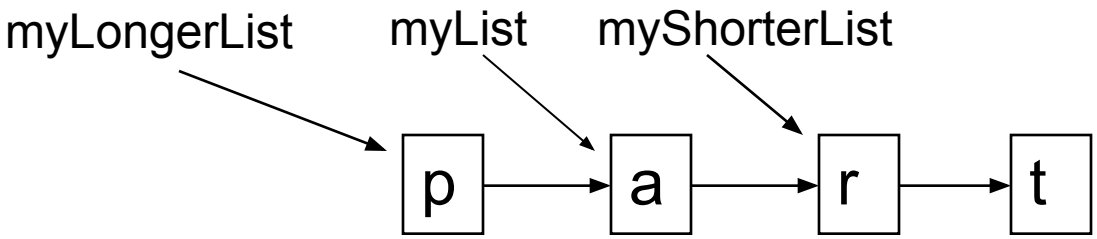
- Lists are composed of values linked together
  - All access starts from the head (first element) and follows links
  - Random access takes linear time



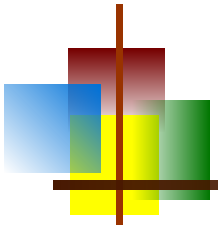


Lists are immutable

- Lists, like Strings, are immutable
- Because all access is via the head, creating a "new" list is a fast operation

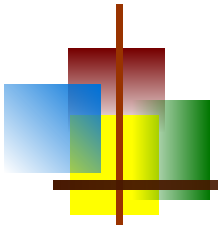


- myLongerList looks like List("p", "a", "r", "t"); the "p" is not visible from myList
- myShorterList looks like List("r", "t")
- myList has not been changed--it is *immutable*



List operations

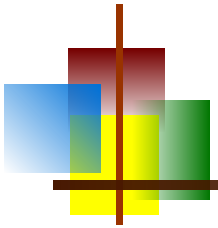
- Basic *fast* (constant time) operations
  - *list*.head (or *list* head) returns the first element in the list
  - *list*.tail (or *list* tail) returns a list with the first element removed
  - *value* :: *list* returns a list with *value* appended to the front
  - *list*.isEmpty (or *list* isEmpty) tests whether the list is empty
- Some *slow* (linear time) operations
  - *list*(*i*) returns the i^{th} element (starting from 0) of the list
  - *list*.last (or *list* last) returns the last element in the list
  - *list*.init (or *list* init) returns a list with the last element removed
    - This involves making a complete copy of the list
  - *list*.length (or *list* length) returns the number of elements in the list
  - *list*.reverse (or *list* reverse) returns a new list with the elements in reverse order
- In practice, the slow operations are hardly ever needed



Stepping through a list

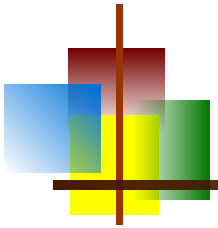
- ```
  def printList1(myList: List[Any]) {
      for (i <- 0 until myList.length) {
          println(myList(i))
      }
  }
  ```
  - What is the time complexity of this method?

- ```
  def printList2(myList: List[Any]) {
      if(! myList.isEmpty) {    // the dot is required here
          println(myList head)
          printList2(myList tail)
      }
  }
  ```
  - What is the time complexity of this method?



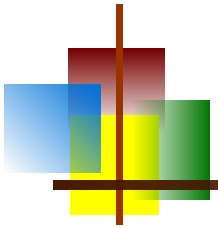
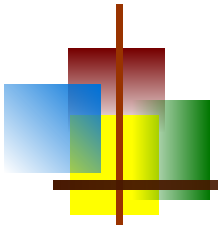
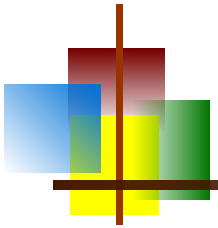
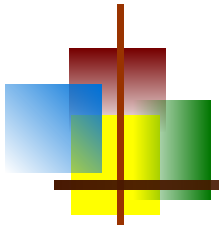
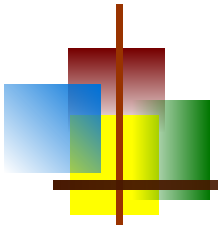
List construction with :: and Nil

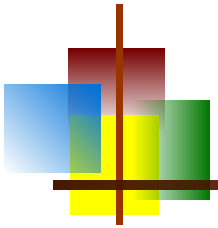
- Lists are homogeneous: All elements have the same type
  - However,
    ```
    scala> "abc" :: List(1, 2, 3)
    res15: List[Any] = List(abc, 1, 2, 3)
    ```
  - The newly-created list has a type which is the least upper bound
- An empty list has “nothing” in it
  - ```
    scala> List()
    res16: List[Nothing] = List()
    ```
- The “name” of the empty list is Nil
  - ```
    scala> Nil
    res17: scala.collection.immutable.Nil.type = List()
    ```
- Lists are built from Nil and the :: operator (which is right-associative)
  - ```
    scala> 1 :: 2 :: 3 :: Nil
    res18: List[Int] = List(1, 2, 3)
    ```
  - ```
    scala> 1 :: (2 :: (3 :: Nil))
    res19: List[Int] = List(1, 2, 3)
    ```



Basic recursion

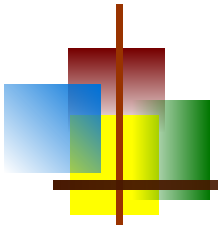
- Recursion is when a method calls itself
- Here's the basic formula for working with a list:
  - if the list is empty
    return some initial value (often an empty list)
    else
    process the head
    recur with the tail

- ```
  def printList2(myList: List[Any]) {
      if(! myList.isEmpty) {
          println(myList head)
          printList2(myList tail)
      }
  }
  ```
  ```



Again, with pattern matching

- Here's our same method again:
  - ```
    def printList2(myList: List[Any]) {
        if(! myList.isEmpty) {
            println(myList head)
            printList2(myList tail)
        }
    }
    ```
- Here it is with pattern matching:
  - ```
    def printList3(myList: List[Any]) {
        myList match {
            case h :: t =>
                println(myList head)
                printList3(myList tail)
            case _ =>
        }
    }
    ```



map

- map applies a one-parameter function to every element of a List, returning a new List
  - scala> List(1, 2, 3, 4) map (n => 10 * n)
    res0: List[Int] = List(10, 20, 30, 40)
- The result list doesn't have to be of the same type
  - scala> List(1, 2, 3, 4) map (n => n % 2 == 0)
    res1: List[Boolean] = List(false, true, false, true)
- Since an element of the list is the only parameter to the function, and it's only used once, you can abbreviate the function
  - scala> List(1, 2, 3, 4) map (10 * _ + 6)
    res2: List[Int] = List(16, 26, 36, 46)
- Of course, you don't have to use a literal function; you can use any previously defined function (yours or Scala's)
  - scala> List(-1, 2, -3, 4) map (_ abs)
    res3: List[Int] = List(1, 2, 3, 4)



flatMap

- flatten "flattens" a list (removes one level of nesting)
  - scala> val nested = List(List(1, 2, 3), List(4, 5))
    nested: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
  - scala> nested flatten
    res0: List[Int] = List(1, 2, 3, 4, 5)
- flatMap is like map, but the function given to flatMap is expected to return a list of values; the resultant list of lists is then "flattened"
- Syntax:
  - def map[B](f: (A) => B): List[B]
  - def flatMap[B](f: (A) => Traversable[B]): List[B]
- Example:
  - scala> val greeting = List("Hello".toList, "from".toList, "Scala".toList)
    greeting: List[List[Char]] = List(List(H, e, l, l, o), List(f, r, o, m), List(S, c, a, l, a))
  - scala> greeting map (word => word.toList)
    res2: List[List[Char]] = List(List(H, e, l, l, o), List(f, r, o, m), List(S, c, a, l, a))
    scala> greeting flatMap (word => word.toList)
  - res3: List[Char] = List(H, e, l, l, o, f, r, o, m, S, c, a, l, a)



filter

- filter is used to remove unwanted elements from a list, returning a new list
  - scala> List(1, -2, 3, -4) filter (_ > 0)
    res3: List[Int] = List(1, 3)
- There is a corresponding (less often used) filterNot method
  - scala> List(1, -2, 3, -4) filterNot (_ > 0)
    res4: List[Int] = List(-2, -4)



foldl, foldr

- The “fold” functions apply a binary operator to the values in a list, pairwise, starting from the left or starting from the right
  - ```
    scala> val list = List(10, 1, 2, 3)
    list: List[Int] = List(10, 1, 2, 3)
    ```
  - ```
    scala> list.foldLeft(0)(_ - _)
    res3: Int = -16
    ```
  - ```
    scala> list.foldRight(0)(_ - _)
    res4: Int = 8
    ```
  - ```
    scala> ((((0 - 10) - 1) - 2) - 3)
    res6: Int = -16
    ```
  - ```
    scala> (10 - (1 - (2 - (3 - 0))))
    res8: Int = 8
    ```
    ```

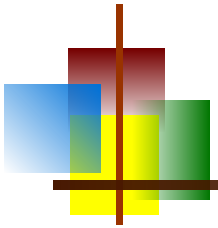
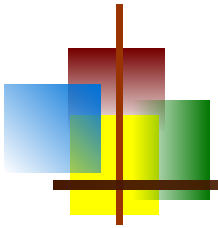


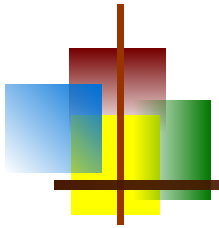
for

- Scala's for comprehension can be used like Java's for loop
  - scala> for (ch <- "abcde") print(ch + "*")
    a*b*c*d*e*
- The ch <- "abcde" is a generator; you can have more than one
  - scala> for { x <- 1 to 5
    | y <- 10 to 30 by 10 } print((x + y) + " ")
    11 21 31 12 22 32 13 23 33 14 24 34 15 25 35
  - The above needs braces, { }, not parentheses, ( )
- You can have definitions (not the same as declarations):
  - scala> for (i <- 1 to 10;
    | j = 100) print ((i + j) + " ")
    101 102 103 104 105 106 107 108 109 110
  - j = 100 is a definition
  - In this example, the semicolon preceding the definition is required
- You can also have guards:
  - scala> for (i <- 1 to 10
    | if i != 7) print(i + " ")
    1 2 3 4 5 6 8 9 10



Another for example

- You need to start with a generator, and after that you can have more generators, definitions, and guards
  - ```
    scala> for { i <- 1 to 5 if i % 2 == 0
         |       k = 100
         |       j <- 1 to 5
         |       if j * k < 450 } print((k + 10 * i + j) + " ")
    121 122 123 124 141 142 143 144
    ```
    ```



for-yield

- The value of a for comprehension, without a yield, is ()
- With a yield, the value is a list of results (one result for each time through the loop)
- The syntax is: for (*sequence*) yield *expression*
- Examples:
  - scala> for (i <- 1 to 5) yield 10 * i
    res12: scala.collection.immutable.IndexedSeq[Int] = Vector(10, 20, 30, 40, 50)
  - scala> for (n <- List("one", "two", "three")) yield n.substring(0, 2)
    res2: List[java.lang.String] = List(on, tw, th)



Another for-yield example

- Here's a more complete example (Odersky, p. 125):
  - ```
    val forLineLengths =
      for {
        file <- filesHere // 'filesHere' is an array of files
        if file.getName.endsWith(".scala")
        line <- fileLines(file)  // get an Iterator[String]
        trimmed = line.trim
        if trimmed.matches(".*for.*")
      } yield trimmed.length // get an Array[Int]
    ```
  - The above method:
    - gets each file from an array of files
    - considers only the file with the .scala extension
    - gets an iterator for the lines in the file
    - removes whitespace from the beginning and end of the line
    - looks for "for" within the line (using a regular expression)
    - counts the number of characters in the line
    - returns an array of line lengths of lines containing "for" in scala files
    ```



toList

- scala> Array(1, 2, 3, 4) toList
  res12: List[Int] = List(1, 2, 3, 4)

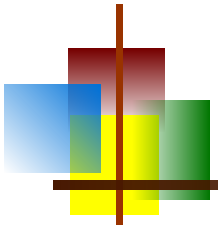
- scala> "abc" toList
  res13: List[Char] = List(a, b, c)

- scala> Map("apple" -> "red", "banana" -> "yellow") toList
  res14: List[(java.lang.String, java.lang.String)] = List((apple,red), (banana,yellow))

- scala> Set("abc", 123) toList
  res16: List[Any] = List(abc, 123)

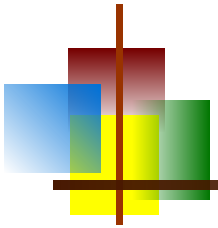
- scala> List(1, 2, 3) toList
  res17: List[Int] = List(1, 2, 3)

- Also: toArray, toString, toSet, toMap



Pattern matching

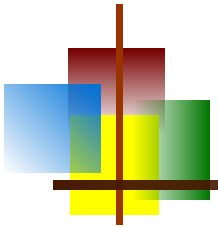
- Given this definition:
  - scala> val myList = List("a", "b", "c")
    myList: List[java.lang.String] = List(a, b, c)
- This works:
  - scala> val List(x, y, z) = myList
    x: java.lang.String = a
    y: java.lang.String = b
    z: java.lang.String = c
- But it's pretty useless unless you know the exact number of items in the list
- Here's a better way:
  - scala> val hd :: tl = myList
    hd: java.lang.String = a
    tl: List[java.lang.String] = List(b, c)



Example program

```
object EnglishToGerman {

   def main(args: Array[String]) {
     println(translate("Scala is a wonderful language !"))
   }

   def translate(english: String) = {
     val dictionary = Map("a" -> "ein", "is" -> "ist",
          "language" -> "Sprache", "wonderful" -> "wunderbar")
     def lookup(word: String) = {
       if (dictionary contains word) dictionary(word) else word
      }
      (english.split(" ") map (lookup(_))).mkString(" ")
   }
}
```

Output: `Scala ist ein wunderbar Sprache !`



The End