Outline

- Review
  - Functions, apply method, Tuples, Arrays as functions
  - Patterns, unapply method
  - Option, List
- List Examples
- For Comprehensions
- Scala 2.8 Collections
  - scala.collection package
Function Syntax: the apply method

- function types
  - 1 arg: \( X \Rightarrow Y \)
  - 2 args: \((X_1, X_2) \Rightarrow Y\)
  - \( n \) args: \((X_1, X_2, \ldots, X_n) \Rightarrow Y\)

- are type aliases for the generic types
  - Function1\([X,Y]\)
  - Function2\([X_1,X_2,Y]\)
  - Functionn\([X_1,X_2,\ldots,X_n,Y]\)

- each of these Function\(n\) classes defines an apply method:
  - \texttt{def apply}(arg1:X_1, arg2:X_2, \ldots argn:X_n) : Y
  - function application is usually written as \( f (x_1,x_2) \)
  - this is translated to \( f.apply(x_1,x_2) \)
Products and Tuples

- the Productn[A1,A2, ... An] trait models an n-dimensional cartesian product (for a given n)
- it defines $n$ get methods
  - _1 : A1
  - _2 : A2
  - _n : An
- the Tuplen case class provides the standard implementation

- shorthand for tuple types:
  - Tuple2[Int, Double] can be written as (Int, Double)
- for tuple values
  - Tuple2(20, 3.0) can be written as (20, 3.0)
Function Syntax: arrays as functions

- The predefined class `Array[A]` extends `Function1[Int,A]` and defines `apply (i: Int) : a`
  - To be array indexing
- So, given `val x : Array[A]`
  - We get elements via function application `x(i)`
- For array update, we can also write
  - `x (i) = rhs`
  - This is equivalent to a call `x.update (i, rhs)`
- The same syntax can be used for any (mutable) function type
  - Example: hash tables (see trait `collections.mutable.Map`)
  - You can also define your own by defining `apply`, and optionally, `update` methods
Pattern Matching: unapply

- can define pattern matching independently of case classes
- use so-called extractor objects
  - object with a one arg apply method
  - and an unapply method
    - which returns the argument of the apply method when (un)applied to its result

- also
  - a block of pattern matching cases can be used as a function
    \[
    \{ \text{case } P_1 \Rightarrow E_1 \ldots \text{ case } P_n \Rightarrow E_n \}\]
  - is equivalent to the anonymous function
    \[
    x \Rightarrow x \text{ match } \{ \text{case } P_1 \Rightarrow E_1 \ldots \text{ case } P_n \Rightarrow E_n \}\]
object Twice {
    def apply(x: Int) = x * 2
    def unapply(z: Int) =
        if (z%2 == 0) Some(z/2) else None
}

object TwiceTest {
    val x = Twice(21)
    x match {
        case Twice(n) => println(n)
    } // prints 21
Optional Results

- optional values of type A defined by type Option[A]
  - useful for returning a result when there is none!
  - in Scala you should avoid using a null as a "no-result"

```scala
final case class Some[+A](x: A) extends Option[A] {
  def isEmpty = false
  def get = x
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```
Lists in Scala

- basic immutable sequence structure
  - built from two constructors
    - Nil and :: (which associates to the right)
  - can also be built using the apply method of the List object
  - Nil and :: can be used in pattern matching

- also, in Scala, symbolic operators ending in `:` are applied to their right argument, so
  \[ x :: y = y ::\ (x) \]
- but, other operators are applied to their left argument
  \[ x + y = x .+\ (y) \]
val fruit: List[String] =
    List("apples", "oranges", "pears")

val nums : List[Int] =
    List(1, 2, 3, 4)

val diag3: List[List[Int]] =
    List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))

val empty: List[Int] = List()
val fruit: List[String] =
    "apples" :: "oranges" :: "pears" :: Nil

val nums : List[Int] =
    1 :: 2 :: 3 :: 4 :: Nil

val diag3: List[List[int]] =
    (1 :: 0 :: 0 :: Nil) ::
    (0 :: 1 :: 0 :: Nil) ::
    (0 :: 0 :: 1 :: Nil) ::
    Nil

val empty: List[Int] = Nil
Basic Methods on Lists

- `isEmpty`, `length`
- `head`, `tail`, `last`, `init`
- `take(n)`, `drop(n)`, `split(n)`, `apply(n)`
  ```python
def split(n: int): (List[A], List[A]) =
    (take(n), drop(n))
def apply(n: int): a = drop(n).head
  ```
- `list1.zip(list2)` yields a "zipped" list of pairs
- list concatenation `:::`
  - note that this symbolic method name ends in `:` so is right associative
Higher Order Methods on Lists

- transforming elements of a list in some way
- for `List[A]`,
  - `map[B](f: A => B) : List[B]`
    - construct result by applying map to each element of this list
  - special case, for functions with no (useful) result, do not produce a list result:
    - `foreach(f: A => Unit) : Unit`
Higher Order Methods on Lists

- extracting elements according to some criterion
- for `List[A]`,
  - `filter(p: A => Boolean) : List[A]`
    - result only includes this list's elements that satisfy `p`
  - special cases, with modified results:
    - `forall(p: A => Boolean) : Boolean`
    - `exists(p: A => Boolean) : Boolean`
Higher Order Methods on Lists

- accumulator for list elements using some operator
  - e.g. summing all elements
    
    ```scala
    def sum = foldLeft (0) ((x,y)=>x+y)
    ```

- for non-empty lists
  
  ```scala
  def reduceLeft(op: (a, a) => a): a
  def reduceRight(op: (a, a) => a): a
  ```

- for possibly empty lists, provide initial value
  
  ```scala
  def foldLeft[b](z: b)(op: (b, a) => b): b
  def /:[b](z: b)(f: (b, a) => b): b = foldLeft(z)(f)
  ```

  ```scala
  def foldRight[b](z: b)(op: (a, b) => b): b
  def :\[b](z: b)(f: (a, b) => b): b = foldRight(z)(f)
  ```
For Comprehensions

- given any structure that supports the underlying methods
  - map
  - flatMap
  - filter
  - foreach

- Scala provides special for loop syntax for generating new structures from existing ones
  - these are called for comprehensions
For Comprehension Syntax

- `for ( s ) yield e`
  - `s` generates values from some structure
    - with optional filtering
    - and further definition of local variables
  - `e` is an expression that depends on values bound by the generator `s`

- similar to expressiveness of SQL queries
  - but are applicable to any data structure that provide the key underlying methods
  - inspired by "monadic" comprehensions in Haskell
Example
with translation to filter / map

for (p <- persons if p.age > 20) yield p.name

- for each person in a list of persons
  - if the person's age is > 20
    - add the person's name to the resultant list

- it is equivalent to
  - the use of the following filter / map combination

persons filter (p => p.age > 20)
  map (p => p.name)
Rolling your Own

- by defining the required underlying methods for your classes
  - you can use for comprehensions for iterating over your structures
- many standard Scala types support for-comprehension
  - e.g. Option
    - iterating over an optional value generates an optional value
  - `for (i <- optional if i % 2 == 0) yield i`
    - will return `None`
      - if optional is `None` or is `Some i` where `i` is odd
    - will return `Some i`
      - otherwise
Collections in Scala

- collections package divided into
  - immutable
  - mutable
- immutable collections
  - add and remove methods produce "new" collections
  - because they are immutable, results may be shared
- walkthrough of Scala 2.8 New Collection Classes document
  - by Odersky at http://docs.scala-lang.org/overviews/collections/introduction.html
  - also see StackOverflow: http://stackoverflow.com/questions/1722137/scala-2-8-collections-design-tutorial
Homework

- read documentation on Scala Collections
  - see last slide for links
  - also browse scaladoc for Collections API
    - search for Scala Collections API

- next lecture
  - Parser Combinators in Scala