CompSci 220

Programming Methodology

02: Arrays, Maps, and Tuples

Today's Goals

- Assignments
- Loops
- For Comprehensions
- Functions
- Arrays
- Maps
- Tuples
- Scaladoc

Block Expressions and Assignments

- In Java and C++, a block statement is a sequence of statements enclosed in { }.
- You use a block statement whenever you need to put multiple actions in the body of a branch or loop statement.
- In Scala, a { } block contains a **sequence of expressions**, and the result is also an expression.
- The value of a block is the value of the last expression.

```
val distance = { val dx = x - x0
                 val dy = y - y0
                 sqrt(dx * dx + dy * dy) }
```

Assignments

- In Scala, assignments have no value…
- Well, strictly speaking, they have a value of type `Unit`.
- `Unit` is similar to `void` in Java and C++.
- A block that ends in an assignment, such as

  ```
  { r = r * n; n -= 1 }
  ```

  has a `Unit` value.

- This is not a problem, but it will come up again when we learn how to define functions.

Assignments (i-Clicker)

- What do you think this does?

```
var x = 2
var y = 4
x = y = 24
```

a) It assigns 24 to x and y.
b) It assigns 24 to y, but leaves x unchanged.
c) It assigns 24 to x, but leaves y unchanged.
d) It fails to compile.

Input and Output

- To print a value use `print` or `println`.

```
print(“Answer: “)
println(42)
```

- Yields the same output as,

```
println(“Answer: “ + 42)
```

Input and Output

- To read a line of input from the console use the `readLine` function.
- To read a numeric value use `readInt`, `readDouble`, `readByte`, `readShort`, `readLong`, `readFloat`, `readBoolean`, or `readChar`.

```
import scala.io.StdIn._
val name = readLine("Your name: ")
print("Your age: ")
val age = readInt()
println(s"Hello, $name! Next year you will be ${age + 1}")
```

Loops

- Scala has the same while and do loops as Java and C++.

```
while (n > 0) {
  r = r * n
  n -= 1
}
```

- Scala does not have a Java/C++ `for (`*`initialize; test; update`*`)` loop.
- Rather, you can use a for statement like this:

```
for (i <- 1 to n)
  r = r * i
```

Loop Patterns

- When traversing a string or array, you often need a range from 0 to n – 1. In that case, use the `until` method instead of the `to` method.

```
val x = "Hello"
var sum = 0
for (i <- 0 until s.length)
  sum += s(i)
```

- In the above example, it turns out there is a simpler way:

```
var sum = 0
for (ch <- "Hello") sum += ch
```

NOTE: In Scala, loops are not used as much as in other languages. As you will see later in the course, you can often process the values in a sequence by applying a function to all of them.

Advanced for Loops

- For loops can have multiple *generators*:

```
for (i <- 1 to 3; j <- 1 to 3) print((10*i*j)+" ")
// Prints 11 12 13 21 22 23 31 32 33
```

- Each generator can have a *guard*:

```
for (i <- 1 to 3; j <- 1 to 3 if i != j) print((10*i+j)+" ")
// Prints 12 13 21 23 31 32
```

- You can have any number of *definitions*:

```
for (i <- 1 to 3; from = 4-i; j <- from to 3) print((10*i+j)+" ")
// Prints 13 22 23 31 32 33
```

for Comprehensions

- When the body of the for loop starts with yield, then the loop constructs a collection of values, one for each iteration:

```
for (i <- 1 to 10) yield i % 3
```

- This type of loop is called a *for comprehension*.
- The generated collection is compatible with the first generator.

```
for (c <- "Hello"; i <- 0 to 1) yield (c+i).toChar
// Yields "Hieflmlmop"
```

for Comprehensions

- Because for comprehensions can get busy, there is alternate syntax using { } along with newlines to make it more clear:

```
for {
  i <- 1 to 3
  from = 4 – i
  j <- from to 3
} yield (10*i+j)+”“

// Yields Vector(13, 22, 23, 31, 32, 33)
```

Functions

- To define a function, you specify the function's name, parameters, and body like this:

```
def abs(x: Double) = if (x >= 0) x else -x
```

- If the body is more than one expression:

```
def fac(n: Int) = {
  var r = 1
  for (i <- 1 to n) r = r * i
  r
}
```

- There is no need for a return keyword

Recursive Functions

- With a recursive function you need to specify the return type:

```
def fac(n: Int): Int =
  if (n <= 0) 1 else n * fac(n - 1)
```

- Without the return type, the Scala compiler couldn't verify that the type of `n * fac(n - 1)` is an Int.

Default Arguments

- You can provide default arguments for functions that are used when you don't specify explicit values.

```
def decorate(str: String,
             left: String = "[",
             right: String = "]") =
  left + str + right
```

- If you call `decorate("Hello")` you get `"[Hello]"`.
- If you call `decorate("Hello", "<", ">")` you get `"<Hello>"`.

Named Arguments

- You can also specify the parameter names when you supply the arguments, like this:

```
decorate(left = "<<<", str = "Hello", right = ">>>")
```

Or like this

```
decorate("Hello", right = "]>>>")
```

Variable Arguments

- Sometimes, it is convenient to implement a function that can take a variable number of arguments.

```
def sum(args: Int*) = {
  var result = 0
  for (arg <- args) result += arg
  result
}
```

- You can call this function with as many arguments as you like:

```
sum(1, 4, 9, 16, 25)
```

Procedures

- Scala has a special notation for a function that returns no value. If the function body is enclosed in braces without a preceding = symbol, then the return type is `Unit`.
- The following procedure prints a string inside a box:

```
-------
|Hello|
-------
```

```
def box(s: String) {
  val border = "-" * s.length + "--\n"
  println(border + "|" + s + "|\n" + border
}
```

i-clicker question!

Which of the following types represent a Scala fixed-length array?

a) T[]
b) Array[T]
c) ArrayBuffer[T]
d) ArrayList[T]
e) List[T]

i-clicker question!

Which of the following types represent a Scala variable-length array?

a) T[]
b) Array[T]
c) ArrayBuffer[T]
d) ArrayList[T]
e) List[T]

i-clicker question!

If **arr** is a Scala array of type **Int**, what syntax is used to access an element at index **i** in the array **arr**?

a) arr.get(i)
b) arr(i)
c) arr.take(i)
d) arr[i]
e) arr -> i

Scala Arrays

- Scala has arrays!
- Java and C++ programmers often choose an array or a close relative (array list or vectors) when they need to collect a bunch of elements.
- In Scala, there are many choices, but for now we will start with Arrays.

Fixed-Length Arrays

- If you need an array whose length doesn't change, use the **Array** type in Scala.

```
val nums = new Array[Int](10)
  // An array of ten integers, all initialized with zero
val a = new Array[String](10)
  // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
  // Note: No new when you supply initial values
s(0) = "Goodbye"
  // Array("Goodbye", "World")
  // Use () instead of [] to access elements
```

Fixed-Length Arrays

- If you need an array whose length doesn't change, use the **Array** type in Scala.

```
val nums = new Array[Int](10)
  // An array of ten integers, all initialized with zero
val a = new Array[String](10)
  // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
  // Note: No new when you supply initial values
s(0) = "Goodbye"
  // Array("Goodbye", "World")
  // Use () instead of [] to access elements
```

Fixed-Length Arrays

- If you need an array whose length doesn't change, use the **Array** type in Scala.

```
val nums = new Array[Int](10)
  // An array of ten integers, all initialized with zero
val a = new Array[String](10)
  // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
  // Note: No new when you supply initial values
s(0) = "Goodbye"
  // Array("Goodbye", "World")
  // Use () instead of [] to access elements
```

Fixed-Length Arrays

- If you need an array whose length doesn't change, use the **Array** type in Scala.

```
val nums = new Array[Int](10)
  // An array of ten integers, all initialized with zero
val a = new Array[String](10)
  // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
  // Note: No new when you supply initial values
s(0) = "Goodbye"
  // Array("Goodbye", "World")
  // Use () instead of [] to access elements
```

Fixed-Length Arrays

- If you need an array whose length doesn't change, use the **Array** type in Scala.

```
val nums = new Array[Int](10)
  // An array of ten integers, all initialized with zero
val a = new Array[String](10)
  // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
  // Note: No new when you supply initial values
s(0) = "Goodbye"
  // Array("Goodbye", "World")
  // Use () instead of [] to access elements
```

i-clicker question!

What is the name of the class/type used in Java for variable-length arrays?

a) ExtensibleArray<T>
b) LinkedList<T>
c) ArrayList<T>
d) ArrayBuffer<T>
e) None of these

i-clicker question!

What is the name of the class/type used in Java for variable-length arrays?

a) ExtensibleArray<T>
b) LinkedList<T>
c) ArrayList<T>
d) ArrayBuffer<T>
e) None of these

Variable-Length Arrays

- Java has **ArrayList<T>** and **C++** has vector that grow and shrink on demand.
- The equivalent in Scala is called **ArrayBuffer[T]**.
- To use an **ArrayBuffer** you need to import it from Scala's mutable collections package.

```
import scala.collection.mutable.ArrayBuffer
```

ArrayBuffer: Creation

- It is easy to create a new empty ArrayBuffer

```
val b = ArrayBuffer[Int]()
  // Or new ArrayBuffer[Int]
  // An empty array buffer, ready to hold integers
```

ArrayBuffer: Appending an Element

- ArrayBuffer defines a method **+=** that appends elements to the end.

```
b += 1
  // ArrayBuffer(1)
  // Add an element at the end with +=
```

ArrayBuffer: Appending Many Elements

- The **+=** method is overloaded to allow many elements to be appended to the end of an ArrayBuffer.

```
b += (1, 2, 3, 5)
  // ArrayBuffer(1, 1, 2, 3, 5)
  // Add multiple elements at the end by enclosing them in parentheses
```

ArrayBuffer: Appending Collections

- The **++=** method is used to append one Collection to another.

```
b ++= Array(8, 13, 21)
  // ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
  // You can append any collection with the ++= operator
```

ArrayBuffer: Trimming

- The **trimEnd** method is used to remove elements at the end.

```
b ++= Array(8, 13, 21)
  // ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
  // You can append any collection with the ++= operator


b.trimEnd(5)
  // ArrayBuffer(1, 1, 2)
  // Removes the last five elements
```

- Adding/Removing elements at the end of an array buffer is an efficient constant time operation.

ArrayBuffer: insert/remove

```
b.insert(2, 6)
  // ArrayBuffer(1, 1, 6, 2)
  // Insert before index 2
b.insert(2, 7, 8, 9)
  // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
  // You can insert as many elements as you like
b.remove(2)
  // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
  // ArrayBuffer(1, 1, 2)
  // The second parameter tells how many elements to remove
```

- You can also insert and remove elements at specific locations.
- However, these operations require shifting all elements after the inserted or removed element – making them not as efficient.

ArrayBuffer: insert/remove

```
b.insert(2, 6)
  // ArrayBuffer(1, 1, 6, 2)
  // Insert before index 2
b.insert(2, 7, 8, 9)
  // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
  // You can insert as many elements as you like
b.remove(2)
  // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
  // ArrayBuffer(1, 1, 2)
  // The second parameter tells how many elements to remove
```

- You can also insert and remove elements at specific locations.
- However, these operations require shifting all elements after the inserted or removed element – making them not as efficient.

ArrayBuffer: insert/remove

```
b.insert(2, 6)
  // ArrayBuffer(1, 1, 6, 2)
  // Insert before index 2
b.insert(2, 7, 8, 9)
  // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
  // You can insert as many elements as you like
b.remove(2)
  // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
  // ArrayBuffer(1, 1, 2)
  // The second parameter tells how many elements to remove
```

- You can also insert and remove elements at specific locations.
- However, these operations require shifting all elements after the inserted or removed element – making them not as efficient.

ArrayBuffer: insert/remove

```
b.insert(2, 6)
  // ArrayBuffer(1, 1, 6, 2)
  // Insert before index 2
b.insert(2, 7, 8, 9)
  // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
  // You can insert as many elements as you like
b.remove(2)
  // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
  // ArrayBuffer(1, 1, 2)
  // The second parameter tells how many elements to remove
```

- You can also insert and remove elements at specific locations.
- However, these operations require shifting all elements after the inserted or removed element – making them not as efficient.

ArrayBuffer: insert/remove

```
b.insert(2, 6)
  // ArrayBuffer(1, 1, 6, 2)
  // Insert before index 2
b.insert(2, 7, 8, 9)
  // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
  // You can insert as many elements as you like
b.remove(2)
  // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
  // ArrayBuffer(1, 1, 2)
  // The second parameter tells how many elements to remove
```

- You can also insert and remove elements at specific locations.
- However, these operations require shifting all elements after the inserted or removed element – making them not as efficient.

Array and ArrayBuffer Conversion

- Often you want to build up an array, but you do not yet know how many elements you will need.
- In that case, make an ArrayBuffer the call the following:

```
b.toArray
  // Array(1, 1, 2)
```

- Likewise, if you need to extend an array you would do this:

```
a.toBuffer
```

Traversing Arrays and ArrayBuffers

- In Java and C++ there are several syntactic differences between arrays and array lists or vectors.
- Scala is more uniform – you can often use the same code for both.
- Here is how you traverse an array of array buffer with a for loop:

```
for (i <- 0 until a.length)
  println(i + ": " + a(i))
```

- The variable **i** goes from **0** to **a.length – 1**.

Traversing Arrays and ArrayBuffers

- In Java and C++ there are several syntactic differences between arrays and array lists or vectors.
- Scala is more uniform – you can often use the same code for both.
- Here is how you traverse an array of array buffer with a for loop:

```
for (i <- 0 until a.length)
  println(i + ": " + a(i))
```

The until method belongs to the RichInt class. It returns all numbers up to (but not including) the upper bound. For example:

```
0 until 10
  // Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

- The variable **i** goes from **0** to **a.length – 1**.

Looping Patterns: Skipping

- The construct

  ```
  for (i <- range)
  ```

  makes the variable ***i*** traverse all values of the range. In our case, the loop variable ***i*** assumes the values 0, 1, and so until ***a.length-1***.

- To visit every second element, let ***i*** traverse

  ```
  0 until (a.length, 2)
    // Range(0, 2, 4, ...)
  ```

Looping Patterns: Reversing / No Indexing

- To visit elements starting from the end of the array, traverse

```
(0 until a.length).reverse
  // Range(..., 2, 1, 0)
```

- If you don't need the array index in the loop body, visit the array elements directly, like this:

```
for (elem <- a)
  println(elem)
```

Transforming Arrays

- In the previous examples we saw how to manipulate arrays and array buffers in a style that resembles idioms from languages such as Java and C++.
- Scala, however, can go further in how arrays, array buffers, and many other collections can be transformed in a convenient and concise way.
- Such transformations do not modify the original array, rather they *yield* a new one.
- Transformations on collections is a common technique that is prolific in *functional programming*.
  We will see much more on this later in the semester…

Transforming Arrays: for comprehension

- Use a for comprehension to transform an array:

```
val a = Array(2, 3, 5, 7, 11)
val result = for (elem <- a) yield 2 * elem
  // result is Array(4, 6, 10, 14, 22)
```

- The for (…) yield loop creates a new collection of the same type as the original collection. If you started with an array, you get another array. If you started with an array buffer, you get another array buffer.

- The result contains the expressions after the yield, one for each iteration of the loop.

Common Algorithms: sum

- It is often said that a large percentage of business computations are nothing but computing sums and sorting. Fortunately, Scala has built-in functions for these tasks.

```
Array(1, 7, 2, 9).sum
  // 19
  // Works for ArrayBuffer too
```

- In order to use the **sum** method, the element type must be a numeric type: integral, floating-point, BigInteger, or BigDecimal.

Common Algorithms: min/max

- Similarly, the **min** and **max** methods yield the smallest and largest element in an array or array buffer.

```
ArrayBuffer("Mary", "had", "a", "little", "lamb").max
  // "little"
```

Deciphering Scaladoc

- Because Scala has a richer type system than Java, you may encounter some strange-looking syntax as you browse the Scala documentation.

- Fortunately, you don’t have to understand all the nuances of the type system to do useful work.

Scaladoc: 0 or more arguments

```
def append(elems: A*): Unit
```

- This method takes zero or more arguments of type A. For example, b.append(1, 7, 2, 9) appends four elements to b.

Scaladoc: TraversableOnce

```
def appendAll(xs: TraversableOnce[A]): Unit
```

- The xs parameter can be any collection with the **TraversableOnce** trait, the most general trait in the Scala collections hierarchy.
- Other common traits include **Traversable** and **Iterable**.
- All Scala collections implement these traits – simply think “any collection” when you see one of these.

Scaladoc: This

```
def += (elem: A): ArrayBuffer.this.type
```

- This method returns *this*, which allows you to chain calls. For example, **b += 4 -= 5**.

- When you work with an ArrayBuffer you can simply think of the method as

```
def += (elem: A): ArrayBuffer[A]
```

Scaladoc: Decoder Ring

- Take a look in Chapter 3 in Scala for the Impatient to see other common patterns in deciphering Scaladoc.

- As we progress with Scala your understanding of Scaladoc will improve and you will become more comfortable with navigating Scala's type system.

Multi-Dimensional Arrays

- Like in Java, multi-dimensional arrays are implemented as arrays of arrays.
- For example, a 2-dimensional array of **Double** values has the type **Array[Array[Double]]**.

- To construct such an array you would do this:

```
val matrix = Array.ofDim[Double](3, 4) // Three rows, four columns
```

- To access an element, use two pairs of parenthesis:

```
matrix(row)(column) = 42
```

i-clicker question!

Which of the following types represent a Scala fixed-length array?

a) T[]
b) Array[T]
c) ArrayBuffer[T]
d) ArrayList[T]
e) List[T]

i-clicker question!

Which of the following types represent a Scala variable-length array?

a) T[]
b) Array[T]
c) ArrayBuffer[T]
d) ArrayList[T]
e) List[T]

i-clicker question!

If **arr** is a Scala array of type **Int**, what syntax is used to access an element at index **i** in the array **arr**?

a) arr.get(i)
b) arr(i)
c) arr.take(i)
d) arr[i]
e) arr -> i

Scala Maps and Tuples

- A classic programmer's saying is,
  "If you can only have one data structure, make it a hash table"

- Hash tables – or more generally – maps are among the most versatile data structures. Scala makes it particularly easy to use them.

- **Maps are collections of key/value pairs.**
  Keys map to values

- **Tuples are aggregates of *n* objects, not necessarily of the same type.**
  A tuple of *n* objects is called an n-tuple.
  A tuple of 2 objects is called a pair.

Constructing a Map

- You can construct a map as

```
val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
```

- This constructs an *immutable* **Map[String, Int]**.

- If you want a *mutable* map, use

```
val scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
```

- Of course, you can import scala.collection.mutable.Map:

```
val scores = Map(“Alice” –> 10, “Bob” –> 3, “Cindy” –> 8)
```

Constructing Pairs

- The -> operator makes a pair, the value of

```
"Alice" -> 10
```

is

```
("Alice", 10)
```

- You could have similarly defined the map as

```
val scores = Map(("Alice", 10), ("Bob", 3), ("Cindy", 8))
```

Accessing Map Values

- The analogy between functions and maps is particularly close because you use the () notation to look up key values.

```
val bobsScore = scores("Bob") // Like scores.get("Bob") in Java
```

- If the map doesn't contain a value for the requested key, an exception is thrown.
- To check whether there is a key with a given value, do this

```
val bobsScore = if (scores.contains("Bob")) scores("Bob") else 0
```

- Since this is common there is a short cut that you see often:

```
val bobsScore = scores.getOrElse("Bob", 0)
```

Option Value

- The call map.get(key) returns an Option object that is either Some(value for key) or None.

```
val v = scores.get(“Bob”).getOrElse(0)
```

- We discuss this further when we look at functional programming techniques and how we handle exceptional control flow.

Updating Mutable Map Values

- In a mutable map, you can update a map value, or add a new one, with a () to the left of an = sign:

```
scores("Bob") = 10
```

- This either updates the existing value for the key “Bob”, or if the key does not exist it creates a new key/value pair in the map.
- You can also do this:

```
scores += ("Bob" -> 10, "Fred" -> 7)
scores -= "Alice"
```

Immutable Map Values

- You can't update an immutable map, but you can do this to create a newly constructed map:

```
val newScores = scores + ("Bob" -> 10, "Fred" -> 7)
```

- The newScores map contains the same associations as scores, except that "Bob" has been updated and "Fred" added.
- Instead of saving the result as a new value, you can use a var:

```
var scores = ...
scores = scores + ("Bob" -> 10, "Fred" -> 7)
```

Iterating Over Maps

- The following amazingly simple loop iterates over all key/value pairs of a map:

```
for ((k, v) <- map) process k and v
```

- The magic here is that you can use *pattern matching* in a Scala for loop. We will cover the mechanics of this when we talk about functional programming.
- If you need to just visit the key or values you do this:

```
scores.keySet
for (v <- scores.values) println(v)
```

Tuples

- A tuple value is formed by enclosing individual values in parenthesis:

```
(1, 3.14, "Fred")
```

  is a tuple of type **Tuple3[Int, Double, java.lang.String]**.
  Also written as:

```
(Int, Double, java.lang.String)
```

- If you have a tuple:

```
val t = (1, 3.14, "Fred")
```

  then you can access its components with the methods _1, _2, _3, …

```
val second = t._2 // Sets second to 3.14
```

Tuples and Pattern Matching

- You can use pattern matching, also known as destructuring assignment, to easily assign tuple contents to variables:

```
val (first, second, third) = t
```

- If you do not need the third part, you can leave it out:

```
val (first, second, _) = t
```