Programming Methodology

Control Abstraction

UMASSCS

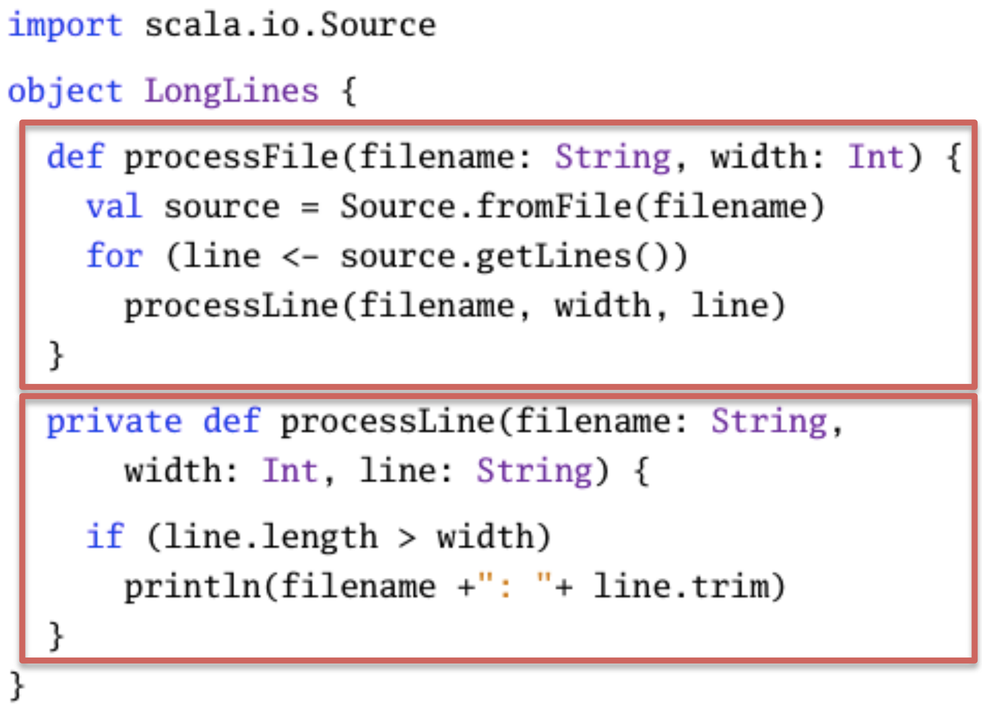
SCHOOL OF COMPUTER SCIENCE

Objectives

- **Functions and Closures (Review)**
  - Local Functions
  - Anonymous Functions
  - First-Class Functions
  - Function Literal Short Cuts
  - Partial Application
  - Closures
- **Reducing Code Duplication**
  - Simplify Libraries: Functional Composition
  - Simplify Client Code: Control Abstraction
- **Writing New Control Structures**
  - Currying
  - By-name Parameters

Methods

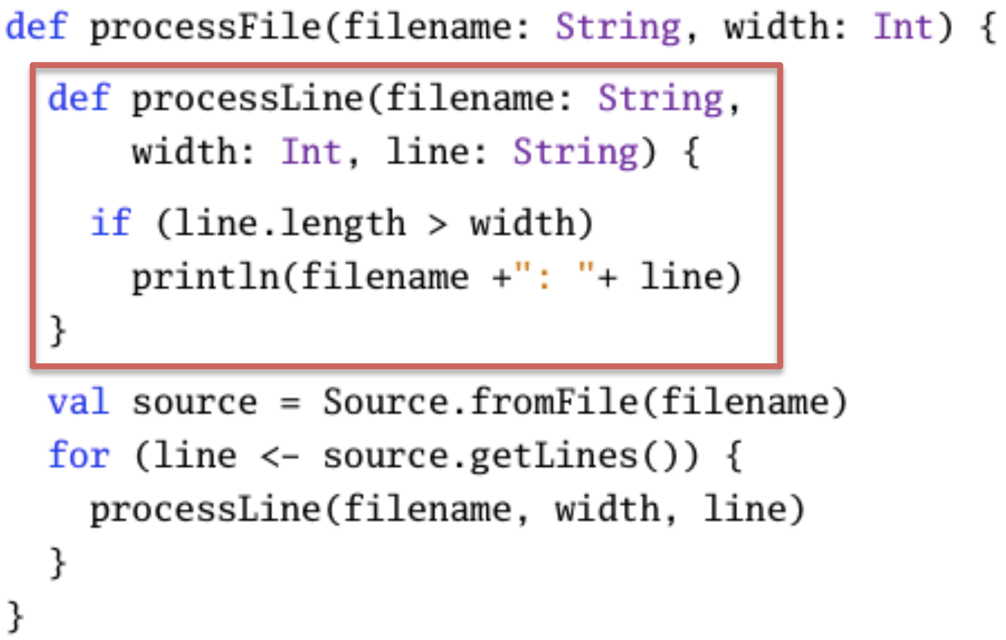
Defn: *A function that is defined as a member of some class/object.*



```
import scala.io.Source

object LongLines {
  def processFile(filename: String, width: Int) {
    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(filename, width, line)
  }
  private def processLine(filename: String,
      width: Int, line: String) {

    if (line.length > width)
      println(filename +": "+ line.trim)
  }
}
```

Local Functions

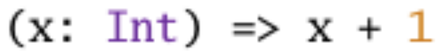
Defn: *A function that is defined inside other functions.*



```
def processFile(filename: String, width: Int) {
  def processLine(filename: String,
      width: Int, line: String) {
    if (line.length > width)
      println(filename +": "+ line)
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines()) {
    processLine(filename, width, line)
  }
}
```

Anonymous Functions

Defn: *A function definition that is not bound to an identifier.*



```
(x: Int) => x + 1
```

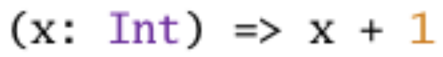
Parameters on the left.

Body on the right.

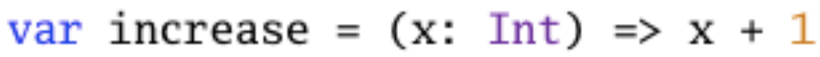
Anonymous Functions

Defn: *A function definition that is not bound to an identifier.*

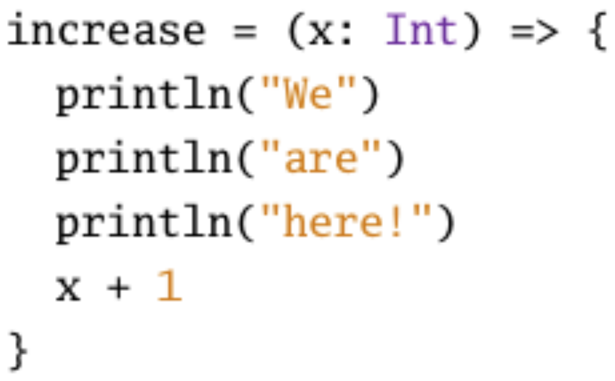
Also knows as a **function literal** or **lambda abstraction**.



```
(x: Int) => x + 1
```



```
var increase = (x: Int) => x + 1
```

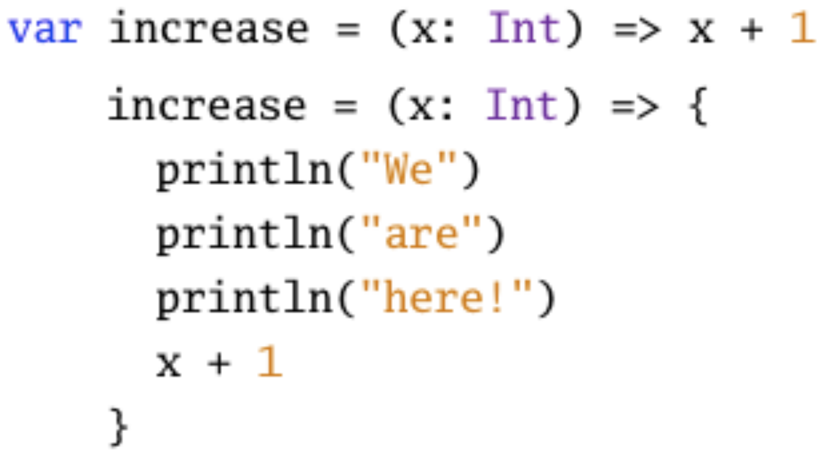


```
increase = (x: Int) => {
  println("We")
  println("are")
  println("here!")
  x + 1
}
```

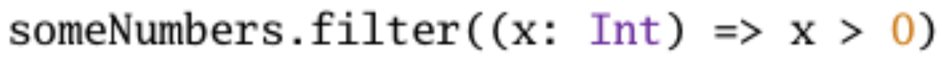
First-Class Functions

Defn: *Functions that are treated as values.*

- Functions that can be assigned to variables.
- Functions that are arguments to other functions.
- Functions that are return values.
- Functions that can be stored in data structures.



```
var increase = (x: Int) => x + 1

    increase = (x: Int) => {
      println("We")
      println("are")
      println("here!")
      x + 1
    }
```



```
someNumbers.filter((x: Int) => x > 0)
```

```
someNumbers.map(increase)
```

Function Literal Short Cuts

There are a number of different *short cuts* in writing function literals in Scala.

1. Elide parameter type

```
someNumbers.filter((x) => x > 0)
```

Function Literal Short Cuts

There are a number of different *short cuts* in writing function literals in Scala.

1. Elide parameter type
2. Elide parameter parens

```
someNumbers.filter((x) => x > 0)

someNumbers.filter(x => x > 0)
```

Function Literal Short Cuts

There are a number of different *short cuts* in writing function literals in Scala.

1. Elide parameter type
2. Elide parameter parens
3. Elide parameter name
   type is inferred

```
someNumbers.filter((x) => x > 0)

someNumbers.filter(x => x > 0)

someNumbers.filter(_ > 0)
```

Function Literal Short Cuts

There are a number of different *short cuts* in writing function literals in Scala.

1. Elide parameter type
2. Elide parameter parens
3. Elide parameter name
   type is inferred

```
someNumbers.filter((x) => x > 0)

someNumbers.filter(x => x > 0)

someNumbers.filter(_ > 0)

val f = _ + _
```

Function Literal Short Cuts

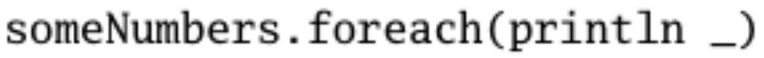
There are a number of different *short cuts* in writing function literals in Scala.

1. Elide parameter type
2. Elide parameter parens
3. Elide parameter name
   type is inferred

```
someNumbers.filter((x) => x > 0)

someNumbers.filter(x => x > 0)

someNumbers.filter(_ > 0)

val [illegible] + _

val f = (_: Int) + (_: Int)
```

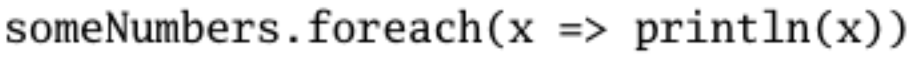
Partial Application

Defn: *Partial application is the process of fixing a number of arguments to a function to produce a new function with the same of fewer parameters.*



```
someNumbers.foreach(println _)
```

Which is the same as…

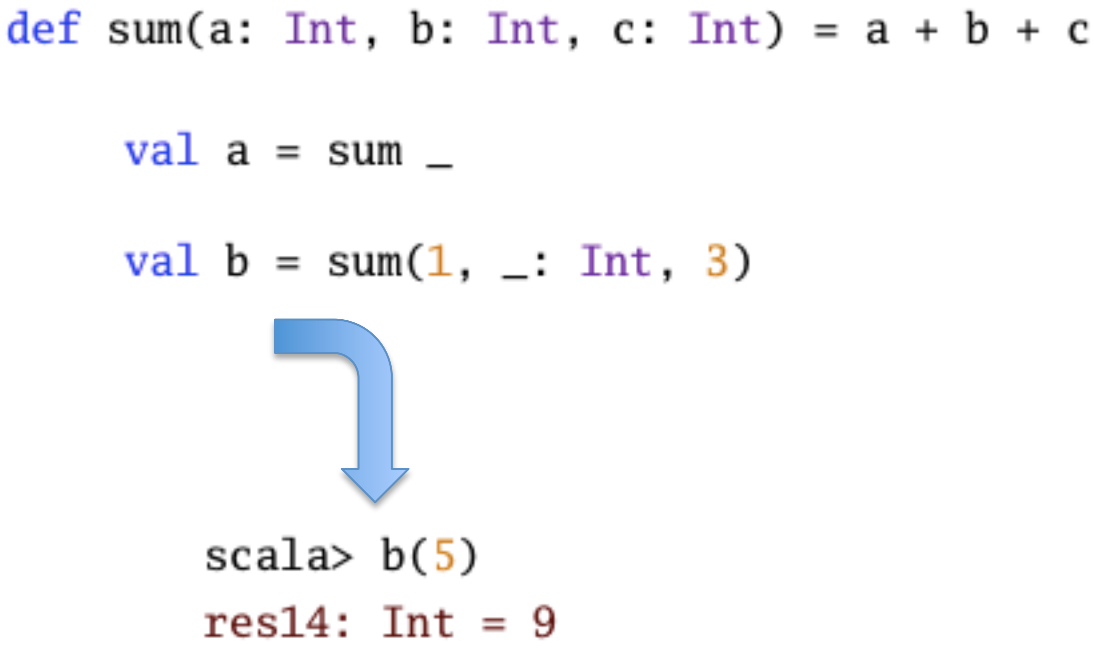


```
someNumbers.foreach(x => println(x))
```

Partial Application

Defn: *Partial application is the process of fixing a number of arguments to a function to produce a new function with the same of fewer parameters.*

This example makes it more clear:



```
def sum(a: Int, b: Int, c: Int) = a + b + c

    val a = sum _

    val b = sum(1, _: Int, 3)
```

```
scala> b(5)
res14: Int = 9
```

Closure

Defn: *A function literal that "captures" the bindings of its "free" variables from an outer scope.*

```
def makeIncreaser(more: Int) =
  (x: Int) => x + more

scala> val a = makeIncreaser(2)
scala> a(2)
4
scala> val b = makeIncreaser(3)
scala> b(5)
8
```

Repeated Parameters

Addition Scala Details

Scala allows you to indicate that the last parameter to a function may be repeated.

This is known as a *variadic function* – a function in which its *arity* is indefinite

```
def echo(args: String*) =
  for (arg <- args) println(arg)

scala> echo()

scala> echo(“hello”, “world”)
hello
world
```

Repeated Parameters

Addition Scala Details

Scala allows you to indicate that the last parameter to a function may be repeated.

This is known as a *variadic function* – a function in which its *arity* is indefinite

```
def echo(args: String*) =
  for (arg <- args) println(arg)

scala> echo()

scala> echo(“hello”, “world”)
hello
world

val arr = Array(“hello”, “world”)
scala> echo(arr)
```

Repeated Parameters

Addition Scala Details

Scala allows you to indicate that the last parameter to a function may be repeated.

This is known as a *variadic function* – a function in which its *arity* is indefinite

```
def echo(args: String*) =
  for (arg <- args) println(arg)

scala> echo()

scala> echo(“hello”, “world”)
hello
world

val arr = Array(“hello”, “world”)
scala> echo(arr)

scala> echo(arr: _*)
```

Named Arguments

Addition Scala Details

Scala allows you to name the parameters so you can invoke a function with its parameters in any order.

```
def speed(dist: Float, time: Float) =
  distance / time

scala> speed(100, 10)
10.0

scala> speed(time = 10, dist = 100)
10.0

scala> speed(dist = 100, time = 10)
```

Default Parameter Values

Addition Scala Details

Scala allows you to provide default parameters.

The argument for such a parameter can optionally be omitted from a function call.

```
def printTime(out: PrintStream =
                   Console.out) =
  out.println(“time = “ +
    System.currentTimeMillis())

scala> printTime()
```

Default Parameter Values

Addition Scala Details

Scala allows you to provide default parameters.

The argument for such a parameter can optionally be omitted from a function call.

```
def printTime(out: PrintStream =
                Console.out) =
  out.println(“time = “ +
    System.currentTimeMillis())

scala> printTime()

def printTime2(
      out: PrintStream = Console.out,
      divisor: Int = 1) =
  out.println(“time = “ +
    System.currentTimeMillis()/divisor)

scala> printTime2
scala> printTime2(divisor = 1000)
```

Reducing Code Duplication

All functions are separated into **common parts**, which are the same in every invocation of the function, and **non-common parts**, which may vary from one function invocation to the next.

- Common parts exist in the body of the code.
- Non-common parts are supplied by parameters.

Sometimes, these non-common parts could in fact be algorithms themselves – **higher-order functions** (functions that take functions as parameters) give you opportunities to reduce code duplication.

File Matching

Imagine you are writing a file browser and you want to provide an API that allows users to search for files matching some criterion:

- Search for files with names ending with a particular string.
- Search for files with names containing a string.
- Search for files with names matching a regular expression.

func-comp/src/main/scala/cs220/files

API Implementation

This example demonstrates that *higher-order* functions can help reduce code duplication as you **implement** an API.

Another important use of *higher-order* functions is to use them with the API itself to make **client code** more concise – thus reducing code duplication for the client of the API.

Client API Usage

What if we wanted to determine if a list contains a negative number.

Here is one way of doing it…

How could we change this to take advantage of the client API of the List class to simplify this code?

```
def containsNeg(nums: List[Int]) = {
  var exists = false
  for (num <- nums)
    if (num < 0)
      exists = true
  exists
}
```

func-comp/src/main/scala/cs220/client

Control Abstraction

The List class (and many others from the Scala collection classes) provide methods that are abstractions of typical control flow.

At the same time, these methods are clearly **methods**.

Is it possible to write methods that look like regular control flow constructs?

```
def containsNeg(nums: List[Int]) = {
  var exists = false
  for (num <- nums)
    if (num < 0)
      exists = true
  exists
}

def containsNeg(nums: List[Int]) =
  nums.exists(_ < 0)
```

func-comp/src/main/scala/cs220/client

Currying

To create our own control structures we must first understand currying…

Defn: Currying is the processes of evaluating a function that takes multiple parameters into evaluating a sequence of functions where each take a single parameter.

```
def a(x, y, z) = x + y + z
```

```
def b(x) = (y) => (z) => x + y + z

b(1, 2, 3) // evaluates to 6
a(1)(2)(3) // evaluates to 6
```

func-comp/src/main/scala/cs220/curry

Operating on Resources

We saw the filesMatching function that defined a very specialized control pattern.

Consider a more widely used coding pattern:

1. Open a resource
2. Operate on a resource
3. Close the resource

```
type Operation = PrintWriter => Unit
def withPrintWriter(file: File, op: Operation) {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}

withPrintWriter(
  new File(“date.txt”),
  writer => writer.println(new Date)
)
```

func-comp/src/main/scala/cs220/control/Control01.scala

Using {} instead of ()

Although this looks ok, we would really prefer to use {} instead of () to make it look more like a “control” construct.

Turns out, Scala allows single parameter functions to use {} instead of ()!

```
withPrintWriter {
  new File(“date.txt”),
  writer => writer.println(new Date)
}
```

```
scala> println(“Hello, World”)
Hello, World

scala> println { “Hello, World” }
Hello, World
```

Using {} instead of ()

Although this looks ok, we would really prefer to use {} instead of () to make it look more like a “control” construct.

Turns out, Scala allows single parameter functions to use {} instead of ()!

What about this?

```
withPrintWriter {
  new File(“date.txt”),
  writer => writer.println(new Date)
}
```

```
scala> println(“Hello, World”)
Hello, World

scala> println { “Hello, World” }
Hello, World

scala> “Hello, World”.substring { 7, 9 }
```

What do you think?

So, if Scala only allows functions with single parameters to use {} instead of ()…

What do you think is a possible approach to making this work?

func-comp/src/main/scala/cs220/control/Control02.scala

Currying for Control

So, the use of currying along with Scala’s special treatment of single parameter functions gets us closer to something that looks like a control structure provided by the language.

How might we implement something without an argument to look like if?

```
type Operation = PrintWriter => Unit

def withPrintWriter(file: File)(op: Operation) {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}

val file = new File(“date.txt”)
withPrintWriter(file) {
  writer => writer.println(new Date)
}
```

By-Value Parameters

Arguments to functions are typically passed by **value**. That is, they are evaluated before they are passed to the function body.

```
def add(x: Int, y: Int) = x + y
```

```
add(1+2+3, 4) => add(6, 4) => 10
```

By-Name Parameters

Another possibility is to not evaluate the arguments before they are passed to the function, rather, they are evaluated inside of the function body. This is called passing by **name**.

```
def add(x: Int, y: Int) = x + y
```

```
add(1+2+3, 4) => 1+2+3+4 => 10
```

Asserting Control

What if we wanted to implement an *assert* statement that only executes its argument if assertions are enabled?

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
```

func-comp/src/main/scala/cs220/control/Control03.scala

Asserting Control

What if we wanted to implement an *assert* statement that only executes its argument if assertions are enabled?

Well, this is a little awkward!

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError

myAssert(() => 5 > 3)
```

func-comp/src/main/scala/cs220/control/Control03.scala

Asserting Control

What if we wanted to implement an *assert* statement that only executes its argument if assertions are enabled?

Well, this is a little awkward!

Can we do this?

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError

myAssert(() => 5 > 3)

myAssert(5 > 3)
```

func-comp/src/main/scala/cs220/control/Control04.scala

Asserting Control

Ok, this is cool – but, how can I create a control abstraction that looks like a built-in **if statement**?

```
var assertionsEnabled = true

def myAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError

myAssert {
  5 > 3
}
```

func-comp/src/main/scala/cs220/control/Control04.scala

Asserting Control

Ok, this is cool – but, how can I create a control abstraction that looks like a built-in **if statement**?

Imagine, we want to have something like this

```
unless(5 > 4) {
  println(“I AM AWESOME”)
}
```

func-comp/src/main/scala/cs220/control/Control05.scala