Scala Course

STUDENT GUIDE

Instructor: Ed Robinson, PhD

Table of Contents

Why Scala?

- Multiparadigm
- Fully supports functional programming
- Functions are first-class citizens
- Everything is an object
    - Every value is an object
    - Every function is a value (is an object)
- Less Code
    - Type-safe language - statically typed
    - Built-in type inference engine
    - Clean code without redundant type information where compiler can deduce the type by the context where it is used
- Can interact with java code and runs on JVM

http://www.scala-lang.org/api/current/index.html
http://docs.scala-lang.org/index.html

Setup Development Environment

- Setup a basic Scala development environment
- Setup IntelliJ IDEA

Exercise - Scala Setup

Setup Scala.
https://www.scala-lang.org/download/

Exercise - IntelliJ

Setup IntelliJ Community Edition on your machine.
https://www.jetbrains.com/idea/download/#section=mac
https://www.jetbrains.com/idea/download/#section=windows

Exercise - Scala Open Source Project

Setup an IntelliJ project with the open source code for Scala.
https://github.com/scala/scala

1. Open a terminal window (shell) in a directory where you would like to keep your Scala projects.

2. Clone the open source code for Scala.
   git clone https://github.com/scala/scala.git

3. Open the Scala project in IntelliJ.

4. Explore. See https://www.scala-lang.org/api/current/index.html for the Scala standard library documentation.

REPL (Read-Evaluate-Print Loop)

```
All commands can be abbreviated, e.g., :he instead of :help.

:completions <string>    output completions for the given string
:edit <id>|<line>        edit history
:help [command]          print this summary or command-specific help
:history [num]           show the history (optional num is commands to show)
:h? <string>             search the history
:imports [name name ...] show import history, identifying sources of names
:implicits [-v]          show the implicits in scope
:javap <path|class>      disassemble a file or class name
:line <id>|<line>        place line(s) at the end of history
:load <path>             interpret lines in a file
:paste [-raw] [path]     enter paste mode or paste a file
:power                   enable power user mode
:quit                    exit the interpreter
:replay [options]        reset the repl and replay all previous commands
:require <path>          add a jar to the classpath
:reset [options]         reset the repl to its initial state, forgetting all session entries
:save <path>             save replayable session to a file
:sh <command line>       run a shell command (result is implicitly => List[String])
:settings <options>      update compiler options, if possible; see reset
:silent                  disable/enable automatic printing of results
:type [-v] <expr>        display the type of an expression without evaluating it
:kind [-v] <type>        display the kind of a type. see also :help kind
:warnings                show the suppressed warnings from the most recent line which had any
```

- Type ‘Scala’ at the command line in a terminal window

```
~ $ Scala
Welcome to Scala 2.12.1 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_121).
Type in expressions for evaluation. Or try :help.

Scala> :help
```

Scala Fundamentals

Variables

- Immutable “val” -> (use immutable whenever possible)

```
Scala> val y = 3
y: Int = 3

Scala> y = 4
<console>:12: error: reassignment to val
       y = 4
         ^
```

- Mutable “var” (use sparingly)

```
Scala> var x = 3
x: Int = 3

Scala> x = 4
x: Int = 4
```

- Specify type explicitly using “:” after variable name

```
Scala> val name: String = “Betty"
Scala> name.getClass
res1: Class[_ <: String] = class java.lang.String
```

REPL

```
All commands can be abbreviated, e.g., :he instead of :help.
:edit <id>|<line>        edit history
:help [command]          print this summary or command-specific help
:history [num]           show the history (optional num is commands to show)
:h? <string>             search the history
:imports [name name ...] show import history, identifying sources of names
:implicits [-v]          show the implicits in scope
:javap <path|class>      disassemble a file or class name
:line <id>|<line>        place line(s) at the end of history
:load <path>             interpret lines in a file
:paste [-raw] [path]     enter paste mode or paste a file
:power                   enable power user mode
:quit                    exit the interpreter
:replay [options]        reset the repl and replay all previous commands
:require <path>          add a jar to the classpath
:reset [options]         reset the repl to its initial state, forgetting all
session entries
:save <path>             save replayable session to a file
:sh <command line>       run a shell command (result is implicitly =>
List[String])
```

- Type :help
- Use tab (or esc) key for completion. ([tab][tab] suggests more)

Exercise - REPL

Practice using the following REPL commands.

- :history 5
- :h? val
- :save filename
- :reset
- :load filename
- :replay

Scala Fundamentals Cont’d

Data types

- Scala and Java basic types have the same range and meanings

- Basic types
  - Byte - 8-bit signed two's complement integer (-2^7 to 2^7 - 1)
  - Short - 16-bit signed two's complement integer (-2^15 to 2^15 - 1)
  - Int - 32-bit signed two's complement integer (-2^31 to 2^31 - 1)
  - Long - 64-bit signed two's complement integer (-2^63 to 2^63 - 1)
  - Char - 16-bit unsigned Unicode character (0 to 2^16 - 1)
  - String - sequence of Chars
  - Float - 32-bit IEEE 754 single-precision float
  - Double - 64-bit IEEE 754 double-precision float
  - Boolean - true or false

- Char and String literals - ‘c’, “hello”, … (same as Java)

- Use the escape character ‘\’ for special characters, “\’”, “\t”

```
Scala> val special = "Left \t\t\t Right"
special: String = Left              Right
```

- Raw Strings = """ … “""

```
Scala> val rawExample = """Hello "Bobby"/'abc' are usually
escaped."""
rawExample: String = Hello "Bobby"/'abc' are usually escaped.
```

- String interpolation -> use ’s’ and ‘$’

```
Scala> val person = "Tom Turkey"
person: String = Tom Turkey

Scala> val greeting = s"Mr. $person"
greeting: String = Mr. Tom Turkey
```

- Formatted String interpolation -> use ’f’

```
Scala> println(f"${math.Pi}%3.30f")
3.141592653589793000000000000000

Scala> println(f"${math.Pi}%3.3f")
3.142
```

http://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

- Value Casting - value types can be cast from Byte to Short to Int to Long to Float to Double and a Char can be cast to an Int

```
Scala> val x = 10
x: Int = 10

Scala> val y: Float = 10
y: Float = 10.0

// but not down casting
Scala> val z: Int = y
<console>:12: error: type mismatch;
 found   : Float
 required: Int
       val z: Int = y

Scala> val xxx:Int = 'c'
xxx: Int = 99
```

Operators

- Scala operators are actually methods

- Type “1.<tab>” on the command line to see all the methods for type Int. Use the + method as shown below.

```
Scala> 1.+(3)
res6: Int = 4
```

- Infix operators also include other methods like “indexOf” for String

```
Scala> val name = "Tom Turkey"
name: String = Tom Turkey

Scala> name indexOf 'u'
res1: Int = 5
```

- Scala methods that take a single argument can be used as infix operators

```
Scala> "Turkey".contains("r")
res0: Boolean = true

Scala> "Turkey" contains 'r'
res2: Boolean = true
```

- Scala methods that take multiple arguments can also be used as infix operators if the args are in parenthesis

```
Scala> "Turkey" subSequence (1,3)
res7: CharSequence = ur
```

- Numerical, logical, and bitwise operators are basically the same as Java

- Operator precedence is the same as Java

- Rich wrapper classes extend basic Java types making them better.

Exercise - Rich Types

1. Open and inspect the following Rich types in IntelliJ from the open source Scala project previously set up.

    - Scala.runtime.RichByte,
    - Scala.runtime.RichShort
    - Scala.runtime.RichInt
    - Scala.runtime.RichLong

Functions

- Scala functions are close to methods except they don't belong to a class. They are complete objects by themselves.
  - Note: methods start with the keyword "def" - to be covered in the OO section later in this course

- Functions are values with a type and can be passed around just like any other variable.

- Function names can be made up of characters like:
  +, ++, ~, &,-, --, \, /, :

- Functions have supporting methods (i.e., apply, andThen, compose, …) see https://www.scala-lang.org/api/current/scala/index.html Function1 definition

```
// function with a side effect – it's value is Unit
Scala> val f1 = () => println("Hello")
f1: () => Unit = $$Lambda$1492/1072723231@79a945d3

Scala> f1.apply
Hello
```

- () => xxx or "LEFT arrow RIGHT" means "take LEFT then do RIGHT" or "goes to" or "implies" or "transforms"
  - Functions and methods don't need an explicit return value however they actually return a value. It's best to use an explicit return type in the function definition line for external APIs.

- Pure Functions do not introduce side effects.

```
// function without side effect

// explicit declaration
scala> val greeting : String => String = (name: String) =>
"Hello " + name
greeting: String => String = $$Lambda$1237/1589547066@7c88d04f

// implicit redeclaration
scala> val greeting = (name: String) => "Hello " + name
greeting: String => String = $$Lambda$1238/1040647394@14d298d6

Scala> greeting("Betty")
res1: String = Hello Betty

Scala> println(greeting("Betty"))
Hello Betty
```

- Pure functions are **Referentially Transparent** - *they can be replaced with its corresponding value without changing the program's behavior*. For example 2 + 3 can be replace by 5 in the program and nothing would change.
    - Enforces the substitution model
- Procedures are impure functions that produce side effects whereas a pure function produces a result from an input with no side effects. Procedures and functions look identical in a Scala declaration.

Looping, Branching, Flow Control

- if, while, do while, for, try/catch, match
    - All return a value
- Minimize using temporary variables to hold values

If conditional

- Returns a value
- Use ‘if’ like a ternary op in Java to store results in a **val** instead of a **var**

```
// OLD JAVA WAY but uses var instead of val
var arg1 = "Empty"
if (!args.isEmpty)
  arg1 = args(0)
println(arg1)

// BETTER uses val which is better than var
val arg2 = if(!args.isEmpty) args(0) else "default.txt"
println(arg2)

// BEST - eliminate the placeholder val if only used once
println(if (!args.isEmpty) args(0) else “default.txt")
```

Exercise - If conditional

1. Create a new IntelliJ Scala project, e.g. LoopingAndBranching.
2. Create a new Scala Object - IfConditional
3. Create a "def main"
   ```
   def main(args: Array[String]): Unit = { }
   ```
4. Create a "max10" as a pure function that takes an Int, returns the number. If it's over 10 then it returns 10.
5. Create some test calls to test max10.

While and Do/While loops

- Behave the same as in other languages
- Returns Unit as while loops are all about side effects
    - Unit is a subtype of Scala.AnyVal. It is similar in concept to void in Java

while loop example:

```
var x = 0
while(x<5) {
  println(x)
  x = x + 1
}
```

do/while loop example:

```
var x = 0
do {
  println(x)
  x = x + 1
} while(x < 5)
```

- While loops are used to update vars or do I/O. For functional programming try to limit their use if possible.

Exercise - While Loops

1. Create a new Scala object named WhileLoops in your LoopingAndBranching app.
2. Modify the WhileLoops object to extend ‘App’
   ```
   object WhileLoops extends App {  …
   ```
3. Create a function named ‘printNTimes’ that takes two arguments, the string and the number of times. The function prints the string on new lines n times.

For expressions

- Similar to other languages
- Use a “generator <-“ and a range to iterate over collections

```
val digits = 0 to 9
for(i <- 0 to digits.length) println(i)  // Error: off by one
for(i <- 0 until digits.length) println(i)
```

- Or better yet, eliminate the temporary val digits

```
for (x <- 1 to 10) println(x)
for (x <- 1 until 10) println(x)
```

Exercise - For expressions

1. Create a new Scala object named ForExpression in your LoopingAndBranching app.
2. Modify the ForExpression object to extend ‘App’
3. Create a function named ‘printNTimes’ that takes two arguments, the string and the number of times. The function prints the string on new lines n times. Use a for expression.

For expressions with filters

- Filters can be in the parentheses

```
    val files = List("Person.Scala", "Address.Scala",
"PersonController.Scala",     "AddressController.Scala",
"pic.png")
    for (
      file: String <- files
      if file.endsWith("Scala")
      if file.contains("Controller")
    ) println(file)
```

Exercise - For expressions with filters

1. Create a new Scala object named ForExpressionFiltered in your LoopingAndBranching app.
2. Modify the ForExpressionFiltered object to extend ‘App’
3. Create a function named ‘printOddsIn’ that takes one argument as a Range. The function prints the odd numbers. Use a for expression with a filter.

For-Comprehensions in Scala

- For-Comprehensions let you generate sequences inside the for parenthesis. Yield creates the resulting collection.
    - The <- “binds” the value to the variable

- **for** (enumerators) **yield** e
    - enumerators refers to a semicolon-separated list of enumerators - generators and/or filters

Using generators only:

```
Scala> for (i <- 1 to 10; j <- 1 to i) yield (i, j)
res6: Scala.collection.immutable.IndexedSeq[(Int, Int)] =
Vector((1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), (4,2),
(4,3), (4,4), (5,1), (5,2), (5,3), (5,4), (5,5), (6,1), (6,2),
(6,3), (6,4), (6,5), (6,6), (7,1), (7,2), (7,3), (7,4), (7,5),
(7,6), (7,7), (8,1), (8,2), (8,3), (8,4), (8,5), (8,6), (8,7),
(8,8), (9,1), (9,2), (9,3), (9,4), (9,5), (9,6), (9,7), (9,8),
(9,9), (10,1), (10,2), (10,3), (10,4), (10,5), (10,6), (10,7),
(10,8), (10,9), (10,10))
```

Using generators and a filter:

```
Scala> for (i <- 1 to 10; j <- 1 to i; if j % 2 == 0) yield (i,
j)
res1: Scala.collection.immutable.IndexedSeq[(Int, Int)] =
Vector((2,2), (3,2), (4,2), (4,4), (5,2), (5,4), (6,2), (6,4),
(6,6), (7,2), (7,4), (7,6), (8,2), (8,4), (8,6), (8,8), (9,2),
(9,4), (9,6), (9,8), (10,2), (10,4), (10,6), (10,8), (10,10))
```

- For-Comprehensions combined with Futures enable parallelism (see Futures section on day 2 for how to do this).

Exercise - For Comprehensions

1. Create a new Scala object named ForComprehension in your LoopingAndBranching app.
2. Modify the ForComprehension object to extend ‘App’
3. Create a pure function named ‘oddsIn(range:Range)’ that takes one argument as a Range and generates the odd numbers from within the range. Use a for comprehension with a filter.

Match Statements

- Similar to other languages switch statements, but with very powerful matching for case statements

```
val x: Any = 1
x match {
  case "one" => println("ONE of type string”)
 case 1 => println("one of type integer")
  case "two" => println("two")
}
```

- No break statement is needed in Scala as the break is inferred. There is no fall through.

- There are no “break” or “continue” statements in Scala

- You can also use a conditional filter on a case statement

Exercise - Match Statements

1. Create a new Scala object named MatchStatements in your LoopingAndBranching app.
2. Modify the MatchStatements object to extend ‘App’
3. Create a pure function named ‘parseMessage’ that takes one argument as a String. The function tests messages for all lower case, or all upper case, or an empty string. Return an appropriate message for each case. Hint: use if conditions in case statements, e.g., case string if(…) => …
4. Be sure to check for the “all others” case using “case _ => …” as the last statement or you will get a runtime error for certain cases.

Try / Catch / Finally

- Scala has exception handling similar to Java other languages. However, there is a better, functional approach described later

```
try {
      val f = new FileReader(“myFile")
      // do something here
     …
   } catch {
      case ex: FileNotFoundException => // Handle missing file
      case ex: IOException => // Handle other I/O error
   }  finally {
     // optional clean up code
   }
```

Classes and Objects

Classes

- Classes are a blueprint for objects (instances of a class)

- Instance variables
    - var x = 0
    - val birthday = “Jan. 1”

- Methods - encapsulated inside of an object
    - def methodName(args): return type = {…}
    - Don’t need an explicit return
    - Recommended style - avoid using multiple explicit returns - refactor if necessary. (don’t use return statements)
    - For single line methods you don’t need the curly braces

```
class Person(val firstName: String = "joe",
             val lastName: String = "schmoe",
             var age: Int = 0)  {

  def fullName(): String = {
    firstName + " " + lastName
  }

  override def toString(): String = {
    firstName + " " + lastName + ", Age: " + age
  }
}
```

- Use “new” to create a new instance

```
val me = new Person(firstName="Bobby", "Bender", 10)
```

- Return type doesn’t have to be explicitly declared if compiler can determine type from the method implementation

```
def fullName() = {
  firstName + " " + lastName
}
```

Singleton Objects

- “object” defines a singleton object

- Create simple tests (singletons) using “object” and extend App

```
object Test1 extends App {
  val me = new Person(firstName="Bobby", "Bender", 10)
  me.age = 20
  println(me)
  println(me.fullName)
}

object Test2 extends App {
  val me = new Person(age=30)
  println(me)
  println(me.fullName)
}
```

Class Companion Object

- Class Companion Object is a type of singleton
- Scala doesn’t have class behavior or class variables -> use a Companion Object instead
    - Factory methods like apply, etc.
- Requirements for a Companion Object:
    - Companion Object has the same name as the class
    - Companion Object must be defined in the same file as the class

```
class Person(val firstName: String = "joe",
             val lastName: String = "schmoe",
             var age: Int = 0) {
  Person.population += 1
  // …
}

// Companion Object
object Person {
  var population = 0
}

// test Person
object Test3 extends App {
  val bob = new Person("Bob", "Jones", 50)
  val betty = new Person("Betty", "Smith", 30)
  val joe = new Person("Joe", "Smith", 31)
  println(s"Total Population ${Person.population}")
}
```

Access Level Modifiers and Qualifiers

- Modifiers
    - No access modifier results in public scope. This is the default access for instance variables and methods. There is no “public” keyword in Scala!
    - “protected” - class, companion, and subclass (different than java)
    - “private” - class and companion
        - “private class” is private to the package and nested packages (different than java)
- Qualifiers
    - “protected” and “private” can be qualified to a package when specified with brackets. (name between the square brackets must be a simple name - not a dot name)

Example: make the Person.population only available to the “model” package.

```
package com.aboutobjects.model
…
// Companion Object
object Person {
  protected[model] var population = 0
}
```

Class Constructors

- Primary Constructors and Auxiliary Constructors
- Primary (Best -> use this with parameter defaults)
  - Defined using the body of the class definition itself
  - Class definition may include parameters (instance parameters)
    - **val** - read only; **var** - read/write; **nothing** - private
  - Statements in the class body are executed as part of the primary constructor

```
class Person(val firstName: String = "joe",
             val lastName: String = "schmoe",
             var age: Int = 0) {

  // primary constructor code
  Person.population += 1
  println(s"Population is now: ${Person.population}")

  def fullName() = {
    firstName + " " + lastName
  }
  override def toString() = {
    firstName + " " + lastName + ", Age: " + age
  }
}

// Companion Object
object Person {
  private[model] var population = 0
}
```

- Auxiliary Constructors (avoid using these)
    - Defined using methods named “def this(…)”
    - Has to call a previous constructor in its first statement
    - Each has a different signature (parameter list)
    - **Avoid using these if possible. Use Primary with default parameters instead.**

```
class Person(val firstName: String = "joe",
             val lastName: String = "schmoe",
             var age: Int = 0) {

  // Example of an auxiliary constructor
  def this( months: Int) {
    this(age = months/12)
  }
…
}
```

Class Inheritance

- “extends” keyword in class definition
- “super” keyword for parent
- “this” keyword for this instance
- “override” keyword for methods
- Single inheritance only

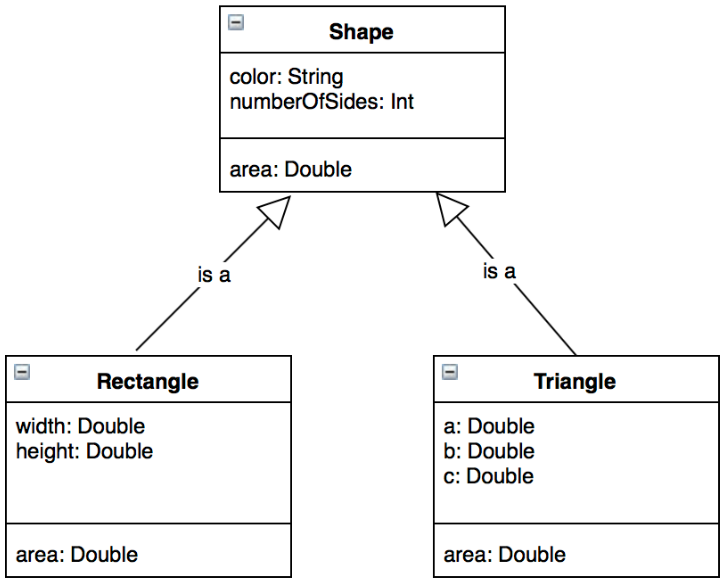
- Use default parameters
    - Without “var” compiler won’t generate an attribute for subclass and use super class attribute
    - With “var” compiler will generate an attribute for subclass

```
class Woman(first: String, last: String, age: Int = 0 ) extends
Person(first, last, age) {
  override def toString() = {
    "Ms. or Mrs. " + super.toString()
  }
}

object testWoman1 extends App {
  val woman = new Woman("Betty", "Boop", 35)
  println(woman)
}
```

Exercise - Classes and Objects

1. Create a new Scala project named ExerciseClassesAndObjects.
2. Create a package for your source code, e.g., com.aboutobjects.
3. Create the Shape and Rectangle classes from the diagram below.



4. Override the default toString method for each class.
5. Create objects to test the following classes.
6. Create the Triangle class. Use Hebron's formula for finding the area of a triangle with 3 sides where c is the long side (or the base) of the triangle.

$$A = \frac{1}{4}\sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}$$

Case Classes

- Case classes are immutable by default
- "**case** class" causes the compiler to add a few goodies to your class - but more importantly, a way to do fancy pattern matching
    - Adds a factory method for the name of the class so you don't have to use "new" to create an instance
    - While you can specify a "var", it won't be an immutable instance so we avoid this
    - A "canEqual" method -> Boolean … (subclasses can override if they want to not allow being equal to the parent class or sibling classes)
    - Some methods to inspect the args - "productArity", "productElement(index)", "productIterator", and "productPrefix"
    - A "copy" operator
        - Use named and default args to choose what is copied and what is new

```
Scala> case class Dog(name:String, breed:String, age:Int)
defined class Dog

// Don't need “new” to create an instance
Scala> val buford = Dog("Buford","Irish Setter", 1)
buford: Dog = Dog(Buford,Irish Setter,1)

// methods generated by the compiler
Scala> buford.
age     canEqual   equals     name          productElement
productPrefix
breed   copy       hashCode   productArity  productIterator
toString

// copy and override a named parameter
Scala> val sally = buford.copy(name = "Sally")
sally: Dog = Dog(Sally,Irish Setter,1)

Scala> sally.toString()
res16: String = Dog(Sally,Irish Setter,1)
```

When to use Case Classes

- Use case classes for immutable objects that represent primarily data type objects (DTOs).
    - Equality for “case class” comparison is redefined to compare the values of the aggregated stateful information - uses the hash.
    - Simplify matching for case statements
- Use regular classes when inheritance is involved and/or state changes.

- Object comparison compares the object's identity. This is different than case classes.
- Pattern matching doesn't work with regular classes.

```
case class Dog (name: String, age: Int, breed: String)
case class Fish (name: String, age: Int, breed: String)

object testAnimals extends App {
  val buford = Dog("Buford", 2, "Irish Setter")
  val bubbles = Fish("Bubbles", 1,  "Guppie")
  println(buford.name + " and " + bubbles.name + " are my 
pets.")
}
```

Exercise - Classes and Objects

1. Create a new Scala project named ExerciseCaseClasses.
2. Create a package for your source code, e.g., com.aboutobjects.shapes.
3. Create a Shapes.scala file and object called testShapes.
4. Add Rectangle and Triangle as case classes with the same attributes as the previous exercise. You can add them in the same file as the testShapes object.
5. Define the area method inside of each case class.
6. Create objects to test the case classes.

Traits

- Traits are a type of mixins. Objects can ‘mixin' fields and behavior.
    - Can declare fields and maintain state.
    - Can override functions

- Just like a class except:
    - Can't declare constructor type arguments
    - “super" refers to the object that mixes in the trait (not a superclass). Dynamically bound.

- Used to thin out class api by refactoring common fields and methods by ‘mixin' type - turn a thin interface into a rich one.

- Stackable modifications - traits further to the right take effect first. You can override default behavior by stacking traits.

```
trait Pet {
  val name: String
}

trait HasLegs {
  var numberOfLegs = 2
}

trait MakesNoise {
  def speak(): Unit = { println(“silence") }
}
```

```
case class Dog(name: String, age: Int, breed: String) extends
Pet with MakesNoise with HasLegs {
  numberOfLegs = 4
  override def speak() = println("Woof Woof")
}

case class Bird(name: String, age: Int, breed: String) extends
MakesNoise with HasLegs {
  override def speak(): Unit = println("Chirp Chirp")
}

case class Lizard(breed: String) extends HasLegs {
  numberOfLegs = 4
}

object test1 extends App {
  val buford = Dog("Buford", 2, "Irish Setter")
  println(s"My name is ${buford.name}.")
  buford.speak()
  println(s"I have ${buford.numberOfLegs} legs.")

  val tweetie = Bird("Tweetie", 1, "Sparrow")
  println(s"My name is ${tweetie.name}")
  tweetie.speak()
  println(s"I have ${tweetie.numberOfLegs} legs.")

  val liz = Lizard("Iguana")
  println(s"I'm a wild Lizard of type: ${liz.breed}")
}
```

- Use traits when attributes and behavior might be used by more than one unrelated class.

Generics / Parameterized Types

- Everything Java has and more
- Generic classes take types as parameters
- Type parameter is in [A] where A represents the passed in type

```
class Stack[A] {
  private var elements: List[A] = Nil

  def push(x: A) {
    elements = x :: elements
  }

  def peek: A = elements.head

  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}

object testStack extends App {
  case class Person(firstName: String, lastName: String)
  val john = Person("John", "Smith")
  val sue = Person("Sue", "Jones")
  val lifo = new Stack[Person]
  lifo.push(john)
  lifo.push(sue)
  println(lifo.pop())
  println(lifo.pop())
}
```

Generic Wrappers To Handle Exceptions

Try/Catch/Finally exceptions are essentially side effects. They can be problematic when many parallel threads are executing or computations are distributed across many processors. We can encapsulate side effects information using a generic wrapper to handle the possible states.

Try[T]

- Scala.util.Try type which is a class that returns Success[T] or Failure[T]
    - Failure[T] has to be of type Throwable
- Use this over try/catch/finally as allows you to postpone exception handling.
- Try[T] can be passed around to concurrent executing parts of the application.

```
object TryExample extends App {

  def textFrom(filename: String): Try[List[String]] = {
    Try(Source.fromFile(filename).getLines.toList)
  }

  val filename = "/Users/bobbybender/.bashrc" // my shell file
  //val filename = "/.file" // permission not granted to read
this file

  textFrom(filename) match {
    case Success(lines) => lines.foreach(println)
    case Failure(f) => println(f)
  }
}
```

Option

- Option[T] is a special kind of error handling where it handles the absence of some value. It returns Some(value) or None.
- Rids your code of try/catch blocks and/or null checks
- However, it hides the error.

```
object OptionExample extends App {

  case class Person private (firstName: String, lastName:
String, age: Int)

  object Person {
    def create(firstName: String, lastName: String, age: Int):
Option[Person] = {
      for {
        lastName <- validateNotEmpty(lastName)
        age <- validateIntInRange(age, 0 to 110)
      } yield Person(firstName, lastName, age)
    }
  }

  def validateNotEmpty(string: String): Option[String] = {
    if(string.isEmpty) None
    else Some(string)
  }

  def validateIntInRange(x: Int, range: Range): Option[Int] = {
    if(x <= range.start || x > range.last) None
    else Some(x)
  }

  val tooYoung = Person.create("Bob", "Smith", 0)
  val tooOld = Person.create("Bob", "Smith", 111)
  val noLastName = Person.create("Bob", "", 1)
  val bob = Person.create("Bob", "Smith", 10)

  val peeps = List(tooYoung, tooOld, noLastName, bob)
  peeps.foreach(println(_))
}
```

Exercise - Option

1. Modify the ExerciseCaseClasses to return an Option[A] for triangles and rectangles returning None for invalid cases. (Hint: use a companion class factory method like ‘create’).
2. Modify the case statements to handle None and Some.

Enumerations

Using Scala.Enumeration

- Like Java, Scala has an Enumeration class but it is not as powerful as using sealed traits and case objects and not used as much in the field

```
object WeekDay extends Enumeration {
  type WeekDay = Value
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

object TestEnum extends App {
  import WeekDay._
  def isWorkingDay(d: WeekDay) = ! (d == Sat || d == Sun)
  WeekDay.values filter isWorkingDay foreach println
}
```

Using traits

- For rich enumeration use sealed trait and case object

```
package com.aboutobjects.enums {
  sealed trait TrafficLightColor
  case object Red extends TrafficLightColor
  case object Yellow extends TrafficLightColor
  case object Green extends TrafficLightColor
}

object TestTrafficLight extends App {
  import com.aboutobjects.enums._
  println(Red)
  println(Yellow)
}
```

Powerful Pattern Matching

selector match { *alternatives* }

- Constant patterns (case 1 => “one”)

```
case 10 => “ten”
case true => “true”
```

- Wildcard pattern (_) or variable pattern matches any object

```
// wildcard
case _ => “something else”
// or variable
case thing => s”you gave me a $thing”
```

- Constructor patterns

```
case class Dog (name: String, age: Int, breed: String)
case class Fish (name: String, age: Int, breed: String)
…
animal match {
      case Dog => "This doesn't work"
      case Dog("Buford", _, _) => "My name is Buford"
      case Dog(_, _, _) => "Ruff Ruff"
      case Fish(name, age, breed) => "Blub blub”
}
val buford = Dog("Buford", 2, "Irish Setter”)
val rover = buford.copy(name = "Rover")
val buford2 = Dog("Buford", 2, "Irish Setter")
val bubbles = Fish("Bubbles", 1,  “Guppie")
// :: cons operator adds element on front of list
val animals = buford :: rover :: buford2 :: bubbles :: Nil
animals match {
     case x :: xs => println(s"head: ${x}, tail: ${xs}”)
     case _ => println("nothing")
}
```

- Sequence patterns - fixed length and * length for List, Array, Vector, etc.

```
      case List(0, _, _) => "three-element list with 0 as the 
first element"
      case List(1, _*) => "list beginning with 1, having any 
number of elements"
      case Vector(1, _*) => "vector beginning with 1 and having 
any number …"
      case Array(1, _*) => "array beginning with 1 and having 
any number …"
```

- Tuple patterns

```
      case (x, y, z) => s"tuple with values $x, $y, and $z"
      case (r, g, b, _) => s"tuple: red $r, blue $b, green $g"
```

- Typed patterns

```
      case dog: Dog => s"you gave me a dog named ${dog.name}"
```

- Use **pattern guards** to refine the pattern - if ...

```
      case name: String  if name(0) == ‘A’ => …
```

- Option type (optionals)

```
    def show(name: Option[String]) = x match {
           case Some(name) => name
           case None => "?"
     }
```

- Use variable definition for a pattern - @ sign

```
     case matchingList @ List(1, _*) => s"$matchingList"
```

Exercise - Classes and Objects (Continued)

1. Modify the previous ExerciseCaseClasses project test cases to take a list of shapes and print specific messages for rectangles and triangles using **case statements, typed patterns,** and **pattern guards** for different patterns. The output should be something like the following:

```
Rectangle with area > 10: 10.0, 20.0, area: 200.00
Rectangle: Width: 1.0, Height: 1.0, Color: black, Sides: 4,
Area: 1.0
Triangle with area > 10: 10.0, 20.0, 20.0, area: 96.82
Triangle: a: 1.0, b: 1.0, c: 10.0, Color: black, Sides: 3,
Area: NaN
```

Collections in Scala

Lists

- Lists are ordered immutable collections of the same type.
    - A Nil terminated linked list.
    - Fast access to head but not last

- Constructor x :: xs where x is prepended to the list xs where “::” is the “cons" operator

```
Scala> val pets: List[String] = List("dogs", "cats",
"fish")
pets: List[String] = List(dogs, cats, fish)

Scala> val numbers = List(1,2,3)
numbers: List[Int] = List(1, 2, 3)

// use the cons operator to construct a list
Scala> val numbers2 = 1 :: (2 :: (3 :: Nil))
numbers2: List[Int] = List(1, 2, 3)

// simpler still
Scala> 1 :: 2 :: 3 :: Nil
res10: List[Int] = List(1, 2, 3)
```

- Concatenation operator “:::”

```
Scala> List(1, 2, 3, 4) ::: List(3, 4, 5)
res5: List[Int] = List(1, 2, 3, 4, 3, 4, 5)
```

- Some important methods
    - count, drop, filter, find, flatMap, fold, foreach, head, tail, indexOf, last, length, map, max, min, reverse, slice, sortBy, sortWith, splitAt, sum, union, zip, unzip, map, reduce, …

Arrays

- Arrays are an ordered immutable collection of the same type. You can efficiently access an element at an arbitrary position.

```
Scala> Array(1,2,3)
res3: Array[Int] = Array(1, 2, 3)

Scala> Array(1,2,"hello")
res5: Array[Any] = Array(1, 2, hello)

Scala> println(res5(2))
hello
```

List Buffers

- List buffers are ordered mutable collections of the same type

- Constant time to append or prepend

```
Scala> import Scala.collection.mutable.ListBuffer
import Scala.collection.mutable.ListBuffer

Scala> val buffer = new ListBuffer[Int]
buffer: Scala.collection.mutable.ListBuffer[Int] =
ListBuffer()

Scala> buffer += 1
res8: buffer.type = ListBuffer(1)
```

```
Scala> buffer += 2
res9: buffer.type = ListBuffer(1, 2)
```

Array Buffers

- Array buffers are an ordered mutable collection.
    - Constant time to append or prepend

```
Scala> import Scala.collection.mutable.ArrayBuffer
import Scala.collection.mutable.ArrayBuffer

Scala> val arrayBuf = new ArrayBuffer[Int]
arrayBuf: Scala.collection.mutable.ArrayBuffer[Int] =
ArrayBuffer()

Scala> arrayBuf += 2
res13: arrayBuf.type = ArrayBuffer(2)

Scala> arrayBuf += 4
res14: arrayBuf.type = ArrayBuffer(2, 4)
```

Sets

- Sets have no duplicates and can be either mutable or immutable.

```
Scala> import scala.collection.mutable
import scala.collection.mutable

Scala> val set = Set(1,2,3)
set: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

Scala> val mSet = mutable.Set(1,2,3)
mSet: scala.collection.mutable.Set[Int] = Set(1, 2, 3)
```

```
Scala> mSet += 10
res2: mSet.type = Set(1, 2, 3, 10)
```

Maps

- Maps can be either mutable or immutable.

```
Scala> import Scala.collection.mutable
import Scala.collection.mutable

// Mutable Map
Scala> val map = mutable.Map.empty[String, Int]
map: Scala.collection.mutable.Map[String,Int] = Map()

Scala> map("one") = 1

Scala> map("two") = 2

Scala> map
res6: Scala.collection.mutable.Map[String,Int] = Map(one ->
1, two -> 2)

// Immutable Map
Scala> val map2 = Map("one" -> 1, "two" -> 2)
map2: Scala.collection.immutable.Map[String,Int] = Map(one
-> 1, two -> 2)

Scala> map2("one")
res7: Int = 1
```

Sorted Sets and Maps

- Use TreeSet and TreeMap to get sorted collections

```
Scala> import Scala.collection.mutable._
```

```
import Scala.collection.mutable._

Scala> val sortedSet = TreeSet(3,2,1,9,8)
sortedSet: Scala.collection.mutable.TreeSet[Int] =
TreeSet(1, 2, 3, 8, 9)

Scala> val sortedMap = TreeMap("one" -> 1, "two" -> 2,
"three" -> 3)
sortedMap: Scala.collection.mutable.TreeMap[String,Int] =
TreeMap(one -> 1, three -> 3, two -> 2)
```

- Create a mutable map from an immutable map

```
Scala> val mutableSortedMap = mutable.Map() ++ sortedMap
mutableSortedMap: Scala.collection.mutable.Map[String,Int]
= Map(one -> 1, three -> 3, two -> 2)
```

Map, Reduce, Filter Collection Operators

Map

- The map operator transforms the collection into a new collection.

```
Scala> import Scala.collection.mutable._
import Scala.collection.mutable._

Scala> val men = List("Bobby", "Raffaele", "Greg")
men: List[String] = List(Bobby, Raffaele, Greg)

Scala> men map ("Mr. " + _)
res0: List[String] = List(Mr. Bobby, Mr. Raffaele, Mr.
Greg)
```

Reduce

- The reduce operator reduces the collection into a single value.

```
Scala> men reduce (_ + ", " +  _)
res9: String = Bobby, Raffaele, Greg
```

Filter

- The reduce operator reduces the collection into a single value.

```
Scala> men filter (_.size > 4)
res5: List[String] = List(Bobby, Raffaele)
```

Exercise - Map, Filter, Reduce Practice

1. Create a new project called ExerciseMapFilterReduce. Print the sum of the even numbers between 1 and 20 inclusive.
2. Create a Person class with name, age, and gender. Create a list of people multiple males and females.
3. Print a list of men over 20 yrs. old sorted by age oldest first.
4. Create a copy of the original peeps list but replace the men's name with Mr. <Name>.

Advanced Functional Programming

Functional programming is built in to Scala as the default. You should use functional programming concepts as much as possible for both distributed and parallel computing on a multiprocessor chip and in a distributed cluster.

- Characteristics of Functional Programming:
    - Declarative vs Imperative programming style.
    - Program with expressions instead of statements
    - Expression-oriented programming - Nearly every construction is an expression that yields value.
    - Use higher-order functions
    - Use pure functions
    - Leverage recursion
    - Avoids state and mutable data
    - Avoid assignment statements

Expressions vs Statements

- In concept a **statement** is a standalone unit of execution that doesn't return anything. It's purpose is to have a side effect.

- The purpose of an **expression** is to generate a value. It may or may not have side effects depending on the implementation.

- In Scala, everything is an expression, even flow control statements. This promotes the ability to “compose” or combine expressions into a pipeline type logic.

- Expression Oriented Programming:
    - Inputs go in, a result is returned, and there are no side effect
    - Easier to reason about and test
    - Concise and expressive code
    - Can be executed in any order - supports parallelism on multi-processor machines

Pure functions

- Act on input, have no side effects, and only return a value

- Pure functions are easier to test, reuse, parallelize, generalize, and reason about, less prone to bugs

- Easier to reason about because you don’t have to keep in your head when state is changing

Anonymous functions or Function Literals

- Functions or computations that are declared inline - in curly braces
- No assignment statement needed
- Anonymous functions are used inline. The function literal is turned into a value at runtime.

```
Scala> val numbers = List(1,2,3,4,5,6,7,8,9,10)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// anonymous function with parameter in parenthesis
Scala> numbers.filter( (x) => x % 2 == 0 )
res5: List[Int] = List(2, 4, 6, 8, 10)

// anonymous function with parameter and no parenthesis
Scala> numbers.filter( x => x % 2 == 0 )
res6: List[Int] = List(2, 4, 6, 8, 10)

// anonymous function with placeholder _
Scala> numbers.filter( _ % 2 == 0 )
res7: List[Int] = List(2, 4, 6, 8, 10)
```

- (x) => x % 2 == 0 means “take LEFT then do RIGHT” or “x is mapped to the function on the right”

Functions as Variables

- Functions can be assigned to a variable

```
Scala> val isEven = (x: Int) => x % 2 == 0
isEven: Int => Boolean = $$Lambda$1805/1655078594@57c26a40
```

```
Scala> List range(1, 21) filter isEven
res13: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

Higher-order functions

- Functions that take other functions as parameters or returns a function or both
- Abstraction of functions at a higher level. Come about when refactoring code - using the DRY principle

parameterName: (parameterType(s)) => returnType

Example:
f:(String) => Int

```
Scala> def evens(list: List[Int]) = list filter { _ % 2 == 0 }
evens: (list: List[Int])List[Int]

Scala> evens(List.range(1,11))
res0: List[Int] = List(2, 4, 6, 8, 10)

Scala> def odds(list: List[Int]) = list filter { _ % 2 != 0 }
odds: (list: List[Int]) List[Int]

Scala> odds(List.range(1,11))
res1: List[Int] = List(1, 3, 5, 7, 9)

// Function that takes a function as an arg
Scala> def perform( list: List[Int], f: (List[Int]) => List[Int]
) = f(list)
perform: ( list: List[Int], f: List[Int] => List[Int] )
List[Int]
```

```
Scala> perform(List.range(1,11), odds)
res2: List[Int] = List(1, 3, 5, 7, 9)

Scala> perform(List.range(1,11), evens)
res3: List[Int] = List(2, 4, 6, 8, 10)
```

Nested Functions

- Scala supports nested functions that can access variables in the scope of its parent function - to hide a nested function

- Only use nested functions when they reduce complexity else use private functions inside of a class or object

Curried Function

- How? Rewrite a function to take only a single argument at a time.

- Why would you use a curried function? To define a general function

- Contains multiple argument lists

```
// instead of def add(x:Int, y:Int) = x + y
def add(x:Int) = (y:Int) => x + y

// OR Scala lets you do this instead (simpler)
def add(x:Int)(y:Int) => x + y
```

```
Scala> def line(m: Int, b: Int, x: Int): Int = m * x + b
line: (m: Int, b: Int, x: Int)Int

Scala> def curriedLine(m: Int)( b: Int)( x: Int): Int =
line(m,b,x)
curriedLine: (m: Int)(b: Int)(x: Int)Int

Scala> val xValues = (1 to 10).toList
xValues: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Scala> def line(x:Int):Int = curriedLine(2)(0)(x)
line: (x: Int)Int

Scala> xValues.map(line)
res1: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

Partially Applied Functions

- An expression that doesn't apply all the arguments needed by a function

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

Scala> val partial = sum(1, 2, _: Int)
partial: Int => Int = $$Lambda$1181/612223930@535be281

Scala> partial(3)
res14: Int = 6
```

- Example: XML partially applied function

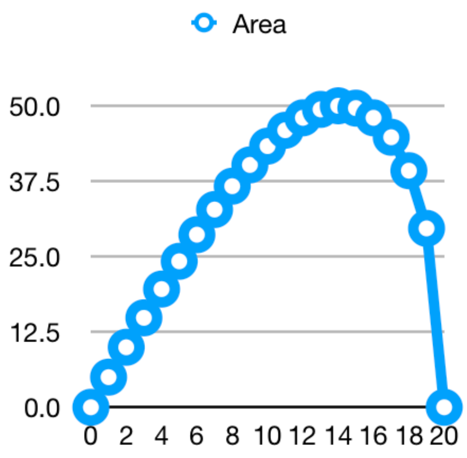
```
Scala> def wrapXml(prefix: String, content: String, suffix:
String) = { prefix + content + suffix }
wrapXml: (prefix: String, content: String, suffix: String)String
```

```
Scala> val authorXml = wrapXml("<author>", _: String, "<\
\author>")
authorXml: String => String = $$Lambda$1224/1313152922@7828bc6b

Scala> println(authorXml("Bobby Bender"))
<author>Bobby Bender<\author>
```

Exercise - Partially Applied Function

1. Create a new project called ExerciseFunctional.
2. Create an object "partiallyAppliedFunction" and extend App.
3. Create a function for Harod's formula to calculate the area of a triangle given 3 sides a,b,c with c being the longest side (the base).
   ```
   1D/4 * Math.sqrt((a+b+c)*(-a+b+c)*(a-b+c)*(a+b-c))
   ```
4. Create a partially applied function where a = 10, b equal 10, and c is not yet defined.
   ```
   … (10, 10, _:Double)
   ```
5. For the range 0 to 20, print the areas using your partially applied function.



Closure

- A function literal that contains “free variables” from the context of where the function is defined.
- “Bound variables” are bound explicitly as function arguments.
- Scala’s closures capture the variables by reference, not by value.
    - Closed over variables won’t be garbage collected until the closure itself is released as well as the scope where the variable was created.

```
Scala> var x = 0
x: Int = 0

// x is a free variable captured by the closure
Scala> val addToX = (y: Int) => x = x + y
addToX: Int => Unit = $$Lambda$1231/521524038@7fb70e94

Scala> addToX(10)

Scala> x
res1: Int = 10

Scala> addToX(20)

Scala> x
res3: Int = 30

Scala> addToX(x)

Scala> x
res5: Int = 60
```

Variable Argument List

- Function can take a variable argument list for the last argument only.
- Use <TYPE>* - asterisk

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

Scala> sum(1,2,3)
res0: Int = 6
```

Tail Recursion

- Scala will optimize tail recursion if and only if the method or nested function calls itself directly as its last operation. Without tail recursion, the stack will overflow for large collections.

```
object Functions extends App {

  def echo(x: Int, text: String): Unit = {
    println(text)
    if(x == 1) return
    else echo(x-1, text)      // tail recursion
  }

  echo(3,"Hello")
}
```

Exercise - Refactor Into Tail Recursion

1. Turn the following recursion into tail recursion.

```
  // first attempt at recursion
  def productOfInts(ints: List[Int]): Int = ints match {
    case Nil => 1
    case head :: tail => head * productOfInts(tail)    // not
tail recursive here
  }
```

Exercise - Tail Recursion - Sum of Squares

1. Create a tail recursive function that computes the sum of the squares of integers. The results should be a Double.

Refactoring code to make it functional

1. First, make all entities immutable (Value objects).

2. If you need to make the application more performant, then and only then, consider the tradeoffs for a mutable entity (Referential object). Encapsulate mutable entities behind a referentially transparent wrapper function.

3. Remote client's (e.g. other microservice or UI) only get a copy (Value object).

4. Instead of the imperative style of changing data in place, use the functional style of mapping input values to output values.

5. Refactor functions that have side effects.

```
package com.aboutobjects
object EliminateSideEffects extends App {
  case class Person(name: String, age: Int)
  def printMessage(person: Person) = person match {
    case Person(_, age) if age >= 18 => println(s"Please go vote 
${person.name}")
    case Person(_,_) => println(s"${person.name}, please 
participate in the elections even though you can't vote.")
  }

  val john = Person("John Doe", 21)
  val betty = Person("Betty White", 16)
  printMessage(john)
  printMessage(betty)
}

object WithoutSideEffect extends App {
  case class Person(name: String, age: Int)
  def message(person: Person) = person match {
    case Person(_, age) if age >= 18 => s"Please go vote $
{person.name}"
    case Person(_, _) => s"${person.name}, please participate in
the elections even though you can't vote."
  }

  val john = Person("John Doe", 21)
  val betty = Person("Betty White", 16)
  println(message(john))
  println(message(betty))
}
```

Asynchronous Programming

Futures, promises, and Actors are the tools used for asynchronous programming in Scala. We will only provide an introduction to futures in this course.

Futures

- A future is concurrent code that returns eventually
    - States: Incomplete, Complete
    - Complete substates: Successful, Failed
- Futures are composable - you don't have to nest them!
- Future is a container type. You can use a for comprehension, map, or flatMap
- Scala.concurrent.ExecutionContext.Implicits.global - the default thread pool used for ForkJoinPool. You can create your own.
    - Default thread size on global is set to the number of processors - Runtime.availableProcessors

```
import Scala.concurrent.{Future}
import Scala.concurrent.ExecutionContext.Implicits.global
import Scala.util.{Failure, Success}
import Scala.util.Random

object Futures extends App {
  def sleep(timeInMillis: Long) = {
    Thread.sleep(timeInMillis)
  }

  println("starting future containing long calculation”)
  val longCalc = Future[String] {
    sleep(Random.nextInt(5000))
    "I'm done calculating"
  }

  println("before onComplete")
  longCalc.onComplete {
    case Success(value) => println(value)
    case Failure(e) => e.printStackTrace
  }

  // do the rest of your work
  println("0 secs ...");
  sleep(1000)
  println("1 sec ...");
  sleep(1000)
  println("2 secs ...");
  sleep(1000)
  println("3 secs ...");
  sleep(1000)
  println("4 secs ...");
  sleep(1000)
  println("5 secs ...");
  sleep(1000)

  sleep(2000)
}
```

Exercise - Futures

1. Create a new project called ExerciseFutures.
2. Create a future that takes a url (i.e., https://www.apple.com), calls the url using the line of code below and returns the time it takes to receive the results and the html as a string.

```
…
val start = System.currentTimeMillis()
val html = Scala.io.Source.fromURL(url).mkString
val stop = System.currentTimeMillis()
…
```

3. Use a callback to handle both the Success and Failure cases.
4. Test out your future with both valid and invalid URLs.
5. Tip: make sure your program doesn't complete before it has had time to get a result by adding this to the bottom of you running object.

```
Thread sleep 3000
```

Futures and For-Comprehensions

- For-Comprehensions let you use generators inside the for parenthesis. When these sequences are Futures, then Yield creates parallel tasks where possible.

- **Important: Futures must be created before used in a For-comprehension or they won't have a chance to run in parallel.**

```
import Scala.concurrent._
import ExecutionContext.Implicits.global
import Scala.util.{Failure, Success}

object ParallelForComprehension extends App {

  val f1 = Future { Thread.sleep(500); 1}
  val f2 = Future { Thread.sleep(1000); 2}
  val f3 = Future { Thread.sleep(1500); 3}

  val startTime = System.currentTimeMillis

  // see https://docs.Scala-lang.org/style/control-
structures.html
  // for comprehensions can take () or {}

  // don't do this ... bad style
//  val results = for (
//    r1 <- f1;
//    r2 <- f2;
//    r3 <- f3
//  ) yield (r1 + r2 + r3)

  // do this ... good style "for comprehensions with multiple
generators"
```

```
  // runs in parallel time should be close to 1500 ms
  val results = for {
    r1 <- f1
    r2 <- f2
    r3 <- f3
  } yield (r1 + r2 + r3)

  // runs sequentially not in parallel ... should be close to
4000 ms
//  val results = for {
//    r1 <- Future { Thread.sleep(1000); 1}
//    r2 <- Future { Thread.sleep(1000); 2}
//    r3 <- Future { Thread.sleep(2000); 3}
//  } yield (r1 + r2 + r3)

  results.onComplete {
    case Success(x) => {
      val runTime = System.currentTimeMillis - startTime
      println(s"result = $x")
      println(s"Processing time: $runTime")
    }
    case Failure(error) => println(s"error: $error")
  }

  Thread.sleep(5000) // for demo keep jvm alive
}
```

Promises

- A Promise is the task of writing the value to a future - a Promise that it will eventually have a value.
    - You normally don't have to create Promises.

- Works in conjunction with a Future. The Promise is the producer and the future is the consumer.

    1. Create a Promise
    2. Complete the Promise by setting a successful value
    3. Implement the unsuccessful case by returning an exception
    4. Return the read side of the Promise - the Future

Implicit Conversions and Parameters

- Use implicit conversion to clean up distracting boiler plate code - whenever you see repetitive redundant boring distracting code.
    - Conversion to an expected type
    - Conversions from the receiver of the method call
    - Implicit classes as wrappers to existing classes to add new methods or rename methods
    - Implicit parameters to add missing parameter to a function call
- The compiler only checks for implicits if the code doesn't type check as written
- Rules:
    - Only definitions marked "implicit" are allowed
    - Implicit variable, object, or function definition
    - Implicit definition must be in scope as a single identifier or be associated with either the source or target of the conversion
    - Only one implicit at a time - no nesting

Sala Coding Guidelines

Taken from the following links:
https://github.com/alexandru/scala-best-practices
https://tpolecat.github.io/2014/05/09/return.html

Avoid using "return" statements in functions.

- Functions return values by definition but the "return" statement abandons the current computation and returns to the caller of the method in which return appears.

- Non-Local Return

- Not referentially transparent

https://tpolecat.github.io/2014/05/09/return.html

TBD add code and examples here…

Use immutable data structures.

- Avoid side effects by using immutable entities. Very rarely do you need to use a mutable object. When you do only use it locally and return an immutable copy.

Avoid using unnecessary temporary variables.

- Leverage functional concepts to eliminate the need for a var prior to a loop or temporary variable used in a function. You can probably eliminate it. Prefer expressions and immutability.

Eliminate useless traits.

- Use traits only when you are defining behaviors that will be used by multiple classes.

Keep Case Classes immutable.

- Don’t use var to override immutable behavior in case class initializers.

Use def over var in abstract classes.

- Use def instead of var in abstract classes. That way subclasses can override it however to meet their needs.

Use Option[T] not null.

- Use the Option[T] type instead of using null. Then the compiler will check for you and your intentions will be clear.

Use java.time instead of Java's Date or Calendar.

- Java's Date and Calendar return mutable objects. Use java.time that was introduced in Java 8 which fixes many problems.

Limit your use Any, AnyRef, isInstanceOf, or asInstanceOf.

- Don't work against strong type checking with Any, AnyRef, isInstanceOf, or asInstanceOf. This typically comes up in deserialization. Instead use case class for different types and matching.

Do not use case classes nested inside of regular classes.

- If you define a nested case class inside of a regular class, when serializing a case class it will serialize over the parent class because of the "this" reference.

Scala Style Checker

Exercise - Scala Style Checker

Setup a new project for using a style checker and unit testing for the next section.

1. Create a new project using IntelliJ called ScalaTestExcercises.
2. Add model code from the ExerciseClassesAndObjects project (shapes).
3. Download the following file and add it to the root of the previous ScalaTestExcercises project.

   http://www.scalastyle.org/scalastyle_config.xml

4. Run Analyze/Inspect Code in IntelliJ.
5. Start fixing warnings. Comment out the rules that don't apply in the scalastyle_config.xml file.

See http://www.scalastyle.org for details on managing the Scala Style Checker.

Scala Unit Testing

- ScalaTest is a powerful framework for testing. See http://www.scalatest.org/user_guide.
- Supports multiple styles of testing - **FlatSpec**, FunSuite, FunSpec, WordSpec, FreeSpec, **PropSpec**, and **FeatureSpec**, and RefSpec.
- Supports tagging of tests (logical groupings)

```
import org.scalatest.Tag
object MyTests extends Tag(“com.aboutobjects].tags.DbTest”)
```

- Supports an abundance of Matchers

```
result should equal (3)
result should === (3)
result should be (3)
result shouldEqual 3
result shouldBe 3
string should startWith ("xxx")
string should endWith ("xxx")
string should include ("xxx")
. . . and many more . . .
```

- Supports Mocks
- Asynchronous testing
- And more - See http://www.scalatest.org/user_guide

Exercise - Creating Unit Tests with ScalaTest

Add unit tests using the FlatSpec style with ScalaTest.

1. Open the ScalaTestExcercises project in IntelliJ.
2. Open sbt build.sbt file and add the following to the end of the file:

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5" % “test"
```

3. Add test classes for Shape, Rectangle, and Triangle. Extend FlatSpec with Matchers for each test class.
4. Run tests.
5. Run tests with coverage.
6. Run all tests -> right click on menu src/test/scala and select “Run scala tests in Scala …”

http://www.scalatest.org/user_guide/using_scalatest_with_intellij

https://docs.scala-lang.org/getting-started-intellij-track/testing-scala-in-intellij-with-scalatest.html

Important Scala Links

Scala Style Guide: https://docs.Scala-lang.org/style/

http://danielwestheide.com/Scala/neophytes.html