Brief Introduction to Scala

Nathaniel Osgood

CMPT 470/816

Scala: Distinguishing Features

- Multi-method language (procedural, object-oriented, functional) with particularly strong support for functional programming
- Use within Java Virtual Machine => interoperability with Java modules & libraries
- Widespread use of type inference
- Sparse syntax & many conveniences
- Widespread use in Data Science
- Semi-declarative syntax (e.g., pattern matching, for comprehensions, etc.)
- Strong support for parallel programming
- Optional support for two types of laziness
- Availability of an interactive REPL (Read-Eval-Print-Loop); invoke via “scala”
  - But note that because this goes line-by-line this is more picky about leading things on the same line (e.g., the “{“ following a class declaration)

Declarations: Values and Variables

- declare value (e.g., a convenient name for a value to be used in subsequent code)

```
val d: Double = sqrt(2.0)
val d = sqrt(2.0)
```

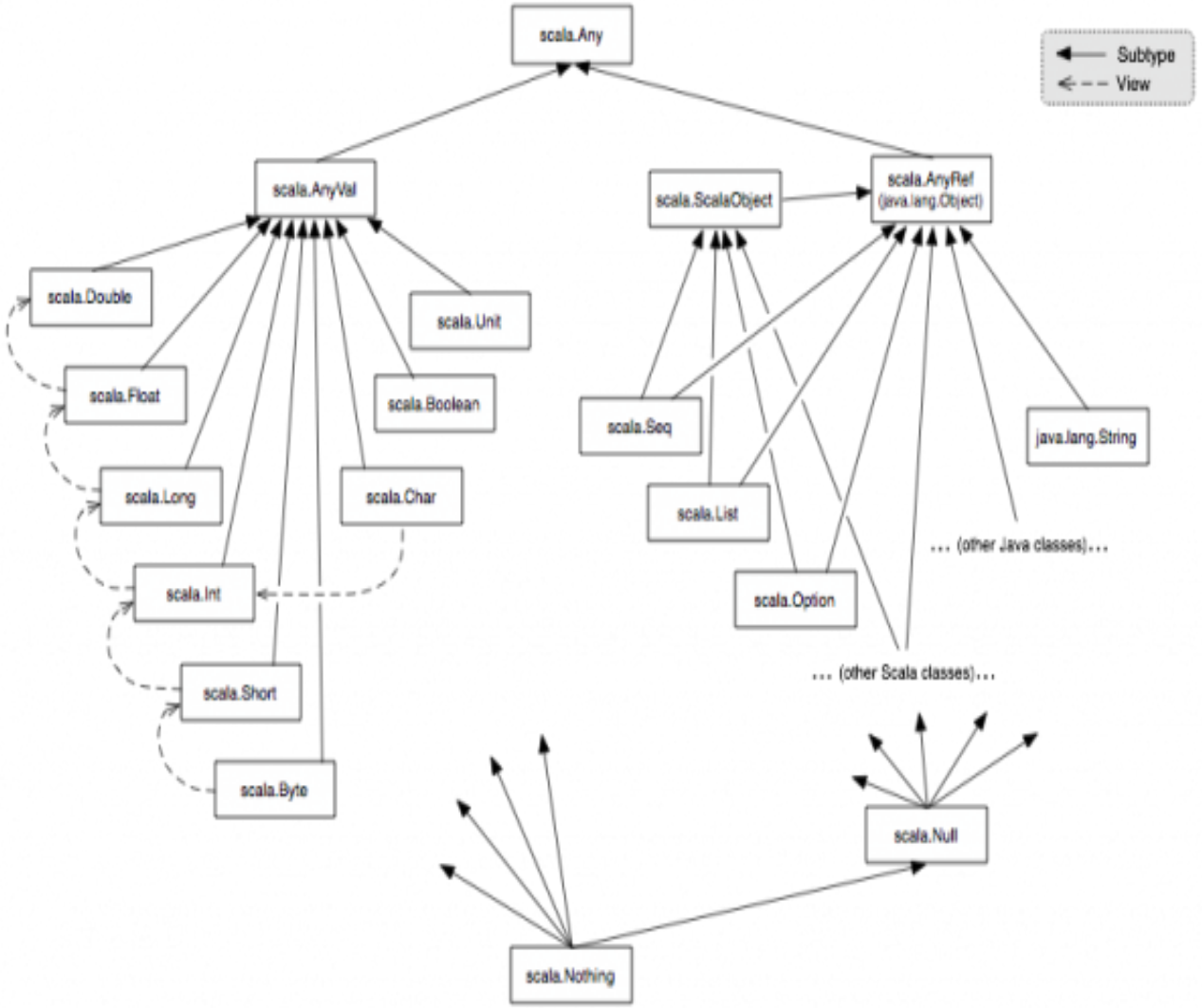
- declare variable (e.g., for an instance variable, or local variable in a method) with var (e.g., var countInserted)

```
var counter = 2;
var counter: Int = 2;
```

Scala Types

- Entirely class based
- Built-in value types beneath AnyVal (Float, Double, Boolean, Char, Int, Short, Byte, Unit)
  - Can build your own value class to avoid overhead
  - Unit allows for uniform handling of cases often handled by void in other langs
- Reference types beneath AnyRef
  - Beneath scala.scalaObject,
    - Other user-defined types
    - Seq, List, etc.
    - Option (like Java 8 Optional)
  - Use Null as a distinguished value

From http://docs.scala-lang.org/tutorials/tour/unified-types



Additional Type Notes

- Extensive use of type inference – don't have to include many type annotations
- Marking a value as a type: Put colon after name, followed by type
  - e.g., val foo: Int = square(2) **or** val foo = square(2)
- Subclassing: Use "extends", and "overrides" for method overriding
- Widespread use of type parameterization (roughly like generics in Java, templates in C++)
  - Notation uses square brackets (e.g., Array[Int], List[Int])
  - Can specify lower and upper type bounds

Scala Methods & Functions 1

- use "def" for a method (both functions and procedures)
  - For functions, have “=“ separating fn header from body
  - e.g., def exampleFunction(int n) = { }
  - e.g. def performAction(int n) { }

- To turn a method fn created w/“def” into standard function for passing or returning as an argument, follow by _
  - This makes it clear that are not simply calling, but seeking to take its value as a function

Referring to Methods as Functions

```
scala> class Foo { var m = 2;    def incr() = { m += 1;   m } }
defined class Foo
scala> val foo = new Foo()
foo: Foo = Foo@419bc62d
scala> foo.incr
res4: Int = 3
scala> foo.incr _
res5: () => Int = <function0>
```

This is a function that, when called, will increment foo

```
scala> floor _
res14: Double => Double = <function1>
```

Refers to the “floor” function (rather than to the method itself)

Method Formal Parameters

- Can include **default values**
  - These values are used if no argument is given for that parameter when the function is called
  - e.g. def join(sep = “,”)
  - This can really cut down the amount of arguments that have to be passed. This offers greater:
    - Convenience
    - Clarity of the key argument values
  - This can also much reduce need for auxiliary constructors
- Can be **specified by name** when *calling*
  - Given specification by name, can give in any order
  - e.g., find(strToLocate = “foo”, strBeingSearched = zap())
  - Can mix specification by just listing and then by name
- Can be specified as subject to *implicit* specification

Type Parameterization

- Here, we are allowing the same definition to be applied to many different *types* of parameters
- Examples (some functionality already captured elsewhere):
  - def interweaveStreams**[T]**(streamA: Stream[T], streamB: Stream[T]) : Stream[T] = streamA.head #:: streamB.head #:: interweaveStreams(streamA.tail, streamB.tail)
    - Example use: interweaveStreams(Stream.continually(1), Stream.continually(2)).take(10).force
  - def iterateStream**[T]**(a: T, f: T => T) : Stream[T] = a #:: iterateStream(f(a), f)
  - def identity**[T]**(a: T) : T = a

Anonymous Functions 1

- Here, can write anonymous functions in syntax (a: Int) => (a*a), a => a*a
- Thus we can write

```
(0 to 10).map(i => i * 3)
```

```
(1 to 30).reduceRight((n, a) => n + a)
```

Succinct Notation for Anonymous Functions

- To avoid writing out full closure when just single argument and only appears once in body, can use “_” to denote parameter in the body alone

  e.g., (0 to 10).map(_* 3)

- If “_” appears in the body twice, it is assumed to represent two different values, e.g.,

  (1 to 30).reduceRight(_ + _)

Statically Defined **Methods** in Scala

```
def square(x: Int) = x*x

def increment(x: Int) = x+1
```

```
scala> square(3)
res6: Int = 9

scala> increment(2)
res5: Int = 3
```

To treat as a **function** (and obtain a corresponding function object referring to the method), follow by _

```
scala> square _
res7: Int => Int = <function1>
```

Anonymous Functions

- Some of the most powerful uses of closures occur with unnamed functions
  - This functions are created to accomplish some particular ad hoc or dynamic task
  - If the task is commonly needed, we can create a name function to abstract that functionality
- Anonymous functions play a very important role in functional programming for higher order fns
  - Passed to functions
  - Returned from functions
- What do these anonymous do?
  - (n:Int) => n+1
  - (n:Int) => n*2
  - scala> (c:Int) => ((n:Int) => n+c) // requires closure
  - res10: Int => (Int => Int) = <function1>

Creating Closures

- A closure binds a function to some scope (environment)
  - Environment: A mapping of names to locations
    - In a purely functional languages, this maps names to values
  - ***When we create a function value, a closure is created with it that captures the values for vars on which function relies***
  - e.g.
    - val IncrementByFunctionGenerator = (c:Int) => ((n:Int) => n+c) // a closure is formed when this (function) value is returned. This closure captures the "binding" between "c" and whatever value was passed as the argument to "foo"

Functions in Scala

- **Parameterizing: Add parameterized (e.g. user-specified) constant to the elements**

```
val FnIncrementBy = (c:Int) => ((n:Int) => n+c)
val FnSimpleIncrement = FnIncrementBy(1)
val FnSimpleIncrement(5)
res1: Int = 6
val FnAdd10=FnIncrementBy(10)
val FnAdd10(5)
res2: Int = 15
val testArray = (-1 to 10)
scala> testArray.map(FnAdd10)
res3: scala.collection.immutable.IndexedSeq[Int] = Vector(9, 10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20)
scala> testArray.map(FnSimpleIncrement)
res4: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7,
  8, 9, 10, 11)
```

Delay

- Wrapping in anonymous function w/ no args offers simple mechanism for delaying evaluation of code
- Allows capturing of some benefits of lazy evaluation

```
scala> val fn = (() => print("foo"))
fn: () => Unit = <function0>

scala> fn()
foo
```

- Cf setting up event handlers
  - arrEventHandlers(iOnMouseSingleClick) = () => PerformCalc1()
  - arrEventHandlers(iOnMouseDoubleClick) = () => PerformCalc2()
- Cf Streams,BigNums
- In Scala, it can often be more natural to capture these with lazy values or call by name

Scala Classes

Scala Classes: Some Characteristics

- Class declaration parameters tersely specifies all of
  - Primary Constructor parameters (a privileged constructor)
  - (Together with method bodies) the associated fields in the class
  - Associated getters (& sometimes setters)
  - **Example**: class Person (var age : Double, var income: Double, val mother: Person)
- Use “this” as name of constructor (rather than name of class)

Primary Constructor

- In Scala, there is a privileged primary constructor
- This primary constructor takes the parameters specified following the class name in the class header

  e.g., class Person (age : Double, income: Double, mother: Person) has a primary constructor this(age : Double, income: Double, mother: Person)
- This constructor executes all code declared directly in the class (i.e., not within a method)
- Auxiliary constructors will often delegate to this primary constructor
  - By using default values for class parameters, can often greatly lessen this need

    Person (age : Double, income: Double, mother: Person, isMarried = false)

Auxiliary Constructors

- Auxiliary constructors are declared as named “this”
- Auxiliary constructors must delegate to one of
  - Primary constructor (with parameters associated with class header)
  - A previously defined auxiliary constructor
  - To call these other constructors, use this (e.g., this(foo, bar))
- Example: Either of the following works
  - class Person2(val isFemale:Boolean, var age:Int, var income:Double) { def isMale = !isFemale; def this(isFemale: Boolean) **=** this(isFemale, 0,0.0) }
  - class Person2(val isFemale:Boolean, var age:Int, var income:Double) { def isMale = !isFemale; def this(isFemale: Boolean) **{** this(isFemale, 0,0.0) **}** }

Implicit Field Creation

- Not all parameters listed in the class header (and used in the primary constructor) are declared as fields
  - For example, these may only be used transiently by the primary constructor
- If such a parameter is used in another method, it is considered to require a field, and such a field (and derivative getters/setters) is created in the class
  - What sort of field is created will depend on the specifications (qualifiers, etc.) in the class header, e.g.,
    - A val parameter will create a val field
    - A var parameter will create a var field
    - Several other qualifiers (private, @BeanProperty, private) affect the getter/setter characteristics

Scala Getters and Setters

- By default, getters and setters are used instead of direct access to underlying variables
- When declare a field (or it is created automatically by rules, Scala by default provides supporting methods
- val field x in class C
  - By default, generates accessor (“getter”) x()
  - Elements of Scala Style: Given an instance c of C, considered better to refer to as c.x
  - **NB: Class parameters without “val” or “var” specified are by default val**
- var field x in class C
  - By default, generates accessor x() and setter x_=()
- **private** class header parameter: any resulting getter and setters will be **private**
- **If want accessor but not setter, just declare parameter with different name than accessor**
- @BeanProperty class header parameter, classic java getters/setters also generated (getX, setX)

Example

```
class Person (age : Double, income: Double, mother: Person)  {
  ….
  def reportProperties() { println(s“age: $age, income: $income,mother:$mother”);
}
```

- Automatically introduces
  - Primary constructor Person(age : Double, var income: Double, mother: Person)
  - val age: Double; var income : Double, val mother : Person
  - Accessors
    - def age() : Double = { age }
    - def income(): Double = { income }
    - def income_=(double v): Double { this.income = v}
    - ef mother() : Person = { mother }

Example with Private Accessor

// correct way to define a situation where we have an aspect of state (changing over time), but we don't want to define a setter for user by clients of this class

class Person2(val isFemale:Boolean, var age:Int, private var **_income**:Double) { def isMale = !isFemale; def this(isFemale: Boolean) = this(isFemale, 0,0.0); def **income** = **_income** }

Case Classes

- Constructor parameters correspond to public fields
  - e.g., class AndSpec(left: Specification, right: Specification)
- Companion object with “apply” and “unapply” and “tupled” operators
  - **apply** method in companion (so can use “call” like syntax to create the objects, rather than new)
    - e.g., AndSpec(leftExpr, rightExpr)
  - **unapply** method (so that can “take apart” and bind value to the values within): Can destructure elements using “match” constructs, such that subfields are matched
    - e.g., case AndSpec(left, right) => left + “ & “ + right
  - **tupled** allows *apply* to be invoked from a tuple (e.g., MyClass.tupled.(tupleOfParamValues))
- These are generally ***immutable***
- These automatically contain
  - val field for each parameter (except if declared as “var” [not recommended])
- Versions of each of equals, copy, toString, hashCode (using default unless overridden)

Apply

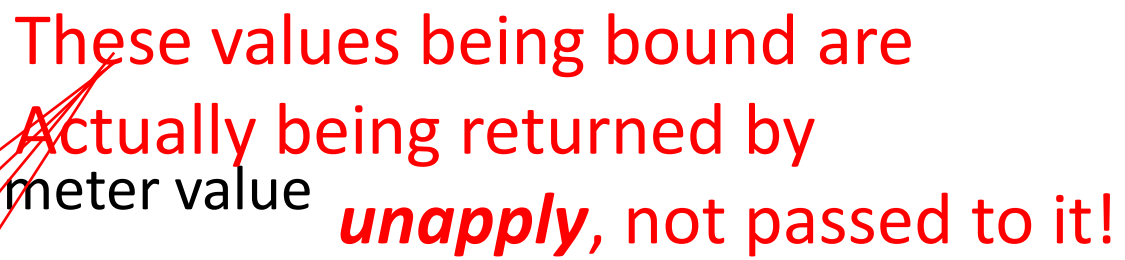
- This function can be defined in either classes or companion objects
  - e.g., def apply(a: Int) { /* Handle calling as if it is a function */ }
- In **companion object**: Here, allows **class name** (rather than instance) to be called like function. This is commonly used to create instances without "new" e.g.,

  CanadianDollar(10.25)
- In **class itself**: Allows *instances of the class* to be called as if they were functions

  fft() //After everything defined, compute the Fast Fourier Transform

“Extraction”: Understanding the “Unapply” Operator

- The “unapply” operator takes in a value to be decomposed (“destructured”), and returns a value back that indicates
  - If the destructuring was successful
  - (If so), what the values extracted from the parameter value
- This role is especially confusing, because the way that unapply seems to be done would erronenously appear to be passing in the values to be extracted TO unapply as arguments! e.g.,
  - array match { … case Array(x, y) =>… // Extracts x, y as 2 elts of an array }
  - str match { case pattern(day, month, year) => // expression involving day, month, year }
  - val rePattern(day,month,year) = str
  - for (PersonCaseClass(age,income,name) <- population)
- **One good way to think about it is that it is almost as if “unapply” is determining the class parameters that would yield the value being considered**



Functions of Unapply Operator

- Thus, “unapply” normally does two things
  - Indicate whether or not a match was made
  - Allow for “destructuring” of values by returning several values that can be bound to named values in the code
- These two functions are interlinked
  - The matching only occurs if a match was successful
  - Which convention is used to indicate success/failure depends on whether it has to return values (if not, it can simply return a Boolean)

Matching By Providing Explicit "Unapply" Operator (Example from Scala.org)

```
object Twice {
    def apply(x: Int): Int = x * 2
    def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None
}
object TwiceTest extends App {
    val x = Twice(21)  /// this is actually calling apply(21)
    x match
   {
   …
   case Twice(n) => Console.println(n) // prints 21
   }
}
```

"Unapply" provides a way to indicate if a criteria is true and (if so) bind one or more values accordingly

If Twice returns a "Some", this case will be matched and "n" bound to unwrapped value, but if it returns None (an Empty value) the case will not match

This is determining the value of *n* for which if we apply Twice(n) we get x

Destructuring (Extraction) is Used for:

- Match expressions
- Bindings/initial values for variables
  - e.g.,
    - val (a,b) = foo()
    - PersonCaseClass(age,income,name) = bar()
- For expressions
  - E.g., for ((key,value) <- map) *body*
  - for (PersonCaseClass(age,income,name) <- population)

This is determining the particular values of *age, income, name* for which if we apply PersonCaseClass(age, income, name) we will obtain the value returned by bar

```
println(s"$name has income $income")
```

Apply

- This function can be defined in either classes or companion objects
  - e.g., def apply(a: Int) { /* Handle calling as if it is a function */ }
- In **companion object**: Here, allows **class name** (rather than instance) to be called like function. This is commonly used to create instances without "new" e.g.,

  CanadianDollar(10.25)
- In **class itself**: Allows *instances of the class* to be called as if they were functions

  fft() //After everything defined, compute the Fast Fourier Transform

Update

- This function can be defined in either classes or companion objects
  - e.g., def update(a: Int) { /* Handle calling as if it is a function */ }
- This is called when apply would normally be, but if we are on the left hand side of an assignment (=) operator
- In **companion object**: Here, allows **class name** (rather than instance) to be called like function on left hand side. This could be used to update static elts

  Buffer(i) = sensorValue()
- In **class itself**: Allows *instances of the class* to be called on left hand side as if they were functions

  myDataStructure(strKey, i) = v // Update the data structure (instance)

“Companion Objects” for Singleton support

- Static values and methods within classes and other singleton functionality can be supported via “companion objects”. Common uses:
  - Define “apply” function to avoid need for explicit “new” keyword
  - Define “unapply” function as extractor to restructure objects
  - Define static variables/fields
  - Define “main” function
  - Define static accessor methods
- Given a class C, we can create a companion object as “object C”
- These are “Friends” to main class, by default can access fields
- Access these via C.foo as if it were associated with the class

Another Common Example: For Static Main

```
object ConwaysGameOfLife extends App
{
    Console.err.printf(s"Starting...\n");

    val life: ConwaysGameOfLife  = new ConwaysGameOfLife()

    life.simulate(args)
}
```

Example Companion Object

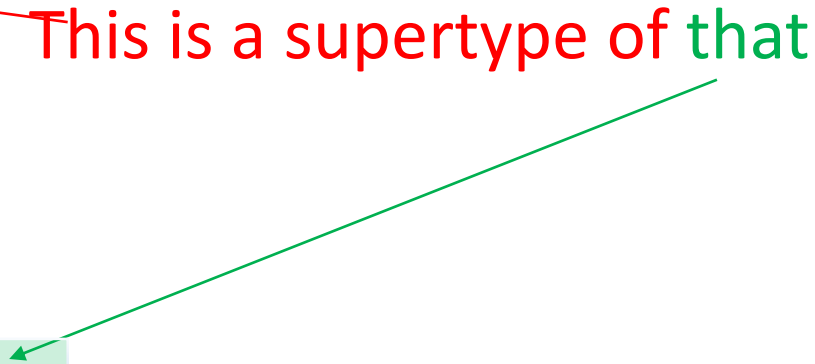
```
class ComputeLivenessRule(val livenessRule: (Boolean, Int) => Boolean)
{
    def isAliveNextTimestep(isCurrentCellAlive: Boolean, countSurroundingAlive: Int) = livenessRule.apply(isCurrentCellAlive, countSurroundingAlive);
}

object ComputeLivenessRule
{
    val ruleClassic = new ComputeLivenessRule(
        (isCurrentCellAlive: Boolean , countSurroundingAlive: Int ) => if (isCurrentCellAlive) (countSurroundingAlive >= 2 && countSurroundingAlive <= 3) e

    val commandLineDispatchTable: Map[String, ComputeLivenessRule] = Map(
        "Classic" -> ruleClassic,
        "Liveness2To4"-> new ComputeLivenessRule((isCurrentCellAlive: Boolean, countSurroundingAlive: Int) => if (isCurrentCellAlive) (countSurroundingAliv
        "Liveness1To3" -> new ComputeLivenessRule((isCurrentCellAlive: Boolean, countSurroundingAlive: Int) => if (isCurrentCellAlive) (countSurroundingAli
        )

    // Returns the rule for the given command line argument.
    // If no matching rule is found, return NULL;
    // PRECONDITIONS:
    //      strCommandLineArg is non-null
    // POSTCONDITIONS:
    //      returns the rule (as a ComputeLivenessRule) associated with the command line argument strCommandLineArg, and null if no such
    //      command line argument is recognized


    def determineRuleFunctionForCommandLineArgument(strCommandLineArg: String) : ComputeLivenessRule =
    {
        commandLineDispatchTable(strCommandLineArg)
    }

}
```

Scala Functions: Contravariance & Covariance

- In contrast to traditional Java or C++, Scala supports more general type rules for functions motivated by the Liskov Substitution Principle
- By LSP, Implementation of a function/method *f* in subtype B of a supertype A must have
  - Contravariant in Parameters: a given parameter of *f* as defined in B must have a type that is equivalent or a supertypes of the corresponding parameter of *f* in supertype A
    - "Contra" in "Contravariant": As to go successive subtypes, the parameters can go to successive supertypes! (As go to successive specialization, parameters can become more general)
  - Covariant in Return values/exceptions: The return value and exceptions thrown by *f* in B can be the same or subtypes of those of *f* in A
- Scala supports the latter of these (specifically, covariant return values)

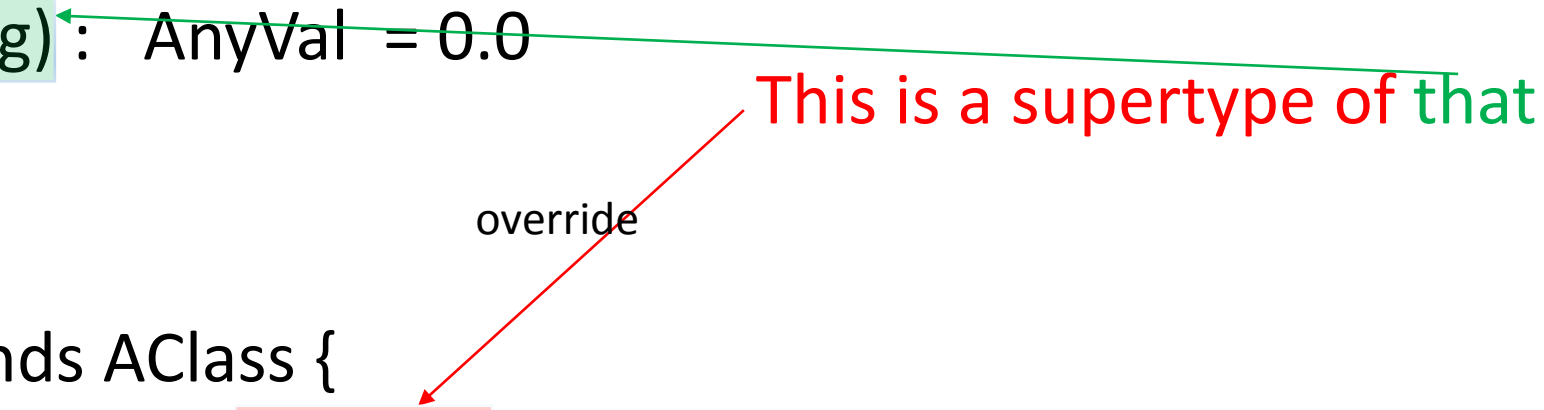
Acceptable to Scala Compiler

```
class AClass {
    def foo(v: AnyRef) : AnyVal = 0.0
}

class BClass extends AClass {
        override def foo(v: AnyRef) : Double =  0.0
}
```



Should be Acceptable, But Refused by Scala Compiler

```
class AClass {
    def foo(v: String) :   AnyVal  = 0.0
}

class BClass extends AClass {
        override def foo(v: AnyRef) : Double =  0.0
}
```



This satisfies the Liskov Substitution Principle, but is still refused by the Scala compiler

Valuable Use of Covariance 1

```
class AClass {
    def setFoo(v: String) :   this.type = { … this }
}

class BClass extends AClass {
        override  def setFoo(v: String) :   this.type = { … this }
}
```

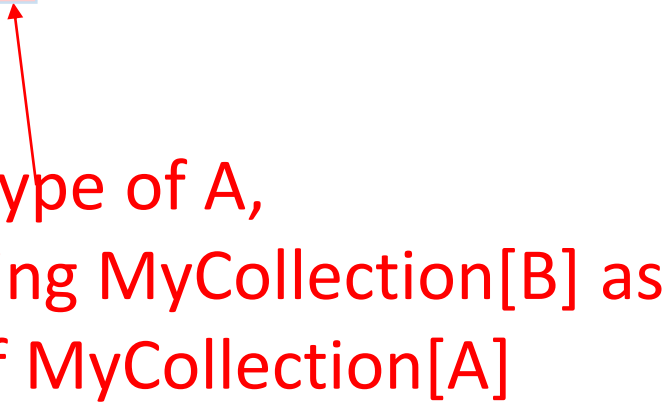
override

This is a subtype of that

This satisfies the Liskov Substitution Principle, and allows the “fluent” style of chaining method calls

Valuable Use of Covariance 2

```
class MyCollection[+A] …
```



Apparent Danger to the Liskov Substitution Principle with Scala Self Types

```
class AClass {
   def bar(v: this.type) : AnyVal  = 0.0  }
class BClass extends AClass {
   override def bar(v: this.type) : AnyVal  = 0.0  // This should not be allowed! }
```

- Ok
  - val a: AClass = new AClass()
  - val b: BClass = new BClass()
  - val apparentAClass : Aclass
  - a.bar(a)
  - b.bar(a)
  - val apparentAClass : AClass = b
- Should work but doesn't
  - apparentAClass.bar(a)
  - apparentAClass.bar(apparentAClass)

This is a "self type", which refers to the type of the enclosing class. By the L SP, Scala should not allow a subtype Method using this as a parameter to override a corresponding supertype method, but does.
This leaves the door open to a gratuitous violation of the LSP!

Scala Functions 2

- Note that can have methods that are themselves polymorphic
- Can be useful to use “this.type” to denote surrounding type
  - For example, making the return value of an overridden function (method) “this.type” allows subtypes to return corresponding subtypes (e.g., subtype A return A) -- covariance
- Can have nested functions: Functions whose scope is limited to and used within another function
- Currying: can explicitly define multiple lists for a function (in successive parentheses), so that provide arguments in staged fashion
  - MyAdd(x:Double)(y:Double)

Scala Functions 3

- Methods that only take a single argument (but which are also applied to some “target” as the method) can be used as infix operators
  - def plus(a: Int): Int = ….
  - e.g., a plus b
- Can use operators that don’t take an argument can be used as a postfix operator
  - def complement(): Set = ….
  - e.g., ASet complement
- Above: *need care & s.t. () to ensure scope of each parameter (e.g., a fn) is clear*
- When call unary function, can omit
- can overload operator syntax, as in C++
  - (e.g., use syntax like "^"), but I believe that can't change precedence easily

Built-in Support for Tuples

- Elements can be of different types
- Constructed implicitly with parentheses
  - e.g., (“Pi”, 3.14159)
- With pattern matching, can put on left hand side of an assignment
  - E.g. val (a,b,c) = fooReturningMultipleValues()
- The underlying type parameterized classes are Tuple2, Tuple3, Tuple4, … Tuple22,etc.

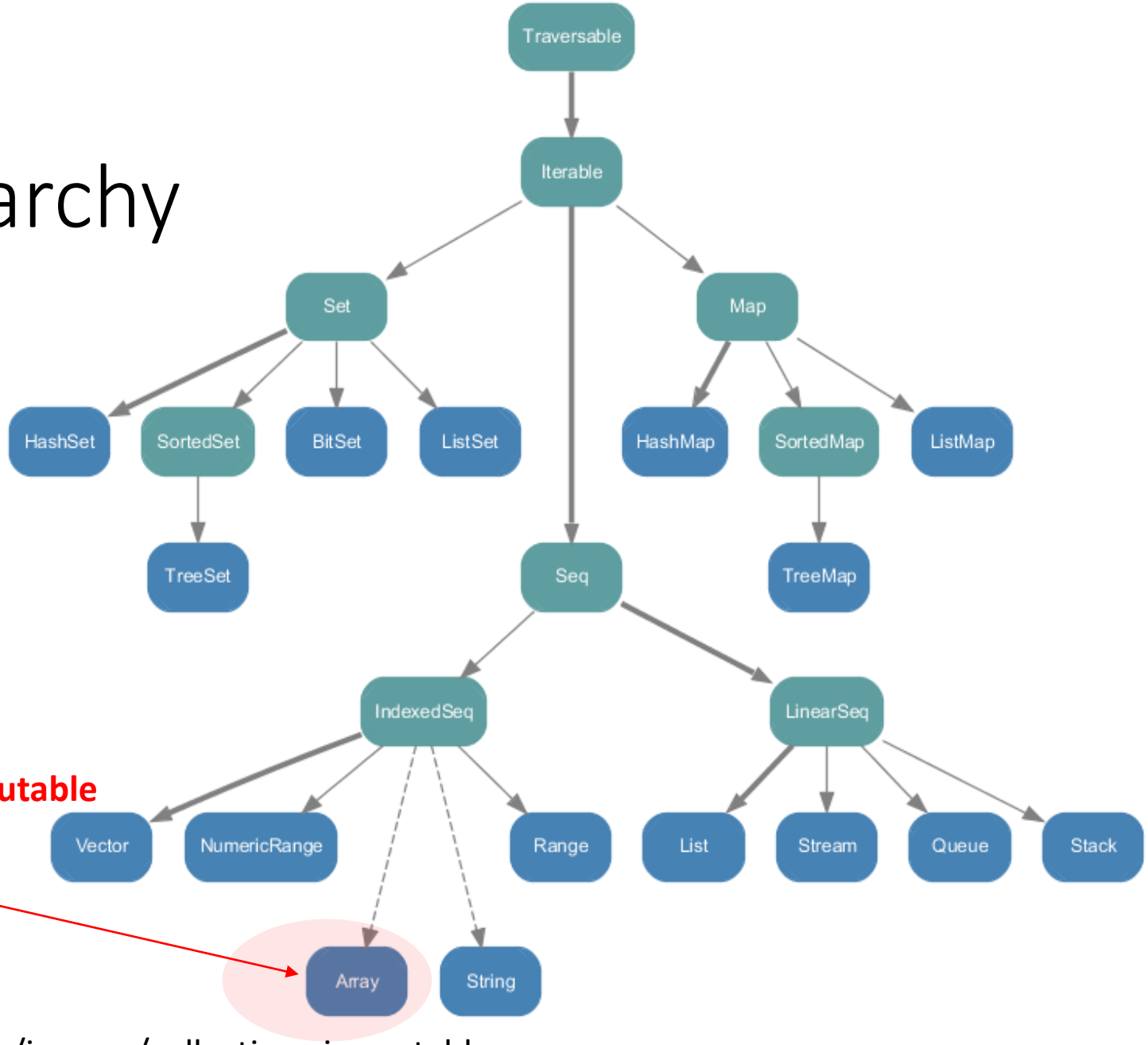
Scala Collections 1

- Scala provides rich set of collections, in 3 categories: seq, set, map
- Generally type parameterized (e.g., Array[T])
- Can create w/name of collection (e.g., Array(1,2,3), Map("a"->1,"b"->2))
- Include all of
  - Lazy vs. not
  - Immutable vs. mutable
- Equality is based on
  - If in different category, then not equal
  - equality of elements (collections of different types/structures are unequal)
- Can use "par" method to render parallel version of collection
- Can use "view" method to render lazy version of collection
- Can use foreach syntax e.g. --- but much more
- Performance characteristics are at: http://docs.scala-lang.org/overviews/collections/performance-characteristics.html

Important Collection Initializers

- Most
  - fill(length)(lazilyEvaluatedExpr)
    - Here, lazilyEvaluatedExpr is reevaluated on an ongoing basis
    - Multidimensional version exists
  - iterate(initial, length)(f)
    - Produces collection of [initial, f(initial), f(f(initial)), f(f(f(initial))), ….]
  - tabulate(length)(f)
    - Produces collection of [f(0), f(1), f(2), f(3), f(4), ….]
      - This can be very useful, as each element can know its next proper position e.g., for lookup, to reference corresponding elements in a related structure, etc.
    - Multidimensional version exists (e.g., giving matrix, where each element is informed of its row&col)
- Streams:
  - continually

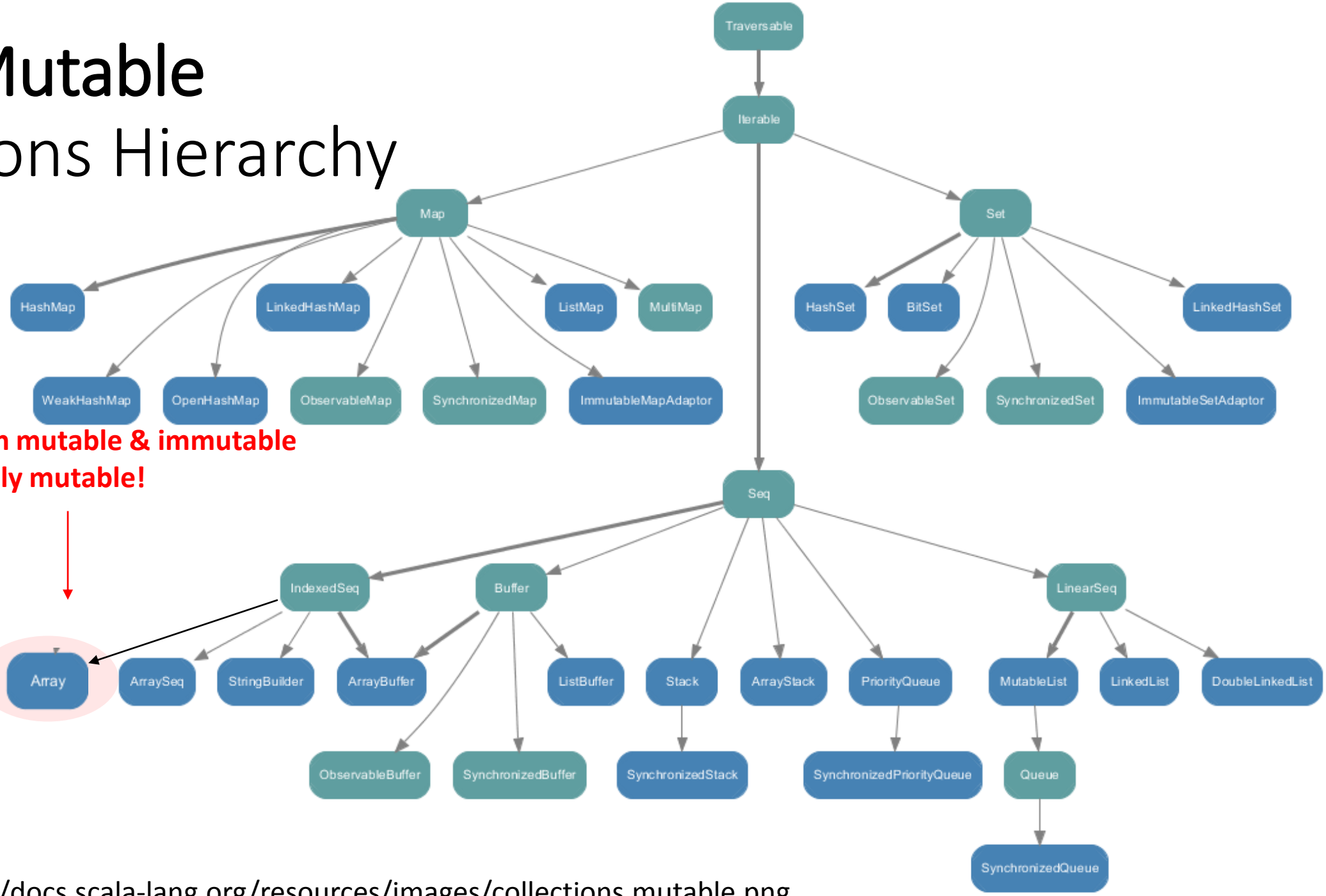
Immutable
Collections Hierarchy



A little bit in between mutable & immutable
Contents are definitely mutable!
Size is immutable.

Adapted from: http://docs.scala-lang.org/resources/images/collections.immutable.png

Mutable
Collections Hierarchy



A little bit in between mutable & immutable
Contents are definitely mutable!
Size is immutable.

Adapted from: http://docs.scala-lang.org/resources/images/collections.mutable.png

Performance

- Detailed performance characteristics can be found at:
- http://docs.scala-lang.org/overviews/collections/performance-characteristics.html
- Recommend Alvin Alexander's Scala Cookbook

http://alvinalexander.com/scala/understanding-performance-scala-collections-classes-methods-cookbook

- Example recommendations (drawn from Scala Cookbook)

	Random Indexing	Sequential Access
Immutable	Vector	List
Mutable	ArrayBuffer	ListBuffer

Collections 2: Example Collections

- **Stream[T]** lazy, use #:: to build from head/tail. Operations are **incremental**
- **Vector[T]**: O(1) time for operations on head, others take time linear to position/depth in sequence. Preferred immutable indexed sequence
- **Range**: **Ints**, created w/"to" and "by" (e.g., 0 to 100 by 2) and until(m,n) (upper:n-1)
- **NumericRange**: **Doubles**, created w/"to" and "by" (e.g., 0.0 to 1 by 0.1) and until
- **ArrayBuffer[T]: Mutable.**
- **Array[T]** Familiar array
  - single dimensional: can allocate an array of Ints of size n using new Array[Int](n)
  - can access with array1D(i), add elements (single or tuple) to with +=, collection with ++=
  - multidimensional: Array.ofDim[Int](countRows, countColumns)
    - have then Array[Array[Int]]
    - access with array2(i)(j)
- **List[T]**: Can use :: operator to build from head/tail (have Nill at end!), or List(a,b,c…)
    - e.g., val list = "foo" :: "bar" :: "baz" :: Nil
- **Map[K, V]**: to create a map from a key to a value (e.g., String to a ComputeLivenessRule)
  - val myMap = Map("square" -> ((a: Int) => (a*a)),
    "cube" -> ((a: Int) => (a*a*a)))

Scala Maps

- Represent key-value mappings (Collections of key-value pairs)
- Can create with normal collection syntax: Map(“a”->1,”b”->2)
  - As with other collections, not absence of “new” operator!
- For mutable map, can add new item (key k and value v) in either of the sort of way
  - map(k)=v (e.g., map(“Scala”) = “Great!”)
  - map += (k -> v) (e.g., map += (“Scala” -> “Great!”))
- Can deconstruct via pattern matching as a (k,v) pair)
  - Iterate over map with (for (k,v) <- map) *body*

Classic Functional Programming Constructs

- **Container[E].map: (E => F) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].flatMap: (E => Container[F]) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].reduce**: (F × E => F) => F
  - Applies a function to each element of a collection and to the previous result of the reduce function
- **Container[E].filter**: (E => Boolean) => Collection[E]
  - Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)
- Other useful Scala:
  - **Container[E].groupBy**: (E => K) => Map[K, **Container[E]]**
    - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k
  - **Container[E].collect**: PartialFuntion(E => K) => **Container[K]**

High-Level Description

- **Reduce/fold** represents “accumulation” – independent of what exactly one is accumulating
- **Filter (filterWith)** represents conditional selection – independent of what exactly one is selecting (the exact rules for selection)
- **Map** represents a mapping (transformation) of each element of a container
- **flatMap** represents a mapping (transformation) of each element of a container, ***where the mapping can itself return a container***
- A “**container**” here can be a
  - Collection (the focus here)
  - Another monad, for example
    - Option
    - Either
    - Try
    - Future
    - Promise
    - Your own monad! (later lecture)

Classic Functional Programming Constructs

- **Container[E].map: (E => F) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].flatMap: (E => Container[F]) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].reduce**: (F × E => F) => F
  - Applies a function to each element of a collection and to the previous result of the reduce function
- **Container[E].filter**: (E => Boolean) => Collection[E]
  - Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)
- Another: **Container[E].groupBy**: (E => K) => Map[K, **Container[E]]**
  - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k

Initial Binding

```
scala> val testArray = (-1 to 10)
testArray: scala.collection.immutable.Range.Inclusive = Range(-1, 0, 1,
2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Map

- **What does this do?**
  - scala> testArray.map(n => n * n)

- **And this?**
  - scala> testArray.map(n => n + 1)

  What is an alternative expression of the same function?

- **What function does the following transformation undertake?**
  scala> testArray.map(n => n*(if (n > 1) 1 else -1))

Map

- **Square the elements of the collection**
  - scala> testArray.map(n => n * n)
  - res8: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
- **Add fixed value (one) to each elements**
  - scala> testArray.map(n => n + 1)
  - res9: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

  An alternative expression of the same function:
  - scala> testArray.map(_ + 1)
  - res10: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
- **Value-specific mapping**

  scala> testArray.map(n => n*(if (n > 1) 1 else -1))

  res13: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 0, -1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Passing closures resulting from higher-order functions, and named closures

```
val FnIncrementBy = (c:Int) => ((n:Int) => n+c)

scala> testArray.map(FnIncrementBy(3))
res17: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
  12, 13)

val FnAdd10=FnIncrementBy(10)

scala> testArray.map(FnAdd10)
res18: scala.collection.immutable.IndexedSeq[Int] = Vector(9, 10, 11, 12, 13, 14, 15,
  16, 17, 18, 19, 20)
```

Flatmap: Container[E].flatMap: (E => **Container[F]**) => Container[F]

- Applies mapping function to each element of collection, but where that mapping function **can itself return a container**
- Example: Create an expression that computes the reciprocal of every number n between 0 and 100 that is not divisible by 2, 3 or 5, as well as the reciprocal of the square of that number
- First try (causes a problem!)
  - (0 to 100).filter(_ %2 != 0).filter(_ % 3 != 0).filter(_ % 5 != 0).map(i => Vector(i, i*i)).map(1.0 / _)
- A workable solution:
  - (0 to 100).filter(_ %2 != 0).filter(_ % 3 != 0).filter(_ % 5 != 0).**flatMap**(i => Vector(i, i*i)).map(1.0 / _)
- Flatmap combines all of the containers returned by the function into one!
  - This will be critical when we consider the functionality of **monads**

Classic Functional Programming Constructs

- Container[E].map: (E => F) => Container[F]
  - Applies mapping function to each element of collection
- Container[E].flatMap: (E => Container[F]) => Container[F]
  - Applies mapping function to each element of collection
- **Container[E].reduce**: (F × E => F) => F
  - **Applies a function to each element of a collection and to the previous result of the reduce function**
- Container[E].filter: (E => Boolean) => Collection[E]
  - Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)
- Another: Container[E].groupBy: (E => K) => Map[K, Container[E]]
  - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k

Reduce

- **Totalling up the elements of the array**

```
>>> reduce(function(a,b) { return(a+b); }, rg)
  10
```

- Concatening array elements

```
>>>reduce(function(a,b) { return(a+""+b); }, rg)
```

- Taking max of elements

```
>>> reduce(function(x,y) { return (x>y) ? x : y; }, rg);
```

Variants

- Reduce
  - reduceLeft
  - reduceRight
- Fold
  - foldLeft
  - foldRight
  - “Curried” to take initial value

Classic Functional Programming Constructs

- Container[E].map: (E => F) => Container[F]
  - Applies mapping function to each element of collection
- Container[E].flatMap: (E => Container[F]) => Container[F]
  - Applies mapping function to each element of collection
- Container[E].reduce: (F × E => F) => F
  - Applies a function to each element of a collection and to the previous result of the reduce function
- **Container[E].filter: (E => Boolean) => Collection[E]**
  - **Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)**
- Another: Container[E].groupBy: (E => K) => Map[K, Container[E]]
  - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k

Filter

- scala> testArray.filter(n => n > 0)
- res19: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  Alternatively,
- scala> testArray.filter(_ > 0)
- res20: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- scala> testArray.filter(_ % 2 == 0)
- res21: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 2, 4, 6, 8, 10)

filterWith is like filter, but doesn't create a new copy of the values that pass – just takes a reference to the items selected!

- Numbers between 0 and 5 (inclusive)
  - scala> testArray.filter(n => (n >= 0 && n <= 5))
  - res22: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5)

GroupBy:
Container[E].groupBy: (E => K) => Map[K, Container[E]]

- Applies to a Container[E], and takes a function f mapping each element to a key K
- Results in a map
  - **Keys** are each value of k produced by applying function to elements of the source container
  - **Values** associated with key k are all elements e of the Container[E] for which f(e) == k
    - Places all elements of the original collection that are associated with the same key the values for that key!

Collect:
Container[E].collect: PartialFunction(E => F) => Container[F]

- Applies to a Container[E],
- takes a *partial* function f mapping some element to type F
  - This function f is not defined over some elements
- Results in a Container[F]
  - **Elements are the value of the partial function for all elements of the** Container[E] for which they were defined

- **This is like performing a map & filter together at once! (Filtering out things that are not mapped to a defined value)**

Parallelization Implications of map, filter, reduce

- The operations discussed here have strong potential for parallelization
- **map, filter** can be performed in parallel across a collection by different processing units
  - Each processing unit takes responsibility for handling a subset of the data, operates in parallel
- **reduce** can be performed in time proportional to the log of the number of elements => very favourable performance scaling
- We can call .par on many collections to create a parallel version of that collection, but this has overhead that must be balanced with the performance benefits of the parallelism so obtained
  - The data structures involved are quite large, and can be more restricted in use
  - There are implications for copies of the data

Laziness in Scala

Laziness in Scala: Scala supports two types of laziness

- "By need" (waits until needed to compute): Supported via "lazy" keyword
  - Here, only evaluate once – **memoizes (caches) the value after that evaluation**, and uses it once it was computed once
  - *Streams* fall into this category
- "By name" (basically, **each time that value is required, it is recomputed**): Supported via "by name" arguments
  - Just like wrapping in a no-parameter lambda: Every time that need, just call no-parameter lambda again to evaluate
  - Parameters: declare with "=>" preceding the type, e.g., def foo(x : => Int)
  - **Support view calling .view on a collection**

Laziness by Name: “view” operator

- Given a collection, can call .view to make it lazy-by-name
  - Can then access particular elements and if they don’t depend on other elements, they alone will be calculated
  - As Laziness by Name, the result is not cached
- e.g.,

```
scala> val lazySquares = (0 to 10).view.map(x => { println("Computed square of " + x); x * x } )
        lazySquares: scala.collection.SeqView[Int,Seq[_]] = SeqViewM(...)

scala> lazySquares(3)
        Computed square of 3
        res1: Int = 9
```

Compare & contrast successively unfolding the following

- (0 to 100).filter(_ % 3 != 0).filter(_ % 5 != 0).map(_ * 0.01).map(x => x*x).sum

vs.

- (0 to 100).**toStream**.filter(_ % 3 != 0).filter(_ % 5 != 0).map(_ * 0.01).map(x => x*x).sum

vs.

(0 to 100).**view**.filter(_ % 3 != 0).filter(_ % 5 != 0).map(_ * 0.01).map(x => x*x).sum

Significance of Laziness for Conserving Memory

- Operations on lazy values can be divided into
  - Transformers (these are also lazy – return a stream); e.g., map, flatMap, filter
  - Terminal operations (these force a computation because they require a value – e.g., to group it, combine it with other values, classify it)
    - In Spark, estimators fall in this category; others: sum, groupBy, etc.
- Key point: Lazy values are **only computed when required (when consumed) by terminal operations**
  - Until a terminal operation is performed *all of the intermediate transformers simply successively accumulate operations to be performed – no actual new "intermediate" collections are created ("materialized") along they way*
  - When a terminal operation is performed, each of the preceding operations is performed in turn – successively operating on the values, and finally yielding the value to be transformed
- The net result is that values from a lazy collection can be processed with greater memory efficiency, as they do not require representing intermediate collections

Example Stream Computations

- Computng Fibonacci Sequence
  - def fibonacci(a:Int, b:Int) : Stream[Int] = a #:: fibonacci(b, a+b)
- def successiveRatios(s : Stream[Int]) : Stream[Double] = (s.head / (0.0+s.tail.head)) #:: successiveRatios(s.tail)

Streams: Some Cautions and Advice

- Streams allow us to express infinite structures without the need to fully materialize such structures
- Streams do save memory compared to materializing all data by allowing incremental computation
- Streams do impose memory costs of their own in that once values in the stream are computed, they are then cached!
- Unless there is a reason to do so, it is important to avoid retaining a reference to the stream (e.g., to the head of the stream)
  - This will allow the cached values within the stream to be freed up over time by the garbage collection or other machinery

Laziness and Language Extension

- The capacity to define lazy by name operators effective allows for extending language constructs
- e.g.,

```
def doWhile(condition : => Boolean, body : => Unit)
        {
        body            // executes the body the first time for sure
        while (condition)
                {
                body
                }
        }
```

Try var n=1; doWhile(n<10, { println("n: " + n); n += 1; })

Reading/Writing a Text File

- Reading a file
  - to read a file, import scala.io.Source; can then first get the source with
    - val source = Source.fromFile(strFileName, "UTF-8")
  - linesIterator = source.getLines
  - can then
    - loop over lines with for (lines <- linesIterator)
    - put lines in an array with linesIterator.toArray
- Writing a file
  - val os = new scala.io.PrintWriter(strFileName)

Strings in Scala

- Use str charAt i to obtain character within string
- To get length, can call ".length()"
- string interpolation: Indicated placing character (s, f, r) before string
  - s: substitute variables
  - f: substitute variables, but also format according to format strings
  - r: "raw" -- like s, but won't interpret escape characters
    - Denoted by surrounding string with three pairs of double quotes (e.g., “””raw string”””)
- e.g.,
  - Console.err.printf(s"Error: Prematurely terminated file '$strFileName'; expected a space of dimension $ROW_COUNT x $COL_COUNT. Terminating prematurely.\n");

Strings and Regular Expressions

- To create a regular expression, just use ".r" method of String, e.g.,
  - val reDate = """([0-9][0-9])\-([0-9][0-9])\-([0-9][0-9] [0-9][0-9])""".r
  - Due to escape character and quotes, raw strings are often best (as indicated by triple double-quotes above)
- Useful methods
  - String(strSep) // split at points where strSep occurs, into collection
  - Finding matches throughout string: reDate.findFirstIn(str), reDate.findAllIn(str)
  - Replacing matches by regular expression: reDate.replaceFirstIn(strInWhichToReplace, strReplacement), reDate.replaceAllIn(strInWhichToReplace, strReplacement)

Matching Regular Expression Groups

- Scala regular expressions offer extremely convenient extractors: destructuring in match constructs (see later), and in val
- When binding variables with “val”:
  - E.g., val reDate(day,month,year) = str
- When in match

```
str match {
    case reDate(day, month, “1999”) => /* expression involving  day, month, year */….}
    case reDate(day, month, “2000”) => /* expression involving  day, month, year */….}
    case reDate(day, month, year) => /* expression involving  day, month, year */….}
```

This “extractor” call is calling “unapply” on rePattern to bind the named values day, month, year

- In loop: for (rePattern(day,month,year) <- rePattern.findAllIn(str))

```
{ /* code using day, month, year */ )
```

Expressions (including “Statements”)

- no "++" and "--" operator
  - instead, use += 1, -= 1
- common "statements" are actually expressions in Scala
  - For statements semicolons as optional
    - Mostly used to separate statements on the same line
  - Last statement of function is the return value of that function
- anonymous function syntax
  - val fn = (d: Double, factor: Double) => d*factor
- While better to handle them with wrapped “Try” values, can handle exceptions like
  - try { code } catch { case rte : RunTimeException => exceptionHandlingCode }
  - because this returns Nothing, type for "try" branch will be used

If Expression

- C-style "ternary operator" is simply implemented using if expression syntax
  - If (cond) a else b

Returning

- returning
  - values are not typically returned explicitly, but are just the value of the body
  - can place an expression at the end if one wants to return this
- Tuples and multiple return values
  - a function can return a tuple of multiple values by surrounding in parentheses
    - e.g., (1,"foo")
  - upon return, can bind to separate elements as either
  - (a,b) = fooReturningMultValues()
  - val (a,b) = fooReturningMultValues()

Example of Multiple Return Values

```
// Determines the arrays to use to encode the current state at time iTime.  Specifically, sets *pArrayCurrentCells to hold a pointer
//   to the current array at that time (i.e., the array that holds the value of the current state at that time), and *pArrayNextCells to hold a pointer
// PRECONDITIONS:
//    0 <= iTime
// POSTCONDITIONS:
//    returns a pair of
//            the current array at timestep iTime (i.e., the array that holds the value of the current state at that time)
//            the next array at at timestep iTime (i.e., to the array that will hold the values for the next iteration)
def determineCurrentAndNextCells(iTime: Int):   (Array[Array[Int]], Array[Array[Int]] ) =
{
    val isEvenStep = (iTime % 2 == 0);

    // now return the appropriate pair
    if (isEvenStep)  (bufferEven, bufferOdd) else (bufferOdd, bufferEven);
}
```

Try/Catch/Finally

- Scala supports a try/catch/finally syntax

```
try {
body
} catch {
   case e1 : Exception1 => …
   case e2 : Exception2 => …
   case _ : Exception => …
} finally {
    // this code executed in either case
}
```

This area is modeled after match functionality, which can use “extractors” (calling “unapply”) on supporting exceptions to destructure appropriate values

Generally, we prefer using values that are instances of a Try[T]

A Better Way: “Try” type

- As with a collection, to create, just use *Try(expr)*
  - If expr evaluates to type T, the result is of type Try[T]
  - Expression expr is taken by name, so not evaluated until within the Try
  - If expr
    - Evaluates without an exception => value is a **Success[T]**
    - throws an exception => value is a **Failure[T]**
- Working with the type Try[T] is like working with an Option[T]
  - getOrElse, etc.
  - Can call isSuccess to determine if succeeded
  - Can deconstruct in match statements
- Import from scala.util.Try

Example of Using **Try**

- val v1 = Try(1/1)
- val v2 = Try(0/0)
- Note: the above works b/c division by 0 causes exception with integers (not Double!)
- v1 match { case Failure(f) => print("Something bad happened"); case Success(s) => print("Succeeded!") }
- Something bad happened
- v2 match { case Failure(f) => print("Something bad happened"); case Success(s) => print("Succeeded!") }
- val vFinal = v1.getOrElse(0)

Or more simply,

- val fractionTry = Try(a/(a+b)).getOrElse(0)

For Expression [Comprehensions]/loop: Anatomy

- Generators, e.g.,
  - for (p <- population)
  - for (nbhd <- getNeighborhoods())
  - for (x <- expr1; y <- expr2)
  - for (x <- expr1; y <- expr2(x))

Key: These variables on the left hand side actually refer to The values *within* the containers to the right – not to the dontainers themselves. These are automatically extracted for us, without needing to write an enclosing function.

Unless constrained, this gives combinatorial semantics, like nested loops: Iterate over all pairs of contained values of x & y

- Embedded definitions of values
  - for (nbhd <- getNeighborhoods(); **meanIncome = nbhd.meanIncome**)
- Filters
  - for (nbhd <- getNeighborhoods(); if nbhd.meanIncome **<= povertyLine**)
- For **for expressions**, these are followed by a yield expression e.g.,

  yield.nbhd.population
- Such for expressions translate into pipelines of calls to
  - **for expressions**: flatMap (all but innermost)/map (innermost)/filterWith
  - **for loop ("statement"):** foreach/filterWith

“for” Loops Basics

Can contain **multiple iterations**

for (int i <- m to n; j <- p to q [if guardcondition]) { body }

Example Guarded condition:

for (int i <- m to n; j <- p to q if i != j) { body }

With destructuring (extraction):

```
for (PersonCaseClass(age,income,name) <- population)
    println(s"$name has income $income")
```

Intermediate variables: for (int i <- m to n; j <- p to q; prod = i*j; prod > 0) { body }

This is very commonly used with **range** expressions such as

m to n

m until n (goes up to n – 1)

m until (n, k) (goes up to n – 1, in steps of k)

Comprehensions via "for" loops

- For "comprehensions" consist of using a for loop to construct a collection
  - These are reminiscent of the comprehensions seen in Python
- Because for loops (like other Scala statements) have ***values***, can create such collections very readily, e.g.,

  values = for (i <- m to n; j <- p to q [if guardcondition]) yield expr
- For collections that include **Option** values, for values that are empty (i.e., that are None) nothing is performed for that value
- Note that the types of functionality possible with such comprehensions can sometimes be handled in with constructs such as map & filter
- **These play key role in making monadic operations transparent**, as they desugar to calls to flatMap, map, and withFilter
  - Key point: we can write the operations in the loop with respect to the unwrapped values
- These define partial function; only values over which this is defined are used

For comprehension exhibiting partial values

- scala> val mapTest = Map("foo" -> 1, "bar" -> -1, "baz" -> 0)

- scala> for { ("foo", v) <- mapTest } yield v
- res4: scala.collection.immutable.Iterable[Int] = **List(1)**

Matches, Case Expressions, and Case Classes

“Match” Expressions

- A bit like "if" expression syntax
- Suppose “switch” like functionality more safely, but because of decomposition, can do much more
- Matches top to bottom until succeeds
- A key nice feature is that these can “decompose” the argument into pieces that can then be bound to variables
  - This is most easily supported by case classes
  - A class can also apply an “unapply” method
  - Here, the return value of the “unapply” method indicates whether a match has occurred and (if that match has data associated with it), any values that can be bound via explicit arguments in the case
- e.g., the “n” in case Twice(n) (see below)
- This value is then subsequently bound

Matching with Variable Binding

```
expr match
  {
  case 1 =>…
  case 2 =>…
  case 3 =>…
  ....
  case n : Int =>… //code including n as a variable, which has now been bound
  case _ => // default case
  }
```

Matching with Variable Binding and “Destructuring”

```
def MagnitudeSquared(list: List[Double]) : Double =
{
    list match
        {
        case Nil => 0
        case head :: tail => head*head + MagnitudeSquared(tail)
        }
}
```

If “list” (the expression being matched) has a head & tail, takes apart that list and binds

- “head” to the head of the list
- “tail” to the tail of the list

Matching with Variable Binding and Conditions

```
expr match
  {
  case x if isEven(x) =>… //code handling case of even numbers, referring to n
  case n if isPrime(n) =>… //code handling case of a prime, referring to n
  case n if isSquare(n) =>… //code handling case of a prime, referring to n
  case _: // default case
  }
```

Match Expressions and Regular Expressions

- Regular expression objects can readily
  - Tested against a string
  - **For strings that are matched**, destructured into the values that they match. Example:

```
val pattern = """([0-9][0-9])\-([0-9][0-9])\-([0-9][0-9])""".r
str match
    {
    case pattern(day, month, year) => // expression involving  day, month, year
    }
```

This indicates that this string is to be Interpreted as a Scala regular expression

This tests if the string matches regular Expression pattern. If so, variables for each of the matched "groups" (for day, month, year) are bound to the corresponding values

Matching Multiple Types

```
exprReturningDifferentObjectTypes match
 {
 case s: String => //code including s as a variable, which has now been bound
 case d: Double => //code including d as a variable, which has now been bound
 case c: Char => //code including c as a variable, which has now been bound
 ....
 case n : Int => //code including n as a variable, which has now been bound
 case _ : Byte => //code handling bytes (regardless of value, which was not captured)
 case _ => // default case
 }
```

Naturally, if we have created the classes being matched,
it is generally better to handle through polymorphism
(each class overrides the same method, which is then dispatched

Destructuring Values: Built-in Types (Tuples)

```
bn: BigNum = fn()
bn /% n match  // returns a pair (tuple) of (quotient, remainder)
  {
  case (quotient, 0) =>…    // Goes in exactly
  case (0, remainder) =>… // Doesn't go in even once
  case (quotient, remainder) =>… // Handle general case
  ....
  case _: // default case, in case of logical error
  }
```

From "Scala for the Impatient" by Horstmann

Destructuring Values: Built-in Types (Optional)

- We use (monad) Option[T] (like Optional<T> in Java) to handle possible values
- While Option values can be handled by case statements:

```
exprReturningOptional match
  {
  case Some(v) =>… // code using v (actual value, not optional wrapped value)
  case None =>… // handles empty optional (includes no value)
 }
```

- It is often instead better to use

```
exprReturningOptional.getOrElse(valueIfEmpty)
```

Destructuring Values: Built-in Types (Arrays)

```
array: Array[Int] = fn ()
array match  // returns a pair (tuple) of (quotient, remainder)
  {
  case Array() =>…    // Handles case of empty array
  case Array(0) =>…    // Handles case of array holding single 0
  case Array(v) =>…    // Handles case of array holding single non-zero value
  case Array(0, 0) =>…    // Handles case of array holding two 0's
  case Array(0, y) =>…    // Handles case of array holding initial 0, thereafter non-zero
  case Array(x, y) =>…    // Handles case of array holding two non-zero values
  case zeroArray if zeroArray.forall(v => (v == 0))… // Handles case of array of all zeros of length >= 2
   case Array(0, _*) =>…    // Handles case of array of length > 2 starting with 0
  case _: // default case, in case of logical error
  }
```

Destructuring Values: Built-in Types (Lists)

```
lst match {
  case 0 :: Nil => "0"
  case x :: y :: Nil => x + " " + y
  case 0 :: tail => "0 ..."
  case _ => "something else"
}
```

From “Scala for the Impatient” by Horstmann, p187

Reminder: Case Classes

- Constructor parameters correspond to public fields
  - e.g., class AndSpec(left: Specification, right: Specification)
- Can just use name of class rather than ”new”
  - e.g., AndSpec(leftExpr, rightExpr)
- Can destructure elements using “match” constructs, such that subfields are matched
  - e.g., case AndSpec(left, right)
- These are generally ***immutable***
- These automatically contain
  - val field for each parameter (except if declared as “var” [not recommended])
  - unapply method (so that can “take apart” and bind value to the values within)
  - apply method (so that can use “call” like syntax to create the objects, rather than new)
- Versions of each of equals, copy, toString, hashCode (using default unless overridden)

Use of Case Classes in Match Expressions

```
v match  // returns a pair (tuple) of (quotient, remainder)
  {
  case AndSpec(leftSpec, rightSpec) =>…  // Handle “and” case, using args
  case OrSpec(leftSpec, rightSpec) =>…   // Handle “or” case , using args
  case NotSpec(spec) =>…                 // handle “not”, using arg
....
  case _: // default case, in case of logical error
  }
```

Naturally, if we have created the classes being matched,
it is generally better to handle this through polymorphism
(each class overrides the same method, which then dispatch)

Performing (and Binding to) Nested Submatches

```
v match  // returns a pair (tuple) of (quotient, remainder)
  {…
  case Bundle(_, _, art @ Article(_, _), rest @_*) /* code referring to art as an article */ …}
```

Both Examples from "Scala for the Impatient" by Horstmann

```
def price(it: Item) : Double = it match {
        case Article(_, p) => p
        case Bundle(_, disc, its @ _*) = its.map(price _).sum – disc }
```

Naturally, if we have created the classes being matched,
it is generally better to handle this through polymorphism
(each class overrides the same method, which then dispatch)

Nested Submatches: Criteria

```
v match  // returns a pair (tuple) of (quotient, remainder)
  {
  case AndSpec(leftSpec @ IsLeaf(), rightSpec @ IsLeaf()) =>…  // Handle
“and” of two leaf nodes
   case AndSpec(leftSpec, rightSpec) =>…  // Handle other “and” casea
  case _: // default case, in case of logical error
  }

object IsLeaf {
  def unapply(spec) { return spec.subtreeDepth == 0; } }
```

Structural Types

Allowing “Duck Typing” via Structural Types

- Some popular dynamic languages (e.g., Python, Javascript, Ruby) support “Duck Typing”, which supports a looser type system for method calls
  - Basic idea is that if an object supports a method that can handle a given call, then it is viewed as suitable (“If it walks like a duck and quacks like a duck, it must be a duck”)
- Scala supports “Duck Typing” via “Structural types” specifying what properties must be adhered to by an argument
  - Can use these in place of a named type
  - Example: addAll(collForAdditions : { def add(n : T) }, Iterable[T] collToAdd)
    { for (v <- collToAdd) collForAdditions.add(n) }

Indicates that parameter is any class Supporting such a method

- Add in other languages, this “dynamic dispatch” is expensive – here, uses JVM Reflection to locate & call appropriate method (far more costly than direct call)

Mixins via Scala Traits

Mixin classes and mixin class composition via Traits 1

- Replaces but greatly generalizes capability of interfaces
- Provides implementation reuse while sidestepping complexities of multiple inheritance
- Can "mix-in" traits for class, or when allocate particular objects
- Traits can inherit from another class, but when using traits to “mix in” extra functionality associated with those classes, only the “delta” (the things actually defined in the trait) get reused
- Elements of traits are not inherited from, but instead serve as “construction instructions”
- "super.methodname()" syntax actually does not call methodname in the superclass directly, but instead calls others which have been added to this object
  - can overule this with e.g. super[TraitX].methodname()

Syntax

- Need to read everything after “extends” as part of the class being extended (“compound class”)
- Due to chaining of calls, order does matter!
- For class
  - class MyExperiment extends SimulationExperiment with Visualizer with RunLogger
  - class MyLogger extends Logger with Timestamped with Truncated with Cloneable
- For object
  - val experiment1 = new SimulationExperiment with Visualizer with RunLogger
  - val experiment2 = new SimulationExperiment with Visualizer with Tracer

Abstract Methods

- By default, a method body that has no implementation in a trait is viewed as “abstract”, and must be provided
  - Those mixing in the traits can then provide an implementation of the abstract method
- Frequently a trait will provide an implementation for certain methods (i.e., these won’t be abstract), which will then call the abstract methods
- Often parent trait includes abstract & non-abstract methods (with the latter calling the former)
- Often provide some methods implementations in traits, which then delegate to abstract methods (to be defined by children)

Example 1 (Variant of Horstmann example)

Abstract: To be filled in

Concrete methods delegating to abstract method

```
trait Logger {
def log(msg: String)
def info(msg: String) { log("Information: " + msg) }
def warn(msg: String) { log("Information: " + msg) }
def severe(msg: String) { log("Information: " + msg) }
}
class SavingsAccount extends Account with Logger {
    def withdraw(amount: Double) {
        if (amount > balance) severe("Insufficient funds")
        else ....
 Could further do
    trait ConsoleLogger extends Logger { override def log(msg:String) { println(msg) }
val acct = new SavingsAccount with ConsoleLogger
val acct2 = new SavingsAccount with FileLogger
```

Log of ConsoleLogger will be called when we call acct.log()

From "Scala for the Impatient" by Horstmann

Self Types

- Via “self-types”, can specify that a given trait has to be mixed in with a given type (particularly a class)
- While most instances of this can also be handled by just having the trait extend a class, some cannot
  - With self types, can handle reciprocal dependencies (e.g., two traits that have to be used together)
  - Can handle structural types, for “duck typing”
- These can be important for achieving “family polymorophism”, where
  - Two or more type hierarchy exhibit parallel structure
  - Members of one type hierarchy refer to corresponding members of the other hierarch(ies)

Options for Declarations to Use Traits

- So classes that use this inherit from one distinguished superclass, but then use “with” to mix in elements from other traits
  - e.g. val foo = new Foo() with Trait
  - can "mix-in" traits when allocate particular objects
- In class, use “Extends” for superclasses (or initial trait if none), but then also "with" to and other loggers
- i.e., with the “with” keyword, can layer this atop something that is already inheriting from the superclass of the trait, and can do so for multiple traits
- All Java interfaces can serve as traits
- Unlike Java interfaces, a trait can provide an implementation
  - E.g., of fields and methods

Repeated Superclasses and Super Traits

- If a trait extends a class C, and is being requested as a trait for class A
  - OK
    - If A Ithe class into which the trait is being mixed) has no (explicit) superclass
    - If C is already present as a superclass of A (the class into which the trait is being mixed)0
      - Here, C is only built once
  - Unacceptable: If A extends a class which is not a subclass of C
- NB: If the compound class being extended (e.g., A with B with C) contains repeated superclasses or traits, these are only built once

Example (From Horstmann)

Simply an empty action

```
trait Logged { def log(msg: String)  { } }
class SavingsAccount extends Account with Logged {
    def withdraw(amount: Double) {
        if (amount > balance) log("Insufficient funds")
        else ....
```

Log of ConsoleLogger will be called when we call acct.log()

```
trait ConsoleLogger extends Logged { override def log(msg:String) { println(msg) }
val acct = new SavingsAccount with ConsoleLogger
val acct2 = new SavingsAccount with FileLogger
```

From "Scala for the Impatient" by Horstmann

Example 1 (Variant of Horstmann example)

Abstract: To be filled in

Concrete methods delegating to abstract method

Overrides “log” in logger

Log of ConsoleLogger will be called when we call acct.log()

```
trait Logger {
def log(msg: String)
def info(msg: String) { log(“Information: “ + msg) }
def warn(msg: String) { log(“Information: “ + msg) }
def severe(msg: String) { log(“Information: “ + msg) }
}
class SavingsAccount extends Account with Logger {
    def withdraw(amount: Double) {
        if (amount > balance) severe(“Insufficient funds”)
        else ….
 Could further do
    trait ConsoleLogger extends Logged {override def log(msg:String) { println(msg) }
val acct = new SavingsAccount with ConsoleLogger
val acct2 = new SavingsAccount with FileLogger
```

Adapted from “Scala for the Impatient” by Horstmann

Trait “Layering”: Handling of calls to “Super”

- In traits, when call super.methodName(…), actually calls next implementations mixed in
- The notion of what is the “next implementation mixed in” depends on the order of the mixing-in constructs and their superclasses, etc.
  - If in a linear sequence, starts with processing last
    - Confusingly, this means that the **“next implementation mixed in” is the *previous one* within the successive “with” clauses**
  - More complete understanding beyond linear sequences can be found in Scala references

Example A (Variant of Horstmann example) Concrete Empty Implementation, Call Chaining

Adapted from "Scala for the Impatient" by Horstmann

Does nothing but not abstract

```
trait Logger { def log(msg: String) {} }
def info(msg: String) { log("Information: " + msg) }
def warn(msg: String) { log("Information: " + msg) }
def severe(msg: String) { log("Information: " + msg) } }
trait ConsoleLogger {  def log(msg: String) { println(msg); super.log(msg); } }
class SavingsAccount extends Account with Logger {
    def withdraw(amount: Double) {
        if (amount > balance) severe("Insufficient funds")
        else ….
```

Delegating to any other loggers

Use:

```
val acct = new SavingsAccount with ConsoleLogger with FileLogger
```

Will allow for successive logging by several loggers

Example B (Variant of Horstmann example) Abstract Base, Call Chaining via Delegation

```
trait Logger { def log(msg: String) }
def info(msg: String) { log(“Information: “ + msg) }…
def severe(msg: String) { log(“Information: “ + msg) } }
trait ConsoleLogger extends Logger {override def log(msg: String) { println(msg); } }
trait TimestampLoggerDecorator extends Logger
   {  abstract override def log(msg: String) { super.log(new java.util.Date() + “: “ + msg); }}
trait CountingLoggerDecorator extends Logger
   { val line=0 abstract override def log(msg: String) { super.log(line + “: “ + msg);  line += 1; }}
class SavingsAccount extends Account with Logger {
     def withdraw(amount: Double) {
          if (amount > balance) severe(“Insufficient funds”)
          else ….
```

Abstract: To be filled in

Delegating to any other decorator and the logger

These **log** methods are both abstract b/c call to super.log relies on some fully concrete ***log*** implementation

Allows for successive logging by several logger Decorators, delegating to a final Logger

Use:

```
val acct = new SavingsAccount with ConsoleLogger with TimestampLoggerDecorator with
CountingLoggerDecorator
```

Adapted from “Scala for the Impatient” by Horstmann

Example C: Example of Trait "Layering"

From "Scala for the Impatient" by Horstmann

```
trait TimestampLogger extends Logged {
  override def log(msg: String) {
    super.log(new java.util.Date() + " " + msg)
  }
}
```

```
trait ShortLogger extends Logged {
  val maxLength = 15 // See Section 10.8 on fields in traits
  override def log(msg: String) {
    super.log(
      if (msg.length <= maxLength) msg else msg.substring(0, maxLength - 3) + "...")
  }
}
```

```
val acct1 = new SavingsAccount with ConsoleLogger with
  TimestampLogger with ShortLogger
val acct2 = new SavingsAccount with ConsoleLogger with
  ShortLogger with TimestampLogger
```

acct1.log will first call ShortLogger's log, and then TimestampLogger's log.

⇒Call of acct1.log(msg) will first truncate message (in ShortLogger.log) and then Timestamp the result.

Call of acct2.log(msg) will first timestamp the message (in TimestampLogger.log) and then truncate the result.

Mixin classes and mixin class composition via Traits 2

- Can have abstract field (must be filled in by subclass)

- Common parents are only constructed once (hence “delta” added)

Fields in Traits

- Traits can contain either concrete or abstract fields
- Concrete field
  - Here, this field is placed into the classes it is mixed in
    - This is not inherited, but is instead added
- Abstract field
  - Here, the field is declared without an initializer
  - An abstract field must be provided by a class into which it is mixed in (much as a method needs to be provided)

Many Issues Not Discussed

- Construction order (particularly when not linear)
- No constructor parameters for traits
- Constructors for traits

Understanding Underlying JVM Implementation: Traits are turned into:

- Interface declaring methods
- Accompanying class that provides method implementation
- Fields get added to class that extends from trait

Abstract Types

- Types here have to be filled in by subclasses
- These can serve as alternative to type parameterization of a class
- In the presence of subclassing and when there are many type parameters that would otherwise be required, this can help avoid overwhelming numbers of type parameters

Scala and Parsers

- Scala supports an extremely powerful capacity to create parsers for context-free languages (corresponding to push down automata)
- Elements
  - Encode grammar in Backus-Naur form
  - Production rules operating off of specified subelements parsed
- Many parser variants (e.g., regexp parser, etc.)

Additional Features Not Covered Here

- Annotations: Like Java, Scala supports user-defined definitions of metadata to be maintained, via annotation classes
- Implicit type conversions & extending types
- Scala's process/shell library, to undertake shell commands
- Scala actors and message passing
- Being deprecated in Scala 3: XML Literals
- Thorough coverage of type system
- Easy support for Inversion of control