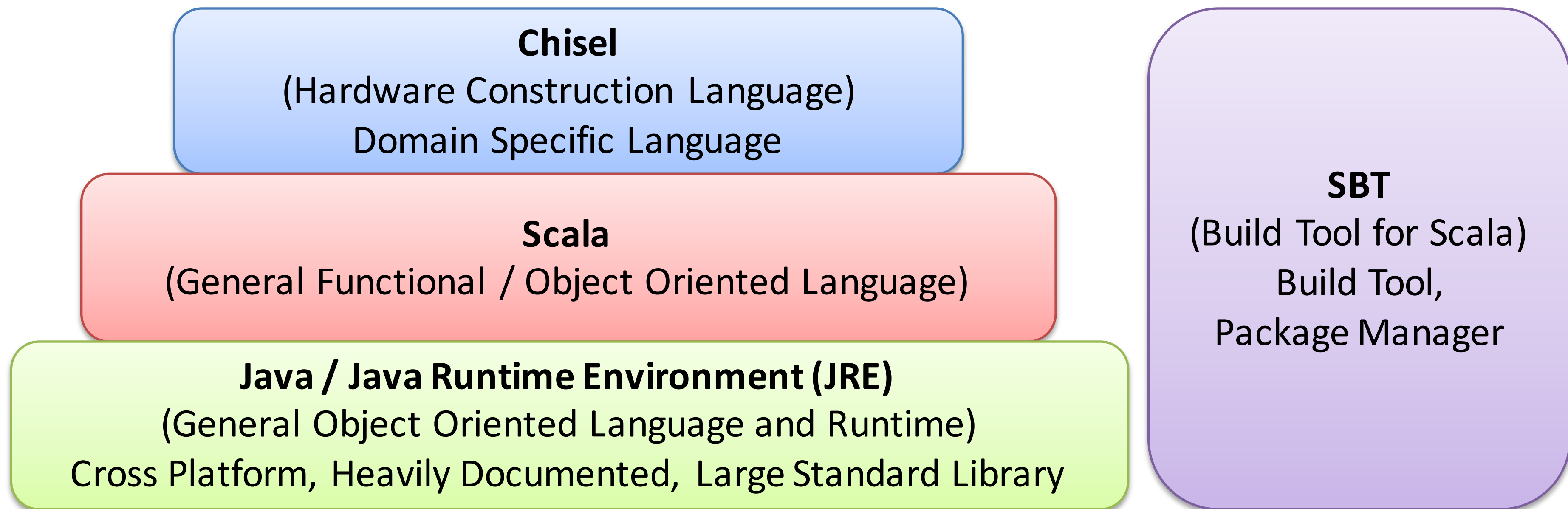
CS250 - Discussion 1
Chisel + Scala Primer

Christopher Yarp

Jan. 2016

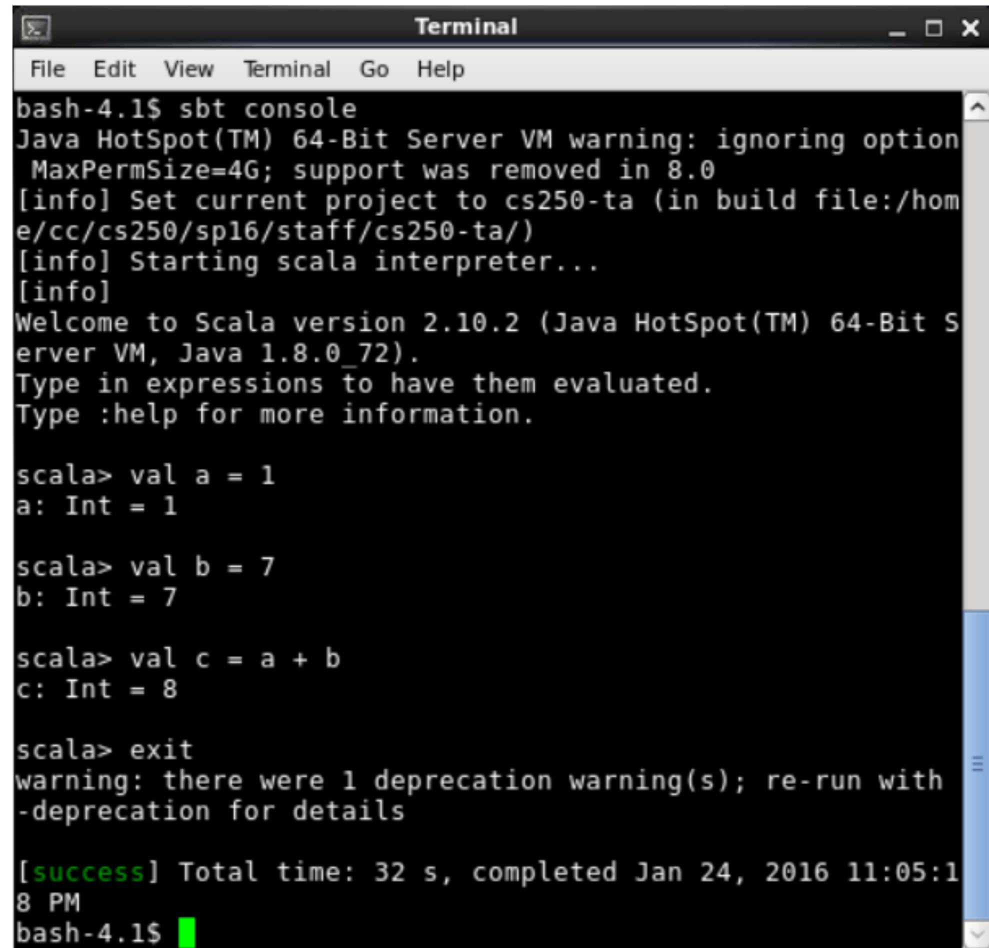
The Chisel Software Architecture



SCALA

Running Scala Interactively

- Scala provides a REPL (Read-Evaluate-Print Loop) interface
- Can be accessed using sbt
- `> sbt console`
- Can use this to try out scala snippets …
  - Or just as a cool calculator!



Semicolons & Ending Statements

- Semicolons are used in C type languages to denote the end of a statement
- Scala uses them for the same purpose
- Scala also infers line ends as the end of a statement
  - Except if the line ends in =, {, (, or an operator
- Semicolons can be used to separate statements in a single line

Valid Scala Syntax:

```
val a = 1 + 2  //preferred

val a = 1 + 2;

val a = 4; val b = 5; val c = a+b

val a = 1 + (
        2 + 3)
```

Information from *Programming Scala, 2nd Edition* by Dean Wampler and Alex Payne

Different Function Call Syntax

- Typical Java syntax is also typically valid scala syntax
- Methods with no arguments can omit ()
  - `object.fun()` is equivalent to `object.fun`
- Methods with no arguments can be called with a postfix syntax (**no longer recommended as semicolons are optional**)
  - `object.fun()` is equivalent to `object fun`
- Methods with one argument can be called with infix notation
  - `object.oneArg(5)` is equivalent to `object oneArg 5`
  - Infix notation should only be used for purely functional method (no side effects) or methods that take a function as a parameter
  - Should be used for high-order functions (map, foreach, …)

Information from *Scala Style Guide*: http://docs.scala-lang.org/style/method-invocation.html

`val` vs. `var`

- `val` defines a constant value
- `var` defines a variable
- vals are preferred in Scala
- Note that if val is a reference to an object, the underlying object may be modified but the reference cannot be changed

Information from *Scala for the Impatient* by Cay S. Horstmann
Scala Documentation: Concrete Mutable Collection Classes: http://docs.scala-lang.org/overviews/collections/concrete-mutable-collection-classes.html

Why are vals preferred? Functional Purity

- A *pure function* is one that has no side effects
- A function has a *side effect* when it relies on or changes some external state
- A pure functions will always yield the same return value with the same input data (arguments)
- *Purely functional languages* (Scala is not one) only allow pure functions
- You have been writing pure functions for a long time … in math class!
  - f(x) = x*x + 2x + 1 is a pure function!
- vars are *mutable* and represent state, vals are *immutable*
- Ideally, one would have some input and *apply* several functions to it

Information from *Scala for the Impatient* by Cay S. Horstmann

Defining Functions

- Functions:
  - `def fctnName(a: Type, b: AnotherType): ReturnType = { function body …}`
  - `def addInt(c: Int, d: Int) = c + d`
  - The return type is optional, the argument types are mandatory
  - If recursion is used, return type is mandatory
- Procedures: Functions without a return value
  - `def procedureName(a: Type, b: AnotherType) {procedure body …}`
  - `def printMe(a: Int) {println("Val="+a)}`

Information from *Scala for the Impatient* by Cay S. Horstmann

class vs. object

- Classes are very similar to Java classes
  - They define a sort of template from which objects can be instantiated.
  - They have member vals, vars, and functions (called methods when defined in a class)
  - When you have an object, you can call it’s class methods using the standard object.method() syntax

Information from *Scala for the Impatient* by Cay S. Horstmann

class vs. object

- The object keyword is used to describe *singleton objects*
  - A singleton object only has only *one* instance
  - You can think of it as defining a class and instantiating it only once

Information from *Scala for the Impatient* by Cay S. Horstmann

class vs. object
Companion Classes

- Why is there an object keyword?
- Java classes allow *static* methods and variables
  - Functions and fields that can be called without an object
    - Class.Function, Class.Feild
  - They can serve many uses including helper functions, factories, and more
- Scala does *not* allow static methods to be declared in classes
- Instead, scala defines *companion* objects with the same name as the class.
- Static methods are placed in the companion object
- Because the companion object has the name name as the class, a call to Class.method is actually a call to the method in the companion object

Information from *Scala for the Impatient* by Cay S. Horstmann

The `apply` function

- The `apply` function is syntactic sugar to overload the () operator.
- You can define an apply method in classes that allow you to use object(arg) in your code
- You can define an apply function in the companion object typically to act as a factory (returns an object of the companion class).

Information from *Scala for the Impatient* by Cay S. Horstmann

The `apply` function

```
class MyClass (name: String, id: Int) {
    var myName = name
    var myId = id
    def printMe() {println(myName + " ID: " + myId)}
    def apply() = myId
    def apply(a: Int) {myId = a}}

object MyClass {
    def apply(s: String) = new MyClass(s, 0)
}
val a = MyClass("bob")
a.printMe()
a(10)
a.printMe()
```

Output:
bob ID: 0
bob ID: 10

When to use `new`?

- `new` will call the constructor
- `val a = MyClass("bob")` calls the apply function in the companion object
- `val b = new MyClass("bob", 10)` calls the constructor

Information from *Scala for the Impatient* by Cay S. Horstmann

Control Statements and Loops

- If/else
  - Like Java

```
if (n<len) {
   //do something}
else if (n==len) {
   //do something
} else {
   //do something
}
```

- While
  - Also like Java

```
while (n<len){
   //do something
}
```

Information from *Scala for the Impatient* by Cay S. Horstmann

Control Statements and Loops

- For
  - Not like Java

```
for (i <- 0 until len){
   //do something
}
```

  - Also supports special syntax for nested loops

`for (i <- 0 to len1; j <- 0 until len2){`
`    //do something`
`}`
	`for (i <- 0 to len1){`
	`    for (j <- 0 until len2){`
	`        //do something`
	`    }`
	`}`

Information from *Scala for the Impatient* by Cay S. Horstmann

Control Statements and Loops

- For
  - Well … *actually* like Java but the *for each* variant
  - Can loop over elements in collection (using an iterator)
  - In the previous for loops, `0 until len` creates a *Range* that can be iterated over.

```
val a = Seq(1, 2, 3, 4, 5)
for (i <- a){
  //i takes the value of each
  //element in the sequence a
  println(i)
}
```

Information from *Learning Scala* by Jason Swartz

Ranges

- Ranges are used in several parts of scala
  - Especially in for loops
- Ranges are sequences of numbers that
  - Start at one number (inclusive)
  - End at another number (inclusive or exclusive)
  - By some increment

- `val a = 0 to 5`
  - a = (0, 1, 2, 3, 4, 5)
- `val b = 0 until 5`
  - b = (0, 1, 2, 3, 4)
- `val c = 0 to 4 by 2`
  - c = (0, 2, 4)
- `val d = 1.5 to 3.0 by 0.5`
  - d = (1.5, 2.0, 2.5, 3.0)

Information from *Programming Scala, 2nd Edition* by Dean Wampler and Alex Payne

Matching

- Like a C switch statement but much more powerful
- Can match on values, types, Boolean expressions, and more!
- A bit advanced for this course but good to know if you come across it
- Related to partial functions

```
val a = Seq(8, 6, 7, 5, 3, 0, 9)

for( i <- a ){
    val rtn = i match {
        case _ if i > 5 => "over"
        case 5          => "five"
        case _          => "under"
    }
    println(rtn)
}
```

Information from *Programming Scala, 2nd Edition* by Dean Wampler and Alex Payne

Tuples

- Tuples are simply ordered collections of data
- The values cannot be iterated over
- Data in tuples do not need to have the same type
- Data from tuples can be extracted using a special syntax where the index starts from 1
- The relation operator -> is syntactic sugar for making 2-tuples and is targeted at key-value pairs

```
val stuff = (3, 5.6, "hello")
```

stuff: (Int, Double, String) = (3,5.6,hello)

```
println(stuff._1)
```

3

```
println(stuff._2)
```

5.6

```
println(stuff._3)
```

hello

```
val keyValPair = "name" -> "Oski"
```

keyValPair: (String, String) = (name,Oski)

Information from *Programming Scala, 2nd Edition* by Dean Wampler and Alex Payne

Functions as "First Class Citizens"

- You have seen that you can assign a number to a val or var
  - `val a = 5`
- You can also assign a function to a val or var!
  - `val a = scala.math.abs _`
  - `a(-5)`
  - 5
  - The space, underscore specifies that you want to assign the *function* to `a` and not the value returned
- Functions can actually be passed around, just like numeric values!

Information from *Scala Cookbook* by Alvin Alexander

High Order Functions

- *High Order Functions* either:
  - take a function as an argument
  - return a function
- Why functions can be passed!

```
def highOrder(a: Int, b: Int, c:Int,
fun:(Int,Int) => Int) = {
    val tmp1 = fun(a, b)
    val tmp2 = fun(b, c)
    fun(tmp1, tmp2)
}

highOrder: (a: Int, b: Int, c: Int, fun:
(Int, Int) => Int)Int

def addInt(a:Int, b:Int) = a + b
addInt: (a: Int, b: Int)Int

val result = highOrder(2, 5, 7, addInt)
result: Int = 19
```

Information from *Becoming Functional* by Joshua Backfield
Scala Cookbook by Alvin Alexander

Anonymous Functions

- It seems silly to define a function like `addInt` if we only pass it to high order functions
- What we would like is define the function we want inside the function call
- We can do this with *anonymous functions*!
- *Anonymous functions* are sometimes called *lambda functions*, *closures*, or *function literals*
- There is a subtle difference between lambda functions and closures

```
def highOrder(a: Int, b:
Int, c:Int, fun:(Int,Int)
=> Int) = {
    val tmp1 = fun(a, b)
    val tmp2 = fun(b, c)
    fun(tmp1, tmp2)
}
```

```
val result = highOrder(2,
5, 7, (a, b) => a+b)
result: Int = 19
```

Types are not needed since the inference engine infers the types of a and b from highOrder

Information from *Becoming Functional* by Joshua Backfield
Scala Cookbook by Alvin Alexander

Anonymous Functions

- There is even more syntactic sugar for anonymous functions!
- Underscores can be used "as positionally matched arguments"

```
def highOrder(a: Int, b:
Int, c:Int, fun:(Int,Int)
=> Int) = {
    val tmp1 = fun(a, b)
    val tmp2 = fun(b, c)
    fun(tmp1, tmp2)
}

val result = highOrder(2,
5, 7, _+_)
result: Int = 19
```

Information from *Scala Style Guide*: http://docs.scala-lang.org/style/method-invocation.html

The Underscore

- The underscore _ is the *wildcard* character in Scala
- It is used in several contexts
  - Import statements to import all sub packages
  - In match statements to act as a placeholder
  - In assigning a function to a val
  - Anonymous functions to act as a placeholder

Useful High Order Functions

These functions take a Sequence or List and perform some operation

- Map
  - Applies a function to each element in a list and returns the resulting transform in a list
- Filter
  - Generates a new list with elements of the original list that fit some filter condition
- foldLeft, foldRight
  - A *reduce* operation where a list is traversed either from the left or from the right. In each step, a function is preformed on the list element and the result of the last fold operation, returning a single value.
  - Results in a single value at the end of evaluation

Information from *Programming Scala, 2nd Edition* by Dean Wampler and Alex Payne

zip, zipWithIndex and unzip

- Sometimes, you want to pair up values in two lists. You can do this with zip

```
val list1 = Seq(1, 2, 3, 4)
val list2 = Seq(5, 6, 7, 8)
val zipped = list1 zip list2
zipped: Seq[(Int, Int)] = List((1,5), (2,6), (3,7), (4,8))
```

- Unzip splits a list of tuples into a tuple of lists

```
val unzipped = zipped.unzip
unzipped: (Seq[Int], Seq[Int]) = (List(1, 2, 3, 4),List(5, 6, 7, 8))
```

- zipWithIndex zips each value with its index in the list

```
scala> val list1WInd = list1.zipWithIndex
list1WInd: Seq[(Int, Int)] = List((1,0), (2,1), (3,2), (4,3))
```

Information from *Scala Cookbook* by Alvin Alexander

CHISEL

Why val and not var?

- Like scala, chisel prefers using vals when possible
- Given that chisel is constructing a circuit, chisel constructs often represent physical things like nets, registers, …
- It can be easier to reason about the circuit when constructs are assigned a unique, constant name (within a scope).

What is going on with :=?

- Since we like to use vals when declaring chisel constructs, how do we make circuit assignments?
- Scala won't let you use =
- := is a special operator defined by chisel to represent circuit connections (similar to assignment in Verilog)
  - It defines which chisel constructs should be connected and in which cases
  - The when block will put in multiplexers as appropriate
- Data about connections is used by chisel to build a graph representation of the circuit

Scala vs. Chisel Types

- Scala and Chisel have different types
- This is because chisel needs information that Scala does not
  - Bit width information
- Unfortunately, the type inference system that Scala uses does not work as well with Chisel types.
  - Type promotion is not necessarily automatic (an artifact of Chisel's implementation
- You may need to cast between Scala and Chisel types
- You may need to cast between Chisel types

```
val a = UInt(1, width=8)
a.toBits //converts to bits
val b = 5
val c = Uint(b, width = 8) //cast from Int to UInt
                           //(actually construct obj)
```

Errors and Chisel

- Because Chisel is built on Scala and Scala is Built on Java, you can get 3 kinds of error messages
  - Scala compiler errors: Type mismatches, syntax, …
  - Chisel checks: Illegal chisel but legal scala
  - Java Stack Trace: Underlying implementation crashed

Exercise in Debugging

- Problem: Types
  - Chisel Type Expected
  - “Compile Time” Error
- Solution
  - Explicitly Cast to Chisel Type

```
val index  = Reg(init = UInt(0, width = log2Up(n)))
val memVal = UInt(width = w)
val done   = !io.en && ((memVal === io.data) || (index === UInt(n-1)))
// ...
when (io.en) {
  index := UInt(0)
} .elsewhen (done === Bool(false)) {
  index := index + 1
}
```

Line Number!

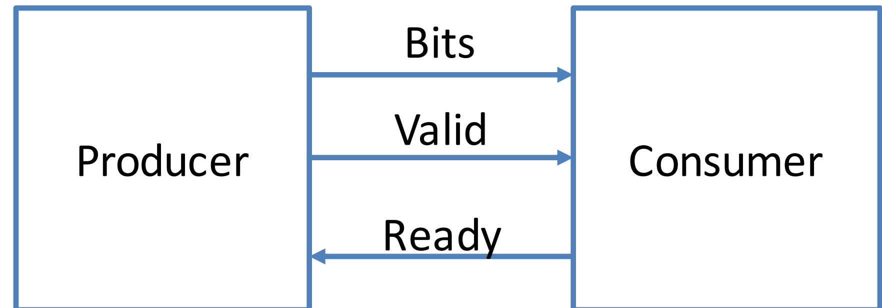
```
ms/DynamicMemorySearch.scala:24: overloaded method value + with alternatives:
[error]   (b: Chisel.SInt)Chisel.SInt <and>
[error]   (b: Chisel.UInt)Chisel.UInt <and>
[error]   (b: Chisel.Bits)Chisel.Bits
[error]  cannot be applied to (Int)
[error]     index := index + 1
[error]                    ^
[error] one error found
[error] (compile:compileIncremental) Compilation failed
[error] Total time: 4 s, completed Jan 18, 2016 6:34:43 PM
make: *** [DynamicMemorySearch.out] Error 1
```

```
val index  = Reg(init = UInt(0, width = log2Up(n)))
val memVal = UInt(width = w)
val done   = !io.en && ((memVal === io.data) || (index === UInt(n-1)))
// ...
when (io.en) {
  index := UInt(0)
} .elsewhen (done === Bool(false)) {
  index := index + UInt(1)
}
```

Decoupled

- The Decoupled interface is a ready/valid interface
- The producer drives the *bits* (data) and *valid* lines
- The receiver drives the *ready* line
- The producer raises the *valid* line when data is on the *bits* line
- The receiver raises the *ready* line when they are ready to receive data
- If *ready* and *valid* are both high, a *transaction* occurred (sometime called *fire*)
  - The receiver has to read in bits during the cycle that both ready and valid or high
  - The producer is allowed to put a new value on the bits line in the next cycle
  - If valid is still high in the next cycle, it is assumed that there is a new value

```
class DecoupledIO[+T <: Data](gen: T) extends Bundle{
    val ready = Bool(INPUT)
    val valid = Bool(OUTPUT)
    val bits  = gen.cloneType.asOutput
    //…
}
```



How do I find out about Chisel's Implementation

- Github provides good search capability
- Search for "class Bits" or "object Bits" for example
- Access at https://github.com/ucb-bar/chisel/
- You also have a copy of the chisel repo for lab1