Scala II

Muhammad Tahir*

Lecture 11

*Professor, Electrical Engineering Department, University of Engineering and Technology Lahore

Contents

Abstract Classes

- The `abstract` modifier with Class signifies that the class may have *abstract members*
- Abstract members:
  - do not have an implementation
  - do not require `abstract` modifier with the method name
- Abstract classes can not be instantiated

Abstract Classes Cont'd

```
abstract class Data_buffer {
    def contents: Array[String]
}
```

```
class Data_buffer_fill(data: Array[String]) extends
    Data_buffer {
    def contents: Array[String] = data
}
```

- `Data_buffer_fill` is a sub (derived) class that implements the method `contents` and as a result is a *concrete class*

Companion Objects

- An object defined using object keyword and has only one instance is called *singleton object*
- When a singleton object has the same name as that of a class, it is the companion object of that class
- The corresponding class is the *companion class*
- A singleton object that does not have a companion class is a *standalone object*

Companion Objects Cont'd

- Scala does not permit static methods to be declared in classes
- Scala defines companion objects with the same name as that of the class.
- Static methods are placed in the companion object
- A call to Class.method is actually a call to the method in the companion object

Companion Object Example

```
19  object Control {
20
21
22    val default =
23      //                                                                    kill                          wb_en  illegal?  en_rs2
24      //            pc_sel  A_sel   B_sel  imm_sel   alu_op   br_type |  st_type ld_type wb_sel  | csr_cmd | en_rs1 |
25      //              |       |       |      |          |          |   |     |       |       |    |  |      |  |     |
26                List(PC_4,   A_XXX,  B_XXX, IMM_X, ALU_XXX   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, Y, N,    N)
27    val map = Array(
28      LUI   -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_COPY_B, BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,    N),
29      AUIPC -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_ADD   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,    N),
30      JAL   -> List(PC_ALU, A_PC,   B_IMM, IMM_J, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, N,    N),
31      JALR  -> List(PC_ALU, A_RS1,  B_IMM, IMM_I, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, Y,    N),
```

```
class Control extends Module {
    val io = IO(new ControlSignals)
    val ctrlSignals = ListLookup(io.inst, Control.default, Control.map)

    // Control signals for Fetch
    io.pc_sel     := ctrlSignals(0)
    io.inst_kill := ctrlSignals(6).toBool

    // Control signals for Execute
    io.A_sel    := ctrlSignals(1)
    io.B_sel    := ctrlSignals(2)
    ...
```

Companion Objects Cont'd

- The apply method, in companion objects, is used to construct objects without keyword `new`
- An apply method takes construction parameters and constructs an object
- An `unapply` method takes an object and extracts values (parameters) from it
- An `extractor` is an object with an `unapply` method

unapply Method Illustration

```
class uModule(val name: String, val bitWidth: Int)

// companion object
object uModule {
     def apply(name: String, bitWidth: Int): uModule = new uModule(
       name, bitWidth)

     def unapply(mod: uModule): Option[(String, Int)] = {
          if (mod.bitWidth == 0) None
          else Some((mod.name, mod.bitWidth))
     }
}

// Using the apply method
val objA = uModule("ALU", 32)
// Extractor using unapply method
val uModule(module_name, module_bitW) = objA
println("Module name is: " + module_name)

// output at the terminal is
Module name is: ALU
```

Case Class

- A `case class` is like a regular class, but has some distinct features
- It adds a factory method with the name of the class
- This factory method manages the construction of object and does not require the use of keyword `new` when instantiating
- All arguments in the parameter list of a case class implicitly get a `val` prefix

Case Class Cont'd

- The compiler adds a *copy* method to the class for making modified copies
- Similar to case classes we have `case objects`
- One of the key advantages of case classes is that they support pattern matching
- The compiler automatically creates an `unapply` method that provides access to all of the class parameters

Case Class: Example

- An example illustration

```
case class register(addr:Int, init: Int)

var c = register(100, 11001100)

// Display the register
println("Reg Addr: " + c.addr + " Initial value: " + c.init)

// the output on the terminal
Reg Addr: 100     Initial value: 11001100
```

- Reassignment error

```
// when tried to update the address
c.addr = 15

// the following error is encountered
reassignment to val error
```

Case Class: copy Method Illustration

- copy method illustration

```
case class register(addr:Int, init: Int)

var c = register(100, 11001100)
var d = c.copy(addr = 104)

// Display the register
println("Reg Addr: " + c.addr + " Initial value: " + c.init)
println("Reg Addr: " + d.addr + " Initial value: " + d.init)
```

- Output on the terminal

```
// the output on the terminal
Reg Addr: 100     Initial value: 11001100
Reg Addr: 104     Initial value: 11001100
```

Case Class: Pattern Match

```
case class uModule(name: String, bitWidth: Int)

// pattern matching using unapply method of case class
def matchObject(obj: uModule) = {
     val result = obj match {
          case uModule(name, 32) => println("Module name is: " + name
          )
          case uModule(name, 16) => println("Module name is: " + name
          )
          case _                 => println(None)
     }
}
// instantiate different modules
val objA = uModule("ALU", 32)
val objI = uModule("Imm", 32)
val objB = uModule("Branch", 16)
val objM = uModule("Mul", 64)

matchObject(objI)

// output at the terminal
Module name is: Imm
```

Scala Map, map and flatMap

- `Map` is an immutable collection type
- `map` and `flatMap` are methods that can be applied to different collections
- `map` is a functional mapping applied to each element
- `flatMap` is a functional mapping applied to each element and flattens the results

Applying map Method

```
// An example list
val uList = List(1, 2, 3, 4, 5)

// map method applied to List
val uList_Twice = uList.map( x => x*2 )
println(s"List elements doubled = $uList_Twice")

// Applying map to List using user defined method
def f(x: Int) = if (x > 2) x*x else None
val uList_Squared = uList.map(x => f(x))
println(s"List elements squared selectively = $uList_Squared")

// The output at the terminal is given below
List elements doubled = List(2, 4, 6, 8, 10)
List elements squared selectively = List(None, None, 9, 16, 25)
```

map and flatMap Illustration

```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

def g(v:Int) = List(v-1, v, v+1)
val uList_Extended = uList.map(x => g(x))
println(s"Extended list using map = $uList_Extended")

val uList_Extended_flatmap = uList.flatMap(x => g(x))
println(s"Extended list using flatMap =
    $uList_Extended_flatmap")

// The output at the terminal is
Extended list using map = List(List(0, 1, 2), List(1, 2, 3),
     List(2, 3, 4), List(3, 4, 5), List(4, 5, 6))

Extended list using flatMap = List(0, 1, 2, 1, 2, 3, 2, 3,
    4, 3, 4, 5, 4, 5, 6)
```

map and flatMap Illustration Cont'd

```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

// Applying map and flatMap to List with builtin Options
    class
def f(x: Int) = if (x > 2) Some(x) else None
val uList_selective = uList.map(x => f(x))
println(s"Selective elements of List with .map =
    $uList_selective")

val uList_selective_flatMap = uList.flatMap(x => f(x))
println(s"Selective elements of List with .flatMap =
    $uList_selective_flatMap")

// Output at the terminal
Selective elements of List using .map = List(None, None,
    Some(3), Some(4), Some(5))
Selective elements of List using .flatMap = List(3, 4, 5)
```

Applying map and flatMap to Map

```
// An example Map using (key, value) pairs
val uMap = Map('a' -> 2, 'b' -> 4, 'c' -> 6)

// Applying .mapValues to Map
val uMap_mapValues = uMap.mapValues(v => v*2)
println(s"Map values doubled using .mapValues = $uMap_mapValues")

def h(k:Int, v:Int) =  Some(k->v*2)

// Applying .map to Map
val uMap_map = uMap.map {
     case (k,v) => h(k,v)
}
println(s"Map values doubled using .map = $uMap_map")

// Applying .flatMap to Map
val uMap_flatMap = uMap.flatMap {
     case (k,v) => h(k,v)
}
println(s"Map values doubled using .flatMap = $uMap_flatMap")

// The output at the terminal
Map values doubled using .mapValues = Map(a -> 4, b -> 8, c -> 12)
Map values doubled using .map = List(Some((97,4)), Some((98,8)), Some((99,12)))
Map values doubled using .flatMap = Map(97 -> 4, 98 -> 8, 99 -> 12)
```

Reading List I

- Read Chapters 15 and 16 of [Odersky et al., 2016]
- The tutorial available at [Tutorial, 2020] is good resource for quick reference
- Consult the [chisel3, 2020] for further details

References

chisel3 (2020).

Chisel3 library reference.

https://www.chisel-lang.org.

Odersky, M., Spoon, L., and Venners, B. (2016).

Programming in Scala.

Artima Incorporation.

Tutorial (2020).

Scala tutorial.

https://www.tutorialspoint.com/scala/index.htm.