19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Arrays, Lists and Tuples in Scala

PARAMETERIZATION

- In Scala, you can instantiate objects, or class instances, using new.
- When you instantiate an object in Scala, you can *parameterize* it with values and types.
- Parameterization means "configuring" an instance when you create it.
- An instance is parameterized with values by passing objects to a constructor in parentheses.
- Example:

```
val big = new java.math.BigInteger("12345")
```

Instantiates a new java.math.BigInteger and parameterizes it with the value "12345"

PARAMETERIZING AN INSTANCE WITH TYPES

- Specify one or more types in square brackets.

Example:

```
object parray
{
  def main(args:Array[String])
  {
    val greetStrings = new Array[String](3)
    greetStrings(0) = "Hello"
    greetStrings(1) = ", "
    greetStrings(2) = "world!\n"
    for (i <- 0 to 2)
      print(greetStrings(i))
  }
}
```

```
D:\PPL\Scala>scalac parray.scala
warning: 1 deprecation (since 2.13.0); re-run
with -deprecation for details
1 warning

D:\PPL\Scala>scala parray
Hello, world!
```

- In this example, greetStrings is a value of type Array[String](an "array of string") that is initialized to length 3 by parameterizing it with the value 3 in the first line of code.

- Note that when you parameterize an instance with both a type and a value, the type comes first in its square brackets, followed by the value in parentheses.

EXPLICIT PARAMETERIZATION

```
val greetStrings: Array[String] = new Array[String](3)
```

- Given Scala's type inference, this line of code is semantically equivalent to previous version: `val greetStrings = new Array[String](3)`
- But this form demonstrates that while the type parameterization portion (the type names in square brackets) forms part of the type of the instance, the value parameterization part (the values in parentheses) does not.
- The type of greetStrings isArray[String], not Array[String](3).

INITIALIZING EACH ELEMENT OF THE ARRAY

greetStrings(0) = "Hello"

greetStrings(1) = ", "

greetStrings(2) = "world!\n“

- Arrays in Scala are accessed by placing the index inside parentheses, not square brackets as in Java.
- Thus the zeroth element of the array isgreetStrings(0), not greetStrings[0].

MEANING OF VAL EXPLAINED…

```
val greetStrings = new Array[String](3)
```

- When a variable with is defined as val, the variable can't be reassigned, but the object to which it refers could potentially still be changed.
- In this case, greetStrings cannot be reassigned to a different array; greetStrings will always point to the same Array[String] instance with which it was initialized.
- But the elements of the Array[String] can be changed over time, hence the array itself is mutable.

THE TO METHOD

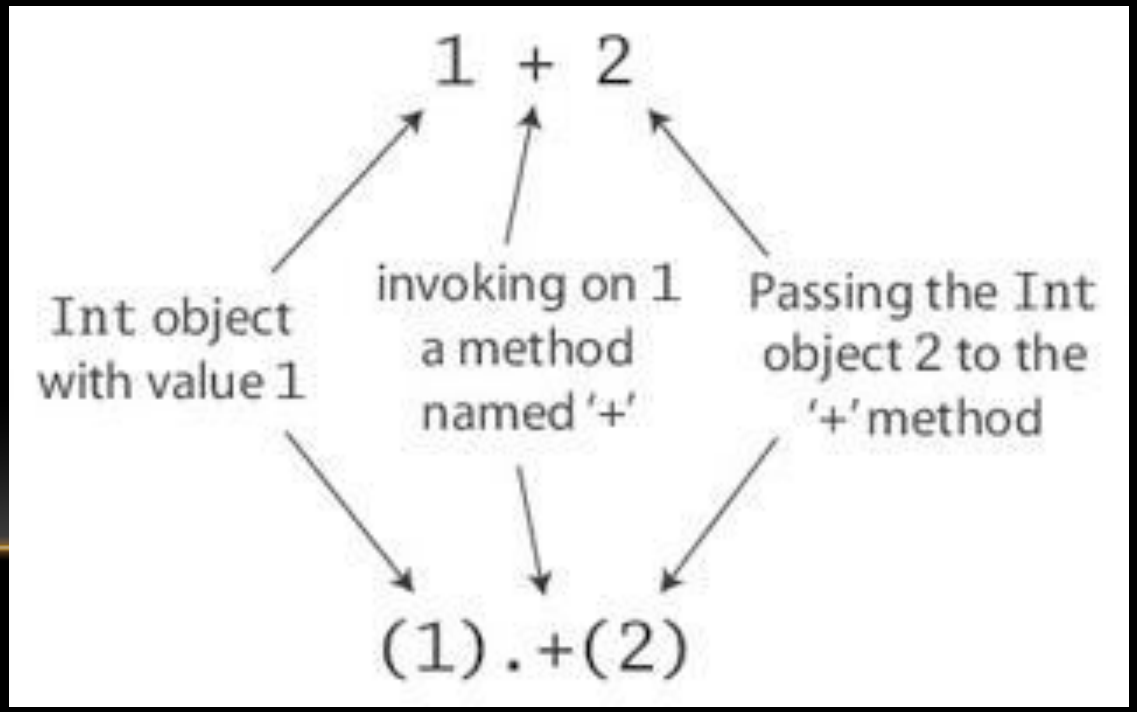
```
for (i <- 0 to 2)
  print(greetStrings(i))
```

- If a method takes only one parameter, you can call it without a dot or parentheses.
- The to in this example is actually a method that takes one Int argument.
- The code 0 to 2 is transformed into the method call (0).to(2).
- Note that this syntax only works if you explicitly specify the receiver of the method call.
- You cannot write "println 10", but you can write "Console println 10".

OPERATOR OVERLOADING

- Technically Scala doesn't have operator overloading, because it doesn't actually have operators in the traditional sense.
- Characters such as +, -, *, and / can be used in method names.
- Hence, when 1 + 2 is typed into the Scala interpreter in Step 1, a method named + is actually invoked on the Int object 1, passing in 2 as a parameter.
- Alternatively one could write 1 + 2 using traditional method invocation syntax, (1)+(2)

All operations are method calls in Scala



ARRAYS ARE CLASSES IN SCALA

- Arrays are accessed with parentheses in Scala.
- Arrays are simply instances of classes like any other class in Scala.
- Scala has fewer special cases than Java.
- When parentheses is applied surrounding one or more values to a variable, Scala will transform the code into an invocation of a method named apply on that variable.

  For e.g., greetStrings(i) gets transformed into greetStrings.apply(i).

- Thus accessing an element of an array in Scala is simply a method call like any other.
- This principle is not restricted to arrays: any application of an object to some arguments in parentheses will be transformed to an apply method call.
- Of course this will compile only if that type of object actually defines an apply method.
- So it's not a special case; it's a general rule.

UPDATE METHOD

- When an assignment is made to a variable to which parentheses and one or more arguments have been applied, the compiler will transform that into an invocation of an update method that takes the arguments in parentheses as well as the object to the right of the equals sign.

For example:

greetStrings(0) = "Hello“ → Transformed to → greetStrings.update(0, "Hello")

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
for (i <- 0 to 2)
print(greetStrings(i))
```

Equivalent

```
val greetStrings = new Array[String](3)
greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")
for (i <- 0.to(2))
print(greetStrings.apply(i))
```

SCALA OBJECTS

- Scala achieves conceptual simplicity by treating everything, from arrays to expressions, as objects with methods.
- There is no need to remember special cases, such as the differences in Java between primitive and their corresponding wrapper types, or between arrays and regular objects.
- Moreover, this uniformity does not incur a significant performance cost.
- The Scala compiler uses Java arrays, primitive types, and native arithmetic where possible in the compiled code.

A MORE CONCISE WAY TO CREATE AND INITIALIZE ARRAYS IN SCALA

```
val numNames = Array("zero", "one", "two")
```

- This is calling a factory method, named apply, which creates and returns the new array.
- This apply method takes a variable number of arguments and is defined on the Array *companion object*
- A more verbose way to call the same apply method is:

```
val numNames2 = Array.apply("zero", "one", "two")
```

USING LISTS IN SCALA

List Creation:

```
val oneTwoThree = List(1, 2, 3)
```

List Concatenation using ::: method:

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " and " + threeFour + " were not mutated.")
println("Thus, " + oneTwoThreeFour + " is a new list.")
```

```
List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new list.
```

CONS OPERATOR IN SCALA

- Prepends a new element to the beginning of an existing list and returns the resulting list
- Example:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

```
List(1, 2, 3)
```

INITIALISING NEW LISTS WITH CONS OPERATOR AND NIL

- Empty list is represented as Nil
- Example:

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

- The above script will produce the same output as List(1, 2, 3)

SOME LIST METHODS AND THEIR USAGES

What it is	What it does
List() or Nil	The empty List
List("Cool", "tools", "rule")	Creates a new List[String] with the three values"Cool", "tools", and "rule"
val thrill = "Will" :: "fill" :: "until" :: Nil	Creates a new List[String] with the three values"Will", "fill", and "until"
List("a", "b") ::: List("c", "d")	Concatenates two lists (returns a new List[String]with values "a", "b", "c", and "d")
thrill(2)	Returns the element at index 2 (zero based) of thethrill list (returns "until")
thrill.count(s => s.length == 4)	Counts the number of string elements in thrill that have length 4 (returns 2)
thrill.drop(2)	Returns the thrill list without its first 2 elements (returns List("until"))
thrill.dropRight(2)	Returns the thrill list without its rightmost 2 elements (returns List("Will"))
thrill.exists(s => s == "until")	Determines whether a string element exists in thrillthat has the value "until" (returns true)

SOME LIST METHODS AND THEIR USAGES

thrill.filter(s => s.length == 4)	Returns a list of all elements, in order, of the thrilllist that have length 4 (returns List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	Indicates whether all elements in the thrill list end with the letter "l" (returns true)
thrill.foreach(s => print(s))	Executes the print statement on each of the strings in the thrill list (prints "Willfilluntil")
thrill.foreach(print)	Same as the previous, but more concise (also prints"Willfilluntil")
thrill.head	Returns the first element in the thrill list (returns"Will")
thrill.init	Returns a list of all but the last element in the thrilllist (returns List("Will", "fill"))
thrill.isEmpty	Indicates whether the thrill list is empty (returnsfalse)
thrill.last	Returns the last element in the thrill list (returns"until")
thrill.length	Returns the number of elements in the thrill list (returns 3)
thrill.map(s => s + "y")	Returns a list resulting from adding a "y" to each string element in the thrill list (returnsList("Willy", "filly", "untily"))
thrill.mkString(", ")	Makes a string with the elements of the list (returns"Will, fill, until")
thrill.filterNot(s => s.length == 4)	Returns a list of all elements, in order, of the thrilllist *except those* that have length 4 (returnsList("until"))
thrill.reverse	Returns a list containing all elements of the thrilllist in reverse order (returnsList("until", "fill", "Will"))
thrill.sort((s, t) => s.charAt(0).toLower < t.charAt(0).toLower)	Returns a list containing all elements of the thrilllist in alphabetical order of the first character lowercased (returns List("fill", "until", "Will"))
thrill.tail	Returns the thrill list minus its first element (returns List("fill", "until"))

TUPLES

- Like lists, tuples are immutable, but unlike lists, tuples can contain different types of elements.
- Whereas a list might be a List[Int] or a List[String], a tuple could contain both an integer and a string at the same time.

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

```
99
Luftballons
```

- To instantiate a new tuple that holds some objects, just place the objects in parentheses, separated by commas.
- Once you have a tuple instantiated, you can access its elements individually with a dot, underscore, and the one-based index of the element.

TUPLES

- The actual type of a tuple depends on the number of elements it contains and the types of those elements.
- Thus, the type of (99, "Luftballons") is Tuple2[Int, String].
- The type of ('u', 'r', "the", 1, 4, "me") is Tuple6[Char, Char, String, Int, Int, String].

ACCESSING THE ELEMENTS OF A TUPLE

- The elements of a tuple cannot be accessed like the elements of a list, for example, with "pair(0)".
- Since list's apply method always returns the same type, but each element of a tuple may be a different type: _1 can have one result type, _2 another, and so on.
- These _N numbers are one-based, instead of zero-based, because starting with 1 is a tradition set by other languages with statically typed tuples, such as Haskell and ML.

ARRAY CREATION AND ACCESS – EXAMPLE

```
import Array._
object arraydemo
{
 val myarr1:Array[Int] = new Array[Int](4);
//Creation method 1
 val myarr2 = new Array[String](5);
//Creation method 2
 val myarr3 = Array(1.5,2.3,6.4);
//Creation method 3
 val myarr4 = Array(1,2,3,4);
 def main(args:Array[String])
 {
    myarr1(0)=20;              //Assignments
    myarr1(1)=50;
    myarr1(2)=10;
    myarr1(3)=30;
    for(i<-myarr1)             //Access Method 1
     {
       println(i);
     }
for(i<- 0 to (myarr1.length-1)) //Access Method
2
    {
      println(myarr1(i));
    }
    for(i<-myarr2)   //Prints default value if
unassigned
      {
        println(i);
      }
     println(myarr3.length);      //no of elements
    val join = concat(myarr1,myarr4);
//concatenates same array type
     for(i<-join)
      {
        println(i);
      }
   }
}
```

ARRAY CREATION AND ACCESS – EXAMPLE OUTPUT

```
D:\PPL\Scala>scala arraydemo
20
50
10
30
20
50
10
30
null
null
null
null
null
3
20
50
10
30
1
2
3
4
```

LIST CREATION AND PROCESSING

```
object listdemo
{
 val mylist1:List[Int] = List(1,2,3,4);
 val mylist2:List[String]=List("John", "Tim", "Ken");
 def main(args:Array[String])
 {
   println(mylist1);
   println(mylist2);

   //mylist1(0)=50;  //List is immutable - cant change
   //println(mylist1);

   println(50::mylist1); //cons :: to prepend
   println(mylist1);     //value does'nt change by ::

   println(1::3::5::Nil); //creates and prints a list

   println(mylist2.head); //to print first item
   println(mylist2.tail); //prints last item

   println(mylist2.isEmpty); //checks if list is empty
   println(mylist1.reverse); //prints list in reverse

   println(List.fill(5)(2)); //creates a list of 5 2's

   println(mylist1.max); //prints max item in list

   mylist2.foreach(println); //iterates thru each elt

   var sum:Int=0;            //sum of list using foreach
   mylist1.foreach(sum += _);
   println(sum);

   for(i<-mylist2)          //iteration using for
    {
      println(i);
    }
      println(mylist2(1));    //list indexing
 }

}
```

LIST CREATION AND PROCESSING - OUTPUT

```
D:\PPL\Scala>scala listdemo
List(1, 2, 3, 4)
List(John, Tim, Ken)
List(50, 1, 2, 3, 4)
List(1, 2, 3, 4)
List(1, 3, 5)
John
List(Tim, Ken)
false
List(4, 3, 2, 1)
List(2, 2, 2, 2, 2)
4
John
Tim
Ken
10
John
Tim
Ken
Tim
```

TUPLE CREATION AND ACCESS - EXAMPLE

```
object tupdemo
{
 val mytuple1 = (1,2,"hi",false);
 val mytuple2 = new Tuple4 (3,4,"hello",true); //no of
items to be specified - allowed upto 22 elements
 val mytuple3 = new Tuple3 (5,"welcome",(1,2));
//Tuple in tuple

 def main(args:Array[String])
 {
   println(mytuple1);
   println(mytuple2._3); //Tuples indexed from 1
   println(mytuple3._3); //To print nested tuple
   println(mytuple3._3._2); //To print nested tuple
element

   mytuple1.productIterator.foreach    //iterating thru
a tuple
   {
    i => println(i);
}
}
}
```

TUPLE CREATION AND ACCESS - OUTPUT

```
D:\PPL\Scala>scala tupdemo
(1,2,hi,false)
hello
(1,2)
2
1
2
hi
false
```

THANK YOU