CSC 735 – Data Analytics

Introduction to Scala

Classes

- A simple class

```
class Counter {
  private var value = 0 // You must initialize the field
  def increment()= { value += 1 }
  def current() = value
}
```

Classes

- A simple class

```
class Counter {
  private var value = 0 // You must initialize the field
  def increment()= { value += 1 }
  def current() = value
}

val myCounter = new Counter
```

Classes

- A simple class

```
class Counter {
  private var value = 0 // You must initialize the field
  def increment()= { value += 1 }
  def current() = value
}

val myCounter = new Counter
myCounter.increment()
```

Classes

- A simple class

```
class Counter {
  private var value = 0 // You must initialize the field
  def increment()= { value += 1 }
  def current() = value
}

val myCounter = new Counter
myCounter.increment()
println(myCounter.current())
```

Getters and Setters

- Scala provides getter and setter methods for every field

```
class Person {
 var age = 0
}
```

- In Scala, the getter and setter methods for age are called **age** and **age_=**

```
val fred = new Person
println(fred.age) // Calls fred.age => prints ?
```

Getters and Setters

- Scala provides getter and setter methods for every field

```
class Person {
 var age = 0
}
```

- In Scala, the getter and setter methods for age are called **age** and **age_=**

```
val fred = new Person
println(fred.age) // Calls fred.age => prints 0
fred.age = 21   // Calls fred.age_=(21)
```

Getters and Setters

- Scala provides getter and setter methods for every field

```
class Person {
 var age = 0
}
```

- In Scala, the getter and setter methods for age are called **age** and **age_=**

```
val fred = new Person
println(fred.age) // Calls fred.age => prints 0
fred.age = 21    // Calls fred.age_=(21)
println(fred.age) // Calls fred.age => prints ?
```

Getters and Setters

- Scala provides getter and setter methods for every field

```
class Person {
 var age = 0
}
```

- In Scala, the getter and setter methods for age are called **age** and **age_=**

```
val fred = new Person
println(fred.age) // Calls fred.age => prints 0
fred.age = 21   // Calls fred.age_=(21)
println(fred.age) // Calls fred.age => prints 21
```

Getters and Setters

- Why Getters and Setters?

```
val fred = new Person
fred.age = 30
fred.age = 21
println(fred.age)
```

Getters and Setters

- If the field is **private** then the automatically generated getter and setter methods will be **private**

Getters and Setters

- If the field is **private** then the automatically generated getter and setter methods will be **private**
- You can always override the provided getters and setters

Getters and Setters

- If the field is **private** then the automatically generated getter and setter methods will be **private**
- You can always override the provided getters and setters
- If the field is **val**, then only a **getter** will be created

Getters and Setters

- If the field is **private** then the automatically generated getter and setter methods will be **private**
- You can always override the provided getters and setters
- If the field is **val**, then only a **getter** will be created
- If you don’t want default getter or setter, declare the field as private[this]

Getters and Setters

- If the field is **private** then the automatically generated getter and setter methods will be **private**
- You can always override the provided getters and setters
- If the field is **val**, then only a **getter** will be created
- If you don’t want default getter or setter, declare the field as private[this]
- Use a different name for private fields and use the normal names for the methods

Getters and Setters

```
class Person {
  private var privateAge = 0  // Make private and rename

  def age = privateAge
  def age_= (newAge: Int)= {
      if (newAge > privateAge) privateAge = newAge;
  }
}
```

Getters and Setters

```
class Person {
  private var privateAge = 0  // Make private and rename

  def age = privateAge
  def age_= (newAge: Int)= {
      if (newAge > privateAge) privateAge = newAge;
  }
}
```

```
val fred = new Person
fred.age = 30 //  fred.age_=(30)
fred.age = 21 //  fred.age_=(20)
println(fred.age)
```

Auxiliary Constructors

- A Scala class can have any number of auxiliary constructors
- They are similar to constructors in Java and C++ but have the name **this**
- Each auxiliary constructor must start with a call to a previously defined auxiliary constructor or the primary constructor

Example - Auxiliary Constructors

```
class Person {
  private var name = ""
  private var age = 0















}
```

Example - Auxiliary Constructors

```
class Person {
  private var name = ""
  private var age = 0

  // An auxiliary constructor
  def this(name: String) ={
    this() // Calls primary constructor
    this.name = name
  }







}
```

Example - Auxiliary Constructors

```
class Person {
  private var name = ""
  private var age = 0

  // An auxiliary constructor
  def this(name: String)= {
    this() // Calls primary constructor
    this.name = name
  }

  // Another auxiliary constructor
  def this(name: String, age: Int)= {
    this(name) // Calls previous auxiliary constructor
    this.age = age
  }
}
```

Example - Auxiliary Constructors

```
class Person {
  private var name = ""
  private var age = 0

  // An auxiliary constructor
  def this(name: String) ={
    this() // Calls primary constructor
    this.name = name
  }

  // Another auxiliary constructor
  def this(name: String, age: Int) ={
    this(name) // Calls previous auxiliary constructor
    this.age = age
  }
}
```

```
val p1 = new Person
```

Example - Auxiliary Constructors

```
class Person {
  private var name = ""
  private var age = 0

  // An auxiliary constructor
  def this(name: String)= {
    this() // Calls primary constructor
    this.name = name
  }

  // Another auxiliary constructor
  def this(name: String, age: Int)= {
    this(name) // Calls previous auxiliary constructor
    this.age = age
  }
}
```

```
val p1 = new Person

val p2 = new Person("Fred")
```

Example - Auxiliary Constructors

```
class Person {
  private var name = ""
  private var age = 0

  // An auxiliary constructor
  def this(name: String)= {
    this() // Calls primary constructor
    this.name = name
  }

  // Another auxiliary constructor
  def this(name: String, age: Int) ={
    this(name) // Calls previous auxiliary constructor
    this.age = age
  }
}
```

```
val p1 = new Person

val p2 = new Person("Fred")

val p3 = new Person("Mark", 42)
```

The Primary Constructor

- Every class has a primary constructor
- It is intertwined with the class definition
- The parameters of the primary constructor are placed immediately after the class name

```
class Person(val name: String = "", val age: Int = 0) {
    // Parameters of primary constructor in (...)
    ...
}
```

- Parameters of the primary constructor turn into fields

The Primary Constructor

```
class Person(val name: String = "", val age: Int = 0) {
    // Parameters of primary constructor in (...)
    ...
}
```

- `name` and `age` become fields of the class
- constructing an object, sets values for the fields

```
val p1 = new Person("Fred", 42)
```

The Primary Constructor

- The primary constructor executes all statements in the class definition

```
class Person(val name: String = "", val age: Int = 0) {
    println("Just constructed another person")
    def description = name + " is " + age + " years old"
}
```

The Primary Constructor

- The primary constructor executes all statements in the class definition

```
class Person(val name: String = "", val age: Int = 0) {
    println("Just constructed another person")
    def description = name + " is " + age + " years old"
}
```

- Using default arguments in the primary constructor can eliminate auxiliary constructors

```
class Person(val name: String = "", val age:
Int = 0)
```

Singleton

- Situations:
  - a home is required for utility functions or constants
  - a single immutable instance needs to be shared efficiently

Singleton

- Situations:
  - a home is required for utility functions or constants
  - a single immutable instance needs to be shared efficiently
- We use **static** members in JAVA or C++; Scala does not support that

Singleton

- Situations:
  - a home is required for utility functions or constants
  - a single immutable instance needs to be shared efficiently
- We use **static** members in JAVA or C++; Scala does not support that
- All you need is a single instance of a class

Singleton

- Singleton objects are defined using the keyword **object**
- The constructor of an object is executed when the object is first used
  - singleton objects cannot take parameters
- Members will be called through the object name

Singleton - Example

```
//file name: singleton.scala
object Accounts {
  private var lastNumber = 0
  def newUniqueNumber() = { lastNumber += 1;
                                        lastNumber }
}

println(Accounts.newUniqueNumber()) //1
println(Accounts.newUniqueNumber()) //2
```

Companion Objects

- In Java or C++, we have a class with both instance methods and static methods
- Scala does not have the static methods
- **Companion objects** are used to achieve the purpose

Companion Objects

- A companion object is an object that has the same name as a class and is placed in the same file as the class
- The class and its companion object have access to each other’s **private** members

Companion Objects - Example

```
//file name: companionObject.scala
class Account {
  val id = Account.newUniqueNumber()
  private var balance = 0.0
  def deposit(amount: Double) ={ balance += amount }
  def description = "Account " + id + " with balance " + balance
}

object Account {// The companion object
  private var lastNumber = 0
  def newUniqueNumber() = { lastNumber += 1;
                                              lastNumber }
}
```

Companion Objects - Example

```
//file name: companionObject.scala
class Account {
  val id = Account.newUniqueNumber()
  private var balance = 0.0
  def deposit(amount: Double) ={ balance += amount }
  def description = "Account " + id + " with balance " + balance
}

object Account {// The companion object
  private var lastNumber = 0
  def newUniqueNumber() = { lastNumber += 1;
                                                lastNumber }
}
val acct = new Account
println(acct.description) //Account 1 with balance 0.0
acct.deposit(1000.0)
println(acct.description) //Account 1 with balance 1000.0
```

Example

```
//file name: companionObjectWithApply.scala
class Account private (val id: Int, initialBalance: Double) {
  private var balance = initialBalance
  def deposit(amount: Double) ={ balance += amount }
  def description = "Account " + id + " with balance " + balance
}

object Account { // The companion object
  def apply(initialBalance: Double) =
    new Account(newUniqueNumber(), initialBalance)
  private var lastNumber = 0
  private def newUniqueNumber() = { lastNumber += 1;
                                                              lastNumber }
}
```

Example

```
//file name: companionObjectWithApply.scala
class Account private (val id: Int, initialBalance: Double) {
  private var balance = initialBalance
  def deposit(amount: Double) ={ balance += amount }
  def description = "Account " + id + " with balance " + balance
}

object Account { // The companion object
  def apply(initialBalance: Double) =
    new Account(newUniqueNumber(), initialBalance)
  private var lastNumber = 0
  private def newUniqueNumber() = { lastNumber += 1;
                                    lastNumber }
}
val acct = Account(1000.0)
println(acct.description) //Account 1 with balance 1000.0
val acct2 = Account(2000.0)
println(acct2.description) //Account 2 with balance 2000.0
```

Application Objects

- An object declaration defines an entry point to an application if it contains a method called **main** that takes an argument of type Array[String] and returns Unit

```
object FirstApp {
    def main(args:Array[String]) {
        println("My first application.")
    }
}
```

Application Objects

- If the object extends the App trait, then you do not need to write a main method;
  - just write the code inside the constructor body

```
object FirstApp extends App{
    println("My first application.")
}
```

Data Structures

Maps

- A Map is a collection of key-value pairs
- Known as a dictionary, associative array, or hash map
- Allows us to index a value by a specific key for fast access

Creating a Map

- val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")
  This creates an immutable map whose contents cannot be changed

Creating a Map

- val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")
  This creates an immutable map whose contents cannot be changed

```
scala> capitals("France") = "Paris"
error: value update is not a member of
scala.collection.immutable.Map[String,String]
```

Creating a Map

- val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")
  This creates an immutable map whose contents cannot be changed

```
scala> capitals("France") = "Paris"
error: value update is not a member of
scala.collection.immutable.Map[String,String]
```

- If you want a mutable map, use
  var scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)

Creating a Map

- val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")
  This creates an immutable map whose contents cannot be changed

```
scala> capitals("France") = "Paris"
error: value update is not a member of
scala.collection.immutable.Map[String,String]
```

- If you want a mutable map, use
  var scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
- scores("George") = 9

Accessing Map Values

```
scala> scores
res342: scala.collection.mutable.Map[String,Int] = Map(George -> 9,
Bob -> 3, Alice -> 10, Cindy -> 8)
```

Accessing Map Values

```
scala> scores
res342: scala.collection.mutable.Map[String,Int] = Map(George -> 9,
Bob -> 3, Alice -> 10, Cindy -> 8)

scala> scores("Cindy")
res343: Int = 8
```

Accessing Map Values

```
scala> scores
res342: scala.collection.mutable.Map[String,Int] = Map(George -> 9,
Bob -> 3, Alice -> 10, Cindy -> 8)

scala> scores("Cindy")
res343: Int = 8

scala> scores("Mark")
java.util.NoSuchElementException: key not found: Mark
```

Accessing Map Values

```
scala> scores
res342: scala.collection.mutable.Map[String,Int] = Map(George -> 9,
Bob -> 3, Alice -> 10, Cindy -> 8)

scala> scores("Cindy")
res343: Int = 8

scala> scores("Mark")
java.util.NoSuchElementException: key not found: Mark

scala> scores.getOrElse("Mark", 0)
res345: Int = 0
```

Iterating over Maps

- Syntax: `for ((k, v) <- map) process k and v`

Iterating over Maps

- Syntax: `for ((k, v) <- map) process k and v`

```
scala> for ((k, v) <- scores) print(s"$k: $v\t")
George: 9	Bob: 3	Alice: 10	Cindy: 8
```

Iterating over Maps

- Syntax: `for ((k, v) <- map) process k and v`

```
scala> for ((k, v) <- scores) print(s"$k: $v\t")
George: 9	Bob: 3	Alice: 10	Cindy: 8

scala> scores.keySet
```

Iterating over Maps

- Syntax: `for ((k, v) <- map) process k and v`

```
scala> for ((k, v) <- scores) print(s"$k: $v\t")
George: 9	Bob: 3	Alice: 10	Cindy: 8

scala> scores.keySet
res347: scala.collection.Set[String] = Set(George, Bob, Alice, Cindy)

scala> scores.values
```

Iterating over Maps

- Syntax: `for ((k, v) <- map) process k and v`

```
scala> for ((k, v) <- scores) print(s"$k: $v\t")
George: 9	Bob: 3	Alice: 10	Cindy: 8

scala> scores.keySet
res347: scala.collection.Set[String] = Set(George, Bob, Alice, Cindy)

scala> scores.values
res348: Iterable[Int] = HashMap(9, 3, 10, 8)

scala> for (v <- scores.values) print(s"$v\t")
9	3	10	8
```

Tuples

- A tuple is a Scala collection which can hold multiple values together
- Unlike an array or list, a tuple can hold objects with different types
- Tuples are immutable
- Tuples are always used when you see parentheses wrapping around data without a specific type
- In Scala, a tuple can have up to 22 elements

Creating Tuples

```
scala> val t = new Tuple3(1, "hello", 20.3)
t: (Int, String, Double) = (1,hello,20.3)
```

Creating Tuples

```
scala> val t = new Tuple3(1, "hello", 20.3)
t: (Int, String, Double) = (1,hello,20.3)

scala> val t = (1, "hello", 20.3)
t: (Int, String, Double) = (1,hello,20.3)
```

Creating Tuples

```
scala> val t = new Tuple3(1, "hello", 20.3)
t: (Int, String, Double) = (1,hello,20.3)

scala> val t = (1, "hello", 20.3)
t: (Int, String, Double) = (1,hello,20.3)

scala> val t = new Tuple4(1, "hello", 20.3, true)
t: (Int, String, Double, Boolean) = (1,hello,20.3,true)
```

Creating Tuples

```
scala> val t = new Tuple3(1, "hello", 20.3)
t: (Int, String, Double) = (1,hello,20.3)

scala> val t = (1, "hello", 20.3)
t: (Int, String, Double) = (1,hello,20.3)

scala> val t = new Tuple4(1, "hello", 20.3, true)
t: (Int, String, Double, Boolean) = (1,hello,20.3,true)

scala> val t = (1, "hello", 20.3, true)
t: (Int, String, Double, Boolean) = (1,hello,20.3,true)
```

Accessing Tuple Values

- Use the methods _1, _2, _3

```
val t = (1, "hello", 20.3)
```

```
val first = t._1 // Sets first to 1
```

- Notice: index of the first element is 1, not 0

Accessing Tuple Values

- Use the methods _1, _2, _3

  ```
  val t = (1, "hello", 20.3)
  ```

  `val first = t._1` // Sets first to 1
- Notice: index of the first element is 1, not 0
- We can also do this:

  ```
      val (first, second, third) = t
  ```

  // Sets first to 1, second to hello, third to 20.3

Accessing Tuple Values

- Use the methods _1, _2, _3

```
val t = (1, "hello", 20.3)
```

`val first = t._1` // Sets first to 1

- Notice: index of the first element is 1, not 0
- We can also do this:

```
     val (first, second, third) = t
```

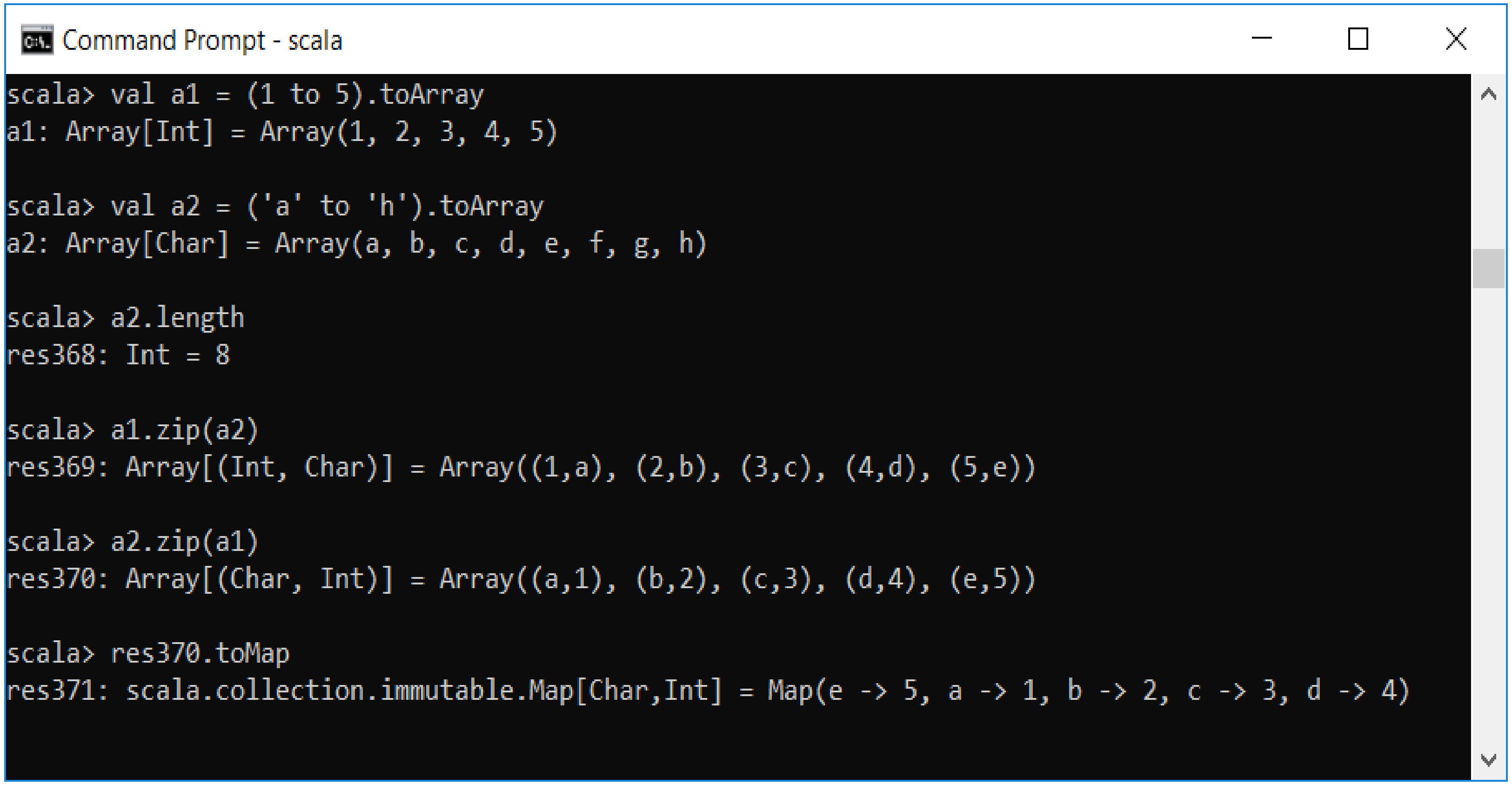
// Sets first to 1, second to hello, third to 20.3

- You can use a _, if you **don't** need all other components:

```
val (first, second, _) = t
```

Zipping

- Zipping allows us to combine the corresponding elements in two collections



```
scala> val a1 = (1 to 5).toArray
a1: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val a2 = ('a' to 'h').toArray
a2: Array[Char] = Array(a, b, c, d, e, f, g, h)

scala> a2.length
res368: Int = 8

scala> a1.zip(a2)
res369: Array[(Int, Char)] = Array((1,a), (2,b), (3,c), (4,d), (5,e))

scala> a2.zip(a1)
res370: Array[(Char, Int)] = Array((a,1), (b,2), (c,3), (d,4), (e,5))

scala> res370.toMap
res371: scala.collection.immutable.Map[Char,Int] = Map(e -> 5, a -> 1, b -> 2, c -> 3, d -> 4)
```