CompSci 220

Programming Methodology

03: Classes, Objects, and Git

Today's Goals

- Maps and Tuples (Highlights)
- IntelliJ: REPL and Worksheets
- Scala Classes
- Scala Objects
- Git Version Control

Scala Maps and Tuples

- A classic programmer's saying is,
  "If you can only have one data structure, make it a hash table"

- Hash tables – or more generally – maps are among the most versatile data structures. Scala makes it particularly easy to use them.

- **Maps are collections of key/value pairs.**
  Keys map to values

- **Tuples are aggregates of *n* objects, not necessarily of the same type.**
  A tuple of *n* objects is called an n-tuple.
  A tuple of 2 objects is called a pair.

Constructing a Map

- You can construct a map as

```
val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
```

- This constructs an *immutable* **Map[String, Int]**.

- If you want a *mutable* map, use

```
val scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
```

- Of course, you can import scala.collection.mutable.Map:

```
val scores = Map(“Alice” –> 10, “Bob” –> 3, “Cindy” –> 8)
```

Constructing Pairs

- The -> operator makes a pair, the value of

```
"Alice" -> 10
```

is

```
("Alice", 10)
```

- You could have similarly defined the map as

```
val scores = Map(("Alice", 10), ("Bob", 3), ("Cindy", 8))
```

Accessing Map Values

- The analogy between functions and maps is particularly close because you use the () notation to look up key values.

```
val bobsScore = scores("Bob") // Like scores.get("Bob") in Java
```

- If the map doesn't contain a value for the requested key, an exception is thrown.
- To check whether there is a key with a given value, do this

```
val bobsScore = if (scores.contains("Bob")) scores("Bob") else 0
```

- Since this is common there is a short cut that you see often:

```
val bobsScore = scores.getOrElse("Bob", 0)
```

Option Value

- The call map.get(key) returns an Option object that is either Some(value for key) or None.

```
val v = scores.get(“Bob”).getOrElse(0)
```

- We discuss this further when we look at functional programming techniques and how we handle exceptional control flow.

Updating Mutable Map Values

- In a mutable map, you can update a map value, or add a new one, with a () to the left of an = sign:

```
scores("Bob") = 10
```

- This either updates the existing value for the key “Bob”, or if the key does not exist it creates a new key/value pair in the map.
- You can also do this:

```
scores += ("Bob" -> 10, "Fred" -> 7)
scores -= "Alice"
```

Immutable Map Values

- You can't update an immutable map, but you can do this to create a newly constructed map:

```
val newScores = scores + ("Bob" -> 10, "Fred" -> 7)
```

- The newScores map contains the same associations as scores, except that "Bob" has been updated and "Fred" added.
- Instead of saving the result as a new value, you can use a var:

```
var scores = ...
scores = scores + ("Bob" -> 10, "Fred" -> 7)
```

Iterating Over Maps

- The following amazingly simple loop iterates over all key/value pairs of a map:

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

- The magic here is that you can use *pattern matching* in a Scala for loop. We will cover the mechanics of this when we talk about functional programming.
- If you need to just visit the key or values you do this:

```
scores.keySet
for (v <- scores.values) println(v)
```

Tuples

- A tuple value is formed by enclosing individual values in parenthesis:

```
(1, 3.14, "Fred")
```

  is a tuple of type **Tuple3[Int, Double, java.lang.String]**.
  Also written as:

```
(Int, Double, java.lang.String)
```

- If you have a tuple:

```
val t = (1, 3.14, "Fred")
```

  then you can access its components with the methods _1, _2, _3, …

```
val second = t._2 // Sets second to 3.14
```

Tuples and Pattern Matching

- You can use pattern matching, also known as destructuring assignment, to easily assign tuple contents to variables:

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

- If you do not need the third part, you can leave it out:

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

IntelliJ

- This is best done by example ☺
- This short tutorial will cover:
  - Creating a new Scala/SBT project
  - Creating a new Scala class/object
  - Using the REPL to access that class/object
  - Using Worksheets to interactively explore code

Simple Classes

- In there simplest form, Scala classes look like Java/C++

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

- In Scala, a class is not declared as **public**.
- A Scala source file can contain multiple classes, and all of them have public visibility.

Using Simple Classes

- To use this class, you construct objects and invoke methods.

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

- If you recall, you can drop parens for parameterless methods.

```
myCounter.increment() // Use () with mutator
println(myCounter.current) // Don't use () with accessor
```

Properties with Getters/Setters

- In Java, we do not like to use public fields:

```
public class Person { // This is Java
  public int age; // Frowned upon in Java
}
```

- This is preferable:

```
public class Person { // This is Java
  private int age;
  public int getAge() { return age; }
  public void setAge(int age) { this.age = age; }
}
```

A getter/setter pair is often called a **property**.

Why is this better?

Properties with Getters/Setters: Goodness

- Setters allow us to protect the values of fields.

```
public void setAge(int newValue) { if (newValue > age) age = newValue; }
  // Can't get younger
```

- This is good, but requires lots of Java boilerplate to do this.
- Scala likes to reduce boilerplate!

Properties: Scala Goodness

- Here is Scala's approach:

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

  def age = privateAge
  def age_=(newValue: Int) {
    if (newValue > privateAge) privateAge = newValue; // Can't get younger
  }
}
```

Properties: Scala Goodness

- Here is Scala's approach:

We can create a private field using **private**.

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

  def age = privateAge
  def age_=(newValue: Int) {
    if (newValue > privateAge) privateAge = newValue; // Can't get younger
  }
}
```

Properties: Scala Goodness

- Here is Scala's approach:

Define a getter…

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

  def age = privateAge
  def age_=(newValue: Int) {
    if (newValue > privateAge) privateAge = newValue; // Can't get younger
  }
}
```

Properties: Scala Goodness

- Here is Scala's approach:

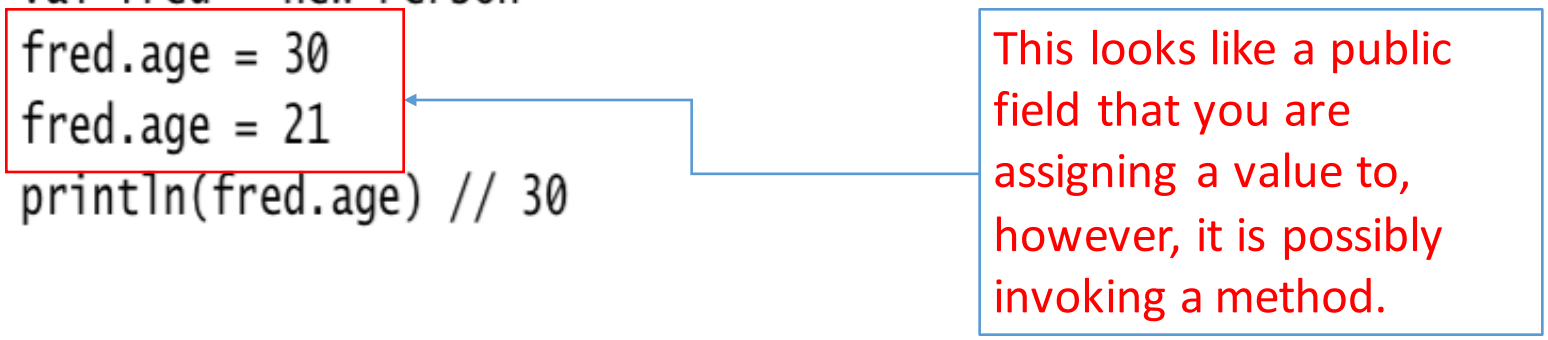
And define a setter…

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

  def age = privateAge
  def age_=(newValue: Int) {
    if (newValue > privateAge) privateAge = newValue; // Can't get younger
  }
}
```

Properties: Scala Goodness

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



Sometimes you need only a getter…

- When you only need a getter and do not want to field to be modified what can you do?

Sometimes you need only a getter…

- When you only need a getter and do not want to field to be modified what can you do?

```
class Message {
  val timeStamp = new java.util.Date
  ...
}
```

Constructors

- In Scala, you may have as many constructors as you wish. There are two types of constructors:
  - ***Primary constructor***: this constructor is the "base" constructor. We will talk more about that in a moment.
  - ***Auxiliary constructor***: these constructors are called *this*. Each auxiliary constructor must start with a call to a previously defined constructor.

Auxiliary Constructors

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

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

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

You define auxiliary constructors with **this**.

Auxiliary Constructors

```
val p1 = new Person // Primary constructor
val p2 = new Person("Fred") // First auxiliary constructor
val p3 = new Person("Fred", 42) // Second auxiliary constructor
```

Primary Constructor

- Every class has a primary constructors that is part of the class definition. The arguments are placed right after the class name.

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

Primary Constructor

- Every class has a primary constructors that is part of the class definition. The arguments are placed right after the class name.

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

The parameters of the primary constructor become fields that are initialized with the construction arguments.

Primary Constructor

- The primary constructor *executes* all statements in the class definition.

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

This would be executed when a new Person object is created:

```
val p = new Person("Pixel", 14)
// Just constructed another person
```

Primary Constructor

- The primary constructor *executes* all statements in the class definition.

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

The parameters are accessed like a any other class field.

Nested Classes

```
import scala.collection.mutable.ArrayBuffer
class Network {
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Member]
  }

  private val members = new ArrayBuffer[Member]

  def join(name: String) = {
    val m = new Member(name)
    members += m
    m
  }
}
```

Scala Objects: Definition

- Scala has not static methods or fields.
- Instead you use the **object** construct.
- An **object** defines a single instance of a class.

```
object Accounts {
  private var lastNumber = 0
  def newUniqueNumber() = { lastNumber += 1; lastNumber }
}
```

Scala Objects: Uses

- You use an object in Scala whenever you would have used a singleton object in Java, C++, etc. In particular,
  - As a home for utility functions or constants
  - When a single immutable instance can be shared efficiently
  - When a single instance is required to coordinate a service

Companion Objects

- In Java and C++ you often have a class with both instance methods and static methods.
- In Scala, you achieve this by having a class and a **companion object** of the same name.
- The class an its companion object can access each other's private features. They must be located in the same source file.

```
class Account {
  val id = Account.newUniqueNumber()
  private var balance = 0.0
  def deposit(amount: Double) { balance += amount }
  ...
}

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

Objects Extending Classes

- It is often useful to specify default objects that can be shared.
- That is, a *special* object that represents a default, an initial setting, or something unique or individual.
- We only need one of these, so we create single object that extends a base class.
- This is called the **flyweight** pattern.

https://en.wikipedia.org/wiki/Flyweight_pattern

```
abstract class UndoableAction(val description: String) {
  def undo(): Unit
  def redo(): Unit
}

object DoNothingAction extends UndoableAction("Do nothing") {
  override def undo() {}
  override def redo() {}
}

val actions = Map("open" -> DoNothingAction, "save" -> DoNothingAction, ...)
  // Open and save not yet implemented
```

Apply Method

- Remember that special method **apply** that is used when you try to use an object as a function call:

$$Object(arg1, \ldots, argN)$$

- For example, the Array object defines **apply** methods that allow array creation with expressions such as:

```
Array("Mary", "had", "a", "little", "lamb")
```

- Why not use **new**?
  It makes doing this nicer: `Array(Array(1, 7), Array(2, 9))`

Apply Method: Example

```
class Account private (val id: Int, initialBalance: Double) {
  private var balance = initialBalance
  ...
}

object Account { // The companion object
  def apply(initialBalance: Double) =
    new Account(newUniqueNumber(), initialBalance)
  ...
}
```

Now you can construct an account as

```
val acct = Account(1000.0)
```

Application Objects

- Come in two flavors…

```
object Hello {
  def main(args: Array[String]) {
    println("Hello, World!")
  }
}
```

```
object Hello extends App {
  println("Hello, World!")
}
```