**Scala Day 1 - Introduction:**

**Scala** **-> Scalable Programming Language**

* Runs on the standard JVM (can run on the JVM anywhere)
* *Functional Programming Language and Object Oriented*
* Spark framework can be run in Scala (65%), Java 8 (25%), Python (ML), and R (Rarely Used)
* Spark is also written in Scala
* Has a specific compiler that creates the same class files as Java
* Has all of the features the same as Java and more
* Scala can take advantage of all Java packages

**Execution/Development Environment:**

* Install Java > 5 (8 recommended)
* Install Scala
* Install Intellij
* Dependency Management
  + Java -> Maven
  + Scala -> SBT (Scala Build Tool)
    - Will download dependencies at compile time

**Java vs. Scala:**

* Java is an OOP language while Scala is a pure OOP language (Pure OOP means there are no primitive data types -> All objects / No operators -> All methods)
* Java defines functions like (public static void main(String[] args) -> Scala can write it this way which is called the Imperative way
* Scala can also write functions the functional way (def functionName(argA, argB:dataType) -> Much less lines of code
* Java always needs to specify the datatype while Scala does not (Scala is a static typed language and dynamically typed (can specify types if you want))
* val -> REPL (Read Evaluate Print Loop) - No reassignment
* var -> Can be initialized and reassigned any amount of times
* Scala does not need semicolon (optional) while Java does
* In Scala, normal operators are implemented as functions (ex. a.sum(b))
* In Scala, . and brackets are also optional (ex. a sum b)
* In Scala, can use normal operators but not normally done
* Java has the diamond inheritance problem and Scala avoids the diamond inheritance problem by using train linearization. Basically, it looks up the method implementation in the traits you extend from right to left

**Scala Day 2 - Basics:**

* Single line comments - //
* Double line comments - /\* \*/
* Keywords (Reserved words) - same rules as other programming languages
* println function same as java
* Variables in Scala (uses type inferencing)
  + var - assigns a value to a variable but can be changed later
  + val - assigns a value to a variable but it becomes immutable
  + var/val variableName:Type (can define type outright)
  + Default data type for decimal numbers is double
  + Default data for normal numbers is integer
  + Sample functions to change datatype
    - .toByte (data type specific)
    - .isInstanceOf[TYPE] (can always be used on any variable)
* Semicolons completely optional
* .Any (Master Class)
  + Any Vale (Subset extends Any)
    - Int, Char, Byte, etc…
* Operators in Scala (operators work the same as scala)
  + Arithmetic
  + Relational
  + Logical
  + Assignment
  + Bitwise
* \*\*\*Can do a concatenation of string and a different data type
* Prefix, indix, postfix all work the same as java
* Can do string interpolation using s”This is a string ${interpolatedVariable}”

**Scala Day 3 - Setting Up IntelliJ:**

* Set up intellij and got a hello world program written

**Scala Day 4 - Functions, Classes, Objects:**

* Look at code ScalaDay4 folder
* build.sbt will contain all of the packages and dependencies for our scala code
* Functions/Methods
  + args: param1, param2, etc - To name the arguments that the function takes
  + :Unit = { … } means that the function is returning nothing
  + Can use scalac to compile the object .scala file
    - Then use scala objectFileName to run the program
  + We don't use normal java class syntax because there is no static keyword in scala but main needs to be static in order to run correctly in java
  + Can omit return keyword
  + You do not need to specify the return type IF you do not specify the return keyword
* Classes / Objects
  + Use class to define a class
  + Can just define the variables outright
  + 3 access levels (Same as Java)
    - Public - access anywhere
    - Private - can access only in the class in which it is defined
    - Protected - can access in class it is defined and any subclasses that inherit from it
  + 3 access levels above also applies to member functions (this is no different than Java)
  + this keyword works the same as Java
* Can run main object by extending App instead of a main method

**Scala Day 5 - Constructors:**

* The constructor consists of a class that takes arguments
* Cannot remove the datatype from the constructor arguments
* Can specify default values for the default constructor by using the arguments as such… argName:type = defaultValue, …..
* Can specify private args in the argument list of the class declaration
* 2 types of constructors
  + Primary Constructor
    - Can have 0 or more parameters
    - args can be var or val
    - Only primary constructor can invoke a super constructor\*\*\*\*
    - Only one primary constructor
  + Auxiliary Constructor
    - Used to initialize the objects state
    - Statements are executed at object creation
    - def this(.....)
    - Can define multiple auxiliary constructors but they must have different parameter lists
    - Can call this(primaryConstructorParams) within the auxiliary constructor to call the primary constructor with the different parameter set

**Scala Day 6 - Singleton Objects, Companion Classes/Objects, and Case Classes/Objects:**

* Singleton Object
  + The main function is written in an Object, not a Class
  + Cannot create an object instance inside of an object (must be a class)
  + You can reference the object directly instead of creating an instance of the object first
  + Even though it is defined as an object, a class is created behind the scenes for the JVM
  + It is actually just an instance of an object in a class behind the scenes
* Companion Class and Object
  + The name of the class and object are the SAME
  + If there is a variable in the object that is not in the class, the variable can be used by the class referenced by the companion name
* Case Class and Object
  + Supports pattern matching
  + OOP → Functional programming
  + Supports functional programming
  + Makes functional programming easier
  + Just a regular class + a lot of extra auto generated code from the compiler
  + No need to write new keyword
  + Define the case class using “case class className”
  + Class parameters are default val so you cannot change the parameter values after object instantiation
  + However, if you change the constructor val to var then everything will work as a normal class
  + Also used for matching
  + The case class auto-generates the apply method that is used for pattern matching
  + Equals and hashcode methods (directly compare two objects) are auto-implemented as well as the toString method is auto-implemented

**Scala Day 7 - Strings:**

* Array of characters just like Java
* Getting length is just like Java
* Accessor Method - any method used to get information about an object
* Pretty much everything in Java for strings works exactly the same as it does in Scala
* == is same as .equals but it also checks to make sure that the two variables are not null
* Multiline strings are instantiated like python but require | per line and the .stripmargin end function to show properly (You want this so you can keep your indentation in the source code but it won't take into account the blank space before the | character in the actual terminal)
* “s” interpolation uses s”$variableName” or s”${variableName/variableName}”
* “f” interpolation uses f”$variableName%8.2f”
* raw interpolation ignores all \ characters
* Many other functions in Java that are all in Scala

**Scala Day 8 - Pattern Matching and Regex:**

* We want to match some sequence of data
* We can match patterns using the match keyword
* variableToMatch match { case match => returnValue, ….. }
* Can do matching with a case class (case class adds functionality that makes it easy to use with matching)
* Will do the first action that matches the case and then will leave the case matching loop
* Can loop through matching as well
* Regular expressions are adopted from Perl and work the same from Java
* Import scala.util.matching.Regex (for regex expressions)
  + Must also create an object of a type Regex that will handle all of the functionality
  + Or you can create a string and append the .r afterwards (ex. “pattern”.r)
* String passed to the regex does not need to be an exact regular expression
* There are a lot of options when it comes to regular expression formatting (see documentation)

**Scala Day 9A - Implicit Class:**

* New feature in scala
* Uses the “implicit” keyword
* Sometimes we have class that we cannot make any code changes (external library, standard libraries, etc)
* What if we want to add a new function to the class in question? We can do this with implicit classes
* Or we can extend the class that we want and add a new function
  + This doesn't work very well because if the class in question is a final class then it cannot be extended
* Implicit class must be defined within an object, cannot be defined at the top level

**Scala Day 9B - Conditional Statements and Loops:**

* Conditional Statements
  + If else is just like how it is in Java
  + Uses all the same operators as Java
* Loops
  + While loops work the same as in Java
  + Do while loops work the same as in Java
  + For loops are much different (SEE DOCUMENTATION AND CODE EXAMPLES)
    - Ex: for (i <- 1 to 10) <= <=
    - for (i <- 1 until 10) <= <
    - Nested for loops work the same as in Java (there is a scala version) for (i <- 1 to 10; j <- 1 to 10) instead of normal paired loop
    - Scala List is a tuple in python
    - Scala array is a list in python
    - Can add filter to loops in the same for loop definition
      * For (i <- listName if i % 2 == 0)
    - Can use the yield keyword to return a value from a list and assign it to a variable
  + Breaks class can break out of a loop (breakObject.breakable { breakObject.break}

**Scala Day 10 - Packages and Imports:**

* Package (are used to create)-> Namespace (modularize the code)
* Packages provide the namespace (directory) so that we can modularize the code
* You can have a package inside of a package, this is more of a c++ way of doing things, not really used in scala
* When creating a package name, we want to invert the website name (com.google.selfdrivingcar.camera)
* package <top level domain>.<domain name>.<project name>.<class name>/<function name>
* Package -> modularizing the code
* Import -> accessing members of the code
* Do not need to import java.lang.\_, scala.\_, and scala.Predef
* Can import all using .\_
* Just like in Java, the class/object name MUST match the file name and will be imported as so
* Can import with an aliased name .{realName=>alias}
* We can use an alias to help the compiler know what constructor to use when instantiating classes with the same name but are from different packages
* You can also specify the entire path with package name when instantiating the “duplicate” object and that will suffice
* In Java, you need to write the import statement at the beginning of the program, but in Scala, you can write it wherever you want
* Definitely common practice to keep the import at the top though\*\*\*\*
* If the import cannot find the method, then it will also search every defined implicit class

**Scala Day 11 - Exception Handling:**

* Want to be able to handle logical errors and any other type of exception
* In Java, exceptions are put into 2 categories
  + Checked expressions - very common errors (file not found, class not found, etc) (always need to check and handle these errors accordingly)
  + Unchecked expressions - everything else
* In Scala, everything is an unchecked expression
* Whenever there is an expression, it is converted into an unchecked expression and is an error
* For handling exceptions, you can throw an error or handle that exception
  + try catch
  + try(class) success/failure
  + catch object
* Can also throw the exception from another program to the parent running program
* try catch works similarly to python, but in the catch block, you use case: exceptionName => { //code }
* Can add a finally block that will execute after the catch
* case \_: will catch all unhandled exceptions
* You don't have to write “case **e**:”, you can write other variables, it is just for the error that is being caught
* See code examples for Try, Success, and Failure
  + You can wrap the vulnerable code in a Try() object and then do a match on the return value for Success(value) and Failure(exception)
* See code examples for catching class exceptions
  + You can create a catching class that specifies the exceptions to catch and then what to do based on that exception, then you can wrap the return class around the “testing” function as is done with Try Success Failure

**Scala Day 12 - Traits, Value Class, and Universal Trait:**

* Traits encapsulate methods and field definitions
  + Must use trait keyword
  + Is a class with some special properties
  + You do not create an object from it
  + Used only for inheritance purposes
  + You cannot extend more than one class, but you can extend a class and one or more traits all at the same time
  + Unimplemented methods in traits should be implemented in class extending given trait (like an interface)
  + class className extends className with traitName
  + Can override methods that are already defined as done in Java (no @Override) just override keyword in the function definition
* Abstract classes are how they are implemented in Java
  + Cannot create an object from it and cannot implement
  + A class can only extend ONE abstract class
* Overriding variables will default to the last class that implements the variable
* You can force other classes to to implement certain traits and other abstract classes when implementing that specific trait
* Value classes
  + Cannot create an object at runtime
  + Value class always has only 1 parameter with type as val
  + You can not extend a value class
  + Value class cannot extend a trait
  + See value class example in code
  + Benefits
    - Less initialization
    - Better performance
    - Less memory usage
    - For performance and memory optimization

**Scala Day 13 - Arrays:**

* A collection of same datatype elements
* Array is a class in Scala just like in Java
* var arrayName:Array[type] = new Array[type](amount)
  + Optional to write on the left but is needed if you give an initializer list on the right
* Access elements in an array using () not []
  + arrName(0) = 10
* All array properties are the same as in Java (cannot change length of array, etc)
* Can have an array of type Any
  + Works like a list in python
* All normal array functions from Java are present
* Can use map and foreach loops (like javascript and c++ respectively)
* arrayName.foreach(function)
  + It will execute a function for each element in the array
* For foreach, you can get multiple outputs or one final output
* With map, you will always get multiple outputs
* If you want to do the same operation on each element of an array, use map for efficiency
* For getting, single output, use foreach for efficiency

**Scala Day 14 - Arrays Part 2:**

* There is a lot more stuff you can do with arrays in scala
* reduceLeft / reduceRight
  + Do compound operations from starting from the left hand side
  + There is shorthand for these and other special functions
  + See examples
* Multidimensional arrays can be made using ofDim[type](val, val, …)
* Indexing is the same as Java
* Can do a foreach on a range
* Can use range function just like in Java
* You can also do literally an array of arrays to create a multidimensional array
* Can make a dynamic length array (vector in c++) using ArrayBuffer

**Scala Day 15 - Functional Programming 1:**

* Functions - group of statements for performing a specific task
* Minor difference between method and function (method is defined within a class (has name, signature, and bytecode)) (function is independent from a class)
* Function definition
  + def functionName(arguments): returnType = { function body … return value}
* Basic functions work the same as in Java
* Can take multiple arguments with args:String\*
  + Variable number argument should always be at the end
  + You cannot have more than one type of variable arguments
* Default parameters are the same as in Java and Python
* Recursive functions work the same as in Java

**Scala Day 16 - Functional Programming 2:**

* Does support anonymous functions
  + var variableName = (arguments) => returnType or {multipleCommands; returnType}
  + See code examples
* Supports first class functions - you can pass functions as a parameter and define function values
  + See code examples
  + funcArgName: (args) => returnType
  + Very much like c++

**Scala Day 17 - Functional Programming 3:**

* Chained functions get called in the same order as Java
* Partially applied function
  + We can write a lambda incomplete function that wraps another function, and can keep the same parameters from call to call
  + Allows for less arguments to have to be passed and for much cleaner code and less repeated code
  + See code example
* Nested functions
  + Can define functions inside of functions just like in Python
* Carrying functions
  + A function that splits the arguments into multiple argument trees
  + def functionName (arglist1) (arglist2) = returnBody
* Carrying function using partially applied function
  + See code example

**Scala Day 18A - Closures:**

* Closures: variables that are defined outside function definition
* Very simple concept
* Variables that are outside of function scope

**Scala Day 18B - Collections:**

* Can be lazy or strict
  + Strict means that data is pulled directly from memory
  + Lazy means that memory is assigned only when you access the particular data
* Can be mutable or immutable (depends on the type of collection)
* Collections are just special classes that are provided to us in Scala
* There is a rich set of collections (Java, Python, and other functionalities)
* Traversable
  + Traverse through particular collection
* Iterable
  + Access elements one by one
* Seq
  + Collection of elements that have a particular order
* Set
  + Collection of unique elements (no duplicates)
* Map
  + Collection of key-value pairs
* Immutable
  + We cannot change the contents of the collection
* Mutable
  + We can change the contents of the collection
* Can use the :+ operator to only overwrite the value of the variable not the immutable list itself

**Scala Day 19 - Collections - List:**

* Similar to array (collection of elements)
* Immutable
* Implemented as a linked list
* Can also create a special list using :: notation and Nil
  + See code
* Can create a list a bunch of different ways with different notations
  + See code
* Simple operations on list are just like in Java
* :+ will create a new copy from an immutable collection and overwrite the previous list
  + listName :+ newAttributeName
  + newAttributeName +: listName
* Complex operations on list
  + See code

**Scala Day 20 - Collections - List:**

* More complex list operations
* Tabulate - you can apply some function to generate each element in a list
* Can use .reverse to reverse the order of a list
  + Sorts in ascending order by default
* Sorting methods
  + See code
* Can replace sorting logic with your own function
* listBuffer
* Many functions are available to a list (just google specification and use as needed)

**Scala Day 21 - Collections - Sets:**

* Can be mutable or immutable
  + Must specify
* Elements are unique
* Will not keep the sequence (will not keep ordering)
* Default immutable
* Mutable set is also known as a hashset
* Use ++ to combine two sets
  + See code
* Will ignore repeated elements
* Works the same as in python
* Iteration operations are much faster
  + Stored as pairwise (not in order)
* Random retrieval of elements is slower than list
* Can use a SortedSet which is actually a TreeSet underneath
* You can also use a LinkedHashSet
  + Preserves the sequence (order) in which elements are added
* There is also a Queue object

**Scala Day 22 - Collections - Map:**

* Key value pair object
  + Same as in C++
* Each key is unique
* Can be mutable or immutable
* Default immutable
* Works just like a dict in python as well
  + Map(key -> value,....)
* Values can be repeated but keys cannot
* Sequence of data is not preserved
* \*\*\*Values must all be of the same type unless map is defined as Any type
* += will return a new map object in the case of an immutable map
  + Can only be saved if map object is of type var
* += will modify the existing map object in the case of a mutable map
* When using foreach, must use mapItem.\_1 for key and mapItem.\_2 for corresponding value
  + See code
* Must specify .valuesIterator. to iterate over values instead of keys
* Dont use special map functions (only use valuesIterator to iterate over values in a map)
* ListMap creates a map using a list based architecture (see documentation)
* There are many types of maps to mess around with (see documentation)

**Scala Day 23 - Tuples, Options, and Iterator:**

* Tuples
  + Fixed number of elements (1 to 22)
  + Immutable
  + tupleName = (elements)
  + Just like in python
  + Can specifically call a numbered tuple class (TupleN - N elements)
  + All basic functions are apart of the tuple class
  + .\_elementNumber (starts from 1) to access each member of the tuple
  + Can use productIterator to iterate over tuple values
  + Normally use case class instead of a tuple (no code readability for tuple)
* Options
  + Option[T] => Some[T] or None
  + var optionName = Some(value)
  + Some or None has similar properties as other collections
  + Just a container with a class defined inside of it\*\*\*\*
  + A parent class with a few other methods
* Iterator
  + Not a collection
  + Gives a way to access the elements within a collection
  + A parent class with a few other methods
  + Next -> give next element
  + hasNext -> checks if the next element is present
  + Has normal functions that are in other parent classes of collections
  + Can buffer the iterator using iteratorName.buffer (buffer in memory)

**Scala Day 24 - File Handling:**

* Either going to do a read operation or a write operation
* Written the exact same way as in Java
* Command line stuff works exactly the same as in Java (same exact functions)
  + Just different locations\*\*\*
* Normally use an absolute path when reading
* Reading uses fromFile
* Writing uses PrintWriter
* Normally use an absolute path when writing
  + FileWriter will abort (try catch)
  + PrintWriter has to have a corresponding check error (costly, slow, flushes output buffer)
  + BufferedWriter (flushes manually, less costly, fast)
* Serialization is the same as in Java
  + See code examples

**FIRST COURSE COMPLETE**