Cheat Sheet

Evaluation Rules

- Call by value: evaluates the function arguments before calling the function
  Ex: ***arg: Double***
- Call by name: evaluates the function first, and then evaluates the arguments if need be
  Ex: ***arg: => Double***

Higher Order Functions

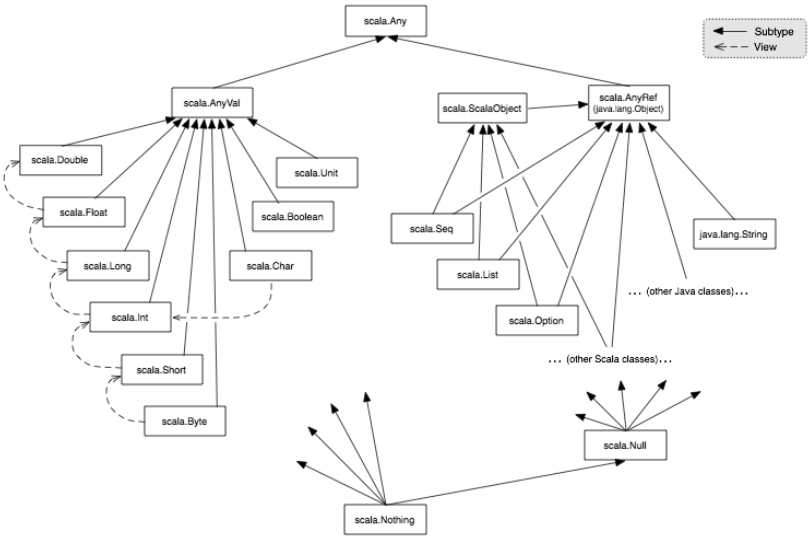
- These are functions that take a function as a parameter or return functions
- Can be an anonymous function or a function with inferred type
  Ex: ***def sum(f: Int => Int)(a: Int, b: Int): Int = f(a) + f(b)***

Currying

- Converting a function with multiple arguments into a function with a single argument that returns another function
  Ex: ***curried, uncurried*** – functions that make the conversion

Classes

- ***this*** references the current object
- For preconditions:
  - ***assert*** – for general assertions
  - ***assume*** – stating an axiom
  - ***require*** – specifically checking inputs
  - ***ensuring*** – a post condition that has also been covered
- Constructors can be implicit or auxiliary
- Methods can be public or private and can override base class methods
- Scala' s class hierarchy:



End Markers

- When the body of a class, object, trait, method or value becomes long, visually inspecting where it ends might become challenging
- In these situations, it is possible to explicitly signal to the reader that the body is over, using the ***end*** keyword with the name of the definition

Operators

- ***myObject myMethod 1*** is the same as calling ***myObject.myMethod(1)***
- Operator names can be alphanumeric, symbolic
- The precedence of an operator is determined by its first character, with the following increasing order of priority:
    - all letters
    - |
    - ^
    - &
    - <, >
    - =, !
    - :
    - +, -
    - *, /, %
    - all other special characters
- The associativity of an operator is determined by its last character: Right-associative if ending with ***:***, left-associative otherwise
- The assignment operators have lowest precedence

Class Hierarchies

- Only abstract classes can have abstract methods
- ***extends*** is used when you need to design an inherited class
- To create an runnable application in Scala:
    - ***@main def run(args: Array[String]) = ???***
    - ***Object Hello extends App = ???***
- ***object*** defines a singleton object; no other instance can be created

Class Organization

- Classes and objects are organized in packages:
  package myPackage
- They can be referenced through import statements:
  import myPackage.myClass, import myPackage.*, import myPackage .{Class1, Class2}
- They can also be directly referenced in the code with the fully qualified name:
  new myPackage.Class1
- All members of packages **scala** and ***java.lang*** as well as all members of the object ***scala.Predef*** are automatically imported
- ***trait*** is similar to a Java interface, except it can have non-abstract members

Type Parameters

- Similar to C++ templates or Java generics
- These can apply to classes, traits or functions
  Ex: ***class MyClass[T](arg: T): …***
- When creating an instance of a class, the type can be inferred; it can be determined based on the value arguments
- It is possible to restrict the type being used
  Ex: ***def myFct[T <: TopLevel](arg: T): T = …***
  def myFct[T >: Level1](arg: T): T = …
  def myFct[T >: Level1 <: TopLevel](arg: T): T = …

Variance

- Given ***A <: B***
  - If ***C[A] <: C[B]***, ***C*** is covariant: **class C[+A]**
  - If ***C[A] >: C[B]***, **C** is contravariant: ***class C[-A]***
  - Otherwise ***C*** is nonvariant: ***class C[A]***
- For a function: if ***A2 <: A1*** and ***B1 <: B2***, then ***A1 => B1 <: A2 => B2***
- Functions must be contravariant in their argument types and contravariant in their result types

Pattern Matching

- Pattern matching is used for decomposing data structures
- Pattern matching can also be used for ***Option*** values; some functions return a value of type ***Option[T]*** which is either a value of type ***Some[T]*** or the value ***None***
- Most of the times when you write a pattern match on an option value, the same expression can be written more concisely using combinator methods of the ***Option*** class
- Pattern matching is also used quite often in anonymous functions

Collections

- Base Classes:
  - ***Iterable***: collections you can iterate on
  - ***Seq***: ordered sequences
- Immutable Collections:
  - ***List***: linked list, provides fast sequential access
  - ***LazyList***: same as List, except that the tail is evaluated only on demand
  - ***Vector***: array-like type, implemented as tree of blocks, provides fast random access
  - ***Range***: ordered sequence of integers with equal spacing
  - ***String***: Java type, implicitly converted to a character sequence, so you can treat every string like a ***Seq[Char]***
  - ***Map:*** collection that maps keys to values
  - ***Set:*** collection without duplicate elements
- Mutable Collections:
  - ***Array***: native JVM arrays at runtime, therefore they are very performant
  - Scala has also mutable maps and sets; these should only be used if there are performance issues with immutable types

Ordering

- There is already a class in the standard library that represents orderings: ***scala.math.Ordering[T]*** which contains comparison functions such as ***lt()*** and ***gt()*** for standard types
- Types with a single natural ordering should inherit from the trait ***scala.math.Ordered[T]***

For-Comprehensions

- A for-comprehension is syntactic sugar for ***map, flatMap, filter*** operations on collections
- The general form is ***for (s) yield e***
    - ***s*** is a sequence of generators and filters
    - ***p <- e*** is a generator
    - ***if f*** is a filter
    - If there are several generators, the last generator varies faster that the first
    - You can use ***{s}*** instead of ***(s)*** if you want to use multiple lines without requiring semicolons
    - ***e*** is an element of the resulting collection
- A for-expression looks like a traditional for loop, but works differently internally
    - ***for (x <- e1) yield e2*** is translated to ***e1.map(x => e2)***
    - ***for (x <- e1 if f; s) yield e2*** is translated to ***for (x <- e1.withfilter(x => f); s) yield e2***
    - ***for(x <- e1; y <- e2; s) yield e3*** is translated to ***e1.flatMap(x =>for(y <- e2; s) yield e3)***
- This means you can use a for-comprehension for your own type, as long as you define ***map, flatMap*** and ***filter***