This cheat sheet originated from the #progfun forum, credits to Laurent Poulain. We copied it and changed or added a few things. There are certainly a lot of things that can be improved! If you would like to contribute, you have two options:

- Click the "Edit" button on this file on GitHub:https://github.com/lrytz/progfun-wiki/blob/gh-pages/CheatSheet.md You can submit a pull request directly from there without checking out the git repository to your local machine.

- Fork the repository https://github.com/lrytz/progfun-wiki and check it out locally. To preview your changes, you need jekyll. Navigate to your checkout and invoke jekyll --auto --server, then open the page http://localhost:4000 /CheatSheet.html.

Evaluation Rules

- Call by value: evaluates the function arguments before calling the function

- Call by name: evaluates the function first, and then evaluates the arguments if need be

```
1  def example = 2      // evaluated when called
2  val example = 2      // evaluated immediately
3  lazy val example = 2 // evaluated once when needed
4
5  def square(x: Double)    // call by value
6  def square(x: => Double) // call by name
7  def myFct(bindings: Int*) = { ... } // bindings is a sequence of int, containing
      a varying # of arguments
```

Higher order functions

These are functions that take a function as a parameter or return functions.

```
1  // sum() returns a function that takes two integers and returns an integer
2  def sum(f: Int => Int): (Int, Int) => Int = {
3    def sumf(a: Int, b: Int): Int = {...}
4    sumf
5  }
6
7  // same as above. Its type is (Int => Int) => (Int, Int) => Int
8  def sum(f: Int => Int)(a: Int, b: Int): Int = { ... }
9
10 // Called like this
11 sum((x: Int) => x * x * x)          // Anonymous function, i.e. does not have a
      name
12 sum(x => x * x * x)                 // Same anonymous function with type
      inferred
13
14 def cube(x: Int) = x * x * x
15 sum(x => x * x * x)(1, 10) // sum of cubes from 1 to 10
16 sum(cube)(1, 10)           // same as above
```

Currying

Converting a function with multiple arguments into a function with a single argument that returns another function.

```
1   def f(a: Int, b: Int): Int // uncurried version (type is (Int, Int) => Int)
2   def f(a: Int)(b: Int): Int // curried version (type is Int => Int => Int)
```

Classes

```
1   class MyClass(x: Int, y: Int) {          // Defines a new type MyClass with a
      constructor
2     require(y > 0, "y must be positive")   // precondition, triggering an
        IllegalArgumentException if not met
3     def this (x: Int) = { ... }            // auxiliary constructor
4     def nb1 = x                            // public method computed every time
        it is called
5     def nb2 = y
6     private def test(a: Int): Int = { ... } // private method
7     val nb3 = x + y                        // computed only once
8     override def toString =                // overridden method
9         member1 + ", " + member2
10    }
11
12  new MyClass(1, 2) // creates a new object of type
```

this references the current object, assert(<condition>) issues AssertionError if condition is not met. See scala.Predef for require, assume and assert.

Operators

myObject myMethod 1 is the same as calling myObject.myMethod(1)

Operator (i.e. function) names can be alphanumeric, symbolic (e.g. x1, *, +?%&, vector_++, counter_=)

The precedence of an operator is determined by its first character, with the following increasing order of priority:

```
1   (all letters)
2   |
3   ^
4   &
5   < >
6   = !
7   :
8   + -
9   * / %
10  (all other special characters)
```

The associativity of an operator is determined by its last character: Right-associative if ending with :, Left-associative otherwise.

Note that assignment operators have lowest precedence. (Read Scala Language Specification 2.9 sections 6.12.3, 6.12.4 for more info)

Class hierarchies

```
1   abstract class TopLevel {     // abstract class
2     def method1(x: Int): Int   // abstract method
3     def method2(x: Int): Int = { ... }
4   }
5
6   class Level1 extends TopLevel {
7     def method1(x: Int): Int = { ... }
8     override def method2(x: Int): Int = { ...} // TopLevel's method2 needs to be
         explicitly overridden
9   }
10
11  object MyObject extends TopLevel { ... } // defines a singleton object. No other
        instance can be created
```

To create an runnable application in Scala:

```
1   object Hello {
2     def main(args: Array[String]) = println("Hello world")
3   }
```

or

```
1   object Hello extends App {
2     println("Hello World")
3   }
```

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.{MyClass1, MyClass2}, import myPackage.{MyClass1 => A})

- They can also be directly referenced in the code with the fully qualified name (new myPackage.MyClass1)

- All members of packages scala and java.lang as well as all members of the object scala.Predef are automatically imported.

- Traits are similar to Java interfaces, except they can have non-abstract members:trait Planar { ... } class Square extends Shape with Planar

- General object hierarchy:

- **scala.Any** base type of all types. Has methods **hashCode** and **toString** that can be overridden

- **scala.AnyVal** base type of all primitive types. (**scala.Double**, **scala.Float**, etc.)

- **scala.AnyRef** base type of all reference types. (alias of **java.lang.Object**, supertype of **java.lang.String**, **scala.List**, any user-defined class)

- **scala.Null** is a subtype of any **scala.AnyRef** (null is the only instance of type **Null**), and scala.Nothing is a subtype of any other type without any instance.

Type Parameters

Similar to C++ templates or Java generics. These can apply to classes, traits or functions.

```
1   class MyClass[T](arg1: T) { ... }
2   new MyClass[Int](1)
3   new MyClass(1)    // the type is being inferred, i.e. determined based on the
        value arguments
```

It is possible to restrict the type being used, e.g.

```
1  def myFct[T <: TopLevel](arg: T): T = { ... } // T must derive from TopLevel or
     be TopLevel
2  def myFct[T >: Level1](arg: T): T = { ... }   // T must be a supertype of Level1
3  def myFct[T >: Level1 <: Top Level](arg: T): T = { ... }
```

Variance

Given A <: B

If C[A] <: C[B], C is covariant

If C[A] >: C[B], C is contravariant

Otherwise C is nonvariant

```
1  class C[+A] { ... } // C is covariant
2  class C[-A] { ... } // C is contravariant
3  class C[A]  { ... } // C is nonvariant
```

For a function, if A2 <: A1 and B1 <: B2, then A1 => B1 <: A2 => B2.

Functions must be contravariant in their argument types and covariant in their result types, e.g.

```
1  trait Function1[-T, +U] {
2    def apply(x: T): U
3  } // Variance check is OK because T is contravariant and U is covariant
4
5  class Array[+T] {
6    def update(x: T)
7  } // variance checks fails
```

Find out more about variance in lecture 4.4.

Pattern Matching

Pattern matching is used for decomposing data structures:

```
1  unknownObject match {
2    case MyClass(n) => ...
3    case MyClass2(a, b) => ...
4  }
```

Here are a few example patterns

```
1  (someList: List[T]) match {
2    case Nil => ...          // empty list
3    case x :: Nil => ...     // list with only one element
4    case List(x) => ...      // same as above
5    case x :: xs => ...      // a list with at least one element. x is bound to
       the head,
6                             // xs to the tail. xs could be Nil or some other list
                             .
7    case 1 :: 2 :: cs => ... // lists that starts with 1 and then 2
8    case (x, y) :: ps => ... // a list where the head element is a pair
9    case _ => ...            // default case if none of the above matches
10 }
```

The last example shows that every pattern consists of sub-patterns: it only matches lists with at least one element,

where that element is a pair. x and y are again patterns that could match only specific types.

Options

Pattern matching can also be used for Option values. Some functions (like Map.get) return a value of type Option[T] which is either a value of type Some[T] or the value None:

```
1  val myMap = Map("a" -> 42, "b" -> 43)
2  def getMapValue(s: String): String = {
3    myMap get s match {
4      case Some(nb) => "Value found: " + nb
5      case None => "No value found"
6    }
7  }
8  getMapValue("a")  // "Value found: 42"
9  getMapValue("c")  // "No value found"
```

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. For example, the function getMapValue can be written as follows:

```
1  def getMapValue(s: String): String =
2    myMap.get(s).map("Value found: " + _).getOrElse("No value found")
```

Pattern Matching in Anonymous Functions

Pattern matches are also used quite often in anonymous functions:

```
1  val pairs: List[(Char, Int)] = ('a', 2) :: ('b', 3) :: Nil
2  val chars: List[Char] = pairs.map(p => p match {
3    case (ch, num) => ch
4  })
```

Instead of p => p match { case ... }, you can simply write {case ...}, so the above example becomes more concise:

```
1  val chars: List[Char] = pairs map {
2    case (ch, num) => ch
3  }
```

Collections

Scala defines several collection classes:

Base Classes

- Iterable (collections you can iterate on)
- Seq (ordered sequences)
- Set
- Map (lookup data structure)

Immutable Collections

- List (linked list, provides fast sequential access)
- Stream (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 (Scala arrays are native JVM arrays at runtime, therefore they are very performant)

- Scala also has mutable maps and sets; these should only be used if there are performance issues with immutable types

Examples

```scala
1   val fruitList = List("apples", "oranges", "pears")
2   // Alternative syntax for lists
3   val fruit = "apples" :: ("oranges" :: ("pears" :: Nil)) // parens optional, ::
      is right-associative
4   fruit.head   // "apples"
5   fruit.tail   // List("oranges", "pears")
6   val empty = List()
7   val empty = Nil
8
9   val nums = Vector("louis", "frank", "hiromi")
10  nums(1)                    // element at index 1, returns "frank", complexity O
      (log(n))
11  nums.updated(2, "helena")   // new vector with a different string at index 2,
      complexity O(log(n))
12
13  val fruitSet = Set("apple", "banana", "pear", "banana")
14  fruitSet.size    // returns 3: there are no duplicates, only one banana
15
16  val r: Range = 1 until 5 // 1, 2, 3, 4
17  val s: Range = 1 to 5    // 1, 2, 3, 4, 5
18  1 to 10 by 3  // 1, 4, 7, 10
19  6 to 1 by -2  // 6, 4, 2
20
21  val s = (1 to 6).toSet
22  s map (_ + 2) // adds 2 to each element of the set
23
24  val s = "Hello World"
25  s filter (c => c.isUpper) // returns "HW"; strings can be treated as Seq[Char]
26
27  // Operations on sequences
28  val xs = List(...)
29  xs.length    // number of elements, complexity O(n)
30  xs.last      // last element (exception if xs is empty), complexity O(n)
31  xs.init      // all elements of xs but the last (exception if xs is empty),
      complexity O(n)
32  xs take n    // first n elements of xs
33  xs drop n    // the rest of the collection after taking n elements
34  xs(n)        // the nth element of xs, complexity O(n)
35  xs ++ ys     // concatenation, complexity O(n)
36  xs.reverse   // reverse the order, complexity O(n)
37  xs updated(n, x)  // same list than xs, except at index n where it contains x,
      complexity O(n)
38  xs indexOf x     // the index of the first element equal to x (-1 otherwise)
39  xs contains x    // same as xs indexOf x >= 0
40  xs filter p      // returns a list of the elements that satisfy the predicate p
41  xs filterNot p   // filter with negated p
42  xs partition p   // same as (xs filter p, xs filterNot p)
43  xs takeWhile p   // the longest prefix consisting of elements that satisfy p
44  xs dropWhile p   // the remainder of the list after any leading element
      satisfying p have been removed
45  xs span p        // same as (xs takeWhile p, xs dropWhile p)
46
47  List(x1, ..., xn) reduceLeft op    // (...(x1 op x2) op x3) op ...) op xn
48  List(x1, ..., xn).foldLeft(z)(op)  // (...( z op x1) op x2) op ...) op xn
49  List(x1, ..., xn) reduceRight op   // x1 op (... (x{n-1} op xn) ...)
50  List(x1, ..., xn).foldRight(z)(op) // x1 op (... (   xn  z) ...)
51
52  xs exists p    // true if there is at least one element for which predicate p is
      true
53  xs forall p    // true if p(x) is true for all elements
54  xs zip ys      // returns a list of pairs which groups elements with same index
      together
55  xs unzip       // opposite of zip: returns a pair of two lists
56  xs.flatMap f   // applies the function to all elements and concatenates the
      result
57  xs.sum         // sum of elements of the numeric collection
58  xs.product     // product of elements of the numeric collection
59  xs.max         // maximum of collection
60  xs.min         // minimum of collection
61  xs.flatten     // flattens a collection of collection into a single-level
      collection
62  xs groupBy f   // returns a map which points to a list of elements
63  xs distinct    // sequence of distinct entries (removes duplicates)
64
65  x +: xs  // creates a new collection with leading element x
66  xs :+ x  // creates a new collection with trailing element x
67
```

```
68   // Operations on maps
69   val myMap = Map("I" -> 1, "V" -> 5, "X" -> 10)  // create a map
70   myMap("I")       // => 1
71   myMap("A")       // => java.util.NoSuchElementException
72   myMap get "A"   // => None
73   myMap get "I"   // => Some(1)
74   myMap.updated("V", 15)  // returns a new map where "V" maps to 15 (entry is
        updated)
75                           // if the key ("V" here) does not exist, a new entry is
                                 added
76
77   // Operations on Streams
78   val xs = Stream(1, 2, 3)
79   val xs = Stream.cons(1, Stream.cons(2, Stream.cons(3, Stream.empty))) // same as
        above
80   (1 to 1000).toStream // => Stream(1, ?)
81   x #:: xs // Same as Stream.cons(x, xs)
82           // In the Stream's cons operator, the second parameter (the tail)
83           // is defined as a "call by name" parameter.
84           // Note that x::xs always produces a List
```

Pairs (similar for larger Tuples)

```
1   val pair = ("answer", 42)   // type: (String, Int)
2   val (label, value) = pair   // label = "answer", value = 42
3   pair._1 // "answer"
4   pair._2 // 42
```

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].

```
1   import math.Ordering
2
3   def msort[T](xs: List[T])(implicit ord: Ordering) = { ...}
4   msort(fruits)(Ordering.String)
5   msort(fruits)   // the compiler figures out the right ordering
```

For-Comprehensions

A for-comprehension is syntactic sugar for map, flatMap and 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 (equivalent of a nested loop), the last generator varies faster than 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

Example 1

```
1  // list all combinations of numbers x and y where x is drawn from
2  // 1 to M and y is drawn from 1 to N
3  for (x <- 1 to M; y <- 1 to N)
4    yield (x,y)
```

is equivalent to

```
1  (1 to M) flatMap (x => (1 to N) map (y => (x, y)))
```

Translation Rules

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) yield e2 is translated to for (x <- e1.filter(x => f)) yield e2

for (x <- e1; y <- e2) yield e3 is translated to e1.flatMap(x => for (y <- e2) yield e3)

This means you can use a for-comprehension for your own type, as long as you define map, flatMap and filter.

Example 2

```
1  for {
2    i <- 1 until n
3    j <- 1 until i
4    if isPrime(i + j)
5  } yield (i, j)
```

is equivalent to

```
1  for (i <- 1 until n; j <- 1 until i if isPrime(i + j))
2    yield (i, j)
```

is equivalent to

```
1  (1 until n).flatMap(i => (1 until i).filter(j => isPrime(i + j)).map(j => (i, j
      )))
```

Mark as completed