Scala cheat-sheet

Variables

```
var x = 11  // variable
val y = 10  // constant
y = 9 // error
```

Classes

```
class ClassDemo(
    val immutableField: String,
    var mutableField: String,
    private val privateField: Int) {

 def paramLength(param: String): Int = {
    param.lenght
  }
}
val instance = ClassDemo("immutable", "muttable", 100)
instance.mutableField = "new mutable value"
instance.immutableField = "new immutable value" // compilation error
instance.privateField // compilation error
instance.paramLength("123") // -> 3
```

Companion object. Imagine static methods holder

```
object ClassDemo {
  def paramLength(param: String): Int = {
    param.length
  }
}
ClassDemo.paramLength("123") // -> 3
```

Case class

scala adds syntactic conveniences:

- all args in param list implicitly get a val, and become fields
- add implementations of toString, hashCode, and equals
- adds a copy method

```
case class Person(name: String)
val eddy = Person("Eddy")       // constructor
eddy.name                       // getter
eddy.name = "ed"                // compilation error
val ed = eddy.copy(name = "Ed") // another instance
```

case classes are immutable so instead of mutating we need to create new instance of it with copy method

Functions

```
def product(x: Int, y: Int): Int = {
 x*y
}
def hello(msg: String): Unit = {
  println("Returns void")
}
product(5, 4)
product(y = 4, x = 5) // named parameters

def lazyFn(lazyArg: => String) = {
  lazyArg
}
lazyFn("Hi!")
lazyFn { "Hi!" } // lazy evaluation

// lambdas
val concatStr: (String, String) => String = (a, b) => a + b
concatStr("Tieto", "Conference") // -> "Tieto Conference"

(1 to 5).map( x => x*x ) // List(1, 4, 9, 16, 25)
```

Higher order functions take other functions as parameters

```
List(1, 2 ,3).map(number => number * 2)  // List(1, 4, 6)
List(1, 2, 3).flatMap(number => List(number * 2)) // same as above
List(1, 2, 3).filter(number => number % 2 == 0) // List(2)
```

Tuples

can hold multiple values of different types

```
val person = ("John", 30)  // (String, Int)
```

Collections

```
val aList = List(1, 2 ,3)
val aSet = Set(1, 2, 3)
val aMap = Map("key1" -> 1, "key2" -> 2)
```

To perform transformations on collection higher order functions like `map` `filter` could be used.

Collections above are immutable. Mutable collections are present in `scala.collection.mutable` package.

Future

```
trait Future[T] {
  def filter(p: T=>Boolean): Future[T]
  def flatMap[S](f: T=>Future[S]): Future[S]
  def map[S](f: T=>S): Future[S]
  def zip[U](that: Future[U]): Future[(T, U)]
}

object Future {
  def apply[T](body :=>T): Future[T]
  def successful[T](result: T): Future[T]
  def failed[T](exception: Throwable): Future[T]
}
```

Create Future

```
val future: Future[String] = Future {
    "Future result is string"
}
val future = Future("Future result is string") // same as above

val future = Future.successful("Successful future")
val future = Future.failed(new RuntimeException("Something went wrong"))
```

Basic function to work with futures

```
val future = Future.successful("Successful future")

future.map(s => s.split(" ").length) // Future[Int](5)
future.flatmap(s => Future(s.split(" ").length)) // same
future.filter(s => !s.isEmpty) // successful future, not changed
future.filter(s => s.isEmpty) // failed future with NoSuchElementException

// zips two futures into single one - result is tuple
val otherFuture = Future.successful("Another future")
future.zip(otherFuture) // Future[(String, String)]

// if at least one future fails, result is failed future
val failedFuture = Future.failed(new RuntimeException())
future.zip(failedFuture)
```