CHEAT SHEETS HERO

Scala Cheatsheet

A concise reference for Scala syntax, features, and common use cases, designed to boost productivity and aid quick recall.

Basics & Syntax

Variables & Data Types

`val` (Immutable)

Declares an immutable variable. Its value cannot be changed after assignment.

Example:

```
val x: Int = 10
```

`var` (Mutable)

Declares a mutable variable. Its value can be changed after assignment.

Example:

```
var y: String = "Hello"
y = "World"
```

Basic Data Types

`Int`, `Double`, `Boolean`, `String`, `Char`, `Unit` (similar to void in Java)

Type Inference

Scala can often infer the type, so explicit type declarations are optional.

Example:

```
val z = 5  // Int is inferred
```

String Interpolation

Embed variables directly in strings.

Example:

```
val name = "Alice"
println(s"Hello, $name!")
```

Multiline Strings

Use triple quotes to define multiline strings.

Example:

```
val multiline = """This is a
multiline string."""
```

Operators

Scala uses similar operators to Java: arithmetic (`+`, `-`, `*`, `/`, `%`), relational (`==`, `!=`, `>`, `<`, `>=`, `<=`), logical (`&&`, `||`, `!`).

Note that `==` in Scala is structural equality (compares content), not reference equality. Use `eq` for reference equality.

Control Structures

`if` Statement

```
val x = 10
val result = if (x > 5) "Big"
else "Small"
```

`for` Loop

```
for (i <- 1 to 5) {
  println(i)
}
```

`while` Loop

```
var i = 0
while (i < 5) {
  println(i)
  i += 1
}
```

`match` Statement

Powerful pattern matching.

```
val code = 404
val message = code match {
  case 200 => "OK"
  case 404 => "Not Found"
  case _ => "Unknown"
}
```

Functions & Classes

Functions

Function Definition

```
def add(x: Int, y: Int): Int = x + y
```

Anonymous Functions (Lambdas)

```
val multiply = (x: Int, y: Int) => x * y
```

Currying

Transforming a function that takes multiple arguments into a function that takes a single argument and returns another function that accepts the remaining arguments.

```
def multiply(x: Int)(y: Int): Int = x * y
val multiplyByTwo = multiply(2) _
println(multiplyByTwo(5))  // Output: 10
```

Default Arguments

```
def greet(name: String = "World"): Unit =
println(s"Hello, $name!")
greet() // Hello, World!
greet("Alice") // Hello, Alice!
```

Higher-Order Functions

Functions that take other functions as arguments or return them as results.

```
def operate(x: Int, y: Int, f: (Int, Int) => Int): Int
= f(x, y)
val sum = operate(5, 3, (a, b) => a + b)
```

Classes

Class Definition

```
class Person(val name: String, var age: Int)
```

Auxiliary Constructor

```
class Person(val name: String, var age: Int) {
  def this(name: String) = this(name, 0)
}
```

Case Classes

Automatically provides `equals`, `hashCode`, `toString`, and a factory method `apply`.

```
case class Point(x: Int, y: Int)
val p = Point(1, 2)  // No 'new' keyword needed
```

Traits

Similar to interfaces in Java, but can also contain implemented methods and fields.

```
trait Loggable {
  def log(message: String): Unit = println(s"Log:
$message")
}

class MyClass extends Loggable {
  def doSomething(): Unit = log("Doing something...")
}
```

Collections

Common Collections

`List`	An ordered, immutable sequence of elements. `val myList = List(1, 2, 3)`
`Set`	A collection of unique elements. `val mySet = Set(1, 2, 2, 3)  // Set(1, 2, 3)`
`Map`	A collection of key-value pairs. `val myMap = Map("a" -> 1, "b" -> 2)`
`Array`	A mutable, fixed-size sequence of elements. More like Java arrays. `val myArray = Array(1, 2, 3)` `myArray(0) = 4 // Mutable`
`Vector`	Indexed, immutable sequence. Provides fast random access and updates (amortized). `val myVector = Vector(1, 2, 3)`

Collection Operations

Scala collections provide a rich set of operations using higher-order functions. These methods generally return a *new* collection (immutability).

`map` - Applies a function to each element and returns a new collection with the results.

```
List(1, 2, 3).map(x => x * 2)  // List(2, 4, 6)
```

`filter` - Returns a new collection containing only the elements that satisfy a predicate.

```
List(1, 2, 3, 4).filter(x => x % 2 == 0)  // List(2, 4)
```

`flatMap` - Applies a function that returns a collection to each element and concatenates the results.

```
List("a", "b").flatMap(x => List(x, x.toUpperCase)) // List(a, A, b, B)
```

`foreach` - Applies a function to each element (for side effects).

```
List(1, 2, 3).foreach(println) // Prints 1, 2, 3
```

`reduce` - Combines the elements of a collection into a single value using a binary operation.

```
List(1, 2, 3).reduce((x, y) => x + y)  // 6
```

`foldLeft` - Similar to reduce, but takes an initial value.

```
List(1, 2, 3).foldLeft(0)((x, y) => x + y)  // 6
```

Advanced Features

Pattern Matching

Matching Literal Values

```
val x = 10
x match {
  case 10 => println("It's 10!")
  case _ => println("It's something else.")
}
```

Matching on Types

```
def describe(x: Any): String = x match {
  case s: String => s"String: $s"
  case i: Int => s"Int: $i"
  case _ => "Unknown type"
}
```

Matching Case Classes

```
case class Person(name: String, age: Int)
val p = Person("Bob", 30)
p match {
  case Person("Bob", age) => println(s"Bob is $age
years old.")
  case _ => println("Not Bob")
}
```

Guards

Adding conditions to case statements.

```
x match {
  case i: Int if i > 0 => println("Positive integer")
  case i: Int => println("Non-positive integer")
  case _ => println("Not an integer")
}
```

Implicits

Implicit parameters, conversions, and classes allow for powerful type-safe abstractions and DSL creation. Use with caution, as they can make code harder to understand.

Implicit Parameter: A parameter that the compiler can automatically provide if it's not explicitly passed.

```
implicit val timeout: Int = 1000
def run(implicit t: Int): Unit = println(s"Running with timeout $t")
run // runs with timeout 1000
```

Implicit Conversion: Automatically converts one type to another.

```
implicit def stringToInt(s: String): Int = s.toInt
val x: Int = "123"  // String is implicitly converted to Int
```

Implicit Class: Adds methods to an existing class.

```
implicit class StringUtils(s: String) {
  def shout(): String = s.toUpperCase + "!"
}
println("hello".shout())  // HELLO!
```