codecentric

presents the Scala Cheat Sheet, for Scala 2.12

Variables & Methods

```
var x = "mutable"
val y = "immutable"

// Initialized once on first access
lazy val z = "lazy"

// Pattern matching on left side
val (one, two) = ("one", 2)

// Simple method
def add(n: Int, m: Int): Int = n + m

// Curried version (one argument per list)
def add(n: Int)(m: Int): Int = n + m

// By-name parameters, evaluates `a` twice
def twice[A](a: => A) = { a; a }

// Repeated Parameters (Varargs)
def many(ns: Int*): Seq[Int] = ns

// Calling a varargs method with a `Seq`
> many(Seq(1, 2) :_*)
```

Strings

```
val answer: Int = 42
// add `s` prefix for interpolation
s"The answer is: $answer"

// use ${} for more complex expressions
s"The answer is: ${21 * 2}"

// """ enclose a multiline string
"""
Inside triple quotes there
is no need to escape: \
"""
```

Regular Expressions

```
val time = """(\d{1,2}):(\d{2})""".r

> "16:03" match { case time(_*) => "matched!" }

res: String = matched!

// extract matched groups
> "16:12" match { case time(h,m) =>
  s"Hours: $h, minutes: $m"
}

res: String = Hours: 16, minutes: 12
```

Classes

```
// Implicit default constructor
// `AnyRef` plays the role of `Object`
class Foo extends AnyRef {
  val bar: Int = 42
  def foobar: Boolean = true
}

// Parameterized constructor
class Foo(msg: String) { ... }

// Additional constructor
class Foo(msg: String) {
  def this(n: Int) = this(n.toString)
}

// Can inherit from exactly one class
class Bar extends Foo("foo")
```

Objects

Objects hold "*static*" members. When used as a companion object, it is relevant during implicit search. Objects are *singletons*.

```
object Foo {
  val hello: String = "Hello"
  def world: String = "World"
}

// Companion object:
// class and object share name and source file
class Bar
object Bar { ... }
```

Traits

The `sealed` modifier forbids extension of the trait from a different source file than the one it is defined in. This allows exhaustiveness checks while pattern matching.

```
trait Foo {
  // can have abstract members
  def foo: String

  // can have implementations
  def bar: Unit = println("bar")
}

// Multiple traits can be mixed-in
trait Bar
class Foobar extends Foo with Bar {
  override def foo = "foo"
}
```

Case Classes

```
// class definition prefixed with `case`
case class Person(name: String, age: Int)
```

What `case` does:

1. constructor parameters are promoted to fields
2. generates companion object with `apply` and `unapply`
3. generates the `copy` method
4. generates `equals`, `hashCode` and `toString`

For-loop and For-comprehension

For-loops: iterate for side-effects only. For-comprehensions: chain effectful computations.

```
// for-loop
for (i <- 1 to 10) println(i)

// nested for-loop
for (i <- 1 to 10; j <- 1 to 10) println((i, j))

// for-comprehension
for (i <- 1 to 3; j <- 1 to i) yield i * y

// guards in for-loops and for-comprehensions
for (i <- 1 to 5 if i > 4) yield i

// curly braces for multiline expressions
for {
  i <- 1 to 5
  j <- 1 to i
  if j % 2 == 1
} yield i * j
```

Pattern Matching

```
arg match {
  // Variable Patterns
  case x =>
  // Typed Patterns
  case x: String =>
  // Literal Patterns
  case 42 =>
  // Stable Identifier Patterns
  case `foo` =>
  // Constructor Patterns
  case Foo(x,y) =>
  // Tuple Patterns
  case (x,y,z) =>
  // Extractor Patterns
  // (See 'Custom Extractors')
  case NumString(x) =>
  // Pattern Sequences
  case x1 +: x2 +: xs =>
  // Pattern Alternatives
  case true | 42 | "str" =>
  // Pattern Binders
  case pair@(x,y) =>
}
```

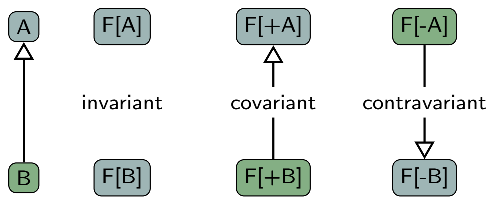
Custom Extractors

```
// Return Boolean from unapply
object Even {
  def unapply(n: Int): Boolean =
    n % 2 == 0
}
> 41 match { case Even() => '!' }
scala.MatchError: 41 ...
> 42 match { case Even() => "even!" }
res: String = "even!"
// Return Option from unapply
object NumString {
  def unapply(s: String): Option[Int] =
    Try(s.toInt).toOption
}
> "42" match { case NumString(n) => n }
res: Int = 42
> "scala" match { case NumString(n) => n }
scala.MatchError: scala
// Alternatively define unapplySeq
object Words {
  def unapplySeq(s: String): Option[Seq[String]]
    Some(s.split("\\s+").to[Seq])
}
> "foo bar baz" match { case Words(ws) => ws }
res: Seq[String] = Vector("foo", "bar", "baz")
> "test" match { case Words("foo" +: _) => 1 }
scala.MatchError: test
```

Type Parameters

```
// Two type parameters A and B
def foo[A, B](a: A, b: B) = ???

// Upper Bound, A has to be a subtype
def foo[A <: String]

// Lower Bound, A has to be a supertype
def foo[A :> String]

// Context Bound
def foo[A: Ordering](x: A, y: A): Boolean = {
  import Ordering.Implicits._
  x < y
}
// Context Bounds desugar to implicit params
def foo[A](x: A, y: A)(
  implicit evidence$1: Ordering[A]): Boolean
```

Variance



codecentric

presents the Scala Cheat Sheet, for Scala 2.12

Implicits

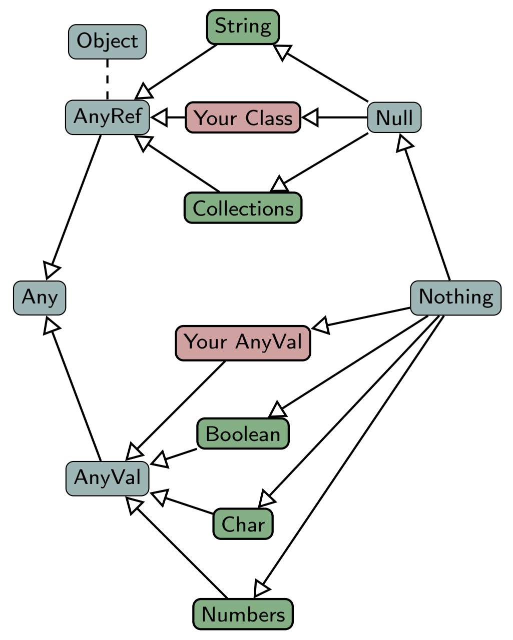
There are two categories of places where Scala searches for implicits:

1. identifiers accessible *without prefix* at the call-site
2. implicit scope: all companion objects of classes associated with the implicit's type

The implicit modifier

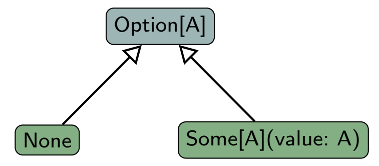
```
// implicit values
implicit val n: Int = 42

// implicit conversions
implicit def f(n: Int): String = n.toString

// implicit classes
implicit class Wrapper[A](val a: A) {
  def printMe: Unit = println(a)
}

// implicit parameters
def foo(implicit ec: ExecutionContext) = ???
```

Types



Option

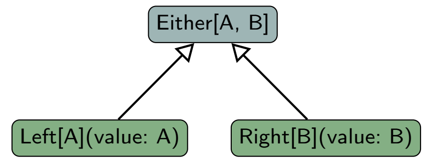
Replaces null, there is only *one* obvious reason for a missing value.



```
val some: Option[Int] = Some(1)
val none: Option[Int] = None

// getOrElse
> some.getOrElse(42)
res: Int = 1
> none.getOrElse(42)
res: Int = 42

// fold
> some.fold("")(_.toString)
res: String = "1"
> none.fold("")(_.toString)
res: String = ""

// orElse
> some.orElse(none)
res: Option[Int] = Some(1)
> none.orElse(Some(42))
res: Option[Int] = Some(42)
```

Either

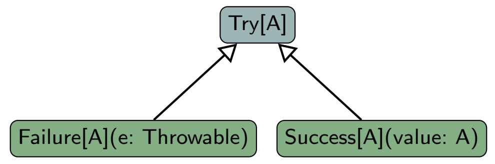
Domain errors that have to be handled, there are *multiple* reasons for an error.



```
val right: Either[String, Int] = Right(1)
val left:  Either[String, Int] = Left("oops")

// getOrElse
> right.getOrElse(42)
res: Int = 1
> left.getOrElse(42)
res: Int = 42

// map on left and right side
> right.map(_ + 1)
res: Either[String, Int] = Right(2)
> right.left.map(_.length)
res: Either[Int, Int] = Right(1)
> left.left.map(_.length)
res: Either[Int, Int] = Left(4)
```

Try

Interact with Java / Legacy Code where exceptions are thrown, a means of last resort.



```
> import scala.util.Try

> Try { "hello".toInt }
res: Try[Int] =
  Failure(java.lang.NumberFormatException)

> Try { "42".toInt }
res: Try[Int] = Success(42)
```

Collections

Prefer immutable collections, falling back to a var first and use mutable collections as a last resort only. Also, prefer Vector over List

Warning: Seq by default allows mutable implementations, import scala.collection.immutable.Seq instead.

```
// Creating a collection via apply:
> List(1, 2, 3)
res: List[Int] = List(1, 2, 3)
> Array('a', 'b')
res: Array[Char] = Array(a, b)
> Map(('a', 1), ('b', 2))
res: Map[Char, Int] = Map(a -> 1, b -> 2)

// Importing mutable collections
> import scala.collection.mutable
> mutable.Buffer(1, 2, 3)
res: Buffer[Int] = ArrayBuffer(1, 2, 3)
```

Important methods

`collect`	filter then map in one
`collectFirst`	find with pattern matching
`count`	count elements with predicate
`exists`	check predicate satisfied >= 1
`find`	find element with predicate
`filter`	filter elements with predicate
`flatMap`	map a function producing a collection
`foldLeft`	recursive traversal
`foldRight`	for right associative ops
`forall`	check predicate holds for all elements
`map`	transform each element
`slice`	select an interval of elements
`take, drop`	remove elements from front/back
`to[Col]`	convert to collection Col

Futures

Don't blindly import Scala's default ExecutionContext, it is optimized for *CPU-bound* tasks!

```
> import scala.concurrent._, duration._
> import ExecutionContext.Implicits.global

// Creating an asynchronous computation
> Future { 5 * 2 }
res1: Future[Int] = Future(<not completed>)

// Modify result with a pure function
> res1.map((n: Int) => n + 11)
res2: Future[Int] = Future(<not completed>)

// Use flatMap to chain Futures
> res2.flatMap((n: Int) => Future { n * 2 })
res3: Future[Int] = Future(<not completed>)

// Register callbacks
> res3.onComplete {
  case Success(r) => println(s"Success: $r")
  case Failure(e) => println(s"Failure: $e")
}

// Block thread for result (anti-pattern)
> Await.result(res3, 1.second)
res4: Int = 42
```

Duration DSL

```
> import scala.concurrent.duration._
> 5.seconds
res: FiniteDuration = 5 seconds
> 2.hours
res: FiniteDuration = 2 hours
```

Scala Plugin for IntelliJ IDEA

Show Type Info	Alt			=
Implicit parameters	Ctrl		Shift ⇑	P
Implicit conversions	Ctrl		Shift ⇑	Q
ScalaDoc stub	Ctrl	Alt		Q
Refactor This	Ctrl	Alt	Shift ⇑	T
Rename			Shift ⇑	F6
Inline	Ctrl	Alt		N
Extract Variable	Ctrl	Alt		V
Extract Field	Ctrl	Alt		F
Extract Method	Ctrl	Alt		M