variables

`var x = 5`	variable

GOOD
`val x = 5`
BAD
`x=6`
	constant

`var x: Double = 5`	explicit type

functions

GOOD
`def f(x: Int) = { x*x }`
BAD
`def f(x: Int) { x*x }`
	define function
	hidden error: without = it's a Unit-returning procedure; causes havoc

GOOD
`def f(x: Any) = println(x)`
BAD
`def f(x) = println(x)`
	define function
	syntax error: need types for every arg.

`type R = Double`	type alias
`def f(x: R)` vs. `def f(x: => R)`	call-by-value call-by-name (lazy parameters)
`(x:R) => x*x`	anonymous function
`(1 to 5).map(_*2)` vs. `(1 to 5).reduceLeft( _+_ )`	anonymous function: underscore is positionally matched arg.
`(1 to 5).map( x => x*x )`	anonymous function: to use an arg twice, have to name it.

GOOD
`(1 to 5).map(2*)`
BAD
`(1 to 5).map(*2)`
	anonymous function: bound infix method. Use `2*_` for sanity's sake instead.

`(1 to 5).map { x => val y=x*2; println(y); y }`	anonymous function: block style returns last expression.
`(1 to 5) filter {_%2 == 0} map {_*2}`	anonymous functions: pipeline style. (or parens too).

`def compose(g:R=>R, h:R=>R) = (x:R) => g(h(x))`
`val f = compose({_*2}, {_-1})`
	anonymous functions: to pass in multiple blocks, need outer parens.

`val zscore = (mean:R, sd:R) => (x:R) => (x-mean)/sd`	currying, obvious syntax.
`def zscore(mean:R, sd:R) = (x:R) => (x-mean)/sd`	currying, obvious syntax
`def zscore(mean:R, sd:R)(x:R) = (x-mean)/sd`	currying, sugar syntax. but then:
`val normer = zscore(7, 0.4) _`	need trailing underscore to get the partial, only for the sugar version.
`def mapmake[T](g:T=>T)(seq: List[T]) = seq.map(g)`	generic type.

`5.+(3); 5 + 3`
`(1 to 5) map (_*2)`
	infix sugar.

`def sum(args: Int*) = args.reduceLeft(_+_)`	varargs.

packages

`import scala.collection._`	wildcard import.

`import scala.collection.Vector`
`import scala.collection.{Vector, Sequence}`
	selective import.

`import scala.collection.{Vector => Vec28}`	renaming import.
`import java.util.{Date => _, _}`	import all from java.util except Date.
`package pkg` *at start of file* `package pkg { ... }`	declare a package.

data structures

`(1,2,3)`	tuple literal. (`Tuple3`)
`var (x,y,z) = (1,2,3)`	destructuring bind: tuple unpacking via pattern matching.
BAD `var x,y,z = (1,2,3)`	hidden error: each assigned to the entire tuple.
`var xs = List(1,2,3)`	list (immutable).
`xs(2)`	paren indexing. (slides)
`1 :: List(2,3)`	cons.

`1 to 5` *same as* `1 until 6`
`1 to 10 by 2`
	range sugar.

`()` *(empty parens)*	sole member of the Unit type (like C/Java void).

control constructs

`if (check) happy else sad`	conditional.

`if (check) happy`
same as
`if (check) happy else ()`
	conditional sugar.

`while (x < 5) { println(x); x += 1}`	while loop.
`do { println(x); x += 1} while (x < 5)`	do while loop.

`import scala.util.control.Breaks._`
`breakable {`
`  for (x <- xs) {`
`    if (Math.random < 0.1)`
`      break`
`  }`
`}`
	break. (slides)

`for (x <- xs if x%2 == 0) yield x*10`
same as
`xs.filter(_%2 == 0).map(_*10)`
	for comprehension: filter/map

`for ((x,y) <- xs zip ys) yield x*y`
same as
`(xs zip ys) map { case (x,y) => x*y }`
	for comprehension: destructuring bind

`for (x <- xs; y <- ys) yield x*y`
same as
`xs flatMap {x => ys map {y => x*y}}`
	for comprehension: cross product

`for (x <- xs; y <- ys) {`
`  println("%d/%d = %.1f".format(x, y, x/y.toFloat))`
`}`
	for comprehension: imperative-ish
	sprintf-style

`for (i <- 1 to 5) {`
`  println(i)`
`}`
	for comprehension: iterate including the upper bound

`for (i <- 1 until 5) {`
`  println(i)`
`}`
	for comprehension: iterate omitting the upper bound

pattern matching

GOOD
`(xs zip ys) map { case (x,y) => x*y }`
BAD
`(xs zip ys) map( (x,y) => x*y )`
	use case in function args for pattern matching.

BAD
`val v42 = 42`
`Some(3) match {`
`  case Some(v42) => println("42")`
`  case _ => println("Not 42")`
`}`
	“v42” is interpreted as a name matching any Int value, and “42” is printed.

GOOD
``val v42 = 42``
``Some(3) match {``
``  case Some(`v42`) => println("42")``
``  case _ => println("Not 42")``
``}``
	”`v42`” with backticks is interpreted as the existing val `v42`, and “Not 42” is printed.

GOOD
`val UppercaseVal = 42`
`Some(3) match {`
`  case Some(UppercaseVal) => println("42")`
`  case _ => println("Not 42")`
`}`
	`UppercaseVal` is treated as an existing val, rather than a new pattern variable, because it starts with an uppercase letter. Thus, the value contained within `UppercaseVal` is checked against `3`, and “Not 42” is printed.

object orientation

`class C(x: R)`	constructor params - `x` is only available in class body

`class C(val x: R)`
`var c = new C(4)`
`c.x`
	constructor params - automatic public member defined

`class C(var x: R) {`
`  assert(x > 0, "positive please")`
`  var y = x`
`  val readonly = 5`
`  private var secret = 1`
`  def this = this(42)`
`}`
	constructor is class body
	declare a public member
	declare a gettable but not settable member
	declare a private member
	alternative constructor

`new{ ... }`	anonymous class
`abstract class D { ... }`	define an abstract class. (non-createable)
`class C extends D { ... }`	define an inherited class.

`class D(var x: R)`
`class C(x: R) extends D(x)`
	inheritance and constructor params. (wishlist: automatically pass-up params by default)

`object O extends D { ... }`	define a singleton. (module-like)

`trait T { ... }`
`class C extends T { ... }`
`class C extends D with T { ... }`
	traits.
	interfaces-with-implementation. no constructor params. mixin-able.

`trait T1; trait T2`
`class C extends T1 with T2`
`class C extends D with T1 with T2`
	multiple traits.

`class C extends D { override def f = ...}`	must declare method overrides.
`new java.io.File("f")`	create object.

BAD
`new List[Int]`
GOOD
`List(1,2,3)`
	type error: abstract type
	instead, convention: callable factory shadowing the type

`classOf[String]`	class literal.
`x.isInstanceOf[String]`	type check (runtime)
`x.asInstanceOf[String]`	type cast (runtime)
`x: String`	ascription (compile time)