Scala 3.5 QuickRef @ Lund University

https://github.com/lunduniversity/introprog/tree/master/quickref

Compiled 2024-08-21. License: CC-BY-SA. Pull requests welcome! Contact: bjorn.regnell@cs.lth.se

Compile and run @main program

In file `hello.scala` (any using-directives must come first)

```
//> using scala 3.5.0
@main def hi(n: Int) = println("hi " * n)
```

Compile and run single file:
`scala run hello.scala -- 42`
Main program arguments after `--`

Compile all scala & java files in current dir:
`scala compile . -w`
where `-w` or `--watch` recompile on change
the dot `.` gives access to code in current dir.

A compilation unit, e.g. `hello.scala`, can have top-level definitions such as **`val`**, **`var`**, **`def`**, **`import`**, **`class object`**, **`trait`**, ... which may be preceded by a package clause, e.g.: **`package`** `x.y.z` creates a name space and places the compiled bytecode in dir `.scala-build` under `x/y/z/`

Run all scala & java files in current dir:
`scala run . -- 42`
Main program arguments after optional `--`

Start REPL: `scala repl .`

Before Scala 3.5 use: `scala-cli repl .`

Definitions and declarations

A **definition** binds a name to a value/implementation, while a **declaration** just introduces a name (and type) of an abstract member. Template bodies with curly braces `{ ... }` are optional and can be replaced by `:` that opens an indentation region with significant leading whitespace. Implementations after `=` also opens an indentation region.

Variable
```
val x = expr                Variable x is assigned to expr. A val can only be assigned once.
val x: Int = 0              Explicit type annotation, expr: SomeType allowed after any expr.
var x = expr                Variable x is assigned to expr. A var can be re-assigned.
val x, y = expr             Multiple initialisations, x and y is initialised to the same value.
val (x, y) = (e1, e2)       Tuple pattern initialisation, x is assigned to e1 and y to e2.
val Seq(x, y) = Seq(e1, e2) Sequence pattern initialisation, x is assigned to e1 and y to e2.
```

Function
```
def f(a: Int, b: Int): Int = a + b          Function f of type (Int, Int) => Int
def f(a: Int = 0, b: Int = 0): Int = a + b  Default arguments used if args omitted, f().
f(b = 1, a = 3)                             Named arguments can be used in any order.
def add(a: Int)(b: Int): Int = a + b        Multiple parameter lists, apply: add(1)(2)
(a: Int, b: Int) => a + b                   Anonymous function value, "lambda".
val g: (Int, Int) => Int = (a, b) => a + b  Types can be omitted in lambda if inferable.
val inc = add(1)      Partially applied function add(1) of add above, where inc is of type Int => Int
def sumAll(xs: Int*) = xs.sum  Repeated parameters: sumAll(1,2,3) or sumAll(Seq(1,2,3)*)
def twice(block: => Unit) = { block; block }  Call-by-name argument evaluated later.
```

Object
```
object Name:          A singleton object is automatically allocated when referenced the first time.
  def member = "hi"   Object basicstyleody can contain definitions of members such as def, val, etc.
```

Class
```
class C(val x: Int):   A template for objects to be allocated with new or apply.
  def myMethod = ???   Members indented after colon, or use curly braces instead of colon.
case class C(x: Int)   Case class parameters become val members,
other case class goodies: equals, copy, hashcode, unapply, nice toString, companion object with apply factory.
```

Trait
```
trait T(val x: Int):          A trait is like an abstract class, but can be mixed in.
  def myAbstractMethod: Int   An abstract member declaration with no implementation.
```

Mixin
```
class C extends D, T   A class can only extend one class, but mix in many traits separated with comma
```

Type Alias
```
type A = AnotherType   Defines an alias A for the type AnotherType. Abstract if no = ...
```

Import
```
import path.to.name                Makes name directly visible. Can be renamed using as
import path.to.*                   Wildcard * imports all.
import path.to.{a, b as x, c as _} Import several names, b renamed to x, c not imported.
```

2 Modifyers on definitions and declarations

Modifier	applies to	semantics
`private`	definitions, declarations	Restricts access to directly enclosing class and its companion.
`override`	definitions, declarations	Mandatory if overriding a concrete definition in a parent class.
`final`	definitions	Final members cannot be overridden, final classes cannot be extended.
`protected`	definitions	Restricts access to subtypes and companion.
`lazy`	val	Delays initialization of val, initialized when first referenced.
`infix`	def	Allow alpha-numeric names in operator notation without warning.
`inline`	def, val	Replaced at compile time by its implementation. Also if, match, params.
`abstract`	class	Abstract classes cannot be instantiated (redundant for traits).
`sealed`	class, trait	Restricts direct inheritance to classes in the same compilation unit.
`open`	class	Signal intent to be used in inheritance hierarchy. Silences warning.
`transparent`	class, trait, def	Inference of class/trait is suppressed. Inference of def type is precise.

Constructors and special methods (getters, setters, apply, update, right-assoc. op.), Companion object

```
class A(initX: Int = 0):
  private var _x = initX
  def x: Int = _x
  def x_=(i: Int): Unit =
    _x = i
  def +: (i: Int) = _x += i
end A
object A:
  def apply(i: Int) =
    new A(i)
  val y = A(1)._x
```

primary constructor, object creation (new is optional): new A(1), A(1), A()
private member only visible in A and its companion object
getter for private field _x (name with _ chosen to avoid clash with x)
special setter syntax of **setter** method enabling assignment syntax:
`val a = A(1); a.x = 2` means `a.x_=(42)`
Right associative operator if ends with colon: `42 +: a` means `a.+:(42)`
optional end marker checked by compiler
becomes a **companion object** if same name and in same code file
apply is optional: A(1) is expanded to A.apply(1)
new is needed here to avoid recursive calls
private members can be accessed in companion

Getters and setters above are auto-generated by **var** in primary constructor: `class A(var x: Int = 0)`
With **val** in primary constructor only getter, no setter, is generated: `class A(val x: Int = 0)`
Private constructor e.g. to enforce use of factory in companion only: `class A private (var x: Int = 0)`
Instead of default arguments, an **auxiliary constructor** can be defined (less common): `def this() = this(0)`

```
class IntVec(private val xs: Array[Int]):
  def update(i: Int, x: Int): Unit = { xs(i) = x }
  def apply(i: Int): Int = xs(i)
```

Special syntax for **update** and **apply**:
v(0) = 0 expands to v.update(0,0)
v(0) expands to v.apply(0)
where val v = IntVec(Array(1,2,3))

Type parameters, type bounds, variance, ClassTag

```
class Box[T](val x: T):
  def pair[U](y: U): (T, U) = (x, y)
val b = Box(0)
val p: (Box[Int], Box[Char]) = b.pair(Box('!'))
```

a **generic class** Box with a **type parameter** T, allowing x to be of any type
a **generic method** with **type parameter** U
T is bound to the type of x, U is free in pairedWith, so y can be of any type
same as (with explicit type parameters): val b: Box**[Int]** = new Box**[Int]**(0)

type bounds `>:` supertype `<:` subtype
+ covariance **- contravariance** `class Box[+T](x: T){ def pair[U >: T](y: U) = (x, y) }`

```
val f: [A] => Seq[A] => A = [A] => xs => xs.head
```

polymorfic function type and lambda

ClassTag needed for generic array constr.: `def mkArr[A:reflect.ClassTag](a: A) = Array[A](a)`

Expressions

literals	`0 0L 0.0 "0" '0' true false`	Basic types e.g. Int, Long, Double, String, Char, Boolean
block	`{ expr1; ...; exprN }`	The value of a block is the value of its last expression
if	`if cond then expr1 else expr2`	Value is expr1 if cond is true, expr2 if false (else is optional)
match	`expr match caseClauses`	Matches expr against each case clause, see pattern matching.
for	`for x <- xs do expr`	Loop for each x in xs, x visible in expr, type Unit
yield	`for x <- xs yield expr`	Yields a sequence with elems of expr for each x in xs
while	`while cond do expr`	Loop expr while cond is true, type Unit
throw	`throw new Exception("Bang!")`	Throws an exception that halts execution if not in try catch
try	`val result = try expr catch f finally doStuff`	Evaluate function f: Throwable => T if exception thrown by expr f for example: `{case e: Exception => someValue}` finally is optional, doStuff always done even if expr throws

Expressions (continued)

Tuples:

Empty tuple, unit value	`()`	the only value of type Unit
2-tuple value	`(1, "hi")`	also: 1->"hi" and Tuple2(1, "hi")
2-tuple type	`(Int, String)`	same as Tuple2[Int, String]

Integer division and remainder:

`a / b` no decimals if Int, Short, Byte
`a % b` fulfills: (a / b) * b + (a % b) == a
Check if x is even: `x % 2 == 0`

Tuple prepend `3 *: (1.0, '!')` of type `Int *: Double *: Char *: EmptyTuple` same as (Int, Double, Char)
Methods on tuples: `apply _1 _2 ... drop take head init tail zip toArray toIArray toList`

Non-referable reference: `null` refers to null object of type Null. Instead prefer Option or unitialized:
Uninitialized: `var x: String = scala.compiletime.uninitialized` mutable AnyRef field set to null
Shorthand assignment: `x += 1` expands to `x = x + 1` if no method += is available, works for all operators
Check arguments: `require(x >= 0)` If condition is false throws IllegalArgumentException. Optional param msg.
Check assertion: `assert(x >= 0)` If condition is false throws AssertionError. Optional param msg.

Evaluation order	`(1 + 2) * 3`	parenthesis control order
Method application	`1.+(2)`	call method + on object 1
Operator notation	`1 + 2`	same as 1.+(2)
Conjunction	`c1 && c2`	true if both c1 **and** c2 true
Disjunction	`c1 || c2`	true if at least one of c1 **or** c2 true
Negation	`!c`	logical **not**, false if c is true
Function application	`f(1, 2, 3)`	same as f.apply(1,2,3)
Function literal	`x => x + 1`	anonymous function, "lambda"
Placeholder syntax	`_ + 1`	same as x => x+1, if arg used once
Object creation	`new C(1,2)`	class args (1,2) new is optional
Self reference	`this`	refers to the object being defined
Supertype reference	`super.m`	refers to member m of supertype

Precedence of operators starting with:

all letters	**lowest**
`|`	
`^`	
`&`	
`= !`	
`< >`	
`:`	
`+ -`	
`* / %`	
other special chars	**highest**

Pattern matching, type tests

`expr match` expr is matched against patterns from top until match found, yielding the expression after =>
`case "hello" => expr` **literal pattern** matches any value equal (in terms of ==) to the literal
`case x: C => expr` **typed variable pattern** matches all instances of C, binding variable x to the instance
`case C(x, y, z) => expr` **constructor pattern** matches values of the form C(x, y, z), args bound to x,y,z
`case (x, y, z) => expr` **tuple pattern** matches tuple values, alias for constructor pattern Tuple3(x, y, z)
`case x +: xs => expr` **sequence extractor patterns** matches head and tail, also x +: y +: z +: xs etc.
`case p1 | ... | pN => expr` **alternative pattern**, match if at least one pattern p1, ..., pN match
`case x@pattern => expr` a **pattern binder** with the @ sign binds a variable to (part of) a pattern
`case x => expr` **untyped variable pattern** matches any value, typical "catch all" at bottom: `case _ =>`
Pattern matching on direct subtypes of a **sealed** class is checked for exhaustiveness by the compiler

Matching with type pattern `x match { case a: Int => a; case _ => 0 }` is preferred over explicit instance test and casting: `if x.isInstanceOf[Int] then x.asInstanceOf[Int] else 0`

Enumerations

```
enum Col:
  case Red, Green, Blue
```
Col is a sealed class, values in companion of type Col: `Col.Red` etc.
`Col.values == Array(Col.Red, Col.Green, Col.Blue)`

`Col.Blue.ordinal` zero-based ordinal number, here 2. The `toString` of `Col.Blue` is `"Blue"`
`Col.valueOf("Red")` value from String, here Red, can throw IllegalArgumentException
`Col.fromOrdinal(0)` value from Int, here Red, can throw NoSuchElementException

```
enum Bin(val toInt: Int):
  case F extends Bin(-1)
  case T extends Bin(1)
```
Parameterized enum. `val` is needed for class param to be externally visible.
get parameter from case value: `Bin.F.toInt == -1`
you can also define case members (def, val, etc) inside enums

```
enum Color(val rgb: Int):
  case Red   extends Color(0xFF0000)
  case Green extends Color(0x00FF00)
  case Blue  extends Color(0x0000FF)
  case Mix(mix: Int) extends Color(mix)
```
Algebraic Data Type (ADT). Parameterized `case` expands to case class.
The exends clause is only needed if parameters are passed.
0x00FF00 is a hexadecimal Int litteral, decimal value 65280
Parameter access: `Color.Blue.rgb == 255`
expanded to:
case class Mix(mix: Int) extends Color(mix) in companion object of trait Color(val rgb: Int)

Extension methods

Extension methods allow adding methods to a type after the type is defined, e.g. add a method to type String:

```
extension (s: String) def shoutBackwards = s.reverse.toUpperCase
```

Can be called using dot notation and normal call: `"hej".shoutBackwards; shoutBackwards("hej")`

Collective extension methods provide multiple extensions to the same type:

```
extension (xs: Seq[Double])                        // note: no trailing colon
  def mean: Double = xs.sum / xs.length            // significant indentation
  def midrange: Double = Seq(xs.max, xs.min).mean

extension [T](xs: List[T])     // generic extension, type param before paren
  def second = xs.tail.head

extension [T: Ordering](xs: List[T]) // generic extension with context bound
  def sortedDescending = xs.reverse.sorted       // sorted requires Ordering
```

Contextual abstractions: given, using, summon[T]

```
enum Lang { case En, Sv }                                  enum assumed in examples below
case class Config(lang: Lang)                              case class assumed in examples below
```

Contextual abstraction allows values to be inferred, and arguments to be filled in, based on expected type.

```
object Config:                                             Given values in companion have lowest priority.
  given default: Config = Config(Lang.En)                  A given instance, the name default: is optional.

def greet(name: String)(using cfg: Config) =               A using parameter, the name cfg: is optional.
  if cfg.lang == Lang.Sv then s"hej $name"                 If unnamed use: summon[Config].lang
  else "hello $name"                                       where summon returns given of specified type
```

A `using` argument can be omited: `greet("Anna")` A value of given type is inferred, here Config.default

Explicit given: `greet("Anna")(using Config(Lang.sv))` Keword `using` is needed for explicit givens.

A "**type class**" in Scala is a `trait` with one type parameter and at least one abstract method. Example:

```
trait CanShow[T]:                                          A trait with one type parameter T
  extension (x: T) def show: String                        An abstract member with a parameter of type T
```

Separately define given instances of type class CanShow for any type in any local scope:

```
given CanShow[Config] with          keyword with needed when implementing members of given types
  extension (x: Config) def show: String = s"Language is " + x.lang
```

Extension method show in type class CanShow is now available on Config implicitly: `Config(Lang.En).show`

Context bound: Use colon after type parameter as a shorthand when using parameter is a type class:

```
def rev[T: CanShow](x: T) = x.show.reverse    CanShow[T] is required; automatically expands to:
                              def rev[T](x: T)(using CanShow[T]) = x.show.reverse
```

Context functions: the type of a function with a context parameter of type U to result type R is denoted `U ?=> R`

```
val f: Config ?=> Int = c ?=> c.lang.ordinal      a context function lambda, c is a using-param.
```

When `f` is referred to without argument it is automatically expanded to: `f(using summon[Config])`

If type `U ?=> E` is expected, an `expr` of type E is expanded to a context function lambda: `(u: U) ?=> expr`

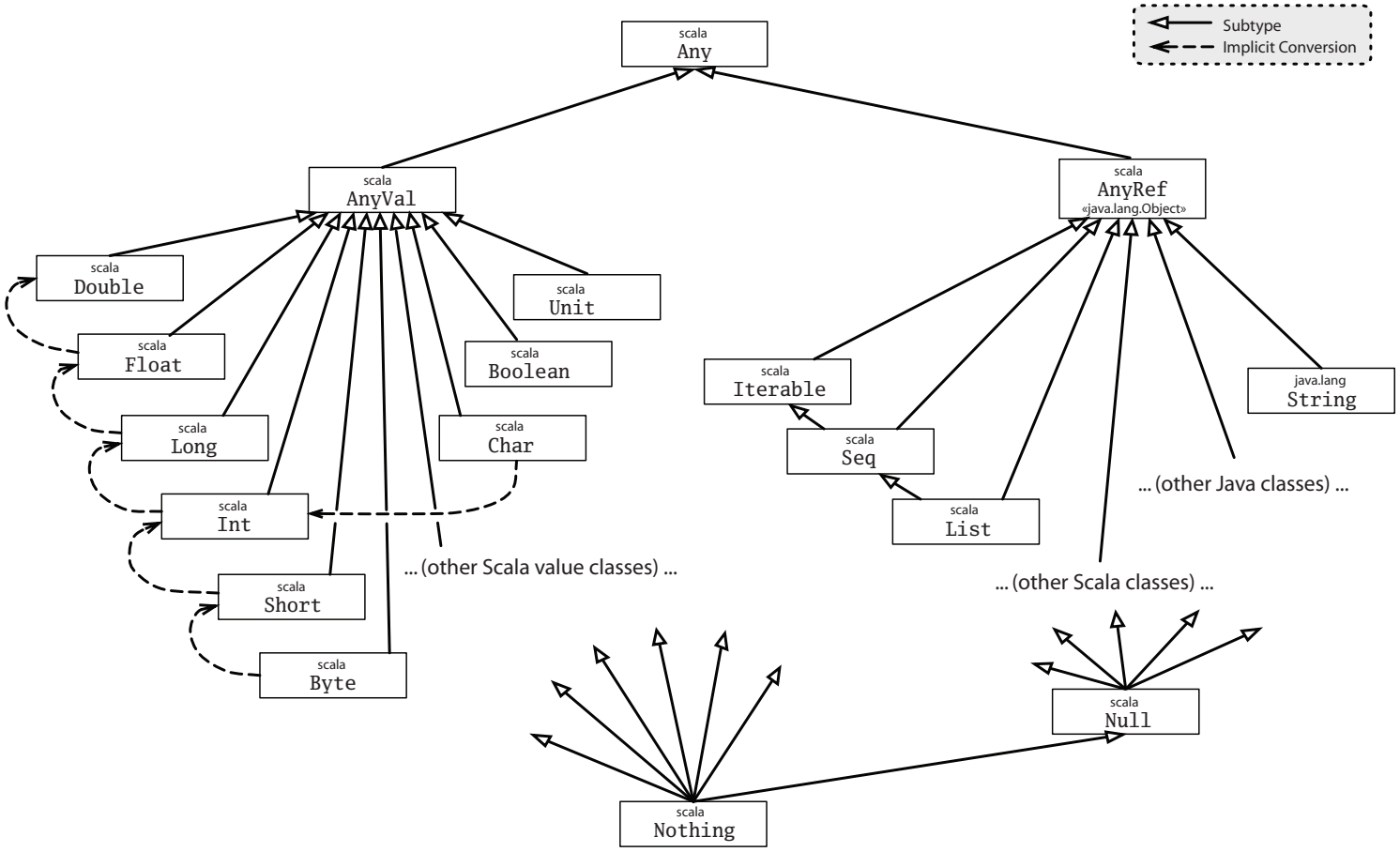
Opaque types: zero-cost abstractions

```
object Physical:                                   Create a local scope for our opaque type alias
  opaque type Distance = Double      alias of Distance is Double, but that is invisible outside Physical
  object Distance:                   In the local scope it is known that Distance is actually Double
    extension (value: Double) def meter: Distance = value
```

Make the opaque type alias member available: `import Physical.Distance.meter`

Type of dist is Distance and members of type Double not available on dist: `val dist = 42.0.meter`

Prefer more versatile: `case class Distance(value: Double)` if zero allocation not performance-critical.



Numbers

Number types

name	# bits	range	literal
`Byte`	8	$-2^7 \ldots 2^7 - 1$	`0.toByte`
`Short`	16	$-2^{15} \ldots 2^{15} - 1$	`0.toShort`
`Char`	16	$0 \ldots 2^{16} - 1$	`'0' '\u0030'`
`Int`	32	$-2^{31} \ldots 2^{31} - 1$	`0 0xF`
`Long`	64	$-2^{63} \ldots 2^{63} - 1$	`0L`
`Float`	32	$\pm 3.4 \cdot 10^{38}$	`0F`
`Double`	64	$\pm 1.8 \cdot 10^{308}$	`0.0`

Some methods in `math` same as in `java.lang.Math`:
`pow(x, y)` x^y `sqrt(x)` $\sqrt{x}$ `exp(x)` e^x
`hypot(x, y)` $\sqrt{x^2 + y^2}$ `log(x)` natural logaritm
`sin(x) asin(x) cos(x) tan(x) atan2(x,y)`
`floorMod(x, y)` similar to x % y but always positive

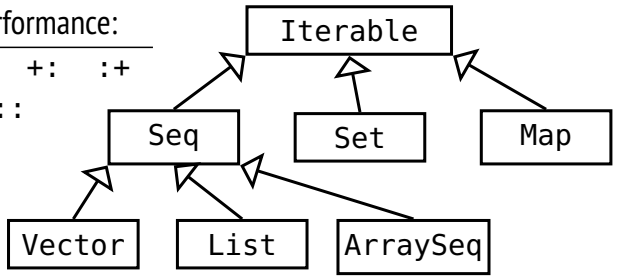
Methods on numbers

`x.abs`	math.abs(x), absolute value
`x.round`	math.round(x), to nearest Long
`x.floor`	math.floor(x), cut decimals
`x.ceil`	math.ceil(x), round up
`x max y`	math.max(x, y), maximum
`x.toInt`	also toByte, toChar, toDouble etc.
`1 to 4`	Range.inclusive(1, 4), incl. 1,2,3,4
`0 until 4`	Range(0, 4), incl. 0,1,2,3
`Int.MinValue`	least possible value of type Int
`Int.MaxValue`	largest possible value of the Int
`math.Pi math.E`	π e
`4.0.toRadians`	also `toDegrees`

scala.util.Random

`Random.nextInt(n)`	A random `Int` uniformly distributed from `0` until n, not including n.
`Random.nextInt()`	A random `Int` uniformly distributed from `-Int.MaxValue` to `Int.MaxValue`.
`Random.nextDouble()`	A random `Double` uniformly distributed from `0.0` to `0.9999999999999999`.
`Random.between(a, b)`	A random number (a, b is Int, Float or Double) uniformly distributed from a until b.
`Random.nextPrintableChar()`	A random `Char` including `A` to `Z` or any other printable char (uniform dist.).
`Random.nextGussian()`	A random `Double` that is **normally** distributed with mean 0 and std dev 1.
`Random.shuffle(xs)`	Returns a new sequence with the elements in xs in any, equally likely, random order.
`Random.setSeed(s)`	Set Random's integer seed `s`; sequence of next "random" values will be same on each run.

scala.collection.immutable.	mutable.	methods with good performance:
Vector	ArrayBuffer	head tail apply +: :+
List	ListBuffer	head tail +: ::
ArraySeq	ArraySeq	head apply
Set	Set	contains + -
Map	Map	apply + -



Factory examples:

```
List(0, 0, 0); List(List(0), List(0))   same as List.fill(3)(0); List.fill(2,1)(0)
Map(1 -> "Sweden", 2 -> "Norway")       same as Map((1, "Sweden"), (2, "Norway"))
Vector.tabulate(3)(i => i + 10)                          gives Vector(10, 11, 12)
Vector.iterate(1.2, 3)(_ + 0.5)                        gives Vector(1.2, 1.7, 2.2)
collection.mutable.Set.empty[Int]        same as collection.mutable.Set[Int]() etc.
```
On **mutable** and immutable Set, Map, ArraySeq, etc.: `toSet`, `toMap`, `toSeq` etc. returns **immutable** collection

String — implicitly `Seq[Char]` via `immutable.StringOps`

Some methods below are from java.lang.String and some methods are implicitly added from StringOps, etc.
Strings are implictly treated as **`Seq[Char]`**, so all `Iterable` and `Seq` methods also work.

```
s(i)  s.apply(i)  s.charAt(i)     Returns the character at index i.
s.capitalize                      Returns this string with first character converted to upper case.
s.compareTo(t)                    Returns x where x < 0 if s < t, x > 0 if s > t, x is 0 if s == t
s.compareToIgnoreCase(t)          Similar to compareTo but not sensitive to case.
s.endsWith(t)                     True if string s ends with string t.
s.replace(s1, s2)                 Replace all occurances of s1 with s2 in s.
s.split(c)                        Returns an array of strings split at every occurance of character c.
s.startsWith(t)                   True if string s begins with string t.
s.stripMargin                     Strips leading white space followed by | from each line in string.
s.substring(i)                    Returns a substring of s with all charcters from index i.
s.substring(i, j)                 Returns a substring of s from index i to index j-1.
s.toIntOption  s.toDoubleOption   Parses s as an Option[Int] or Option[Double] etc. None if invalid.
42.toString   42.0.toString       Converts a number to a String.
s.toLowerCase                     Converts all characters to lower case.
s.toUpperCase                     Converts all characters to upper case.
s.trim                            Removes leading and trailing white space.

val sb = StringBuilder("")        En empty mutable string. (If multi-thread access use StringBuffer.)
sb.append("hello")                Append string in-place. Also for Int, Char, Boolean, etc
sb.insert(i, s)                   Insert s at index i.
sb.delete(i)                      Remove char at index i.
sb.setCharAt(i, ch)               Update char at index i to ch.
sb.toString                       Make an immutable String copy of sb.
```

Escape	char
\n	line break
\t	horisontal tab
\"	double quote "
\'	single quote '
\\	backslash \
\u0041	unicode for A

Special strings

```
   "hello\nworld\t!"   string including escape char for line break and tab
"""a "raw" string"""   can include quotes and span multiple lines
          s"x is $x"   s interpolator inserts values of existing names after $
    s"x+1 is ${x+1}"   s interpolator evaluates expressions within ${}
          f"$x%5.2f"   format Double x to 2 decimals at least 5 chars wide
            f"$y%5d"   format Int y right justified at least five chars wide
```

Iterable[A]

What	Usage	Explanation f is a function, pf is a partial funct., p is a predicate.
Traverse:	`xs.foreach(f)`	Executes f for every element of xs. Return type Unit.
Add:	`xs ++ ys`	A new collection with xs followed by ys (concatenation).
Map:	`xs.map(f)`	A new collection created by applying f to every element in xs.
	`xs.flatMap(f)`	A new collection created by applying f (which must return a collection) to all elements in xs and concatenating the results.
	`xs.collect(pf)`	A new collection created by applying the pf to every element in xs for which it is defined (undefined ignored).
Convert:	`toVector toList toSeq toBuffer toArray`	Converts a collection. Unchanged if the run-time type already matches the demanded type.
	`toSet`	Converts the collection to a set; duplicates removed.
	`toMap`	Converts a collection of key/value pairs to a map.
Array Copy:	`xs.copyToArray(arr,s,n)`	Copies at most n elements of xs to array arr starting at index s (last two arguments are optional). Return type Unit.
Size info:	`xs.isEmpty`	Returns true if the collection xs is empty.
	`xs.nonEmpty`	Returns true if the collection xs has at least one element.
	`xs.size`	Returns an `Int` with the number of elements in xs.
Retrieval:	`xs.head xs.last`	The first/last element of xs (or some elem, if order undefined).
	`xs.headOption xs.lastOption`	The first/last element of xs (or some element, if no order is defined) in an option value, or `None` if xs is empty.
	`xs.find(p)`	An option with the first element satisfying p, or None.
Subparts:	`xs.tail xs.init`	The rest of the collection except xs.head or xs.last.
	`xs.slice(from, to)`	The elements in from index `from` until (not including) `to`.
	`xs.take(n)`	The first n elements (or some n elements, if order undefined).
	`xs.drop(n)`	The rest of the collection except xs take n.
	`xs.takeRight(n) xs dropRight n`	Similar to `take` and `drop` but takes/drops the last n elements (or any n elements if the order is undefined).
	`xs.takeWhile(p)`	The longest prefix of elements all satisfying p.
	`xs.dropWhile(p)`	Without the longest prefix of elements that all satisfy p.
	`xs.filter(p)`	Those elements of xs that satisfy the predicate p.
	`xs.filterNot(p)`	Those elements of xs that do not satisfy the predicate p.
	`xs.splitAt(n)`	Split xs at n returning the pair (xs take n, xs drop n).
	`xs.span(p)`	Split xs by p into the pair (xs takeWhile p, xs.dropWhile p).
	`xs.partition(p)`	Split xs by p into the pair (xs filter p, xs.filterNot p)
	`xs.groupBy(f)`	Partition xs into a map of collections according to f.
Conditions:	`xs.forall(p)`	Returns true if p holds for all elements of xs.
	`xs.exists(p)`	Returns true if p holds for some element of xs.
	`xs.count(p)`	An `Int` with the number of elements in xs that satisfy p.
Folds:	`xs.foldLeft(z)(op) xs.foldRight(z)(op)`	Apply binary operation op between successive elements of xs, going left to right (or right to left) starting with z.
	`xs.reduceLeft(op) xs.reduceRight(op)`	Similar to foldLeft/foldRight, but xs must be non-empty, starting with first element instead of z.
	`xss.flatten`	xss (a collection of collections) is reduced by concatenation.
	`xs.sum xs.product`	Calculates the sum/product of numeric elements.
	`xs.min xs.max`	Finds the minimum/maximum of numeric elements.
	`xs.minBy(f) xs.maxBy(f)`	Finds the min/max of value after applying f to each element.
	`xs.minOption xs.maxOption`	Finds a min/max value based on implicitly available ordering.
	`xs.minByOption(f)`	Finds a min/max value after applying f to each element.

Iterable[A] contiues on next page...

Iterable[A] (continued)

What	Usage	Explanation
Iterators:	`val it = xs.iterator`	An iterator `it` of type `Iterator` that yields each element one by one: `while (it.hasNext) f(it.next)`
	`xs.grouped(size)`	An iterator yielding fixed-sized chunks of this collection.
	`xs.sliding(size)`	An iterator yielding a sliding fixed-sized window of elements.
Zippers:	`xs.zip(ys)`	An iterable of pairs of corresponding elements from xs and ys.
	`xs.zipAll(ys, x, y)`	Similar to `zip`, but the shorter sequence is extended to match the longer one by appending elements x or y.
	`xs.zipWithIndex`	An iterable of pairs of elements from xs with their indices.
Compare:	`xs.sameElements(ys)`	True if xs and ys contain the same elements in the same order.
Make string:	`xs.mkString(start, sep, end)`	A string with all elements of xs between separators sep enclosed in strings start and end; start, sep, end are all optional.

Seq[A]

Indexing and size:	`xs(i)` `xs.apply(i)`	The element of xs at index i.
	`xs.length`	Length of sequence. Same as `size` in `Iterable`.
	`xs.indices`	Returns a `Range` extending from 0 until xs.length.
	`xs.isDefinedAt(i)`	True if i is contained in xs.indices.
	`xs.lengthCompare(n)`	Returns -1 if xs is shorter than n, +1 if it is longer, else 0.
Index search:	`xs.indexOf(x)`	The index of the first element in xs equal to x.
	`xs.lastIndexOf(x)`	The index of the last element in xs equal to x.
	`xs.indexOfSlice(ys)` `xs.lastIndexOfSlice(ys)`	The (last) index of xs such that successive elements starting from that index form the sequence ys.
	`xs.indexWhere(p)`	The index of the first element in xs that satisfies p.
	`xs.segmentLength(p, i)`	The length of the longest uninterrupted segment of elements in xs, starting with xs(i), that all satisfy the predicate p.
	`xs.prefixLength(p)`	Same as `xs.segmentLength(p, 0)`
Add:	`x +: xs` `xs :+ x`	Prepend/Append x to xs. Colon on the collection side.
	`xs.padTo(len, x)`	Append the value x to xs until length len is reached.
Update:	`xs.patch(i, ys, r)`	A copy of xs with r elements of xs replaced by ys starting at i.
	`xs.updated(i, x)`	A copy of xs with the element at index i replaced by x.
	`xs(i) = x` `xs.update(i, x)`	Only available for mutable sequences. Changes the element of xs at index i to x. Return type Unit.
Sort:	`xs.sorted`	A new Seq[A] sorted using implicitly available ordering of A.
	`xs.sortWith(lt)`	A new Seq[A] sorted using less than lt: (A, A) => Boolean.
	`xs.sortBy(f)`	A new Seq[A] sorted by implicitly available ordering of B after applying f: A => B to each element.
Reverse:	`xs.reverse`	A new sequence with the elements of xs in reverse order.
	`xs.reverseIterator`	An iterator yielding all the elements of xs in reverse order.
	`xs.reverseMap(f)`	Similar to map in Iterable, but in reverse order.
Tests:	`xs.startsWith(ys)`	True if xs starts with sequence ys.
	`xs.endsWith(ys)`	True if xs ends with sequence ys.
	`xs.contains(x)`	True if xs has an element equal to x.
	`xs.containsSlice(ys)`	True if xs has a contiguous subsequence equal to ys
	`(xs corresponds ys)(p)`	True if corresponding elements satisfy the binary predicate p.
Subparts:	`xs.intersect(ys)`	The intersection of xs and ys, preserving element order.
	`xs.diff(ys)`	The difference of xs and ys, preserving element order.
	`xs.union(ys)`	Same as `xs ++ ys` in Iterable.
	`xs.distinct`	A subsequence of xs that contains no duplicated element.

Mutation methods in `mutable.{ArraySeq[A], ArrayBuffer[A], ListBuffer[A]}`

`xs(i) = x  xs.update(i, x)`	Replace element at index i with x. Return type Unit.
`xs.insert(i, x)  xs.remove(i)`	Insert x at i, ret. Unit. Remove elem at i, ret. removed elem.
`xs.append(x)  xs.addOne(x)  xs += x`	Insert x at end. Return xs itself.
`xs.prepend(x)  x +=: xs`	Insert x in front. Return xs itself.
`xs -= x`	Remove first occurance of x (if exists). Returns xs itself.
`xs ++= ys  xs.addAll(ys)`	Appends all elements in ys to xs and returns xs itself.
`xs.clear()  xs.sortInPlace`	Remove all elements, return Unit. Sort in place, return itself.
`xs.filterInPlace(p).mapInPlace(f)`	ArrayBuffer, ListBuffer: update in place. Return itself.

`Set[A]`

`xs(x)  xs.apply(x)  xs.contains(x)`	True if x is a member of xs.
`xs.subsetOf(ys)`	True if xs is a subset of ys.
`xs + x  xs - x` `xs + (x, y, z)  xs - (x, y, z)`	Returns a new set including/excluding elements. Addition/subtraction can be applied to many arguments.
`xs.intersect(ys)`	A new set with elements in both xs and ys. Also: &
`xs.union(ys)`	A new set with elements in either xs or ys or both. Also: |
`xs.diff(ys)`	A new set with elements in xs that are not in ys. Also: &~

Mutation methods in `mutable.Set[A]`

`xs += x  xs.addOne(x)  xs -= x`	Returns the same set with included/excluded element x.
`xs ++= ys  xs.addAll(ys)`	Adds all elements in ys to set xs and returns itself.
`xs.add(x)  xs.remove(x)`	Adds/removes x to xs and returns true if xs was mutated.
`xs(x) = b  xs.update(x, b)`	If b is true, adds x to xs, else removes x. Return type Unit.
`xs.filterInPlace(p).mapInPlace(f)`	Update in place, no duplicates remain. Returns itself.

`Map[K, V]`

`ms.get(k)`	The value associated with key k an option, None if not found.
`ms(k)  ms.apply(k)`	The value associated with key k, or exception if not found.
`ms.getOrElse(k, d)`	The value associated with key k in map ms, or d if not found.
`ms.isDefinedAt(k)`	True if ms contains a mapping for key k. Also: ms.contains(k)
`ms + (k -> v)  ms + ((k, v))` `ms.updated(k, v)`	The map containing all mappings of ms as well as the mapping k -> v from key k to value v. Also: ms + (k1 -> v1, k2 -> v2)
`ms - k`	Excluding any mapping of key k. Also: ms - (k, l, m)
`ms ++ ks`	The mappings of ms with the mappings of ks added/removed.
`ms.keys  ms.values  ms.keySet`	An Iterable/Set containing each key/value in ms.
`ms.view.mapValues(f).toMap`	A new Map[K, U] created by applying f: V => U to each value.

Mutation methods in `mutable.Map[K, V]`

`ms(k) = v  ms.update(k, v)`	Adds mapping k to v, overwriting any previous mapping of k.
`ms += (k -> v)  ms -= k`	Add or overwrite k -> v / Remove k if key exists or no effect.
`ms.put(k, v)  ms.remove(k)`	Adds/removes mapping; returns previous value of k as an option.
`ms.mapValuesInPlace(f)`	Update all values in place by applying f: (K, V) => V to each pair.
`ms.filterInPlace(p)`	Filter in place, keep elems that satisfy p. Returns xs itself.

`java.lang.Array[A]` implicitly `mutable.Seq[A]` via `ArrayOps`

`Array` has efficient `update`, but is strange with generics, and gives reference equality on `xs == ys`. Shallow equality test: `xs.sameElements(ys)` Deep equality test: `java.util.Arrays.deepEquals(xs, ys)`
`val xs = new Array[Int](n)` Allocate n Int values initialized to default value for number types: 0
`val ys = new Array[String](n)` n String references initialized to default value of reference types: `null`
Copy from start: `Array.copyOf(xs, newLen)` From pos: `Array.copy(xs, pos, xs2, toPos, n)`
`Array.ofDim[Int](3,2)` gives `Array(Array(0, 0), Array(0, 0), Array(0, 0))`
Prefer: "normal" collection: `immutable.ArraySeq`, `mutable.ArraySeq`, or immutable `IArray`

scala.{Option, Some, None} to handle missing values

Option[T] is like a collection with zero or one element. **Some[T]** and **None** are subtypes of Option.

```
def rnd(): Option[String] = if math.random() > 0.9 then Some("bingo") else None
rnd().getOrElse(expr)   if rnd() == Some[T] then x else expr
rnd().map(f)            if rnd() == Some(x) then Some(f(x)) else None
rnd().get               if rnd() == Some[T] then x else throws NoSuchElementException
rnd() match                 a match on Option where expr1 if Some(x) or expr2 if None
  case Some(x) => expr1     or use if rnd().isDefined then expr1 else expr2
  case None => expr2        or use if rnd().isEmpty then expr2 else expr1
                            or use if rnd().nonEmpty then expr1 else expr2
```

Some collection methods also work on **Option**: map, foreach, filter, toVector, flatten, flatMap, ... with None discarded.

scala.util.{Either, Left, Right} to handle errors as values

```
def rnd(): Either[String, Int] = // normally returns Int but can report error
  if math.random() > 0.9 then Right(42) else Left("Bad") // Left wraps message
```

Either[E, V] wraps a normal value or error. **Left[E]** error of type E. **Right[V]** normal value of type V.

`rnd().isRight` Predicate that tests if normal value. To test if error: `rnd().isLeft`
`rnd().merge` Unraps any value, will get precice common supertype, here union type: `String | Int`
`rnd().swap` Turns a Right to Left and vice versa, with swapped type parameters.

Some collection methods also work on **Either**: map, foreach, flatMap, ... with Left discarded.

scala.util.{Try, Success, Failure} to handle exceptions as values

Try[T] is like a collection with **Success[T]** or **Failure[E]**. `import scala.util.{Try, Success, Failure}`

```
Try{ ...; ...; expr1 }.getOrElse(expr2)      evaluates to expr1 if successful or expr2 if exception
Try(expr1).recover{case e: Exception => expr2} Success(expr2) if exception else Success(expr1)
Try(1/0) match
  case Success(x) => println("happy path")                  or use predicate: isSuccess
  case Failure(e) => println("exception")                   or use predicate: isFailure
```

Some collection methods also work on **Try**: map, foreach, flatMap, ... with Failure discarded.

scala.io.{Source, StdIn} java.nio.file.{Path, Paths, Files}

Alternative to scala.io, java.nio: use **Scala toolkit** os.write os.read etc, see page 12, which normally is preferred.

Read string of lines from **file**, `fromFile` gives BufferedSource, `getLines` gives Iterator[String]

```
val source = scala.io.Source.fromFile("f.txt", "UTF-8")   or fromURL(adr, enc)
val lines = try source.getLines.mkString("\n") finally source.close
```

Read string from **standard in** (prompt string is optional) using readLine; **write** to **standard out** using println:

```
val input = scala.io.StdIn.readLine("> ")  Reads keystrokes in terminal after printing prompt >
```

Write string to **file** using java.nio:

```
import java.nio.file.{Path, Paths, Files}
import java.nio.charset.StandardCharsets.UTF_8

def save(fileName: String, data: String): Path =
  Files.write(Paths.get(fileName), data.getBytes(UTF_8))
```

Other common character encoding: `java.nio.charset.StandardCharsets.ISO_8859_1`

scala.jdk.CollectionConverters

Enable `.asJava` and `.asScala` with `import scala.jdk.CollectionConverters.*`

`xs.asJava` on a **Scala** collection of type:		`xs.asScala` on a **Java** collection of type:
`Iterator`	⟷	`java.util.Iterator`
`Iterable`	⟷	`java.lang.Iterable`
`Iterable`	←	`java.util.Collection`
`mutable.Buffer`	⟷	`java.util.List`
`mutable.Set`	⟷	`java.util.Set`
`mutable.Map`	⟷	`java.util.Map`
`mutable.ConcurrentMap`	⟷	`java.util.concurrent.ConcurrentMap`

java.util.Scanner

A **Scanner** reads tokens separated by whitespace from a String or File. Throws **NoSuchElementException** if end of file.

```
val s = java.util.Scanner("hello 42 42.0 world")          Create a Scanner from a String.
val f = java.util.Scanner(java.io.File("f.txt"))          Create a Scanner from a File.
val in = java.util.Scanner(System.in)                     Create a Scanner that reads from standard in.
s.hasNext      s.hasNextLine                              True if at least one more token or line is available.
s.hasNextInt   s.hasNextDouble   s.hasNextBigDecimal etc. True if number tokens is next.
s.next()       s.nextLine()                               Read next token or rest of line as a String.
s.nextInt()    s.nextDouble()    s.nextBigInteger etc.    Parse next token as number type.
```

Create a single page web app using ScalaJS

In file `hello-js.scala` Note double colon before version when using dependencies for ScalaJS.

```
//> using scala 3.4.2
//> using platform scala-js
//> using dep org.scala-js::scalajs-dom::2.8.0

import org.scalajs.dom

@main def hello: Unit =
  val p = dom.document.createElement("p")  // create a paragraph node
  p.textContent = "Hello ScalaJS!"
  dom.document.addEventListener("DOMContentLoaded",
    (e: dom.Event) => dom.document.body.appendChild(p))
```

Compile and package javascript to hello.js:

```
scala package hello-js.scala
```

In file `page.html`

```
<!DOCTYPE html>
<head><script src="hello.js"></script></head>
</html>
```

Open file `page.html` in browser e.g. Firefox.

Reserved words in Scala

These words and symbols have special meaning. Can be used as identifiers if put within `backticks`.

```
abstract as case catch class def derives do else end enum export extends extension false
final finally for forSome given if implicit import infix inline lazy macro match new null
object opaque open override package private protected return sealed super then this throw
trait transparent true try type using val var while with yield
 _   :   =   =>   <-   <:   <%   >:   #   @
```

Using the Scala toolkit — os, munit, ujson, upickle

`scala run hello.scala --toolkit default` Use stable vers. of Scala toolkit, an extended std libary.
`//> using toolkit default` Directive in source. `//> using toolkit latest` for unstable vers.

Manage files and os processes with `os-lib`

```
os.pwd   os.root   os.home     Absolute path of type os.Path to working directory, root, and home dir.
os.root/"tmp"      os.pwd/"src"                 Absolute paths to /tmp dir and src in working dir.
val wd = os.pwd/"myDir";  val p = wd/"myFile.txt"            Absolute paths to dir and file.
p.ext  p.segments     p.last            Methods on os.Path, returns "txt", iterator, last segment.
os.remove.all(wd)     os.makeDir(wd)       Remove all files in wd if exists. Create wd if not exists.
os.write(os.pwd/"f.txt", "hi")   os.read(os.pwd/"f.txt") Write text to f.txt. Read as String.
os.exists(os.pwd/"f.txt")        os.list(os.pwd)             true if exists. List files as Seq.
os.write.append(os.pwd / "f.txt", "text to append")     Append text at end of existing file.
os.write.over(os.pwd / "f.txt", "replace contents")        Overwrite text to existing file.
os.copy(os.pwd/"f.txt", os.pwd/"cpy.txt", replaceExisting = false)  conditional copy
os.move(os.pwd/"f.txt", os.pwd/"f2.txt", replaceExisting = false)  conditional move
```

Call OS process in host shell in wd and get output as String:

```
val output = os.proc("cat", wd/"f.txt", wd/"cpy.txt").call(cwd = wd).out.text()
```

Write and run tests with `munit`

compatible with junit, scala, sbt, metals, vscode etc

In file `MyTests.test.scala` in a class extending `munit.FunSuite` as follows:
Run with: `scala test MyTests.test.scala --toolikt deafult`

```
class MyTests extends munit.FunSuite:
  test("sum of two integers"):
    val sum = 2 + 2
    assert(sum == 4)
```

Read and write JSON with `ujson` and `upickle`

```
val jsonString = """{"name": "Anna", "age": 42, "pets": ["Cat", "Dog"]}"""
val json: ujson.Value  = ujson.read(jsonString)
val name = json("name").str                // "Anna"
val nameOpt = json("name").strOpt          // Some("Anna")
val pets = json("pets").arr                // ArrayBuffer("Cat", "Dog")
val petsOpt = json("pets").arrOpt          // Some(ArrayBuffer("Cat", "Dog"))
val m: Map[String, Int] =  Map("Dog" -> 3, "Cat" -> 5)
val js: String = upickle.default.write(m) // """{"Dog":3,"Cat":5}"""
import upickle.default.*
case class PetOwner(name: String, pets: List[String]) derives ReadWriter
val po = PetOwner("Kim", List("Cat", "Dog"))
val jsPo: String = write(po) // """{"name":"Kim","pets":["Cat","Dog"]}"""
val po2: PetOwner = read[PetOwner](jsPo)
```

Using dependencies — from Maven central, Github

Using library dependency from Maven central, here `introprog` for simple graphics, as using-directive in source:

```
//> using dep "se.lth.cs::introprog:1.4.0"
```

Note: double colon adds Scala version to jar.

Or as argument in terminal: `scala repl --dep "se.lth.cs::introprog:1.4.0"`

Using library dependency from Github, one line per dependency, note single colon:

```
//> using dep "termut:termut:0.1.0,url=https://github.com/bjornregnell/termut/
releases/download/v0.1.0/termut_3-assembly-0.1.0.jar"
```