Scala 3.1 Quick Ref @ Lund University

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

Top-level definitions

```scala
// in file: hello.scala
package x.y.z

val msg = "Hello"

@main def greet(args: String*): Unit =
  println(s"$msg ${args.mkString(" ")}")
```

A compilation unit (here hello.scala) consists of top-level definitions such as val, var, def, import, class and object, which may be preceded by a package clause, e.g.: **`package`** `x.y.z` that places the compiled files in directory x/y/z/

```
Compile: scalac hello.scala
    Run: scala x.y.z.greet Earth Moon
```

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. Below `defsAndDecl` denotes a list of definitions and/or declarations. Template bodies `{ ... }` are optional, can be replaced by `:` that opens an indentation region. `=` 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 addAll(xs: Int*) = xs.sum`	Repeated parameters: addAll(1,2,3) or addAll(Seq(1,2,3)*)
	`def twice(block: => Unit) = { block; block }`	Call-by-name argument evaluated later.
Object	`object Name { defsAndDecl }`	Singleton object auto-allocated when referenced the first time.
Class	`class C(parameters) { defsAndDecl }`	A template for objects to be allocated with **new** or apply.
	`case class C(parameters) { defsAndDecl }`	Case class parameters become val members, other case class goodies: equals, copy, hashcode, unapply, nice toString, companion object with apply factory.
Trait	`trait T(parameters) { defsAndDecl }`	A trait is like an abstract class, but can be mixed in.
	`class C extends D, T`	A class can only **extend** one class but **mix in** many traits separated with **,**
Type	`type A = typeDef`	Defines an alias A for the type in typeDef. Abstract if no typeDef.
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.

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 definitions	Delays initialization of val, initialized when first referenced.
`infix`	def definitions	Allow alpha-numeric functions in operator notation without warning.
`abstract`	class definitions	Abstract classes cannot be instantiated (redundant for traits).
`sealed`	class definitions	Restricts direct inheritance to classes in the same source file.
`open`	class definitions	Signal intent to be used in inheritance hierarchy. Silences warning.

Constructors and special methods (getters, setters, apply, update), Companion object

`class A(initX: Int = 0):`	**primary constructor**, new is optional creating objects: A(1), default arg: A()
`  private var _x = initX`	private member only visible in A and its companion object
`  def x: Int = _x`	getter for private field _x (name with _ chosen to avoid clash with x)
`  def x_=(i: Int): Unit =`	special setter syntax to update attribute using assignment:
`    _x = i`	val a = A(1); a.x = 2
`end A`	optional end marker checked by compiler, also allowed: `end class`
`object A:`	becomes a **companion object** if same name and in same code file
`  def apply(i: Int = 0) =`	apply is optional: A.apply(1), A(1), A()
`    new A(i)`	new is needed here to avoid recursive calls
`  val y = A(1)._x`	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)`

```scala
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 expanded to v.update(0,0)
v(0)   expanded to v.apply(0)
where val v = new IntVec(Array(1,2,3))

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 resultOfUnsafeExpr =`	Evaluate function f: Throwable => T if exception thrown by expr
	`  try expr catch f`	f for example: `{case e: Exception => someValue}`
	`  finally doStuff`	finally is optional, doStuff always done even if expr throws

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 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"
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 beginning with:

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

Non-referable reference	`null`	refers to null object of type Null
Uninitialized	mutable AnyRef field set to null	`var x: String = scala.compiletime.uninitialized`
Assignment operator	`x += 1`	expands to `x = x + 1` if no method += is available, works for all operators

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

Integer division and remainder:
`a / b` no decimals if a, b Int, Short, Byte
`a % b` fulfills: (a / b) * b + (a % b) == a

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

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`	matches if at least one **pattern alternative** p1, p2 ... or pN matches
`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:`	Col is a sealed class, values in companion of type Col: `Col.Red` etc.
`  case Red, Green, Blue`	Array of values: `Col.values(Col.Red.ordinal) == Col.Red`
	value from String: `Col.valueOf("Red") == Col.Red`
`enum Bin(val toInt: Int):`	**parameterized enum** val is needed for class param to be externally visible.
`  case F extends Bin(-1)`	get parameter from case value: `Bin.F.toInt == -1`
`  case T extends Bin(1)`	you can also define case members (def, val, etc) inside enums

Type parameters, type bounds, variance, ClassTag

`class Box[T](val x: T):`	a **generic class** Box with a **type parameter** T, allowing x to be of any type
`  def pair[U](y: U): (T, U) = (x, y)`	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
`val b = Box(0)`	same as (with explicit type parameters): val b: Box**[Int]** = new Box**[Int]**(0)
`val p: (Box[Int], Box[char]) = b.pair(Box('!'))`	**type bounds** `>:` supertype `<:` subtype

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

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

scala.{Option, Some, None}, scala.util.{Try, Success, Failure}

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

```
val opt: Option[String] = if math.random() > 0.9 then Some("bingo") else None
```

`opt.getOrElse(expr)`	x: T if opt == Some[T](x) else expr
`opt.map(x => ... }`	apply x => ... to x if opt is Some(x) else None
`opt.get`	x: T if Some[T](x) else throws NoSuchElementException
`opt match { case Some(x) => expr1; case None => expr2 }`	expr1 if Some(x) else expr2

Other collection-like methods on **Option**: foreach, isEmpty, filter, toVector, ..., on **Try**: map, foreach, toOption, ...

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) => x; case Failure(e) => 0 }
```

Reading/writing from file, and standard in/out:

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
```

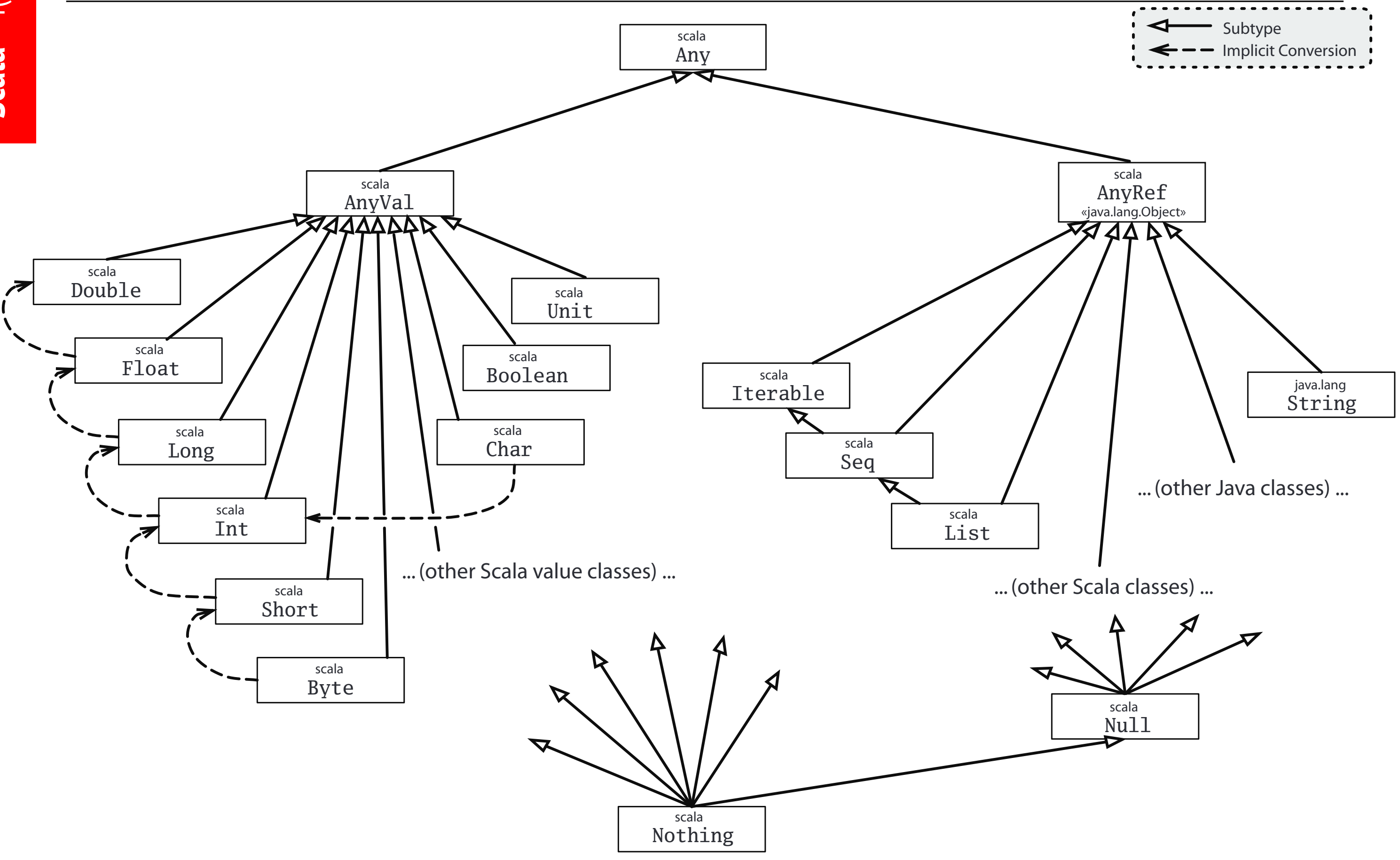
Read string from **standard in** (prompt string is optional) using readLine; **write** to **standard out** using println:

```
val input = scala.io.StdIn.readLine("> ")
println(s"you wrote $input after > using ${input.length} chars")
```

Write string to **file** after 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))
```

The Scala Type System



Number types

name	# bits	range	literal
`Byte`	8	$-2^7 \dots 2^7-1$	`0.toByte`
`Short`	16	$-2^{15} \dots 2^{15}-1$	`0.toShort`
`Char`	16	$0 \dots 2^{16}-1$	`'0'  '\u0030'`
`Int`	32	$-2^{31} \dots 2^{31}-1$	`0  0xF`
`Long`	64	$-2^{63} \dots 2^{63}-1$	`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`:
`hypot(x, y) sin(x) cos(x) tan(x)`
`pow(x, y) sqrt(x) log(x) toRadians(x)`
`floorMod(x, y)` similar to x % y but always positive

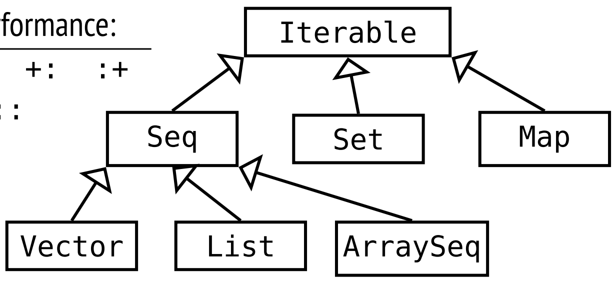
Methods on numbers

`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 cut decimals
`x max y`	math.max(x, y), gives largest, also min
`x.toInt`	also toByte, toChar, toDouble etc.
`1 to 4`	Range.inclusive(1, 4), contains 1,2,3,4
`0 until 4`	Range(0, 4), contains 0,1,2,3
`Int.MinValue`	least possible value of type Int
`Int.MaxValue`	largest possible value of the Int
	similar for all number types.

The Scala Standard Collection Library

`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 + -`



`String` and `Array` has implicit conversions that make sequence methods work as for other sequences.

`Array` has efficient `update`, but strange with generics. Special `Array` allocation syntax: `new Array[Int](n)`
Prefer `ArraySeq` (a "normal" collection, better with generics) or `IArray` (an Array that cannot be updated)

Methods in trait `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.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.

...more methods in trait `Iterable[A]`

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.

Methods in trait 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 trait `mutable.Buffer[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 += x`	Insert x at end. Return type Unit.
`xs.prepend(x)` `x +=: xs`	Insert x in front. Return type Unit.
`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.

Methods in trait `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: &~

Additional mutation methods in trait `mutable.Set[A]`

`xs += x` `xs -= x`	Returns the same set with included/excluded elements.
`xs ++= ys` `xs.addAll(ys)`	Adds all elements in ys to set xs and returns xs itself.
`xs.add(x)` `xs.remove(x)`	Adds/removes x to xs and returns true if xs was mutated, else false.
`xs(x) = b` `xs.update(x, b)`	If b is true, adds x to xs, else removes x. Return type Unit.

Methods in trait `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.

Additional mutation methods in trait `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 by applying f: (K, V) => V to each pair.

Factory examples:

On mutable Set, Map: `toSet`, `toMap` returns immutable; `Vector(0,0,0)` same as `Vector.fill(3)(0)`;
`collection.mutable.Set.empty[Int]` same as `collection.mutable.Set[Int]()`
`Map("se" -> "Sweden","nk" -> "Norway")` same as `Map(("se","Sweden"),("nk","Norway"))`
`Array.ofDim[Int](3,2)` gives `Array(Array(0, 0), Array(0, 0), Array(0, 0))` same as
`Array.fill(3,2)(0)`; `Vector.iterate(1.2, 3)(_ + 0.5)` gives `Vector(1.2, 1.7, 2.2)`
`Vector.tabulate(3)("s" + _)` gives `Vector("s0", "s1", "s2")`

Strings

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 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.

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

scala.jdk.CollectionConverters

Enable `.asJava` and `.asScala` conversions: `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`

Reserved words

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`

`_   :   =   =>   <-   <:   <%   >:   #   @`

Java snabbreferens @ LTH

Vertikalstreck | används mellan olika alternativ. Parenteser ( ) används för att gruppera en mängd alternativ. Hakparenteser [ ] markerar valfria delar. En sats betecknas `stmt` medan `x`, `i`, `s`, `ch` är variabler, `expr` är ett uttryck, `cond` är ett logiskt uttryck. Med `...` avses valfri, extra kod.

Satser

Block	`{stmt1; stmt2; ...}`	fungerar ”utifrån” som **en** sats
Tilldelning	`x = expr;`	variabeln och uttrycket av kompatibel typ

Förkortade	`x += expr;`
	`x++;`
		x = x + expr; även −=, *=, /=
		x = x + 1; även x − −

if-sats	`if (cond) {stmt; ...}` [`else { stmt; ...}`]	utförs om cond är true utförs om false

switch-sats	`switch (expr) {`
	`    case A: stmt1; break;`
	`    ...`
	`    default: stmtN; break;`
	`}`
		expr är ett heltalsuttryck
		utförs om expr = A (A konstant)
		”faller igenom” om break saknas
		sats efter default: utförs om inget case passar

for-sats	`for (int i = a; i < b; i++) {`
	`    stmt; ...`
	`}`
		satserna görs för i = a, a+1, ..., b-1
		Görs ingen gång om a >= b
		i++ kan ersättas med i = i + step

for-each-sats	`for (int x: xs) {`
	`    stmt; ...`
	`}`
		xs är en samling, här med heltal
		x blir ett element i taget ur xs
		fungerar även med array

while-sats	`while (cond) {stmt; ...}`	utförs så länge cond är true

do-while-sats	`do {`
	`    stmt; ...`
	`} while (cond);`
		utförs minst en gång,
		så länge cond är true

return-sats	`return expr;`	returnerar funktionsresultat

Uttryck

Aritmetiskt uttryck	(x + 2) * i / 2 + i % 2	för heltal är / heltalsdivision, % ”rest”
Objektuttryck	new Classname(...) | ref-var | null | function-call | this | super	
Logiskt uttryck	! cond | cond && cond | cond || cond | relationsuttryck | true | false	
Relationsuttryck	expr (< | <= | == | >= | > | !=) expr	för objektuttryck bara == och !=, också typtest med expr instanceof Classname

Funktionsanrop	obj-expr.method(...)
	Classname.method(...)
		anropa ”vanlig metod” (utför operation)
		anropa statisk metod

Array	new int[size] vname[i] vname.length	skapar int-array med size element elementet med index i, 0..length−1 antalet element

Matris	new int[r][c]
	m.length
	m[i].length
		//Skapar matris med r rader och c kolonner
		//Ger matrisens längd (d.v.s. antalet rader)
		//Ger antalet element (längden) på raden i

Typkonvertering	(newtype) expr (int) real-expr (Square) aShape	konverterar expr till typen newtype – avkortar genom att stryka decimaler – ger ClassCastException om aShape inte är ett Square-objekt

Deklarationer

Allmänt	[<protection>] [static] [final] <type> name1, name2, …;	
<type>	byte | short | int | long | float | double | boolean | char | Classname	
<protection>	public | private | protected	för attribut och metoder i klasser (paketskydd om inget anges)
Startvärde	int x = 5;	startvärde bör alltid anges
Konstant	final int N = 20;	konstantnamn med stora bokstäver
Array	<type>[ ] vname = new <type>[10];	deklarerar och skapar array
Matris	<type>[ ][ ] m = new <type>[4][5];	// deklarerar och skapar 4x5 matrisen m

Klasser

Deklaration

```
[ public ] [ abstract ] class Classname
    [ extends Classname1 ] [ implements Interface1, Interface2, … ] {
    <deklaration av attribut>
    <deklaration av konstruktorer>
    <deklaration av metoder>
}
```

Attribut — Som vanliga deklarationer. Attribut får implicita startvärden, 0, 0.0, false, null.

Konstruktor	<prot> Classname(param, …) { stmt; … }	Parametrarna är de parametrar som ges vid new Classname(…). Satserna ska ge attributen startvärden
Metod	<prot> <type> name(param, …) { stmt; … }	om typen inte är void måste en return-sats exekveras i metoden
Huvudprogram	public static void main(String[ ] args) { …}	
Abstrakt metod	Som vanlig metod, men abstract före typnamnet och {. . .} ersätts med semikolon. Metoden måste implementeras i subklasserna.	

Standardklasser, java.lang, behöver inte importeras

Object	Superklass till alla klasser.	
	boolean equals(Object other);	ger true om objektet är lika med other
	int hashCode();	ger objektets hashkod
	String toString();	ger en läsbar representation av objektet
Math	Statiska konstanter Math.PI och Math.E. Metoderna är statiska (anropas med t ex Math.round(x)):	
	long round(double x);	avrundning, även float $\rightarrow$ int
	int abs(int x);	$\|x\|$, även double, …
	double hypot(double x, double y);	$\sqrt{x^2+y^2}$
	double sin(double x);	$\sin x$, liknande: cos, tan, asin, acos, atan
	double exp(double x);	e^x
	double pow(double x, double y);	x^y
	double log(double x);	$\ln x$
	double sqrt(double x);	$\sqrt{x}$
	double toRadians(double deg);	$deg \cdot \pi/180$
System	void System.out.print(String s);	skriv ut strängen s
	void System.out.println(String s);	som print men avsluta med ny rad
	void System.exit(int status);	avsluta exekveringen, status != 0 om fel
	Parametern till print och println kan vara av godtycklig typ: int, double, …	

Wrapperklasser

För varje datatyp finns en wrapperklass: char → Character, int → Integer, double → Double, … Statiska konstanter MIN_VALUE och MAX_VALUE i klassen Integer ger minsta respektive största heltalsvärde. För klassen Double ger MIN_VALUE minsta flyttalet som är större än noll. Exempel med klassen Integer:

Integer(int value);	skapar ett objekt som innehåller value
int intValue();	tar reda på värdet

String

Teckensträngar där tecknen inte kan ändras. ”asdf” är ett String-objekt. s1 + s2 för att konkatenera två strängar. StringIndexOutOfBoundsException om någon position är fel.

int length();	antalet tecken
char charAt(int i);	tecknet på plats i, 0..length()-1
boolean equals(String s);	jämför innehållet (s1 == s2 fungerar inte)
int compareTo(String s);	< 0 om mindre, = 0 om lika, > 0 om större
int indexOf(char ch);	index för ch, -1 om inte finns
int indexOf(char ch, int from);	som indexOf men börjar leta på plats from
String substring(int first, int last);	kopia av tecknen first..last-1
String[ ] split(String delim);	ger array med ”ord” (ord är följder av tecken åtskilda med tecknen i delim)

Konvertering mellan standardtyp och String (exempel med int, liknande för andra typer):

String.valueOf(int x);	x = 1234 → ”1234”
Integer.parseInt(String s);	s = ”1234” → 1234, NumberFormat-Exception om s innehåller felaktiga tecken

StringBuilder

Modifierbara teckensträngar. length och charAt som String, plus:

StringBuilder(String s);	StringBuilder med samma innehåll som s
void setCharAt(int i, char ch);	ändrar tecknet på plats i till ch
StringBuilder append(String s);	lägger till s, även andra typer: int, char, …
StringBuilder insert(int i, String s);	lägger in s med början på plats i
StringBuilder deleteCharAt(int i);	tar bort tecknet på plats i
String toString();	skapar kopia som String-objekt

Standardklasser, import java.util.Classname

List

List<E> är ett gränssnitt som beskriver listor med objekt av parameterklassen E. Man kan lägga in värden av standardtyperna genom att kapsla in dem, till exempel int i Integer-objekt. Gränssnittet implementeras av klasserna ArrayList<E> och LinkedList<E>, som har samma operationer. Man ska inte använda operationerna som har en position som parameter på en LinkedList (i stället en iterator). IndexOutOfBoundsException om någon position är fel.

För att operationerna contains, indexOf och remove(Object) ska fungera måste klassen E överskugga funktionen equals(Object). Integer och de andra wrapperklasserna gör det.

ArrayList	ArrayList<E>();	skapar tom lista
LinkedList	LinkedList<E>();	skapar tom lista
	int size();	antalet element
	boolean isEmpty();	ger true om listan är tom
	E get(int i);	tar reda på elementet på plats i
	int indexOf(Object obj);	index för obj, -1 om inte finns
	boolean contains(Object obj);	ger true om obj finns i listan
	void add(E obj);	lägger in obj sist, efter existerande element
	void add(int i, E obj);	lägger in obj på plats i (efterföljande element flyttas)
	E set(int i, E obj);	ersätter elementet på plats i med obj
	E remove(int i);	tar bort elementet på plats i (efterföljande element flyttas)
	boolean remove(Object obj);	tar bort objektet obj, om det finns
	void clear();	tar bort alla element i listan

Random	Random();	skapar ”slumpmässig” slumptalsgenerator
	Random(long seed);	– med bestämt slumptalsfrö
	int nextInt(int n);	heltal i intervallet [0, n)
	double nextDouble();	double-tal i intervallet [0.0, 1.0)
Scanner	Scanner(File f);	läser från filen f, ofta System.in
	Scanner(String s);	läser från strängen s
	String next();	läser nästa sträng fram till whitespace
	boolean hasNext();	ger true om det finns mer att läsa
	int nextInt();	nästa heltal; också nextDouble(), …
	boolean hasNextInt();	också hasNextDouble(), …
	String nextLine();	läser resten av raden

Filer, import java.io.File/FileNotFoundException/PrintWriter

Läsa från fil	Skapa en Scanner med new Scanner(new File(filename)). Ger FileNotFoundException om filen inte finns. Sedan läser man ”som vanligt” från scannern (nextInt och liknande).
Skriva till fil	Skapa en PrintWriter med new PrintWriter(new File(filename)). Ger FileNotFoundException om filen inte kan skapas. Sedan skriver man ”som vanligt” på PrintWriter-objektet (println och liknande).
Fånga undantag	Så här gör man för att fånga FileNotFoundException:

```
Scanner scan = null;
try {
    scan = new Scanner(new File("indata.txt"));
} catch (FileNotFoundException e) {
    … ta hand om felet
}
```

Specialtecken

Några tecken måste skrivas på ett speciellt sätt när de används i teckenkonstanter:

\n	ny rad, radframmatningstecken
\t	ny kolumn, tabulatortecken (eng. tab)
\\	bakåtsnedstreck: \ (eng. backslash)
\”	citationstecken: ”
\’	apostrof: ’

Reserverade ord

Nedan 50 ord kan ej användas som identifierare i Java. Orden **`goto`** och **`const`** är reserverade men används ej.

```
abstract assert boolean break byte case catch char class const
continue default do double else enum extends final finally float for
goto if implements import instanceof int interface long native new
package private protected public return short static strictfp super
switch synchronized this throw throws transient try void volatile while
```