

Scala Quick Ref @ Lund University

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

License: CC-BY-SA, © Dept. of Computer Science, Lund University.

Pull requests welcome! Contact: bjorn.regnell@cs.lth.se

Top-level definitions

```
// in file: hello.scala
package x.y.z
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hi " + args.mkString(" "))
  }
}
```

A compilation unit (here hello.scala) consists of a sequence of packagings, import clauses, and class and object definitions, which may be preceded by a package clause, e.g.: **package** x.y.z that places the compiled file HelloWorld.class in directory x/y/z/

Compile: scalac hello.scala

Run: scala x.y.z.HelloWorld args
Execution starts in method main.

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.

Variable	val x = expr val x: Int = 0 var x = expr val x, y = expr val (x, y) = (e1, e2) val Seq(x, y) = Seq(e1, e2) val x: Int = _	Variable x is assigned to expr. A val can only be assigned once . Explicit type annotation, expr: SomeType allowed after any expr. Variable x is assigned to expr. A var can be re-assigned . Multiple initialisations, x and y is initialised to the same value. Tuple pattern initialisation, x is assigned to e1 and y to e2. Sequence pattern initialisation, x is assigned to e1 and y to e2. Initialized to default value, 0 for number types, null for AnyRef types.
Function	def f(a: Int, b: Int): Int = a + b def f(a: Int = 0, b: Int = 0): Int = a + b f(b = 1, a = 3) def add(a: Int)(b: Int): Int = a + b (a: Int, b: Int) => a + b val g: (Int, Int) => Int = (a, b) => a + b f _ val inc = add(1) _ def addAll(xs: Int*) = xs.sum def twice(block: => Unit) = { block; block }	Function f of type (Int, Int) => Int Default arguments used if args omitted, f(). Named arguments can be used in any order. Multiple parameter lists, apply: add(1)(2) Anonymous function value, "lambda". Types can be omitted in lambda if inferable. Replacing a parameter list with a space and underscore gives the function itself as a value. Partially applied function add(1) of add above, where inc is of type Int => Int Repeated parameters: addAll(1,2,3) or addAll(Seq(1,2,3): _*) Call-by-name argument evaluated later.
Object	object Name { defsAndDecl }	Singleton object auto-allocated when referenced the first time.
Class	class C(parameters) { defsAndDecl } case class C(parameters) { defsAndDecl }	A template for objects, which are allocated with new . Case class parameters become val members, other case class goodies: equals, copy, hashCode, unapply, nice toString, companion object with apply factory.
Trait	trait T { defsAndDecl } class C extends D with T	A trait is an abstract class without parameters. Can be used as an interface. A class can only extend one normal class but mix in many traits using with .
Type	type A = typeDef	Defines an alias A for the type in typeDef. Abstract if no typeDef.
Import	import path.to.module.name import path.to.{a, b => x, c => _}	Makes name directly visible. Underscore imports all. Import several names, b renamed to x, c not imported.

Modifier	applies to	semantics
private [this]	definitions, declarations	Restricts access to this instance only; also private[p] for package p.
private	definitions, declarations	Restricts access to directly enclosing class and its companion.
protected	definitions	Restricts access to subtypes and companion.
override	definitions, declarations	Mandatory if overriding a concrete definition in a parent class.
abstract	class definitions	Abstract classes cannot be instantiated (redundant for traits).
final	definitions	Final members cannot be overridden, final classes cannot be extended.
lazy	val definitions	Delays initialization of val, initialized when first referenced.
sealed	class definitions	Restricts direct inheritance to classes in the same source file.

Special methods

```
class A(initX: Int = 0) {
  private var _x = initX
  def x: Int = _x
  def x_=(i: Int): Unit = { _x = i }
}

object A {
  def apply(i: Int = 0) = new A(i)
  val a = A(1)._x
}
```

primary constructor: new A(1) or using default arg: new A()
 private member only visible in A and its companion
getter for private field _x (name chosen to avoid clash with x)
 special **setter** assignment syntax: val a = new A(1); a.x = 2

companion object if same name and in same code file
factory method makes new unnecessary: A.apply(1), A(1), A()
 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 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
block	{ expr1; ...; exprN }
if	if (cond) expr1 else expr2
match	expr match caseClauses
for	for (x <- xs) expr
yield	for (x <- xs) yield expr
while	while (cond) expr
do while	do expr while (cond)
throw	throw new Exception("Bang!")
try	try expr catch pf

Basic types e.g. Int, Long, Double, String, Char, Boolean
 The value of a block is the value of its last expression
 Value is expr1 if cond is true, expr2 if false (else is optional)
 Matches expr against each case clause, see pattern matching.
 Loop for each x in xs, x visible in expr, type Unit
 Yields a sequence with elems of expr for each x in xs
 Loop expr while cond is true, type Unit
 Do expr at least once, then loop while cond is true, type Unit
 Throws an exception that halts execution if not in try catch
 Evaluate partial function pf if exception in expr, where pf e.g.:
 {case e: Exception => someBackupValue}

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)	from class C with arguments 1,2
Self reference	this	refers to the object being defined
Supertype reference	super.m	refers to member m of supertype
Non-referable reference	null	refers to null object of type Null
Assignment operator	x += 1	expanded to x = x + 1
	x -= 1	works for any op ending with =
Empty tuple, unit value	()	of type Unit, similar to Java void
	x -= 1	works for any op ending with =
2-tuple value	(1, "hello")	same as new Tuple2(1, "hello")
2-tuple type	(Int, String)	same as Tuple2[Int, String] etc. until Tuple22

Precedence of operators beginning with:

all letters **lowest**

|

^

&

= !

< >

:

+ -

* / %

other special chars **highest**

Integer division and remainder:

a / b no decimals if a, b Int, Short, Byte

a % b fulfills: (a / b) * b + (a % b) == a

Pattern matching, type tests and extractors

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 binders** 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 if exhaustive by the compiler

Matching with typed variable pattern x **match** { **case** a: Int => a; **case** _ => 0 } is preferred over explicit `InstanceOf` tests and casts: **if** (x.isInstanceOf[Int]) x.asInstanceOf[Int] **else** 0

The **unapply** method can be used in **extractor** pattern matching (to avoid extra class & instance), e.g.:

```
object Host {
  def unapply(s: String): Option[String] =
    if (!s.startsWith("http://")) None
    else s.stripPrefix("http://").split('/').headOption
}
str match { case Host(name) => ... }
```

Extractor object
 extractor must return **Option**
None gives no match in patterns
Some(x) matches in patterns

Extractor pattern leads to a call to Host.unapply(str)

Generic classes and methods

```
class Box[T](val x: T){
  def pairedWith[U](y: U): (T, U) = (x, y)
}
val b = new Box(0)
val p = b.pairedWith(new Box("zero"))
```

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)
 the type of p is (Box[Int], Box[String])

Generic types are erased before JVM runtime except for Array, so a `reflect.ClassTag` is needed when constructing arrays from generic type parameters: **def** mkArray[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) 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: Throwable => expr2 } expr2 if exception else Success(expr1)
 Try(1/0) match {case Success(x) => x; case Failure(e) => 0} e here ArithmeticException

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

Read string of lines from **file** (`fromFile` gives `BufferedSource`, `getLines` gives `Iterator[String]`; also from URL):

```
val s = scala.io.Source.fromFile("f.txt", "UTF-8").getLines.mkString("\n")
```

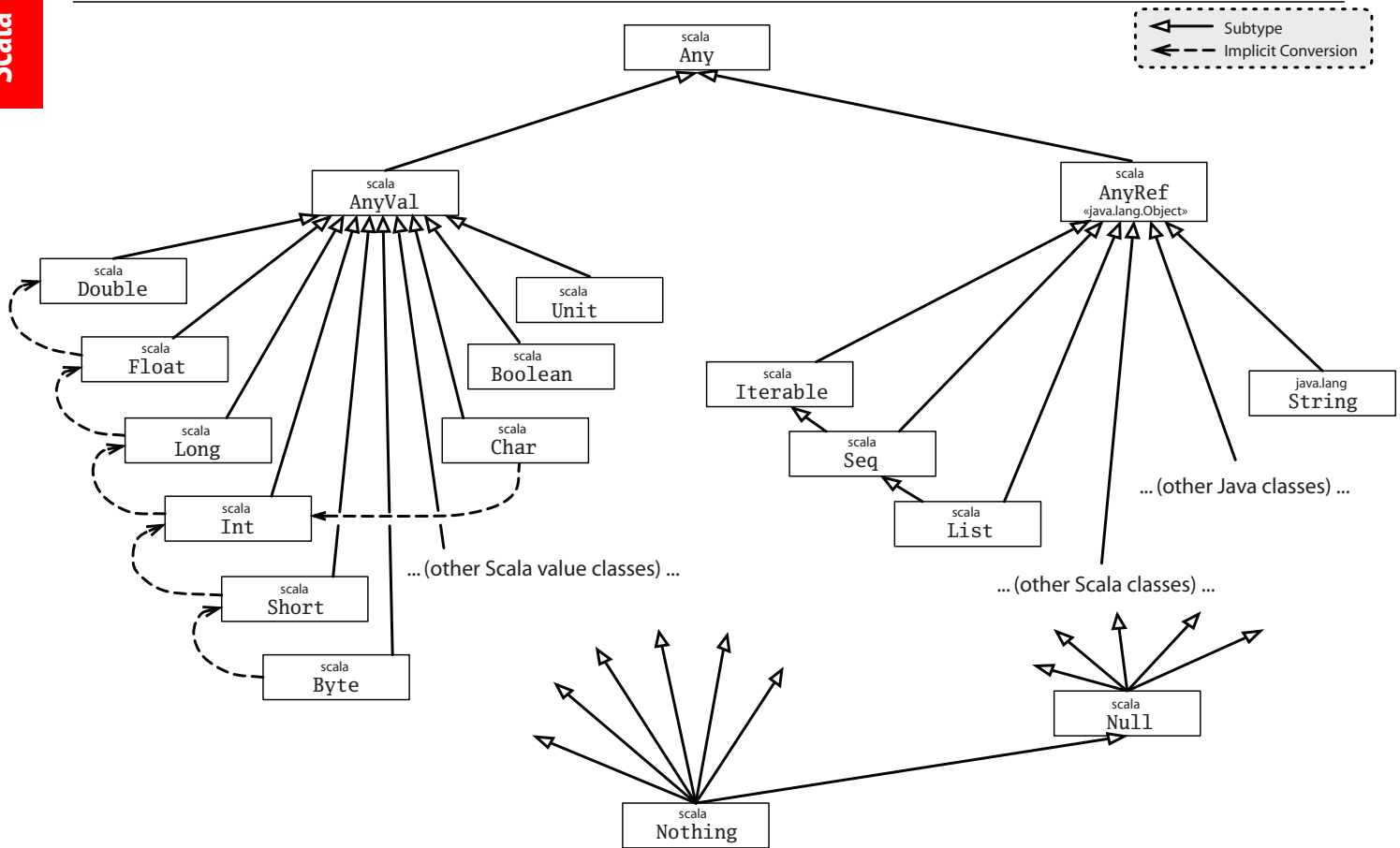
Read string from **standard in** (prompt string is optional) using `readLine`; **write** to **standard out** using `println`:

```
val s = scala.io.StdIn.readLine("prompt"); println("you wrote" + s)
```

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	$-2^7 \dots 2^7 - 1$	<code>0.toByte</code>
Short	16	$-2^{15} \dots 2^{15} - 1$	<code>0.toShort</code>
Char	16	$0 \dots 2^{16} - 1$	<code>'0'</code> <code>'\u0030'</code>
Int	32	$-2^{15} \dots 2^{15} - 1$	<code>0</code> <code>0xF</code>
Long	64	$-2^{15} \dots 2^{15} - 1$	<code>0L</code>
Float	32	$\pm 3.4 \cdot 10^{38}$	<code>0F</code>
Double	64	$\pm 1.8 \cdot 10^{308}$	<code>0.0</code>

Methods on numbers

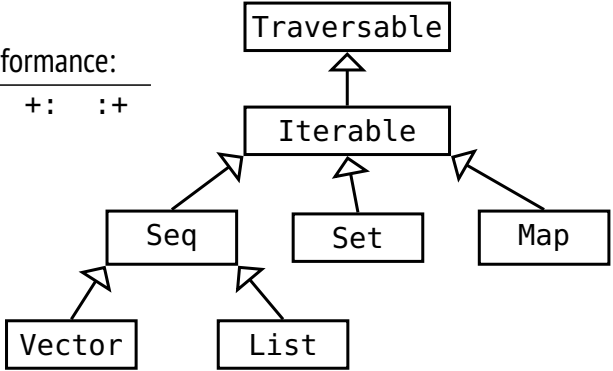
<code>x.abs</code>	<code>math.abs(x)</code> , absolute value
<code>x.round</code>	<code>math.round(x)</code> , to nearest Long
<code>x.floor</code>	<code>math.floor(x)</code> , cut decimals
<code>x.ceil</code>	<code>math.ceil(x)</code> , round up cut decimal
<code>x max y</code>	<code>math.max(x, y)</code> , gives largest, also min
<code>x.toInt</code>	also <code>toByte</code> , <code>toChar</code> , <code>toDouble</code> etc.
<code>1 to 4</code>	<code>Range(1, 2, 3, 4)</code>
<code>0 until 4</code>	<code>Range(0, 1, 2, 3)</code>

The Scala Standard Collection Library

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

String and Array are implicitly converted to Seq making sequence methods work as for other sequences.

Allocate array of Int of size n: `new Array[Int](n)`



Concrete implementations of **Set** include `HashSet`, `ListSet` and `BitSet`; `collection.SortedSet` is implemented by `TreeSet`. Concrete implementations of **Map** include `HashMap` and `ListMap`; `collection.SortedMap` is implemented by `TreeMap`.

Methods in trait Traversable[A]

What	Usage	Explanation <i>f</i> is a function, <i>pf</i> is a partial funct., <i>p</i> is a predicate.
Traverse:	<code>xs foreach f</code>	Executes <i>f</i> for every element of <i>xs</i> . Return type Unit.
Add:	<code>xs ++ ys</code>	A collection with <i>xs</i> followed by <i>ys</i> .
Map:	<code>xs map f</code>	A collection formed by applying <i>f</i> to every element in <i>xs</i> .
	<code>xs flatMap f</code>	A collection obtained by applying <i>f</i> (which must return a collection) to all elements in <i>xs</i> and concatenating the results.
	<code>xs collect pf</code>	The collection obtained by applying the <i>pf</i> to every element in <i>xs</i> for which it is defined (undefined ignored).
Convert:	<code>toVector toList toSeq toBuffer toArray</code>	Converts a collection. Unchanged if the run-time type already matches the demanded type.
	<code>toSet</code>	Converts the collection to a set; duplicates removed.
	<code>toMap</code>	Converts a collection of key/value pairs to a map.
Copy:	<code>xs copyToBuffer buf</code>	Copies all elements of <i>xs</i> to buffer <i>buf</i> . Return type Unit.
	<code>xs copyToArray (arr, s, n)</code>	Copies at most <i>n</i> elements of the collection to array <i>arr</i> starting at index <i>s</i> (last two arguments are optional). Return type Unit.
Size info:	<code>xs.isEmpty</code>	Returns true if the collection <i>xs</i> is empty.
	<code>xs.nonEmpty</code>	Returns true if the collection <i>xs</i> has at least one element.
	<code>xs.size</code>	Returns an Int with the number of elements in <i>xs</i> .
Retrieval:	<code>xs.head xs.last</code>	The first/last element of <i>xs</i> (or some elem, if order undefined).
	<code>xs.headOption xs.lastOption</code>	The first/last element of <i>xs</i> (or some element, if no order is defined) in an option value, or None if <i>xs</i> is empty.
	<code>xs find p</code>	An option with the first element satisfying <i>p</i> , or None.
Subparts:	<code>xs.tail xs.init</code>	The rest of the collection except <i>xs.head</i> or <i>xs.last</i> .
	<code>xs slice (from, to)</code>	The elements in from index <i>from</i> until (not including) <i>to</i> .
	<code>xs take n</code>	The first <i>n</i> elements (or some <i>n</i> elements, if order undefined).
	<code>xs drop n</code>	The rest of the collection except <i>xs take n</i> .
	<code>xs takeWhile p</code>	The longest prefix of elements all satisfying <i>p</i> .
	<code>xs dropWhile p</code>	Without the longest prefix of elements that all satisfy <i>p</i> .
	<code>xs filter p</code>	Those elements of <i>xs</i> that satisfy the predicate <i>p</i> .
	<code>xs filterNot p</code>	Those elements of <i>xs</i> that do not satisfy the predicate <i>p</i> .
	<code>xs splitAt n</code>	Split <i>xs</i> at <i>n</i> returning the pair (<i>xs take n</i> , <i>xs drop n</i>).
	<code>xs span p</code>	Split <i>xs</i> by <i>p</i> into the pair (<i>xs takeWhile p</i> , <i>xs.dropWhile p</i>).
	<code>xs partition p</code>	Split <i>xs</i> by <i>p</i> into the pair (<i>xs filter p</i> , <i>xs.filterNot p</i>)
	<code>xs groupBy f</code>	Partition <i>xs</i> into a map of collections according to <i>f</i> .
Conditions:	<code>xs forall p</code>	Returns true if <i>p</i> holds for all elements of <i>xs</i> .
	<code>xs exists p</code>	Returns true if <i>p</i> holds for some element of <i>xs</i> .
	<code>xs count p</code>	An Int with the number of elements in <i>xs</i> that satisfy <i>p</i> .
Folds:	<code>xs.foldLeft(z)(op)</code> <code>xs.foldRight(z)(op)</code>	Apply binary operation <i>op</i> between successive elements of <i>xs</i> , going left to right (or right to left) starting with <i>z</i> .
	<code>xs.reduceLeft op</code> <code>xs.reduceRight op</code>	Similar to <i>foldLeft/foldRight</i> , but <i>xs</i> must be non-empty, starting with first element instead of <i>z</i> .
	<code>xs.sum xs.product</code> <code>xs.min xs.max</code>	Calculation of the sum/product/min/max of the elements of <i>xs</i> , which must be numeric.
Make string:	<code>xs mkString (start, sep, end)</code>	A string with all elements of <i>xs</i> between separators <i>sep</i> enclosed in strings <i>start</i> and <i>end</i> ; <i>start</i> , <i>sep</i> , <i>end</i> are all optional.

Methods in trait Iterable[A]

What	Usage	Explanation
Iterators:	<code>val it = xs.iterator</code>	An iterator <code>it</code> of type <code>Iterator</code> that yields each element one by one: <code>while (it.hasNext) f(it.next)</code>
	<code>xs grouped size</code>	An iterator yielding fixed-sized chunks of this collection.
	<code>xs sliding size</code>	An iterator yielding a sliding fixed-sized window of elements.
Subparts:	<code>xs takeRight n</code> <code>xs dropRight n</code>	Similar to <code>take</code> and <code>drop</code> in <code>Traversable</code> but takes/drops the last <code>n</code> elements (or any <code>n</code> elements if the order is undefined).
	<code>xs zip ys</code> <code>xs zipAll (ys, x, y)</code>	An iterable of pairs of corresponding elements from <code>xs</code> and <code>ys</code> . Similar to <code>zip</code> , but the shorter sequence is extended to match the longer one by appending elements <code>x</code> or <code>y</code> .
Zippers:	<code>xs.zipWithIndex</code>	An iterable of pairs of elements from <code>xs</code> with their indices.
	<code>xs sameElements ys</code>	True if <code>xs</code> and <code>ys</code> contain the same elements in the same order.

Methods in trait Seq[A]

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

Methods in trait Set [A]

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

Additional mutation methods in trait mutable.Set [A]

<code>xs += x</code>	<code>xs -= x</code>	Returns the same set with included/excluded elements.
<code>xs += (x, y, z)</code>	<code>xs -= (x, y, z)</code>	Addition/subtraction can be applied to many arguments.
<code>xs ++= ys</code>		Adds all elements in ys to set xs and returns xs itself.
<code>xs add x</code>		Adds element x to xs and returns true if x was in xs, else false.
<code>xs remove x</code>		Removes x from xs and returns true if x was in xs, else false.
<code>xs retain p</code>		Keeps only those elements in xs that satisfy predicate p.
<code>xs.clear</code>		Removes all elements from xs. Return type Unit.
<code>xs(x) = b</code>	<code>xs.update(x, b)</code>	If b is true, adds x to xs, else removes x. Return type Unit.
<code>xs.clone</code>		Returns a new mutable set with the same elements as xs.

Methods in trait Map [K, V]

<code>ms get k</code>		The value associated with key k an option, None if not found.
<code>ms(k)</code>	<code>xs apply k</code>	The value associated with key k, or exception if not found.
<code>ms getOrElse (k, d)</code>		The value associated with key k in map ms, or d if not found.
<code>ms isDefinedAt k</code>		True if ms contains a mapping for key k. Also: ms.contains(k)
<code>ms + (k -> v)</code>	<code>ms + ((k, v))</code>	The map containing all mappings of ms as well as the mapping k -> v from key k to value v. Also: ms + (k -> v, l -> w)
<code>ms - k</code>		Excluding any mapping of key k. Also: ms - (k, l, m)
<code>ms ++ ks</code>	<code>ms -- ks</code>	The mappings of ms with the mappings of ks added/removed.
<code>ms.keys</code>	<code>ms.values</code>	An iterable containing each key/value in ms.

Additional mutation methods in trait mutable.Map [K, V]

<code>ms(k) = v</code>	<code>ms.update(k, v)</code>	Adds mapping k to v, overwriting any previous mapping of k.
<code>ms += (k -> v)</code>	<code>ms -= k</code>	Adds/Removes mappings. Also vid several arguments.
<code>ms put (k, v)</code>	<code>ms remove k</code>	Adds/removes mapping; returns previous value of k as an option.
<code>ms retain p</code>		Keeps only mappings that have a key satisfying predicate p.
<code>ms.clear</code>		Removes all mappings from ms.
<code>ms transform f</code>		Transforms all associated values in map ms with function f.
<code>ms.clone</code>		Returns a new mutable map with the same mappings as ms.

Factory examples:

```
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 implicitly treated as `Seq[Char]` so all `Seq` methods also works.

<code>s(i)</code>	<code>s</code> apply <code>i</code>	<code>s.charAt(i)</code>	Returns the character at index <code>i</code> .
<code>s.capitalize</code>			Returns this string with first character converted to upper case.
<code>s.compareTo(t)</code>			Returns <code>x</code> where <code>x < 0</code> if <code>s < t</code> , <code>x > 0</code> if <code>s > t</code> , <code>x</code> is 0 if <code>s == t</code>
<code>s.compareToIgnoreCase(t)</code>			Similar to <code>compareTo</code> but not sensitive to case.
<code>s.endsWith(t)</code>			True if string <code>s</code> ends with string <code>t</code> .
<code>s.replaceAllLiterally(s1, s2)</code>			Replace all occurrences of <code>s1</code> with <code>s2</code> in <code>s</code> .
<code>s.split(c)</code>			Returns an array of strings split at every occurrence of character <code>c</code> .
<code>s.startsWith(t)</code>			True if string <code>s</code> begins with string <code>t</code> .
<code>s.stripMargin</code>			Strips leading white space followed by <code> </code> from each line in string.
<code>s.substring(i)</code>			Returns a substring of <code>s</code> with all characters from index <code>i</code> .
<code>s.substring(i, j)</code>			Returns a substring of <code>s</code> from index <code>i</code> to index <code>j-1</code> .
<code>s.toInt</code> <code>s.toDouble</code> <code>s.toFloat</code>			Parses <code>s</code> as an <code>Int</code> or <code>Double</code> etc. May throw an exception.
<code>42.toString</code> <code>42.0.toString</code>			Converts a number to a <code>String</code> .
<code>s.toLowerCase</code>			Converts all characters to lower case.
<code>s.toUpperCase</code>			Converts all characters to upper case.
<code>s.trim</code>			Removes leading and trailing white space.

Escape	char	Special strings	
<code>\n</code>	line break	<code>"hello\nworld\t!"</code>	string including escape char for line break and tab
<code>\t</code>	horizontal tab	<code>"""a "raw" string"""</code>	can include quotes and span multiple lines
<code>\"</code>	double quote	<code>s"x is \$x"</code>	s interpolator inserts values of existing names
<code>\'</code>	single quote	<code>s"x+1 is \${x+1}"</code>	s interpolator evaluates expressions within <code>\${}</code>
<code>\\</code>	backslash	<code>f"\$x%5.2f"</code>	format <code>Double x</code> to 2 decimals at least 5 chars wide
<code>\u0041</code>	unicode for A	<code>f"\$y%5d"</code>	format <code>Int y</code> right justified at least five chars wide

scala.collection.JavaConverters

Enable `.asJava` and `.asScala` conversions: **import** `scala.collection.JavaConverters._`

<code>xs.asJava</code> on a Scala collection of type:		<code>xs.asScala</code> on a Java collection of type:
<code>Iterator</code>	\longleftrightarrow	<code>java.util.Iterator</code>
<code>Iterable</code>	\longleftrightarrow	<code>java.lang.Iterable</code>
<code>Iterable</code>	\leftarrow	<code>java.util.Collection</code>
<code>mutable.Buffer</code>	\longleftrightarrow	<code>java.util.List</code>
<code>mutable.Set</code>	\longleftrightarrow	<code>java.util.Set</code>
<code>mutable.Map</code>	\longleftrightarrow	<code>java.util.Map</code>
<code>mutable.ConcurrentMap</code>	\longleftrightarrow	<code>java.util.concurrent.ConcurrentMap</code>

Reserved words

These 40 words and 10 symbols have special meaning and cannot be used as identifiers in Scala.

**abstract case catch class def do else extends false final finally for
forSome if implicit import lazy macro match new null object override
package private protected return sealed super this throw trait try true
type val var while with yield _ : = => <- <: <% >: # @**

Java snabbreferens @ Lunds universitet

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	<code>{stmt1; stmt2; ...}</code>	fungerar "utifrån" som en sats
Tilldelning	<code>x = expr;</code>	variabeln och uttrycket av kompatibel typ
Förkortade	<code>x += expr;</code> <code>x++;</code>	<code>x = x + expr</code> ; även <code>--</code> , <code>*=</code> , <code>/=</code> <code>x = x + 1</code> ; även <code>x --</code>
if-sats	<code>if (cond) {stmt; ...}</code> <code>[else { stmt; ...}]</code>	utförs om <code>cond</code> är <code>true</code> utförs om <code>false</code>
switch-sats	<code>switch (expr) {</code> <code>case A: stmt1; break;</code> <code>...</code> <code>default: stmtN; break;</code> <code>}</code>	<code>expr</code> är ett heltalsuttryck utförs om <code>expr == A</code> (<code>A</code> konstant) "faller igenom" om <code>break</code> saknas sats efter <code>default</code> : utförs om inget case passar
for-sats	<code>for (int i = a; i < b; i++) {</code> <code>stmt; ...</code> <code>}</code>	satserna görs för <code>i = a, a+1, ..., b-1</code> Görs ingen gång om <code>a >= b</code> <code>i++</code> kan ersättas med <code>i = i + step</code>
for-each-sats	<code>for (int x: xs) {</code> <code>stmt; ...</code> <code>}</code>	<code>xs</code> är en samling, här med heltal <code>x</code> blir ett element i taget ur <code>xs</code> fungerar även med array
while-sats	<code>while (cond) {stmt; ...}</code>	utförs så länge <code>cond</code> är <code>true</code>
do-while-sats	<code>do {</code> <code>stmt; ...</code> <code>} while (cond);</code>	utförs minst en gång, så länge <code>cond</code> är <code>true</code>
return-sats	<code>return expr;</code>	returnerar funktionsresultat

Uttryck

Aritmetiskt uttryck	<code>(x + 2) * i / 2 + i % 2</code>	för heltal är <code>/</code> heltalsdivision, <code>%</code> "rest"
Objektuttryck	<code>new Classname(...)</code> <code>ref-var</code> <code>null</code> <code>function-call</code> <code>this</code> <code>super</code>	
Logiskt uttryck	<code>! cond</code> <code>cond && cond</code> <code>cond cond</code> <code>relationsuttryck</code> <code>true</code> <code>false</code>	
Relationsuttryck	<code>expr (< <= == >= > !=) expr</code>	för objektuttryck bara <code>==</code> och <code>!=</code> , också typtest med <code>expr instanceof Classname</code>
Funktionsanrop	<code>obj-expr.method(...)</code> <code>Classname.method(...)</code>	anropa "vanlig metod" (utför operation) anropa statisk metod
Array	<code>new int[size]</code> <code>vname[i]</code> <code>vname.length</code>	skapar <code>int</code> -array med <code>size</code> element elementet med index <code>i</code> , <code>0..length-1</code> antalet element
Typkonvertering	<code>(newtype) expr</code> <code>(int) real-expr</code> <code>(Square) aShape</code>	konverterar <code>expr</code> till typen <code>newtype</code> – avkortar genom att stryka decimaler – ger <code>ClassCastException</code> om <code>aShape</code> inte är ett <code>Square</code> -objekt

Deklarationer

Allmänt	<code>[<protection>] [static] [final] <type> name1, name2, ...;</code>	
<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

Klasser

Deklaration	<code>[public] [abstract] class Classname</code> <code>[extends Classname1] [implements Interface1, Interface2, ...] {</code> <deklaration av attribut> <deklaration av konstruktörer> <deklaration av metoder> }	
Attribut	Som vanliga deklARATIONER. Attribut får implicita startvärden, 0, 0.0, false, null.	
Konstruktör	<code><prot> Classname(param, ...) {</code> stmt; ... }	Parametrarna är de parametrar som ges vid new Classname(...). Satserna ska ge attributen startvärden
Metod	<code><prot> <type> name(param, ...) {</code> 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 subclasserna.	

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 → 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 ger minsta respektive största värde. Exempel med klassen Integer:
	Integer(int value); int intValue();
	skapar ett objekt som innehåller value 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(); char charAt(int i); boolean equals(String s); int compareTo(String s); int indexOf(char ch); int indexOf(char ch, int from); String substring(int first, int last); String[] split(String delim);
	antalet tecken tecknet på plats i, 0..length()—1 jämför innehållet (s1 == s2 fungerar inte) < 0 om mindre, = 0 om lika, > 0 om större index för ch, —1 om inte finns som indexOf men börjar leta på plats from kopia av tecknen first..last—1 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); Integer.parseInt(String s);
	x = 1234 → "1234" s = "1234" → 1234, NumberFormatException om s innehåller felaktiga tecken
StringBuilder	Modifierbara teckensträngar. length och charAt som String, plus:
	StringBuilder(String s); void setCharAt(int i, char ch); StringBuilder append(String s); StringBuilder insert(int i, String s); StringBuilder deleteCharAt(int i); String toString();
	StringBuilder med samma innehåll som s ändrar tecknet på plats i till ch lägger till s, även andra typer: int, char, ... lägger in s med början på plats i tar bort tecknet på plats i 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 över-skugga funktionen equals(Object). Integer och de andra wrapperklasserna gör det.
ArrayList	ArrayList<E>();
LinkedList	LinkedList<E>();
	int size(); boolean isEmpty(); E get(int i); int indexOf(Object obj); boolean contains(Object obj); void add(E obj); void add(int i, E obj);
	skapar tom lista skapar tom lista antalet element ger true om listan är tom tar reda på elementet på plats i index för obj, —1 om inte finns ger true om obj finns i listan lägger in obj sist, efter existerande element lägger in obj på plats i (efterföljande element flyttas)
	E set(int i, E obj); E remove(int i);
	ersätter elementet på plats i med obj tar bort elementet på plats i (efter- följande element flyttas)
	boolean remove(Object obj); void clear();
	tar bort objektet obj, om det finns tar bort alla element i listan

Random	Random(); Random(long seed); int nextInt(int n); double nextDouble();	skapar "slumpmässig" slumptalsgenerator – med bestämt slumptalsfrö heltal i intervallet [0, n) double-tal i intervallet [0.0, 1.0)
Scanner	Scanner(File f); Scanner(String s); String next(); boolean hasNext(); int nextInt(); boolean hasNextInt(); String nextLine();	läser från filen f, ofta System.in läser från strängen s läser nästa sträng fram till whitespace ger true om det finns mer att läsa nästa heltal; också nextDouble(), ... också hasNextDouble(), ... 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: <pre> Scanner scan = null; try { scan = new Scanner(new File("indata.txt")); } catch (FileNotFoundException e) { ... ta hand om felet } </pre>

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**