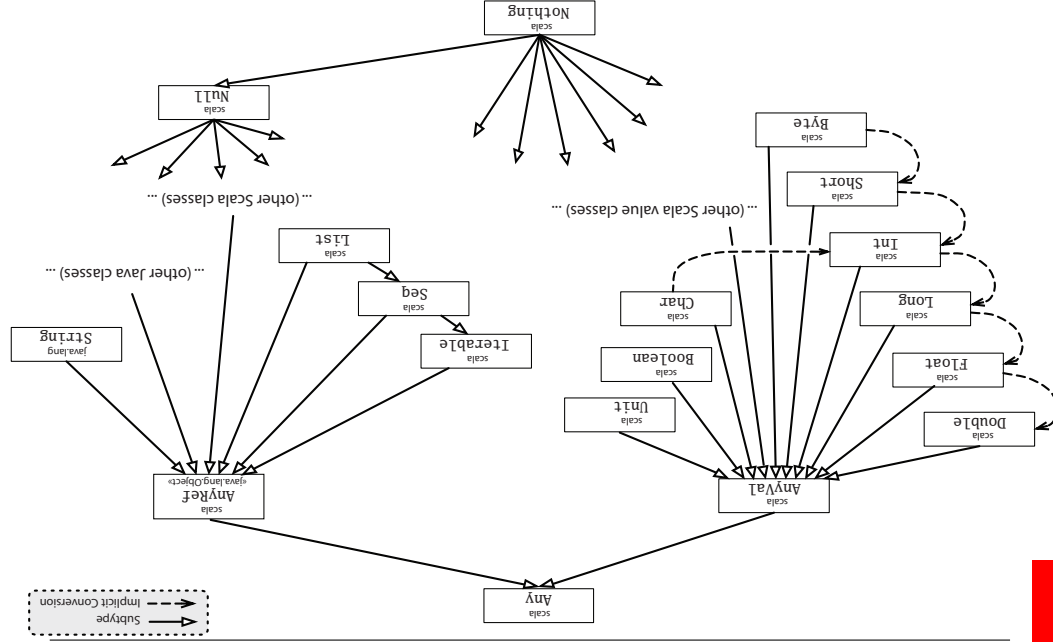


The Scala Type System

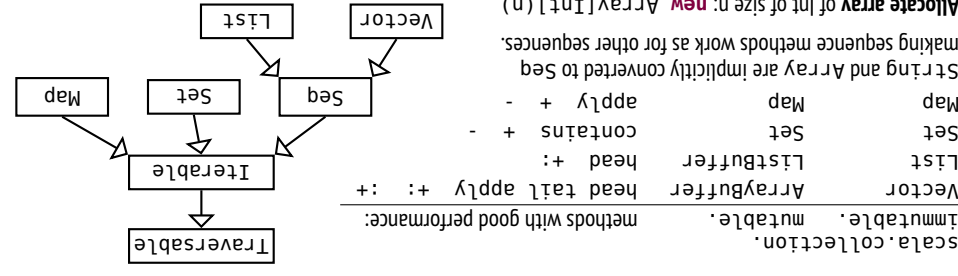


name	# bits	range	literal
Byte	8	$-2^7 \dots 2^7 - 1$	0.toByte ... 2 ¹⁵ - 1
Short	16	$-2^{15} \dots 2^{15} - 1$	0.toShort ... 2 ¹⁵ - 1
Char	16	$0 \dots 2^{16} - 1$	'\u0030' ... '\u0030'
Int	32	$-2^{31} \dots 2^{31} - 1$	0 ... 2 ¹⁵ - 1
Long	64	$-2^{63} \dots 2^{63} - 1$	0L ... 2 ¹⁵ - 1
Float	32	$\pm 3.4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

Methods on numbers

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

The Scala Standard Collection Library



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.

Top-level definitions

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

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 defAndDecl denotes a list of definitions and/or declarations.

Modifier	applies to	semantics
private [this]	definitions, declarations	Restricts access to this instance only; also private[p] for package p.
protected	definitions, declarations	Restricts access to directly enclosing class and its companion.
override	definitions, declarations	Restricts access to subtypes and companion.
abstract	definitions, declarations	Mandatory if overriding a concrete definition in a parent class.
final	definitions	Abstract classes cannot be instantiated (redundant for traits).
lazy	val definitions	Final members cannot be overridden, final classes cannot be extended.
sealed	class definitions	Delays initialization of val, initialized when first referenced.
		Restricts direct inheritance to classes in the same source file.
import	path.to.module.name	Makes name directly visible. Underscore imports all.
type	A = typedDef	Defines an alias A for the type in typedDef. Abstract if no typedDef.
trait	trait T { defAndDecl }	A trait is an abstract class without parameters. Can be used as an interface.
class	class C extends D with T	A class can only extend one normal class but mix in many traits using with.
Object	object Name { defAndDecl }	Singleton object auto-allocated when referenced the first time.
	def twice(block: => Unit) = { block; block }	Call-by-name argument evaluated later.
	def addAll(xs: Int*) = xs.sum	Repeated parameters: addAll(1,2,3) or addAll(Seq(1,2,3): _)
	val inc = add(1) -	Partially applied function add(1) of add above, where inc is of type Int => Int
	f -	Replacing a parameter list with a space and underscore gives the function itself as a value.
	val g: (Int, Int) => Int = (a, b) => a + b	Types can be omitted in lambda if inferable.
	(a: Int, b: Int) => a + b	Anonymous function value, "lambda".
	def add(a: Int)(b: Int): Int = a + b	Multiple parameter lists, apply: add(1)(2)
	f(b = 1, a = 3)	Named arguments can be used in any order.
	def f(a: Int = 0, b: Int = 0): Int = a + b	Default arguments used if args omitted, f()
	def f(a: Int, b: Int): Int = a + b	Function f of type (Int, Int) => Int
	val x: Int = -	Initialized to default value, 0 for number types, null for AnyRef types.
	val Seq(x, y) = Seq(e1, e2)	Sequence pattern initialization, x is assigned to e1 and y to e2.
	val (x, y) = expr	Tuple pattern initialization, x is assigned to e1 and y to e2.
	var x = expr	Multiple initializations, x and y is initialized to the same value.
	val x: Int = 0	Variable x is assigned to expr. A var can be re-assigned.
	val x = expr	Explicit type annotation, expr: Some type allowed after any expr.
	val x = expr	Variable x is assigned to expr. A val can only be assigned once.

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:
 With **val** in primary constructor only getter, no setter, is generated:
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	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) 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) 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) expr	Loop expr while cond is true, type Unit
do while	do expr while (cond)	Do expr at least once, then loop while cond is true, type Unit
throw	throw new Exception("Bang!")	Throws an exception that halts execution if not in try catch
try	try expr catch pf	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 reminder:

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 {
  case "hello" => expr
  case x: C => expr
  case C(x, y, z) => expr
  case (x, y, z) => expr
  case x +: xs => expr
  case p1 | ... | pN => expr
  case x@pattern => expr
  case x => expr
}
```

expr is matched against patterns from top until match found, yielding the expression after =>
literal pattern matches any value equal (in terms of ==) to the literal
typed variable pattern matches all instances of C, binding variable x to the instance
constructor pattern matches values of the form C(x, y, z), args bound to x,y,z
tuple pattern matches tuple values, alias for constructor pattern Tuple3(x, y, z)
sequence extractor patterns matches head and tail, also x +: y +: z +: xs etc.
 matches if at least one **pattern alternative** p1, p2 ... or pN matches
 a **pattern binders** with the @ sign binds a variable to (part of) a pattern
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 **isInstanceOf** 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 fromURL):

```
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))
```

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.

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.replaceAllLiterally(s1, s2)	Replace all occurrences of s1 with s2 in s.
s.split(c)	Returns an array of strings split at every occurrence of character c.
s.startsWith(t)	True if string s begins with string t.
s.stripMargin	Strips leading white space followed by l from each line in string.
s.substring(i)	Returns a substring of s with all characters from index i.
s.substring(i, j)	Returns a substring of s from index i to index j-1.
s.toInt s.toDouble s.toFloat	Parses s as an Int or Double etc. May throw an exception.
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	Special strings
\n line break	"hello\\world\\t"
\t horizontal tab	" "a "raw" string" " " " "
" double quote "	s "x is \$x"
' single quote '	s interpolator evaluates expressions within \${ }
\\ backslash \	format Double x to 2 decimals at least 5 chars wide f"\$x%5.2f"
\u0041 unicode for A	format Int y right justified at least five chars wide f"%5d"

scala.collection.JavaConverters

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

xs.asJava on a **Scala** 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 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 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 - _ : = == > < <: <= % >: # @

Methods in trait Traversable[A]

What	Usage	Explanation
Traverse:	xs foreach f	Executes f for every element of xs. Return type Unit.
Add:	xs ++ ys	A collection with xs followed by ys.
Map:	xs map f	A collection formed by applying f to every element in xs.
	xs flatMap f	A collection obtained by applying f (which must return a collection) to all elements in xs and concatenating the results.
	xs collect pf	The collection obtained by applying the pf to every element in xs for which it is defined (undefined ignored).
Convert:	toVector toList toSeq	Converts a collection. Unchanged if the run-time type already matches the demanded type.
	toBuffer toArray	Converts the collection to a set; duplicates removed.
	toMap	Converts a collection of key/value pairs to a map.
Copy:	xs copyToBuffer buf	Copies all elements of xs to buffer buf. Return type Unit.
	xs copyToArray (arr, s, n)	Copies at most n elements of the collection 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	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 f rom 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 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)	Apply binary operation op between successive elements of xs, going left to right (or right to left) starting with z.
	xs.foldRight(z) (op)	Similar to foldLeft/foldRight, but xs must be non-empty, starting with first element instead of z.
	xs.reduceLeft op	Calculation of the sum/product/min/max of the elements of xs, which must be numeric.
	xs.sum xs.product	xs.min xs.max
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 **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>	Similar to take and drop 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.dropRight n</code>	
Zippers:	<code>xs.zip ys</code>	An iterable of pairs of corresponding elements from <code>xs</code> and <code>ys</code> .
	<code>xs.zipAll (ys, x, y)</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> .
	<code>xs.zipWithIndex</code>	An iterable of pairs of elements from <code>xs</code> with their indices.
Compare:	<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</code> apply <code>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>		The (last) index of <code>xs</code> such that successive elements starting from that index form the sequence <code>ys</code> .
	<code>xs.lastIndexOfSlice 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> .
Add:	<code>xs.prepend x</code>	<code>xs</code> += <code>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>		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.update(i, x)</code>		
Sort:	<code>xs.sorted</code>		A new <code>Seq[A]</code> sorted using implicitly available ordering of <code>A</code> .
	<code>xs.sortWith lt</code>		A new <code>Seq[A]</code> sorted using less than <code>lt</code> : <code>(A, A) => Boolean</code> .
By:	<code>xs.sortBy f</code>		A new <code>Seq[A]</code> sorted/minimized/maximized by implicitly available ordering of <code>B</code> after applying <code>f</code> : <code>A => B</code> to each element.
	<code>xs.maxBy f</code> <code>xs.minBy f</code>		
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>
Subparts:	<code>(xs.corresponds ys)(p)</code>		True if corresponding elements satisfy the binary predicate <code>p</code> .
	<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</code> apply <code>x</code>	True if <code>x</code> is a member of <code>xs</code> . Also: <code>xs</code> contains <code>x</code>
<code>xs.subsetOf ys</code>		True if <code>ys</code> is a subset of <code>xs</code> .
<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 <code>xs</code> and <code>ys</code> . Also: <code>&</code>
<code>xs.union ys</code>		A new set with elements in either <code>xs</code> or <code>ys</code> or both. Also: <code> </code>
<code>xs.diff ys</code>		A new set with elements in <code>xs</code> that are not in <code>ys</code> . Also: <code>&~</code>

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

Methods in trait **Map[K, V]**

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

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

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

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")
```


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

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

Läs från fil	Skapa en Scanner med new Scanner(new File(filename)), Ger FileNotFoundException om filen inte finns. Sedan läser man "som vanligt" från scanneren (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 rad, radframtätningstecken ny kolumn, tabulatorstecken (eng. tab) \\ 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 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

Satser	Block {stmt1; stmt2; ...} Tildelning x = expr; Förkortade x += expr; x = x + 1; även x++, x = x + expr; även x -=, *=, /= if-sats if (cond) {stmt; ...} switch-sats switch (expr) { case A: stmt1; break; ... default: stmtN; break; } for-sats for (int i = a; i < b; i++) { ... } for-each-sats for (int x: xs) { ... } while-sats while (cond) {stmt; ...} do-while-sats do { ... } while (cond); return-sats return expr;	fungerar "utför" som en sats variabeln och uttrycket av kompatibel typ x = x + expr; även x++, /= x = x + 1; även x++ if (cond) {stmt; ...} [else {stmt; ...}] switch (expr) { case A: stmt1; break; ... default: stmtN; break; } for (int i = a; i < b; i++) { satserna görs för i = a, a+1, ..., b-1 Görs ingen gång om a >= b i++ kan ersättas med i = i + step xs är en samling, här med helta x blir ett element i taget ur xs fungerar även med array utförs så länge cond är true utförs minst en gång, så länge cond är true returnerar funktionsresultat
Utttryck	Aritmetiskt uttryck (x + 2) * ! / 2 + ! % 2 Objektuttryck new Classname(...) ref-var null function-call this super Logiskt uttryck i cond cond && cond cond cond relationsuttryck true false Relationsuttryck expr (< > <= == >= < > !=) expr Funktionsanrop obj-expr.method(...) Classname.method(...) anropa statisk metod Array new int[size] vname.length Typkonvertering (int) real-expr (newtype) expr (square) aShape	anropa "vanlig metod" (utför operation) anropa statisk metod skapar int-array med size element elementet med index i, 0.length—1 antalet element konverterar expr till typen newtype avkortar genom att stryka decimaler — ger ClassCastException om aShape inte är ett Square-objekt

Vertikalstreck | används mellan olika alternativa. 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.

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

Klasser

Deklaration	[public] [abstract] class Classname [extends Classname1] [implements Interface1, Interface2, ...] { <deklaration av attribut> <deklaration av konstruktorer> <deklaration av metoder> }	
Attribut	Som vanliga deklarerationer. Attribut får implicita startvärden, 0, 0.0, false, null.	
Konstruktör	<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 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); 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, NumberFormatException 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 över-skugga funktionen equals(Object). Integer och de andra wrapperklasserna gör det.	
ArrayList LinkedList	ArrayList<E>(); skapar tom lista 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 (efter-följande element flyttas) boolean remove(Object obj); tar bort objektet obj, om det finns void clear(); tar bort alla element i listan	