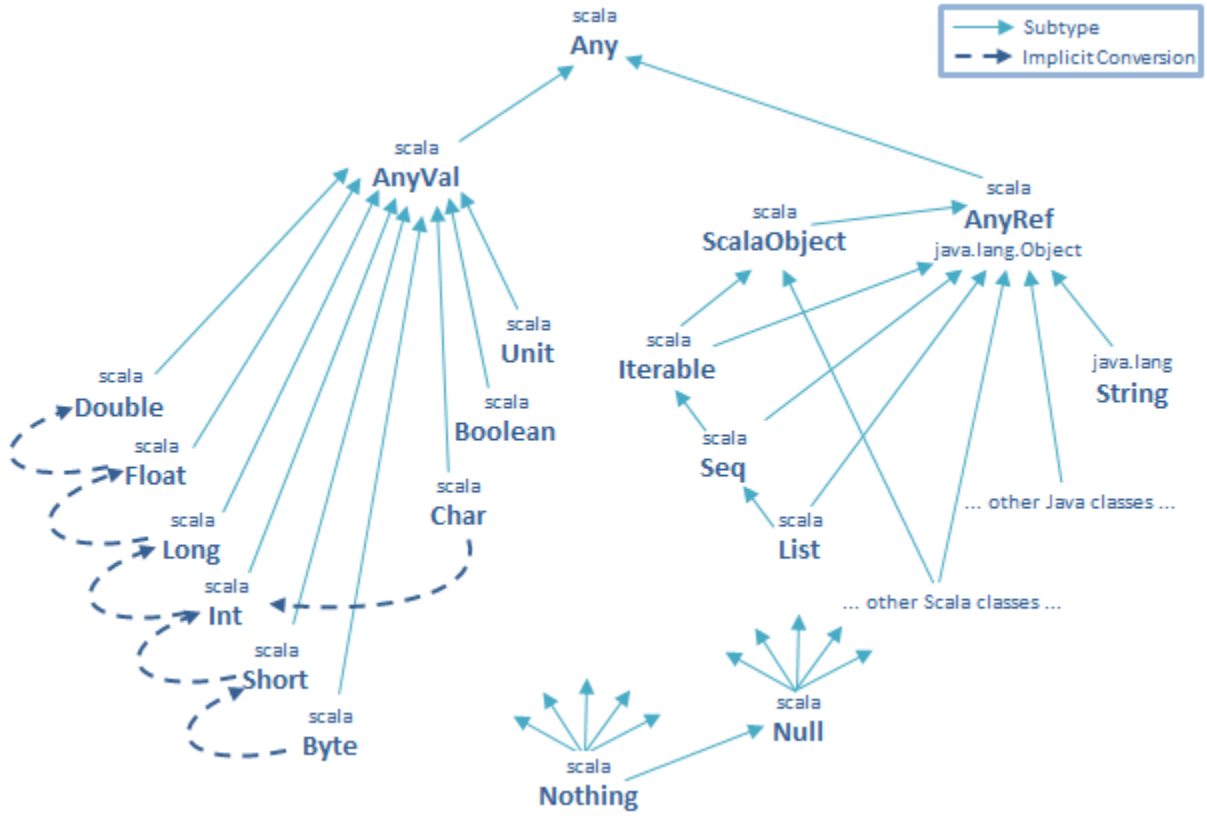
SCALA CHEATSHEET

Vinson Zhang

Scala Reference Book

Scala type hierarchy



Basic Types and Operations

127 - The convention is to include empty parentheses when invoking a method only if that method has side effects

- **Pure methods** are methods that don't have any side effects and don't depend on mutable state (226)
- if the function you're calling performs an operation, use the parentheses, but if it merely provides access to a property, leave out the parentheses

127 - Postfix operator: A method that takes no arguments can be called like this: `"some String" toLowerCase`

127 - Integral types: `Int`, `Long`, `Byte`, `Short`, `Char`

135 - Operator precedence:

`(all other special characters)`

`*` `/` `%`

`+` `-`

`:`

`=` `!`

`<` `>`

`&`

`^` - binary xor

`|`

`(all letters)`
`(all assignment operators)`

Operator precedence is based on the first character of the method used in operator notation, with one exception: If an operator ends with a `=`, and the operator is not one of the comparison operators `<=`, `>=`, `==`, or `!=`, then the precedence of the operator is the same as that of simple assignment `=`, which is last in the list. E.g. `+=`

136 - Associativity

- any method that ends in a `:` character is invoked on its right operand, passing in the left operand. Methods that end in any other character are invoked on their left operand, passing in the right operand. So `a * b` yields `a.*(b)`, but `a ::: b` yields `b.:::(a)`

137 - `a ::: b ::: c` is treated as `a ::: (b ::: c)` (list concatenation)

Functional Objects

141 - Class parameters: Any code placed in the class body (outside methods) will be placed in the ***primary constructor***. When declaring a class you can drop empty `{}`
143 - Precondition is a constraint on values passed into a method or constructor (E.g. `require(d != 0)` in the class body will throw `IllegalArgumentException: requirement failed` when `0` is passed as `d`)
144 - If **Class parameters** are only used inside constructors, the Scala compiler will not create corresponding fields for them
146 - Auxiliary constructors - constructors other than the primary constructor

- every ***auxiliary constructor*** must invoke another constructor **of the same class** (like Java, only Java can also call superclass's constructor instead) as its first action. That other constructor must textually come before the calling constructor

152 - The convention is to use camel case for constants, such as `XOffset`
153 - The Scala compiler will internally "mangle" operator identifiers to turn them into legal Java identifiers with embedded `$` characters. For instance, the identifier `:->` would be represented internally as `$colon$minus$greater`. If you ever wanted to access this identifier from Java code, you'd need to use this internal representation
153 - Mixed identifier consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier, e.g. `unary_+` (used to support ***properties***)
153 - Literal identifier is an arbitrary string enclosed in back ticks. Used to tell Scala to treat a keyword as an ordinary identifier, e.g., writing `Thread.'yield'()` treats `yield` as an identifier rather than a keyword
156 - Implicit conversion definition:

```
implicit def intToRational(x: Int) = new Rational(x)
```

- for an implicit conversion to work, it needs to be in scope. If you place the implicit method definition inside the class `Rational`, it won't be in scope

Built-in Control Structures

163 - Assignment always results with the **unit value**, `()`
164 - In `for (file <- files)` the `<-` is called a **generator**. In each iteration, a new `val` named `file` is initialized with an element value

164 - The `Range` type: `4 to 8`. If you don't want upper bound: `4 until 8`
166 - Filter: `for (file <- files if file.getName.endsWith(".scala"))`

```
// multiple filters example:
for (
   file <- files   // files is a previously defined method that returns array of files
   if file.isFile
   if file.getName.endsWith(".scala")
) println(file)
```

167 - Nested loops and **mid-stream variable binding** example with ***generators*** and ***filters***

```
def fileLines(file: java.io.File) =
   scala.io.Source.fromFile(file).getLines().toList

// curly braces may be used instead of parentheses
// the compiler does not infer semicolons inside regular parentheses
def grep(pattern: String) =
   for {
      file <- files if file.getName.endsWith(".scala")   // semicolons inferred
      line <- fileLines(file)
      trimmed = line.trim   // mid-stream variable
      if trimmed.matches(pattern)
   } println(file + ": " + trimmed)
```

168 - **`yield`** keyword makes `for` clauses produce a value (of the same type as the expression iterated over). Syntax: `for clauses yield body`
174 - **`match case`**

- unlike Java's `select case`, there is no fall through, `break` is implicit and `case` expression can contain any type of value
- `_` is a placeholder for ***completely unknown value***

```
val target = firstArg match {   // firstArg is a previously initialized val
   case "salt" => println("pepper")
   case "chips" => println("salsa")
   case "eggs" => println("bacon")
   case _ => println("waat?")
}
```

175 - Scala doesn't have `break`, nor does it have `continue` statement
180 - Unlike Java, Scala supports ***inner scope variable shadowing***

Functions and Closures

186 - Local functions are functions inside other functions. They are visible only in their enclosing block
188 - Function literal example: `(x: Int) => x + 1`
188 - Every function value is an instance of some class that extends one of `FunctionN` traits that has an `apply` method used to invoke the function (`Function0` for functions with no params, `Function1` for functions with 1 param, ...)
189 - **`foreach`** is a method of `Traversable` trait (supertrait of `List`, `Set`, `Array` and `Map`) which takes

a function as an argument and applies it to all elements
190 - `filter` method takes a function that maps each element to true or false,
e.g. `someNums.filter((x: Int) => x > 0)`
190 - Target typing - Scala infers type by examining the way the expression is used,
e.g. `filter`example can be written: `someNums.filter(x => x > 0)`
191 - Placeholder allows you to write: `someNums.filter(_ > 0)`

- only if each function parameter appears in function literal only once (one placeholder for each param, sequentially)
- sometimes the compiler might not have enough info to infer missing param types:

```
val f = _ + _    // error: missing parameter type for expanded function...
val f = (_: Int) + (_: Int)    // OK: f(5, 10) = 15
```

192 - Partially applied function (PAF)

- an expression in which you don't supply all of the arguments needed by the function. Instead, you supply some, or none:

```
someNums.foreach(println _)
// is equivalent to:
someNums.foreach(x => println(x))
// if a function value is required in that place you can omit the placeholder:
someNums.foreach(println)
def sum(a: Int, b: Int, c: Int) = a + b + c

val a = sum _    // '_' is a placeholder for the entire param list
a: (Int, Int, Int) => Int = <function3>

// they are called partially applied functions because you can do this:
val b = sum(1, _: Int, 3)
b(2)    // Int = 6
```

197 - Closures see the changes to **free variables** and ***vice versa***, changes to ***free variables*** made by ***closure*** are seen outside of ***closure***
199 - Repeated parameters Scala allows you to indicate that the last param to a function may be repeated:

```
def echo(args: String*) = for(arg <- args) println(arg)
// Now `echo` may be called with zero or more params

// to pass in an `Array[String]` instead, you need to
// append the arg with a colon and an `_*` symbol:
echo(Array("arr", "of", "strings"): _*)
```

200 - Named arguments allow you to pass args to a function in a different order:

```
// The syntax is to precede each argument with a param name and an equals sign:
speed(distance = 100, time = 10)

// it is also possible to mix positional and named args
```

```
// in which case the positional arguments, understandably, must come first
```

201 - Default parameter values allows you to omit such a param when calling a function, in which case the param will be filled with its default value:

```
def printTime(out: java.io.PrintStream = Console.out) =
  out.println("time = " +  System.currentTimeMillis())

// now, you can call the function like this:
printTime()
// or like this:
printTime(Console.err)
```

202 - Tail recursion (Tail call optimization)

- if the recursive call is the last action in the function body, compiler is able to replace the call with a jump back to the beginning of the function, after updating param values
- because of the JVM instruction set, tail call optimization cannot be applied for two mutually recursive functions nor if the final call goes to a function value (function wraps the recursive call):

```
val funValue = nestedFun _
def nestedFun(x: Int) {
  if (x != 0) {println(x); funValue(x - 1)}   // won't be optimized
}
```

Control Abstractions

207 - Higher order functions

- functions that take other functions as parameters:

```
/**
 * refactoring imperative code:
 * demonstrates control abstraction (higher order function)
 * that reduces code duplication and significantly simplifies the code
*/
// function receives a String and a function that maps (String, String) => Boolean
def filesMatching(query: String, matcher: (String, String) => Boolean) = {
  for (
    file <- filesHere;   // filesHere is a function that returns an Array of files
    if matcher(file.getName, query)
  ) yield file
}

def filesEnding(query: String) =
  filesMatching(query, (fileName: String, query: String) => fileName.endsWith(query))

def filesContaining(query: String) =
  filesMatching(query, (fileName, query) => fileName.contains(query)) // OK to omit types

def filesRegex(query: String) =
  filesMatching(query, _.matches(_))   // since each 'matcher' param is used only once
```

```
// since the query is unnecessarily passed around,
// we can further simplify the code by introducing a closure
def filesMatching(matcher: String => Boolean) = {
   for(
      file <- filesHere;
      if matcher(file.getName)
   ) yield file
}

def filesRegex(query: String) =
   filesMatching(_.matches(query))    // 'matches' closes over free variable 'query'
```

213 - Currying

- a curried function is applied to multiple argument lists, instead of just one:

```
def curriedSum(x: Int)(y: Int) = x + y
// curriedSum: (x: Int)(y: Int)Int

curriedSum(1)(2)
// Int = 3

/*
 * Curried f produces two traditional function invocations. The first function invocation
 * takes a single 'Int' parameter named 'x', and returns a function value for the second
 * function, which takes the 'Int' parameter 'y'
*/

// This is what the first function actually does:
def first(x: Int) = (y: Int) => x + y    // returns function value
// (x: Int)Int => Int

val second = first(1)    // applying 1 to the first fn yields the second fn
// (x: Int)Int => Int

second(2)    // applying 2 to the second fn yields the final result
// Int = 3

/*
 * You can use the placeholder notation to use curriedSum in a partially applied function
 * expression which returns the second function:
*/
val onePlus = curriedSum(1)_    // '_' is a placeholder for the second param list
// onePlus: (Int) => Int = <function1>  // 'onePlus' does the same thing as 'second'

/*
when using placeholder notation with Scala identifiers you need to put a space between
identifier and underscore, which is why we didn't need space in 'curriedSum(1)_' and we
did need space for 'println _'
*/

// another example of higher order function, that repeats an operation two times
// and returns the result:
def twice(op: Double => Double, x: Double) = op(op(x))
twice(_ + 1, 5)    // f(f(x)) = x + 1 + 1, where x = 5
```

```
// Double = 7.0
```

216 - The Loan pattern

- when some control abstraction function opens a resource and ***loans*** it to a function:

```
// opening a resource and loaning it to 'op'
def withPrintWriter(file: File, op: PrintWriter => Unit) {
   val writer = new PrintWriter(file)
   try {
      op(writer)    // loan the resource to the 'op' function
   } finally {      // this way we're sure that the resource is closed in the end
      writer.close()
   }
}
// to call the method:
withPrintWriter(
   new File("date.txt"),
   writer => writer.println(new java.util.Date)
)

/*
 * In any method invocation in which you're passing in 'exactly one argument'
 * you can opt to use curly braces instead of parentheses to surround the argument
 */

// using 'currying', you can redefine 'withPrintWriter' signature like this:
def withPrintWriter(file: File)(op: PrintWriter => Unit)

// which now enables you to call the function with a more pleasing syntax:
val file = new File("date.txt")
withPrintWriter(file) {  // this curly brace is the second parameter
     writer => writer.println(new java.util.Date)
}
```

218 - By-name parameters

- typically, parameters to functions are ***by-value*** parameters, meaning, the value of the parameter is determined before it is passed to the function
- to write a function that accepts an expression that is not evaluated until it's called within a function, you use ***call-by-name*** mechanism, which passes a code block to the callee and each time the callee accesses the parameter, the code block is executed and the value is calculated:

```
var assertionsEnabled = true
def myAssert(predicate: () => Boolean) =    // without by-name parameter
   if (assertionsEnabled && !predicate())    // call it like this: myAssert(() => 5 > 3)
      throw new AssertionError

// to make a by-name parameter, you give the parameter a type
// starting with '=>' instead of '() =>'
def myAssert(predicate: => Boolean) =         // with by-name parameter
   if (assertionsEnabled && !predicate())    // call it like this: myAssert(5 > 3)
      throw new AssertionError                // which looks exactly like built-in structure

// we could've used a plain-old Boolean, but then the passed expression
```

```
// would get executed before the call to 'boolAssert'
```

Composition and Inheritance

222 - Composition means one class holds a reference to another
224 - `abstract` method does not have an implementation (i.e., no equals sign or body)

- unlike Java, no abstract modifier is allowed on method declarations
- methods that do have an implementation are called **concrete**

224 - Class is said to **declare an abstract method** and that it **defines a concrete method** (i.e. ***declaration*** is ***abstract***, ***definition*** is ***concrete***)
225 - Methods with empty parentheses are called **empty-paren methods**

- this convention (see ***bullet 127*** on top) supports the **uniform access principle**, which says that the client code should not be affected by a decision to implement an attribute as a field or as a method
- from the client's code perspective, it should be irrelevant whether `val` or `def` is accessed
- the only difference is speed, since fields are precomputed when the class is initialized
- but, on the other hand, fields are carried around with the parent object

229 - Fields and methods belong to the same ***namespace***, which makes possible for a field to override a parameterless method, but it forbids defining a field and a method with the same name
230 - ***Java*** has four namespaces: fields, methods, types and packages

- ***Scala*** has two namespaces:
- **values** (fields, methods, packages and singleton objects)
- **types** (classes and traits)

231 - Parametric field

- a shorthand definition for ***parameter*** and ***field***, where ***field*** gets assigned a ***parameter's*** value (the parametric field's name must not clash with an existing element in the same namespace, like a field or a method):

```
class ArrayElement(
  val contents: Array[String]    // could be: 'var', 'private', 'protected', 'override'
)
```

232 - You pass an argument to the superconstructor by placing it in parentheses following the name of the superclass:

```
class LineElement(s: String) extends ArrayElement(Array(s)) {
  override def width = s.length    // 'override' mandatory for concrete member overrides
  override def height = 1
}
```

238 - If you want to disallow for a method to be overridden or for a class to be subclassed, use the keyword **final** (e.g. `final class ...` or `final def ...`)
240 - `++` operator is used to concatenate two arrays
241 - `zip` is used to pair two arrays (make `Tuple2`s), dropping the elements from the longer array that don't have corresponding elements in the shorter array, so:

```
Array(1, 2, 3) zip Array("a", "b") // will evaluate to
```

```
Array((1, "a"), (2, "b"))

// 'zip' usage example
def beside(that: Element): Element =
  new ArrayElement(
    for(
      (line1, line2) <- this.contents zip that.contents   // new Tuple2 in each iteration
    ) yield line1 + line2
  )
```

242 - `mkString` is defined for all sequences (including arrays). `toString` is called on each element of the sequence. Separator is inserted between every two elems:

```
override def toString = contents mkString "\n"
```

Scala's Hierarchy

250 - In Scala hierarchy, **scala.Null** and **scala.Nothing** are the subclasses of every class (thus the name **bottom classes**), just as **Any** is the superclass of every other class

250 - `Any` contains methods:

`==`.....`final`, same as `equals` (except for Java boxed numeric types)

`!=`.....`final`, same as `!equals`

`equals`.....used by the subclasses to override equality

`##`.....same as `hashCode`

`hashCode`

`toString`

251 - Class `Any` has two subclasses:

`AnyVal` the parent class of every built-in **value class** in Scala

`AnyRef` the base class of all **reference classes** in Scala

Built-in **value classes**: `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean` and `Unit`

- represented (except `Unit`) as Java primitives at runtime
- both `abstract` and `final`, so you cannot instantiate them with `new`
- the instances of these classes are all written as literals (e.g. `5` is `Int`)
- `Unit` corresponds to Java's `void` and has a single instance value, `()`
- Implicit conversion from `Int` to `RichInt` happens when a method that only exists in `RichInt` is called on `Int`. Similar **Booster classes** exist for other value types

All **reference classes** inherit from a special marker trait called `ScalaObject`

254 - Scala stores integers the same way as Java, as 32-bit words, but it uses the ***backup*** class `java.lang.Integer` to be used whenever an int has to be seen as object

256 - For **reference equality**, `AnyRef` class has `eq` method, which cannot be overridden (behaves like `==` in Java for reference types). Opposite of `eq` is `ne`

256 - `Null` is a subclass of every reference class (i.e. class that inherits from `AnyRef`). It's not compatible with ***value types*** (`val i: Int = Null // type mismatch`)

257 - `Nothing` is a subtype of every other type (of ***Null*** also). There are no values of this type, it's used primarily to signal abnormal termination:

```
def error(message: String): Nothing =
  throw new RuntimeException(message)

// because of its position in type hierarchy, you can use it like this:
def divide(x: Int, y: Int): Int =        // must return 'Int'
  if(y != 0) x / y                       // Returns 'Int'
  else error("can't divide by zero")    // 'Nothing' is a subtype of 'Int'
```

Traits

258 - Trait

- encapsulates method and field definitions, which can then be reused by mixing them into classes

- ***trait*** can be mixed in using keywords `extends` or `with`. The difference is that, by using `extends`, you implicitly inherit the trait's superclass (`AnyRef` if a trait has no explicit superclass)
- ***trait*** also defines a type which can be used as a regular class
- if you want to mix a trait into a class that explicitly extends a superclass, use `extends` to indicate the superclass and `with` to mix in the trait:
- to mix in multiple traits using `with`:
- a class can override trait's members (polymorphism works the same way as with regular classes):

```
class Animal
class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
  override def philosophize() {
    println("It ain't easy being " + toString)
  }
}
```

261 - ***Traits*** can declare fields and maintain state (unlike Java interfaces). You can do anything in a trait definition that you can do with a class definition, with two exceptions:

- traits cannot have ***class parameters***
- traits have dynamically bound `super` (unlike statically bound `super` in classes)
- the implementation to invoke is determined each time the trait is mixed into class
- key to allowing traits to work as ***stackable modifications***

266 - Ordered trait

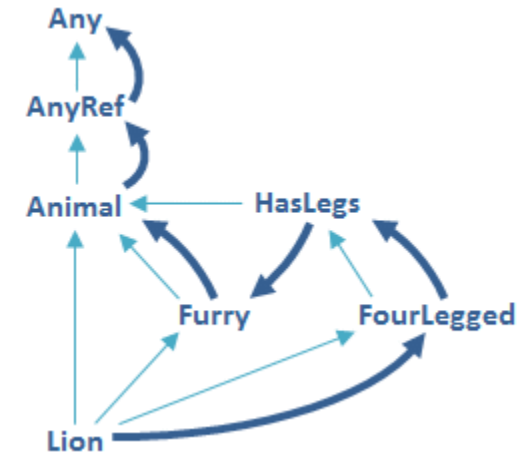
- allows you to implement all comparison operations on a class
- requires you to specify a ***type parameter*** when you mix it in (`extends Ordered[TypeYouCompare]`)
- requires you to implement the `compare` method, which should return `Int`, `0` if the object are the same, negative if receiver is less than the argument and positive if the receiver is greater than the argument
- does not provide `equals` (because of "type erasure")

267 - Stackable modifications

- traits let you modify the methods of a class in a way that allows you to stack those modifications together, by mixing in multiple traits

- when a trait extends a superclass, it means that the trait can only be mixed in in classes that also extend the same superclass
- traits can have `abstract override` methods because of dynamically bound `super` (the call works if the trait is mixed in after another trait or class has already given a concrete definition to that method)
- when you instantiate a class with `new` Scala takes the class and all of its inherited classes and traits and puts them in a single, ***linear*** order, thus the name **linearization**. Then, when you call `super` inside one of those classes, the invoked method is the first implementation up the chain (right in the image bellow)
- the **order of mixins** is significant. Traits further to the right take effect first

```
// mixing in a trait when instantiating with 'new' (no need to create a new class)
val queue = new BasicIntQueue with Doubling with Filtering    // filtering is applied first
// queue: BasicIntQueue with Doubling with Filtering = $anon$1@5fa12d

queue.put(10)    // passes the Filtering and then gets doubled with Doubling trait
queue.put(-1)    // not placed in the queue (negative number filter trait applied)
queue.get()    // Int = 20
```



275 - When to use a `trait` and when an `abstract class`

- if the behavior will not be reused make a concrete class
- if it might be used in multiple, unrelated classes, use a trait
- if you want to inherit from it in Java code, use an abstract class
- a trait with only abstract members translates to Java `interface`
- if you plan to distribute it in compiled form and you expect others to write classes that inherit from your code, use an abstract class (when a trait gains or loses a member, any class that inherit from it must be recompiled)
- if efficiency is very, very important, use a class (in Java, a virtual method invocation of a class member is faster than an interface method invocation)
- if none of the above fits your case, use trait

Packages and Imports

278 - Packages

- can be used like in C#: `package pkg_name { // source... }`, with more packages in a single source file. Also, they can be nested in one another
- a package represents a scope, whose contents is accessed relative to current location
- a top level package that's outside all packages any user can write is called `_root_`
- all names accessible outside packages can be access from inside the package in the same way
- if you stick with one package per file then Java package rules apply

```
// this
package one
package two
// is just syntactic sugar for this
package one {
  package two {

// to import the package (not a specific package member)
import one.two    // and then use objects like this: `two.Two.method`

// which is a shorthand for 'Import selector' syntax:
import one.{two}

// to access all members of a package (underscore instead of Java's star)
import one.two.three._    // could also be written as `import one.two.three.{_}`

// to use import with objects and classes
def showOne(one: One) {    // imports all members of its parameter `one`, of class `One`
  import one._    // use imports wherever you like
  println(name + "s are the best")
}

// to import more than one specific package member use 'Import selectors'
import one.{One1, One2}

// to rename import
import one.{One1 => First, One2}    // `One1` is accessed as `First` (or `one.One1`)

// to import all members and rename one of them
import one.two.{Two => Second, _}    // catch-all must come last in the list

// to import all members except one (useful for ambiguities)
import one.two.{Two => _, _}    // excludes `Two`

// implicit imports (later imports overshadow earlier ones)
import java.lang._
import scala._
import Predef._
```

288 - Access modifiers available in Scala: `Private` and `Protected`

- outer class's access to `private` members of its inner class is forbidden
- Java allows access to `protected` members to classes in the same package even if they don't inherit from the class that declares protected members. Scala don't

289 - Access qualifiers

- a modifier in the form `private[X]` or `protected[X]` means that access is applied "up to X", where `X` designates some enclosing package, class or a singleton
- **object-private** `private[this]` means that access is allowed only from within the the object that contains definition itself, not its instances (`ObjName.privMember` will fail in this case)

291 - Companion objects or Singletons

- a class shares all its access rights with its companion object and vice versa
- `protected` modifier makes no sense since ***Companion objects*** cannot be subclassed

292 - Package objects

- any kind of definition you can put in a class can go in a ***package object***
- each package is allowed to have one ***package object***
- frequently used to hold package-wide ***type aliases*** and ***implicit conversions***
- the top level `scala` package has a package object, which is available to all Scala code
- they are compiled to `package.class` file in that package's directory
- access is the same as for any other package element:

```
// in file 'one/package.scala'
package object one {
  def showSomeone(someone: Someone) {
    import someone._
    println(name + ", I am")
  }
}

// in file View.scala
package view
import one.Someone   // class defined in package 'one'
import one.showSomeone
object ViewDialog {
  def main(args: Array[String]) {
    for(someone <- Someone.dialog) {
      showSomeone(someone)
    }
  }
}
```

Assertions and Unit Testing

295 - Assertions

- written as calls of a predefined method `assert` (defined in the `Predef` singleton)
- assertions and ensuring checks can be enabled/disabled with JVM's `-ea`/`-da` flags
- `assert` methods and `ensuring` convenience methods:

```
assert(condition) // throws AssertionError
assert(condition, explanation: Any) // AssertionError contains explanation.toString

// 'ensuring' example
def widen(w: Int): Element =
  if(w <= width) this
  else {
```

```
    val left = elem(' ', (w - width) / 2, height)
    val right = elem(' ', w - width - left.width, height)
    left beside this beside right
  } ensuring(w <= _.width)   // takes a predicate function
                             // when invoked, it passes return type ('Element') to the
                             // predicate function that returns 'Boolean'
// if predicate evaluates to 'true', 'ensuring' results with 'Element' on which it was
// invoked since this is the last expression of the method, 'widen' returns the 'Element'
// throws AssertionError if predicate returns 'false'
```

297 - Unit testing

- there are many options for unit testing in Scala, e.g. Java `JUnit` and `TestNG` tools or tools written in Scala, e.g. `ScalaTest`, `specs` and `ScalaCheck`
- ScalaTest
- the simplest way to test with ***ScalaTest*** is to extend `org.scalatest.Suite` and define test methods in those classes. Methods start with `test`:

```
import org.scalatest.Suite
import Element.elem

class ElementSuite extends Suite {
  def testUniformElement() {
    val e = elem('x', 2, 3)
    assert(e.width == 2)
  }
}

// ScalaTest offers a trait 'FunSuite', which overrides 'execute'
// so you can define tests as function values, rather than methods
class ElementSuite extends FunSuite {
  // test is a method in FunSuite which is invoked by ElementSuite's primary constructor
  test("elem result should have passed width") {   // name of test
    // curly - function passed as by-name parameter to 'test'
    // which registers it for later execution
    val e = elem('x', 2, 3)
    assert(e.width == 2)   // if fails you see error message with line number
  }
}

// triple equals, if assert fails, returns nice error msg. e.g. "3 did not equal 2":
assert(e.width === 2)

// alternatively, 'expect' can be used:
expect(2) {   // yields "expected 2, but got 3" in the test failure report
  e.width
}

// if you want to check whether a method throws expected exception use 'intercept'
// if the code does not throw expected exception or doesn't throw at all
// 'TestFailedException' is thrown, along with a helpful error msg
intercept[IllegalArgumentException] {   // returns caught exception
  elem('x', -2, 3)
}
```

- although ScalaTest includes Runner application, you can also run Suite directly from the Scala interpreter by invoking `execute` on it (trait Suite's `execute` method uses reflection to discover its test methods and invoke them):

```
(new ElementSuite).execute()
Test Starting - ElementSuite.testUniformElement
Test Succeeded - ElementSuite.testUniformElement
```

- in **BDD**, the emphasis is on writing human-readable specifications of the expected code behavior, along with the accompanying tests that verify that behavior
- for that purpose, ScalaTest includes several traits: Spec, WordSpec, FlatSpec and FeatureSpec

```
import org.scalatest.FlatSpec
import org.scalatest.matchers.ShouldMatchers
import Element.elem

class ElementSpec extends FlatSpec with ShouldMatchers {
  "A UniformElement" should "have a width equal to the passed value" in {
    val e = elem('x', 2, 3)
    e.width should be (2)
  }
  it should "have a height equal to the passed value" in {   // 'specifier clause'
    val e = elem('x', 2, 3)
    e.height should be (3)
  }
  it should "throw an IAE if passed a negative width" in {   // or 'must' or 'can'
    evaluating {
      elem('x', -2, 3)
    } should produce [IllegalArgumentException]
  }
}
```

Case Classes and Pattern Matching

310 - Case classes

- for classes with `case` modifier, Scala compiler adds some syntactic sugar:
- a factory method with the same name as the class, which allows you to create new object without keyword `new` (`val m = MyCls("x")`)
- all class parameters implicitly get a `val` prefix, so they are made into fields
- compiler adds "natural" implementations of methods `toString`, `hashCode` and `equals`, which will print, hash and compare a whole tree of the class and its arguments
- `copy` method is added to the class (used to create modified copies). To use it, you specify the changes by using ***named parameters*** and for any param you don't specify, the original value is used:

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
val op = BinOp("+", Number(1), Var("x"))
```

```
// op: BinOp = BinOp(+,Number(1.0),Var(x))

// copy method example
op.copy(operator = "-")
// BinOp = BinOp(-,Number(1.0),Var(x))
```

312 - Pattern matching

- the biggest advantage of ***case classes*** is that they support ***pattern matching***

```
// written in the form of 'selector match {alternatives}'
def simplifyTop(expr: Expr): Expr = expr match {
   case UnOp("-", UnOp("-", e))  => e
   case BinOp("+", e, Number(0)) => e
   case BinOp("*", e, Number(1)) => e
   case _ => expr
}

// the right hand side can be empty (the result is 'Unit'):
case _ =>
```

- `match` expression is evaluated by trying each of the patterns in the order they are written. The first pattern that matches is selected and the part following the fat arrow is executed
- `match` ***is an expression*** in Scala (always results in a value)
- there is ***no fall through*** behavior into the next case
- if ***none of the patterns match***, an exception `MatchError` is thrown

315 - Constant patterns

- matches only itself (comparison is done using `==`)
- any literal, `val` or singleton object can be used as a constant

```
def describe(x: Any) = x match {
   case 5 => "five"
   case true => "truth"
   case "hello" => "hi"
   case Nil => "the empty list"   // built-in singleton
   case _ => "something unexpected"
}
```

316 - Variable patterns

- matches any object, like wildcard
- unlike the wildcard, Scala binds the variable to whatever the object is and then a variable refers to that value in the right hand side of the `case` clause

```
import math.{E, Pi}

val pi = math.Pi

E match {
```

```
  case 0 => "zero"
  case Pi => "strange! " + E + " cannot be " + Pi
  case `pi` => "strange? Pi = " + pi   // will be treated as constant ('val pi')
  case pi => "That could be anything: " + pi   // variable pattern ('val pi')
  case _ => "What?"   // Compiler reports "Unreachable code" error
}
/*
 * How does Scala know whether 'Pi' is a constant from 'scala.math' and not a variable?
 * A simple lexical rule is applied:
 *  - If a name starts with a lowercase letter Scala treats it as a variable pattern.
 *  - All other references are treated as constants
 *  - With exception of fields: 'this.pi' and 'obj.pi', and lowercase names in back ticks
 */
```

314 - Wildcard patterns

- `_` matches every value, but it doesn't result with a variable

```
// in this example, since we don't care about elements of a binary operation
// only whether it's a binary operation or not, we can use wildcard pattern:
expr match {
  case BinOp(_, _, _) => println(expr + "is a binary operation")
  case _ => println("It's something entirely different")
}
```

318 - Constructor patterns

- Scala first checks whether the object is a member of the named ***case class*** and then checks that the constructor params of the object match the patterns in parentheses
- **Deep matching** means that it looks for patterns arbitrarily deep

```
// first checks that the top level object is a 'BinOp', then whether the third
// constructor param is a 'Number' and finally that the value of that number is '0'
expr match {
  case BinOp("+", e, Number(0)) => println("a deep match")   // checks 3 levels deep
  case _ =>
}
```

318 - Sequence patterns

- `List` and `Array` can be matched against, just like ***case classes***

```
// checks for 3 element list that starts with zero:
expr match {
  case List(0, _, _) => println("zero starting list of three elements")
}

// to check against the sequence without specifying how long it must be:
expr match {
  case List(0, _*) => println("zero starting list")
  case List(_*) => println("any list")
}
```

319 - Tuple patterns

```
("a ", 3, "-tuple") match {
     case (a, b, c) => println("matched " + a + b + c)   // matched a 3-tuple
     case _ =>
   }
```

319 - Typed patterns

- used for convenient type checks and type casts

```
def generalSize(x: Any) = x match {
   case s: String => s.length   // type check + type cast - 's' can only be a 'String'
   case m: Map[_, _] => m.size
   case _ => -1
}                                                    //> generalSize: (x: Any)Int

generalSize("aeiou")                                 //> Int = 5
generalSize(Map(1 -> 'a', 2 -> 'b'))                //> Int = 2
generalSize(math.Pi)                                 //> Int = -1

// generally, to test whether expression is an instance of a type:
expr.isInstanceOf[String]    // member of class 'Any'
// to cast
expr.asInstanceOf[String]    // member of class 'Any'

def isIntToIntMap(x: Any) = x match {
   // non-variable type Int is unchecked since it's eliminated by erasure
   case m: Map[Int, Int] => true
   case _ => false
}

isIntToIntMap(Map(1 -> 2, 2 -> 3))                   //> Boolean = true
isIntToIntMap(Map("aei" -> "aei"))                   //> Boolean = true !!!

// the same thing works fine with arrays since their type is preserved with their value
```

- **Type erasure**
- erasure model of generics, like in Java, means that no information about type arguments is maintained at runtime. Consequently, there is no way to determine at runtime whether a given Map object has been created with two Int arguments, rather than with arguments of any other type. All the system can do is determine that a value is a Map of some arbitrary type parameters

323 - Variable binding

- allows you to, if the pattern matches, assign a variable to a matched object
- the syntax is `var_name @ some_pattern'

```
case UnOp("abs", e @ UnOp("abs", _)) => e    // if matched, 'e' will be 'UnOp("abs", _)'
```

324 - Pattern guards

- in some circumstances, syntactic pattern matching is not precise enough
- a pattern guard comes after a pattern and starts with an `if`

- can be arbitrary boolean expression and typically refers to pattern variables
- the pattern matches only if the guard evaluates to `true`

```
// match only positive integers
case n: Int if 0 < n => n + " is positive"
// match only strings starting with the letter ‘a’
case s: String if s(0) == 'a' => s + " starts with letter 'a'"
```

- e.g. if you'd like to transform `x + x` to `2 * x` with patterns:

```
// this won't work, since a pattern variable may only appear once in a pattern:
def simplifyAdd(e: Expr) = e match {
      case BinOp("+", x, x) => BinOp("*", x, Number(2))    // x is already defined as value x
      case _ => e
}

// so instead:
   def simplifyAdd(e: Expr) = e match {
      // matches only a binary expression with two equal operands
      case BinOp("+", x, y) if x == y => BinOp("*", x, Number(2))
      case _ => e
   }                              //> simplifyAdd: (e: Expr)Expr

   val add = BinOp("+", Number(24), Number(24))
                                  //> add  : BinOp = BinOp(+,Number(24.0),Number(24.0))
   val timesTwo = simplifyAdd(add)
                                  //> timesTwo  : Expr = BinOp(*,Number(24.0),Number(2.0))
```

325 - Pattern overlaps

- patterns are tried in the order in which they are written

```
// recursively call itself until no more simplifications are possible
def simplifyAll(expr: Expr): Expr = expr match {
   case UnOp("-", UnOp("-", e)) =>
      simplifyAll(e)  // '-' is its own inverse
   case BinOp("+", e, Number(0)) =>
      simplifyAll(e)  // '0' is a neutral element for '+'
   case BinOp("*", e, Number(1)) =>
      simplifyAll(e)  // '1' is a neutral element for '*'
   case UnOp(op, e) =>
      UnOp(op, simplifyAll(e))
   case BinOp(op, l, r) =>
      BinOp(op, simplifyAll(l), simplifyAll(r))
   case _ => expr
}

implicit def intToNumber(x: Int) = new Number(x)
implicit def numberToInt(x: Number) = x.num.toInt

val allMin = UnOp("-", UnOp("-", 4))
val allAdd = BinOp("+", 244, 0 + Number(0))
val allMul = BinOp("*", 24, 1)
```

```
simplifyAll(allMin)     //> Expr = Number(4.0)
simplifyAll(allAdd)     //> Expr = Number(244.0)
simplifyAll(allMul)     //> Expr = Number(24.0)
```

326 - Sealed classes

- how can you be sure you covered all the cases when using pattern matching, since a new `case class` may be created in any time, in another compilation unit?
- you make the ***superclass*** of your ***case class*** `sealed`, which then means that a class cannot have any new subclasses added except the ones in the same file
- this way, when using pattern matching, you only need to worry about the subclasses you know of
- also, when you match against subclasses of a sealed class, you get the compiler support, which will flag missing combinations of patterns with a warning message:

```
// if you make 'Expr' class sealed
sealed abstract class Expr

// and leave out some patterns when matching
def describe(e: Expr): String = e match {
   case Number(_) => "a num"
   case Var(_) => "a var"
}

/* you'll get a compiler warning:
 * warning: match is not exhaustive
 * missing combination  UnOp
 * missing combination  BinOp
 *
 * which is telling you that you might get 'MatchError'
 * because some possible patterns are not handled
 */
//
//

// to get rid of the warning, in situations where you're sure that no such pattern
// will ever appear, throw in the last catch-all case:
case _ => throw new RuntimeException    // should never happen

// the same problem can be solved with more elegant solution, without any dead code
// using 'unchecked' annotation:
def describe(e: Expr): String = (e: @unchecked) match {
   case Number(_) ...
}
```

328 - The Option type

- `Option` is type for optional values, which can be of two forms:
- `Some(x)`, where `x` is the actual value
- `None` object, which represents non-existent value
- optional values are produced by some of the standard operations on collections, e.g. the `Map`'s `get` method produces `Some(value)` or `None` if there was no given key
- the common way to distinguish between optional objects is through pattern matching:

```
def show(x: Option[String]) = x match {
  case Some(s) => s
  case None => "?"
}
```

330 - Patterns in variable definitions

- patterns could be used for `Tuple` destructuring:

```
val myTuple = (123, "abc")
val (number, string) = myTuple    // multiple variables in one assignment
```

- you can deconstruct a ***case class*** with a pattern:

```
val exp = new BinOp("*", Number(5), Number(10))
val BinOp(op, left, right) = exp
/*
 * op: String = *
 * left: Expr = Number(5.0)
 * right: Expr = Number(10.0)
*/
```

331 - Case sequences as partial functions

- a sequence of cases can be used anywhere a function literal can
- essentially, a case sequence is a function literal, only more general
- instead of having a single entry point and list of params, a case sequence has multiple entry points, each with their own list of params

```
val withDefault: Option[Int] => Int = {
  case Some(x) => x
  case None => 0
}
```

- a sequence of cases gives you a ***partial function***

```
// this will work for list of 3 elements, but not for empty list
val second: List[Int] => Int = {
  case x :: y :: _ => y
}   // warning: match is not exhaustive! missing combination    Nil

second(List(1, 2, 3))   // returns 2
second(List())          // throws MatchError

/*
 * type 'List[Int] => Int' includes all functions from list of integers to integers
 * type 'PartialFunction[List[Int], Int]' includes only partial functions
 */

// to tell the compiler that you know you're working with partial functions:
val second: PartialFunction[List[Int], Int] = {
```

```
  case x :: y :: _ => y
}

// partial functions have a method 'isDefinedAt':
second.isDefinedAt(List(5, 6, 7))   // true
second.isDefinedAt(List())   // false

/*
 * these expressions above get translated by the compiler to a partial function
 *   by translating the patterns twice, once for the implementation of the real function
 *   and once to test whether the function is defined or not
 */

// e.g. the function literal
{ case x :: y :: _ => y }

// gets translated to the following partial function value:
new PartialFunction[List[Int], Int] {
  def apply(xs: List[Int]) = xs match {
    case x :: y :: _ => y
  }
  def isDefinedAt(xs: List[Int]) = xs match {
    case x :: y :: _ => true
    case _ => false
  }
}

// the translation takes place whenever the declared type of a function literal is
//   a 'PartialFunction'
// if the declared type is just 'Function1', or is missing, the function literal gets
//   translated to a complete function

// if you can, use a complete function, because partial functions allow for runtime
//   errors that the compiler cannot spot

// if you happen to e.g. use a framework that expects partial function, you should
//   always check 'isDefinedAt' before calling the function
```

334 - Patterns in `for` expressions

```
for((country, city) <- capitals)
  println("The capital of " + country + " is " + city)

// in the above example, 'for' retrieves all key/value pairs from the map
//   each pair is then matched against the '(country, city)' pattern

// to pick elements from a list that match a pattern:
val results = List(Some("apple"), None, Some("orange"))
for(Some(fruit) <- results) println(fruit)
// apple
// orange

// 'None' does not match pattern 'Some(fruit)'
```

Working with Lists

344 - List literals

- lists are ***immutable*** (list elements cannot be changed by assignment)
- lists are ***homogeneous*** (all list elements have the same type)
- list type is ***covariant*** (if `S` is subtype of `T`, then `List[S]` is a subtype of `List[T]`)
- `List[Nothing]` is a subtype of any other `List[T]`
- that is why it's possible to write `val xs: List[String] = List()`
- they have two fundamental building blocks, `Nil` and `::` (cons), where `Nil` represents an empty list `val nums = 1 :: 2 :: 3 :: 4 :: Nil`

346 - Basic operations on lists

- all operations on lists can be expressed in terms of the following three methods:
- `head` - returns the first list element (defined for non-empty lists)
- `tail` - returns all elements except the first one (defined for non-empty lists)
- `isEmpty` - returns `true` if the list is empty
- these operations take constant time, `O(1)`

```
// insertion sort implementation:
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]): List[Int] =
  if (xs.isEmpty || x <= xs.head) x :: xs
  else xs.head :: insert(x, xs.tail)
```

347 - List patterns

- lists can be deconstructed with pattern matching, instead of with `head`, `tail` and `isEmpty`

```
val fruit = "apples" :: "oranges" :: "pears"
val List(a, b, c) = fruit   // matches any list of 3, and binds them to pattern elements
// a: String = apples
// b: String = oranges
// c: String = pears

// if you don't know the number of list elements:
val a :: b :: rest = fruit   // matches list with 2 or more elements
// a: String = apples
// b: String = oranges
// rest: List[String] = List(pears)

// pattern that matches any list:
List(...)   // instance of library-defined 'extractor' pattern
```

- normally, infix notation (e.g. `x :: y`) is equivalent to a method call, but with patterns, rules are different. When seen as a pattern, an infix operator is treated as a constructor:
- `x :: y` is equivalent to `::(x, y)` (not `x.::(y)`)
- there is a class named `::`, `scala.::` (builds non-empty lists)
- there is also a method `::` in class `List` (instantiates class `scala.::`)

```
// insertion sort implementation, written using pattern matching:
```

```
def isort(xs: List[Int]): List[Int] = xs match {
   case List() => List()
   case x :: xs1 => insert(x, isort(xs1))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs match {
   case List() => List(x)
   case y :: ys =>
      if(x <= y) x :: xs
      else y :: insert(x, ys)
}
```

349 - First-order methods on class List

- a method is ***first order*** if it doesn't take any functions as arguments
- **Concatenating two lists**
- `xs ::: ys` returns a new list that contains all the elements of `xs`, followed by all the elements of `ys`
- `:::` is implemented as a method in class `List`
- like `cons`, list concatenation associates to the right:

```
// expression like this:
xs ::: ys ::: zs
// is interpreted as:
xs ::: (ys ::: zs)
```

- **Divide and Conquer principle**
- design recursive list manipulation algorithms by pattern matching and deconstruction:

```
// my implementation of list concatenation
def append[T](xs: List[T], ys: List[T]): List[T] = xs match {
   case List() => ys
   case x :: rest => x :: append(rest, ys)
}
```

- **List length**
- `length` on lists is a relatively expensive operation, when compared to arrays (`O(n)`)
- **Accessing the end: `init` and `last`**
- `last` returns the last element of a list
- `init` returns a list consisting of all elements but the last one
- same as `head` and `tail`, they throw an exception when invoked on an empty list
- slow when compared to `head` and `tail` since they take linear time, `O(n)`
- **Reversing lists: `reverse`**
- it's better to organize your data structure so that most accesses are at the head of a list
- if an algorithm demands frequent access to the end of a list, it's better to reverse the list first

```
// 'reverse' implemented using concatenation
def rev[T](xs: List[T]): List[T] = xs match {
   case List() => xs
   case x :: rest => rev(rest) ::: List(x)    // slow: n-1 recursive calls to concatenation
```

```
    // alt. with my concatenation
    // case x :: rest => append(rev(rest), List(x))
}
```

- **Prefixes and suffixes: `drop`, `take` and `splitAt`**
- `xs take n` returns the first `n` elements of the list `xs` (if `n > xs.length` the whole `xs` is returned)
- `xs drop n` returns all elements of the list `xs` except the first `n` ones (if `n > xs.length` the empty list is returned)
- `xs splitAt n` will return two lists, the same as `(xs take n, xs drop n)`, only it traverses the list just once
- **Element selection: `apply` and `indices`**
- `xs apply n` returns ***n-th*** element
- as for all other types, `apply` is implicitly inserted when an object appears in the function position in a method call, so the example from the line above can be written as `xs(n)`
- `apply` is implemented as `(xs drop n).head`, thus it's slow (`O(n)`) and rarely used on lists, unlike with arrays
- `List(1, 2, 3).indices` returns `scala.collection.immutable.Range(0, 1, 2)`
- **Flattening a list of lists: `flatten`**
- takes a list of lists and flattens it out to a single list
- it can only be used on lists whose elements are all lists (compilation error otherwise)

```
fruit.map(_.toCharArray).flatten
// List(a, p, p, l, e, s, o, r, a, n, g, e, s, p, e, a, r, s)
```

- **Zipping lists: `zip` and `unzip`**
- `zip` takes two lists and pairs the elements together, dropping any unmatched elements
- a useful method is also `zipWithIndex`, which pairs every element with its index
- `unzip` converts list of tuples to tuple of lists

```
List('a', 'b') zip List(1, 2, 3)
// List[(Char, Int)] = List((a,1), (b,2))

val zipped = List('a', 'b').zipWithIndex
// zipped: List[(Char, Int)] = List((a,0), (b,1))

zipped.unzip
// (List[Char], List[Int]) = (List(a, b), List(1, 2))
```

- **Displaying lists: `toString` and `mkString`**
- members of the `Traversable` trait, which makes them applicable to all other collections

```
// 'toString' returns canonical representation of a list:
val abc = List("a", "b", "c")
abc.toString    // List(a, b, c)

// 'mkString' is more suitable for human consumption:
mkString(pre, sep, post)    // returns:
```

```
pre + xs(0) + sep + xs(1) + sep + ... + sep + xs(xs.length - 1) + post

// also:
xs mkString sep   // equals
xs mkString("", sep, "")   // also, you can omit all arguments (default to empty string)

// a variant of 'mkString' which appends string to a 'scala.StringBuilder' object:
val buf = new StringBuilder
abc addString(buf, "(", "; ", ")")   // StringBuilder = (a; b; c)
```

- **Converting lists: `iterator`, `toArray` and `copyToArray`**
- `toArray` converts a list to an array and `toList` does the opposite

```
val arr = abc.toArray   // Array(a, b, c)
val xs = arr.toList   // List(a, b, c)

// to copy all elements of the list to an array, beginning with position 'start':
xs copyToArray (arr, start)

// before copying, you must ensure that the array is large enough:
val arr2 = new Array[String](7)
xs copyToArray (arr2, 3)   // produces 'Array(null, null, null, a, b, c, null)'

// to use an iterator to access list elements:
val it = abc.iterator   // it: Iterator[String] = non-empty iterator
it.next   // String = "a"
it.next   // String = "b"
```

- **Merge sort example**
- faster than ***insertion sort*** for lists - `O(n log(n))`

```
  def msort[T](less: (T, T) => Boolean)(xs: List[T]): List[T] = {
    def merge(xs: List[T], ys: List[T]): List[T] = (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if(less(x, y)) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }

    val n = xs.length / 2
    if(n == 0) xs
    else {
      val (ys, zs) = xs splitAt n
      merge(msort(less)(ys), msort(less)(zs))
    }
  }

// call it like this:
val res = msort((x: Int, y: Int) => x < y)(9 :: 1 :: 8 :: 3 :: 2 :: Nil)
```

- ***currying*** helps us to create specialized functions, predetermined for a particular comparison operation:

```
// reverse sort (underscore stands for missing arguments list, in this case,
// a list that should be sorted)
val reverseIntSort = msort((x: Int, y: Int) => x > y) _
// reverseIntSort: (List[Int]) => List[Int] = <function>

reverseIntSort(9 :: 1 :: 8 :: 3 :: 2 :: Nil)
```

361 - Higher-order methods on class List

- allow you to express useful list operation patterns in a more concise way

- **Mapping over lists: `map`, `flatMap` and `foreach`**
- `xs map f`, where `xs` is some `List[T]` and `f` is a function of type `T => U`, applies the function `f` to each list element and returns the resulting list

```
List(1, 2, 3) map (_ + 1)   // returns 'List(2, 3, 4)'

val words = List("the", "quick", "brown", "fox")
words map (_.length)   // 'List(3, 5, 5, 3)'
words map (_.toList.reverse.mkString)  // 'List(eht, kciuq, nworb, xof)'

// 'flatMap' takes a function returning a list of elements as its right operand,
// which it then applies to each list element and flattens the function results
words flatMap (_.toList)   // 'List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x)'

// 'map' and 'flatMap' together:
List.range(1, 5) flatMap (i => List.range(1, i) map (j => (i, j)))
// List[(Int, Int)] = List((2,1), (3,1), (3,2), (4,1), (4,2), (4,3))
// 'range' creates a list of all integers in some range, excluding second operand

// equivalent to:
for(i <- List.range(1, 5);
    j <- List.range(1, i)) yield (i, j)

// difference between 'map' and 'flatMap':
List(1, 2, 3, 4) map (_ :: 8 :: Nil)   // List(List(1,8), List(2,8), List(3,8), List(4,8))
List(1, 2, 3, 4) flatMap (_ :: 8 :: Nil)   // List(1, 8, 2, 8, 3, 8, 4, 8)

// 'foreach' takes a procedure (a function resulting with Unit) as its right operand,
// and applies the procedure to each list element. The result is Unit, not a new list
var sum = 0
List(1, 2, 3, 4, 5) foreach (sum += _)  // sum: Int = 15
```

- **Filtering lists: `filter`, `partition`, `find`, `takeWhile`, `dropWhile` and `span`**

```
// 'filter' takes a list and a predicate function and returns the new list containing
// the elements that satisfy the predicate
val xs = List(1, 2, 3, 4, 5)
xs filter (_ % 2 == 0)   // List(2, 4)

// 'partition' returns a pair of lists, one with elements that satisfy the predicate
// and the other with ones that don't: '(xs filter p, xs filter (!p(_)))'
xs partition (_ % 2 == 0)   // (List(2, 4), List(1, 3, 5))
```

```
// 'find' is similar to 'filter', but returns only the first element, or 'None'
xs find (_ % 2 == 0)   // Some(2)
xs find (_ <= 0)   // None

// 'takeWhile' returns the longest prefix that satisfy the predicate
val ys = List(1, 2, 3, 4, 3, 2)
ys takeWhile (_ <= 3)   // List(1, 2, 3)

// 'dropWhile' is similar to 'takeWhile', but it drops the elements and returns the rest
ys dropWhile (_ <= 3)   // List(4, 3, 2)

// 'span' returns a pair of lists, the first 'takeWhile' and the second 'dropWhile'
// like 'splitAt', 'span' traverses the list only once
ys span (_ <= 3)   // (List(1, 2, 3), List(4, 3, 2))
```

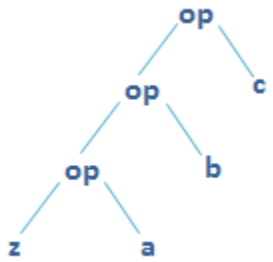
- **Predicates over lists:** `forall` **and** `exists`

```
// 'forall' takes a list and a predicate and returns 'true' if all elements satisfy the predicate
// 'exists' is similar to 'forall', but it returns 'true' if there's at least one element
// that satisfies the predicate
def hasZeroRow(m: List[List[Int]]) =
  m exists (row => row forall (_ == 0))

val y = List(0, -1)
val z = List(0, 0, 0)
val zz = List(y, y, z)   //> List(List(0, -1), List(0, -1), List(0, 0, 0))

hasZeroRow(zz)              //> Boolean = true
```

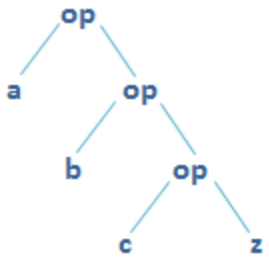
- **Folding lists:** `/:` **and** `:\`
- ***folding*** combines the elements of a list with some operator
- there are equivalent methods named `foldLeft` and `foldRight` defined in class `List`

```
sum(List(1, 2, 3))   // equals 0 + 1 + 2 + 3
// which is a special instance of a fold operation:
def sum(xs: List[Int]): Int = (0 /: xs) (_ + _)   // equals 0 + el1 + el2 + ...

def product(xs: List[Int]): Int = (1 /: xs) (_ * _)   // equals 1 * el1 * el2 * ...
```

- **fold left** operation `(z /: xs)(op)` involves 3 objects:
- start value `z`
- list `xs`
- binary operation `op`
- `(z /: List(a, b, c))(op)` equals `op(op(op(z, a), b), c)`



- **fold right** operation `(z \: xs)(op)` is the reflection of ***fold left***
- consists of the same 3 operands, but the first two are reversed, list comes first
- `(List(a, b, c) :\ z)(op)` equals `op(a, op(b, op(c, z)))`




```
// implementation of the 'flatten' methods with folding:
def flattenLeft[T](xss: List[List[T]]) =
  (List[T]() /: xss) (_ ::: _)   // less efficient, since it copies 'xss' 'n - 1' times

def flattenRight[T](xss: List[List[T]]) =
  (xss :\ List[T]()) (_ ::: _)

// '(xs ::: ys)' takes linear time 'xs.length'
// '[T]' is required due to a limitation in type inferencer

// list reversal implemented using fold left (takes linear time):
def reverseLeft[T](xs: List[T]) =
  (List[T]() /: xs) {(ys, y) => y :: ys}   // "snoc" ("cons" reversed)

// how we implemented the function:

/* First we took smallest possible list, 'List()':
  equals (by the properties of reverseLeft)
    reverseLeft(List())
  equals (by the template for reverseLeft)
    (startvalue /: List())(operation)
  equals (by the definition of /:)
    startvalue
*/

/* Then we took the next smallest list, 'List(x)':
  equals (by the properties of reverseLeft)
    reverseLeft(List(x))
  equals (by the template for reverseLeft)
    (List() /: List(x)) (operation)
  equals (by the definition of /:)
    operation(List(), x)
```

```
*/
```

- **Sorting lists:** `sortWith`
- `xs sortWith before`, where `before` is a function that compares two elements
- `x before y` should return `true` if `x` should come before `y` in a sort order
- uses ***merge sort*** algorithm

```
list(1, -2, 8, 3, 6) sortWith (_ < _)
// List(-2, 1, 3, 6, 8)

words sortWith (_.length > _.length)
// List(quick, brown, the, fox)
```

369 - Methods of the `List` object

- all the methods above are implemented in class `List`, whereas the following ones are defined in globally accessible, `List` class's companion object `scala.List`
- **Creating lists from their elements:** `List.apply`
- `List(1, 2)` is in fact `List.apply(1, 2)`
- **Creating a range of numbers:** `List.range`

```
// simplest form
List.range(1, 4)          // List(1, 2, 3)
List.range(1, 7, 2)       // List(1, 3, 5)
List.range(9, 1, -3)      // List(9, 6, 3)
```

- **Creating uniform lists:** `List.fill`
- creates a list consisting of zero or more copies of the same element

```
// use currying when invoking it:
List.fill(3)('a')     // List(a, a, a)
List.fill(2)("oy")    // List(oy, oy)

// with more than one argument in the first arg list, it'll make multi-dimensional list
List.fill(2, 3)('b')    // List(List(b, b, b), List(b, b, b))
```

- **Tabulating a function:** `List.tabulate`
- similar to `fill`, only element isn't fixed, but computed using supplied function

```
val squares = List.tabulate(5)(n => n * n)    // one list with 5 elements
// List(0, 1, 4, 9, 16)

val multiplication = List.tabulate(3, 4)(_ * _)    // 3 lists with 4 elements
// List[List[Int]] = List(List(0, 0, 0, 0), List(0, 1, 2, 3), List(0, 2, 4, 6))
/*
   0 1 2 3
0  0 0 0 0
1  0 1 2 3
```

```
2 0 2 4 6
*/
```

- **Concatenating multiple lists:** `List.concat`

```
List.concat(List(), List('b'), List('c'))    // List(b, c)
List.concat()    // List()
```

371 - Processing multiple lists together

- `zipped`, `map`, `forall`, `exists`
- `zipped` method (defined on tuples) combines the tuple elements sequentially, same as `zip`, first with first, second with second, ...
- it is used with other (multiple list) methods to apply an operation to combined elements

```
(List(4, 6, 1), List(5, 8)).zipped.map(_ * _)      // List(20, 48)

(List("on", "mali", "debeli"), List(2, 4, 5)).zipped
    .forall(_.length == _)                          //> false (two matches)
(List("on", "mali", "debeli"), List(1, 4, 9)).zipped
    .exists(_.length == _)                          // true (one matches)
(List("on", "mali", "debeli"), List(2, 4, 6)).zipped
    .exists(_.length != _)                         // false (all matches)
```

372 - Understanding Scala's type inference algorithm

- the goal of type inference is to enable users of your method to give as less type information possible, so that function literals are written in more concise way
- type inference is flow based
- in a method application `m(args)`, the inferencer:
- first checks whether the method `m` has a known type
- if it has, that type is used to infer the expected type of arguments
- e.g. in `abcde.sortWith(_ > _)` the type of `abcde` is `List[Char]`
- so it knows `sortWith` takes `(Char, Char) => Boolean` and produces `List[Char]`
- thus, it expands `(_ > _)` to `((x: Char, y: Char) => x > y)`
- if the type is not know
- e.g. in `msort(_ > _)(abcde)`, `msort` is curried, polymorphic method that takes an argument of type `(T, T) => Boolean` to a function from `List[T]` to `List[T]` where `T` is some as-yet unknown type
- the `msort` needs to be instantiated with a specific type parameter before it can be applied to its arguments
- inferencer changes its strategy and type-checks method arguments to determine the proper instance type of the method, but it fails, since all it has is `(_ > _)`
- one way to solve the problem is to supply `msort` with explicit type parameter
- `msort[Char](_ > _)(abcde) // List(e, d, c, b, a)`
- another solution is to rewrite `msort` so that its parameters are swapped:

```
def msortSwapped[T](xs: List[T])(less:
   (T, T) => Boolean): List[T] = {
   // impl
}

msortSwapped(abcde)(_ > _)    // succeeds to compile
// List(e, d, c, b, a)
```

- generally, when tasked to infer type parameters of a polymorphic method, the inferencer consults the types of all value arguments in the first parameter list, but it doesn't go beyond that
- so, when we swapped the arguments, it used the known type of the first parameter `abcde` to deduce the type parameter of `msortSwapped`, so it did not need to consult the second argument list in order to determine the type parameter of the method
- **suggested library design principle:**
- when designing a polymorphic method that takes a non-function and function arguments, place the function argument last in a curried parameter list by its own
- that way, the method's correct instance type can be inferred from the non-function arguments, and then that type can be used to type-check the function argument

Collections

377 - Sequences

- groups of data lined up in order, which allows you to get the 'n-th' element
- **Lists** (immutable linked list)
- support fast addition and removal of items to the beginning of the list
- slow in manipulating the end of a sequence (add to front and reverse in the end)
- do not provide fast access to arbitrary indexes (must iterate through the list)
- **Arrays**
- fast access of an item in an arbitrary position (both, get and update)
- represented in the same way as Java arrays (use Java methods that return arrays)

```
// to create an array whose size you know, but you don't know element values
val fiveInts = new Array[Int](5)    // Array(0, 0, 0, 0, 0)

// to initialize an array when you know the element values:
val fiveToOne = Array(5, 4, 3, 2 , 1)

// read and update:
fiveInts(0) = fiveToOne(4)
fiveInts    // Array(1, 0, 0, 0, 0)
```

- **List buffers** (mutable)
- used when you need to build a list by appending to the end
- constant time append and prepend operations
- `+=` to append, and `+=:` to prepend
- when you're done, you can obtain a list with the `toList` method of `ListBuffer`
- if your `List` algorithm is not tail recursive, you can use `ListBuffer` with `for` or `while` to avoid the potential stack overflow

```
import scala.collection.mutable.ListBuffer
val buf = new ListBuffer[Int]

buf += 22     // ListBuffer(22)
11 +=: buf    // ListBuffer(11, 22)
buf.toList    // List(11, 22)
```

- **Array buffers** (mutable)
- like an array, but you can add and remove elements from the beginning and the end
- all `Array` operations are available, though little slower, due to wrapping layer in the implementation
- addition and removal take constant time on average, but require linear time if the array needs to be expanded

```
import scala.collection.mutable.ArrayBuffer
val abuf = new ArrayBuffer[Int]()

// append using '+='
abuf += 8                          // ArrayBuffer(8)
abuf += 4                          // ArrayBuffer(8, 4)

abuf.length                        // Int = 2
abuf(1)                            // Int = 4
```

- **Strings** (via **StringOps**)
- since `Predef` has an implicit conversion from `String` to `StringOps`, you can use any string like a sequence

```
def hasUpperCaseLetter(s: String) = s.exists(_.isUpper)    // String doesn't have 'exists'

hasUpperCaseLetter("glupson 1")   // false
hasUpperCaseLetter("glupsoN 1")   // true
```

381 - Sets and maps

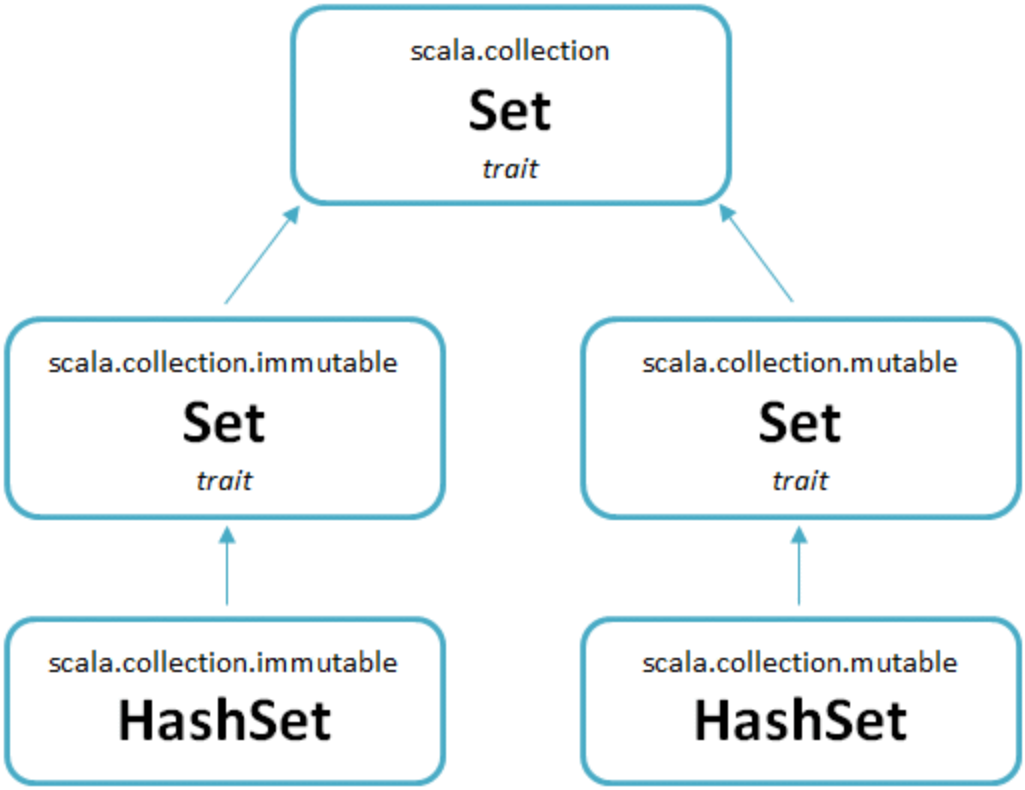
- by default, when you write `Set` or `Map`, you get an immutable object
- for mutable objects, you need explicit import

```
object Predef {
   type Map[A, +B] = collection.immutable.Map[A, B]
   type Set[A] = collection.immutable.Set[A]
   val Map = collection.immutable.Map
   val Set = collection.immutable.Set
   // ...
}
// the 'type' keyword is used in 'Predef' to define aliases for fully qualified names
// the 'vals' are initialized to refer to the singleton objects
// so 'Map' == 'Predef.Map' == 'scala.collection.immutable.Map'

// to use the immutable and mutable in the same source file:
import scala.collection.mutable
val mutaSet = mutable.Set(1, 2)   // scala.collection.mutable.Set[Int]
```

```
val set = Set(1, 2)                    // scala.collection.immutable.Set[Int]
```

- **Using sets**
- the key characteristic is that they maintain uniqueness of their elements, as defined by `==`

```
val text = "run Forest, run. That's it Forest! Run!"
val wordsArray = text.split("[ !,.]+")   // Array(run,Forest,Run,That's,it,Forest,Run)
import scala.collection.mutable
val set = mutable.Set.empty[String]    // Set()

for(word <- wordsArray)
  set += word.toLowerCase

set    // Set(it, run, that's, forest)
```

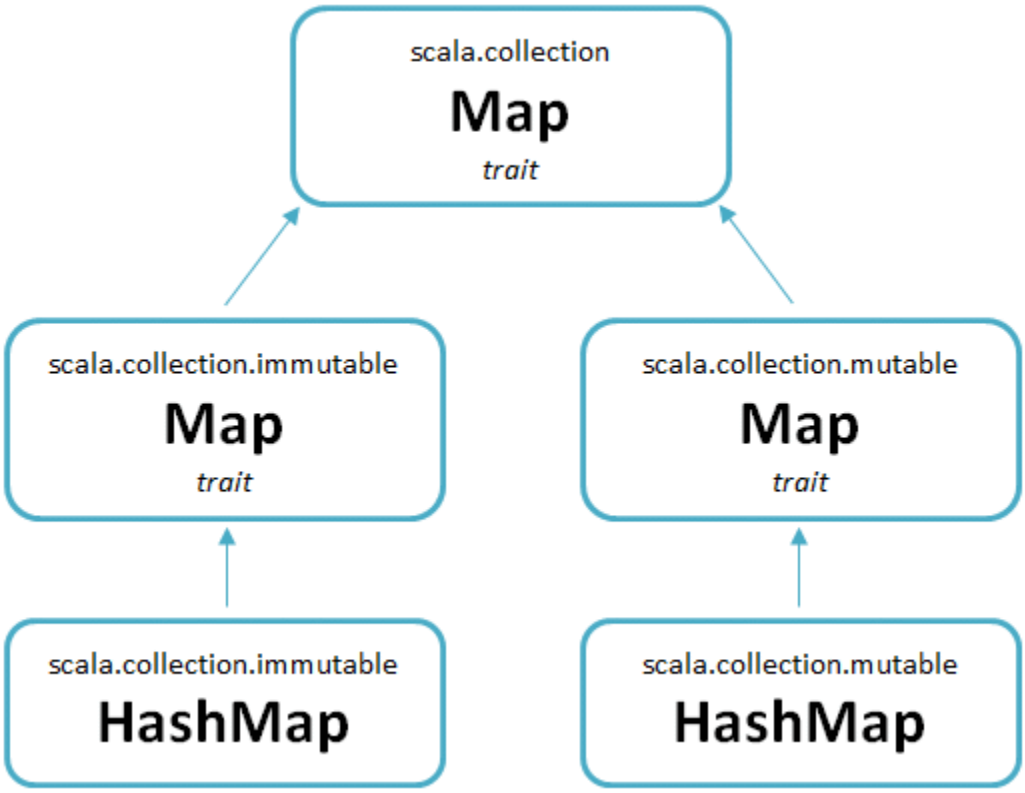



```
/*****************************************************************************/
/*********************      Common operations for sets      *********************/
/*****************************************************************************/

val nums = Set(1, 2, 3)                // crates an immutable set
nums.toString                          // returns Set(1, 2, 3)
nums + 5                               // adds an element (returns Set(1, 2, 3, 5))
nums - 3                               // removes the element (returns Set(1, 2))
nums ++ List(5, 6)                     // adds multiple elements (returns Set(1, 2, 3, 5, 6))
nums -- List(1, 2)                     // removes multiple elements
```

```
nums & Set(1, 3, 5, 7)              // returns the intersection of two sets (Set(1, 3))
nums.size                                 // returns the size of the set
nums.contains(3)                        // checks for inclusion

import scala.collection.mutable   // makes the mutable collections easy to access
val words = mutable.Set.empty[String]    // creates an empty, mutable set (HashSet)
words.toString                           // returns Set()
words += "the"                            // adds an element (Set(the))
words -= "the"                           // removes an element (Set())
words ++= List("do", "re", "mi") // adds multiple elements
words --= List("do", "re")         // removes multiple elements
words.clear                               // removes all elements
```

- **Using maps**
- when creating a map, you must specify two types (key, value)

```
// word count using a map
def main(args: Array[String]): Unit = {
  val count = countWords("run forest, run fast! run forest!")
  println(count.toString)   // Map(fast -> 1, run -> 3, forest -> 2)
}

import scala.collection.mutable
def countWords(text: String): mutable.Map[String, Int] = {
  val wordsArray = text.split("[ !,.]+")
  val map = mutable.Map.empty[String, Int]

  for(w <- wordsArray)
    if(map.contains(w))
      map(w) = map(w) + 1
    else
      map += (w -> 1)

  map
}
```




```
/*****************************************************************************************/
/*********************      Common operations for maps      **********************/
/*****************************************************************************************/

val m = Map("i" -> 1, "ii" -> 2)   // crates an immutable map
m.toString                                  // returns Map(i->1, ii->2)
m + ("vi" -> 6)                            // adds an entry (returns Map(i->1, ii->2, vi->6)
m - "ii"                                    // removes the entry (returns Map(i->1, vi->6))
m ++ List("iii" -> 5, "v" -> 5)    // adds multiple entries
m -- List("i", "ii")                     // removes multiple entries
m.size                                      // returns the size of the map
m.contains("ii")                          // checks for inclusion
m("ii")                                     // returns 2
m.keys                                      // returns Iterable over keys ("i" and "ii")
m.keySet                                   // returns keys as a set (Set(i, ii))
m.values                                   // returns Iterable over values (1, 2)
m.isEmpty                                  // indicates whether the map is empty

import scala.collection.mutable     // makes the mutable collections easy to access
val w = mutable.Map.empty[String, Int]    // creates an empty, mutable map (HashMap)
w.toString                                  // returns Map()
w += ("one" -> 1)                         // adds an entry (Map(one->1))
w -= "one"                                  // removes an entry (Map())
w ++= List("st" -> 1, "nd" -> 2, "rd" - 3)   // adds multiple entries
w --= List("st", "nd")                  // removes multiple entries
w.clear                                     // removes all entries
```

- **Default sets and maps**

- `mutable.Map()` factory method returns `mutable.HashMap` (analogous for mutable set)
- for immutable sets and maps, it depends on how many elements you pass to it:

```
// rules for sets (the same applies for maps)
Number of elements   Implementation (used to maximize performance)
0                    scala.collection.immutable.EmptySet
1                    scala.collection.immutable.Set1
2                    scala.collection.immutable.Set2
3                    scala.collection.immutable.Set3
4                    scala.collection.immutable.Set4
5 or more            scala.collection.immutable.HashSet (implemented as trie)
```

- **Sorted sets and maps**
- a set or map whose iterator returns elements in a particular order
- for this purpose, `collections` library provides traits `SortedSet` and `SortedMap`, which are implemented using `TreeSet` and `TreeMap`, and which use a **red-black tree** to keep `TreeSet`elements and `TreeMap` keys in order
- the order is determined by the `Ordered` trait, which the element type of set, or key type of map must either mix in or be implicitly convertible to

```
import scala.collection.immutable.TreeSet
val ts = TreeSet(9, 2, 5, 1, 8, 6, 4, 3)   // TreeSet(1, 2, 3, 4, 5, 6, 8, 9)

import scala.collection.immutable.TreeMap
val tm = TreeMap(8 -> 'e', 7 -> 'a', 1 -> 'w')   // Map(1 -> w, 7 -> a, 8 -> e)
val otm = tm + (2 -> 'u')   // Map(1 -> w, 2 -> u, 7 -> a, 8 -> e)
```

390 - Selecting mutable versus immutable collections

- immutable collections can usually be stored more compactly, especially small maps and sets, e.g. empty mutable map, in its default representation, HashMap, takes around 80 bytes, with 16 bytes for every new element, while immutable Map1 takes only 16 bytes, and Map4 around 40 bytes
- immutable map is a single object that's shared between all references, so referring to it costs just a single pointer field
- to make the switch between mutable and immutable, Scala provides some syntactic sugar:

```
// if you declare immutable set or map as 'var':
var toys = Set("bear", "car")   // scala.collection.immutable.Set[String] = Set(bear, car)
toys += "doll"   // a new set is created and then 'toys' is reassigned to it
toys   // scala.collection.immutable.Set[String] = Set(bear, car, doll)

toys -= "bear"   // works with any other operator method ending with '='
toys   // scala.collection.immutable.Set[String] = Set(car, doll)

// then, if you want to switch to mutable the only thing you need is:
import scala.collection.mutable.Set

// this works with any type, not just collections
```

392 - Initializing collections

- the common way to create and initialize a collection is to pass the initial elements to a factory method on the companion object (invokes `apply`)
- when an inferred type is not what you need, explicitly set type of your collection:

```
val stuff = mutable.Set[Any](42)
stuff += "green"    // scala.collection.mutable.Set[Any] = Set(42, green)
```

394 - Converting to array or list

- to convert a collection to array, simply call `toArray`
- to convert a collection to list, simply call `toList`
- there is a speed penalty, since all the elements need to be copied during the conversion process, which may be problematic for large collections
- the order of elements in the resulting list or array will be the order produced by an iterator obtained by invoking `elements` on the source collection
- in case of sorted collections, the resulting list or array will also be sorted:

```
val ts = TreeSet(8, 3, 4, 1)
val a = ts.toArray    // Array(1, 3, 4, 8)
val l = ts.toList     // List(1, 3, 4, 8)
```

395 - Converting between mutable and immutable sets and maps

- create a collection of the new type using the `empty` method and then add the new elements using method for adding multiple entries (`++` for mutable and `++=` for immutable)

```
// converting immutable TreeSet to a mutable set and back
val ts = TreeSet(9, 2, 5, 1, 8, 6, 4, 3)
val ms = mutable.Set.empty ++= ts    // mutable.Set[Int] = Set(9, 1, 5, 2, 6, 3, 4, 8)
val is = Set.empty ++ mts            // immutable.Set[Int] = Set(5, 1, 6, 9, 2, 3, 8, 4)
```

396 - Tuples

- a tuple combines a fixed number of items together so they can be passed around as a whole
- unlike arrays and lists, tuple can hold objects of different types
- tuples save you the effort of defining simplistic, data-heavy (as opposed to logic-heavy) classes
- since they can contain objects of different types, they don't inherit from `Traversable`
- a common usage pattern of tuples is returning multiple values from a method
- to access elements of a tuple you can use methods `_1`, `_2`, ...
- you can deconstruct a tuple like this: `val (word, idx) = someTuple2` (if you leave off the parentheses, both `word` and `idx` vals are assigned with a whole tuple)

```
def findLongest(words: Array[String]): Tuple2[String, Int] = {
  var len = -1
  var index = -1

  for(word <- words) {
    if(word.length > len) {
      index = words.indexOf(word)
```

```
      len = word.length
    }
  }
  (words(index), index)

}   // findLongest: (words: Array[String])(String, Int)

var toys = Set("bear", "car", "doll", "loading truck")
val tup = findLongest(toys.toArray)   // tup: (String, Int) = (loading truck,3)
```

Stateful Objects

402 - Reassignable variables and properties

- every non-private `var x` member of an object implicitly defines getter and setter
- getter is named `x` and setter is named `x_=`
- getter and setter inherit access from their `var`

```
var hour = 6
// generates
private[this] var h = 6
// and getter
hour
// and setter
hour_=

// the following two class definitions are identical:
class Time {
  var hour = 6
  var minute = 30
}

class Time {
  private[this] var h = 6   // access qualifier: private up to this (invisible outside)
  private[this] var m = 30

  def hour: Int = h
  def hour_=(x: Int) { h = x }

  def minute: Int = m
  def minute_=(x: Int) { m = x }
}

// you can define getters and setters explicitly
class Time {
  private[this] var h = 6
  private[this] var m = 30

  def hour: Int = h
  def hour_=(x: Int) {
    require(0 <= x && x < 24)
    h = x
  }
}

// getters and setters can be defined without the accompanying field:
class Thermometer {
```

```
  var celsius: Float = _    // initialized to default value (0)

  def fahrenheit = celsius * 9 / 5 + 32

  def fahrenheit_= (f: Float) {
    celsius = (f - 32) * 5 / 9
  }

  override def toString = fahrenheit + "F/" + celsius + "C"
}
```

- initializer sets a variable to default value of that type:
- `0` - for numeric types
- `false` - for booleans
- `null` - for reference types
- works the same way as uninitialized variables in Java

405 - Case study: Discrete event simulation

- internal DSL is a DSL implemented as a library inside another language, rather than being implemented on its own

Type Parameterization

422 - Information hiding

- to hide a primary constructor add `private` modifier in front of the class parameter list
- private constructor can only be accessed from withing the class itself or its companion object

```
class ImmutableQueue[T] private (
  private val leading: List[T]
  private val trailing: List[T]
)

new Queue(List(1, 2), List(3))    // error: ImmutableQueue cannot be accessed in object $iw

// now one possibility is to add auxiliary constructor, e.g.:
def this() = this(Nil, Nil)    // takes empty lists

// auxiliary constructor that takes 'n' parameters of type 'T':
def this(elems: T*) = this(elems.toList, Nil)    // 'T*' - repeated parameters

// another possibility is to add a factory method
// convenient way of doing that is to define an ImmutableQueue object
// that contains 'apply' method
// by placing this object in the same source file with the ImmutableQueue class
// it becomes its companion object
object ImmutableQueue {
  // creates a queue with initial elements 'xs'
  def apply[T](xs: T*) = new ImmutableQueue[T](xs.toList, Nil)
}

// since a companion object contains method 'apply', clients can create queues with:
ImmutableQueue(1, 2, 3)    // expands to ImmutableQueue.apply(1, 2, 3)
```

428 - Private classes

- more radical way of information hiding that hides a class itself
- then, you export a trait that reveals the public interface of a class:

```
trait Queue[T] {
  def head: T                          // public
  def tail: Queue[T]
  def enqueue(x: T): Queue[T]
}


object Queue {
  def apply[T](xs: T*): Queue[T] =
    new QueueImpl[T](xs.toList, Nil)

  private class QueueImpl[T](    // private inner class
    private val leading: List[T],
    private val trailing: List[T]
  ) extends Queue[T] {              // mixes in the trait, which has access to private class

    def mirror =
      if (leading.isEmpty)
        new QueueImpl(trailing.reverse, Nil)
      else
        this

    def head: T = mirror.leading.head

    def tail: QueueImpl[T] = {
      val q = mirror
      new QueueImpl(q.leading.tail, q.trailing)
    }

    def enqueue(x: T) =
      new QueueImpl(leading, x :: trailing)
  }

}
```

429 - Variance annotations

- `Queue`, as defined in previous listing is a trait, not a type, so you cannot create variables of type `Queue`
- instead, trait `Queue` enables you to specify parameterized types, such as `Queue[String]`, `Queue[AnyRef]`
- thus, `Queue` is a trait, and `Queue[String]` is a type
- this kind of traits are called **type constructors** (you can construct a type by specifying a type parameter, which is analogous to plain-old constructor with specified value parameter)
- ***type constructors*** generate a family of types
- it is also said that the `Queue` is a **generic trait**
- in Scala, **generic types have *nonvariant* (rigid) subtyping**
- consequently, `Queue[String]` is not a subtype of `Queue[AnyRef]`
- however, you can demand **covariant** (flexible) subtyping by prefixing a type parameter with `+`:

```
trait Queue[+T] { ... }
```

- besides `+` **parameter's variance annotation**, there's also a `-`, which indicates **contravariant**subtyping:

```
trait Queue[-T] { ... }
```

- then, if `T` is a subtype of `S`, this would imply that `Queue[S]` is a subtype of `Queue[T]`

432 - Variance and arrays

- arrays in Java are treated as covariants:

```
// Java
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(8);   // ArrayStroreException (Integer placed in String[])
String s = a1[0];
```

- because of that, arrays in Scala are nonvariant:

```
val a1 = Array("abc")
val a2 = Array[Any] = a1   // error: type mismatch, found Array[String], required Array[Any]
```

- to interact with legacy methods in Java that use an `Object` array as a means to emulate generic array, Scala lets you cast an array of `T`s to an array of any supertype of T:

```
val a2: Array[Object] = a1.asInstanceOf[Array[Object]]
```

433 - Checking variance annotations

- to verify the correctness of variance annotations, the compiler classifies all positions in a class or trait body as **positive**, **negative** or **neutral**
- a ***position*** is any location in the class (or trait) body where a type parameter may be used
- e.g. every method value parameter is a position, because it has a type and therefore a type parameter could appear in that position type parameters annotated with `+` can only be used in ***positive*** positions, `-` in negative, and a type parameter without variance annotation may be used in any position (so it's the only one that can be used in ***neutral*** positions)
- compiler classifies positions like this:
- the positions at the top level of the class are classified as positive
- positions at deeper nesting levels are classified the same as their enclosing level, but with exceptions where the classifications changes (flips):
- method value parameter positions are classified to the flipped classification relative to positions outside the method (when flipped, neutral stays the same, negative position becomes positive, and vice versa)
- classification is also flipped at the type parameters of methods
- it is sometimes flipped at the type argument position of a type (e.g. `Arg` in `C[Arg]`), depending on the variance of the corresponding type parameter (if C's type param is annotated with `+`, then the classification stays the same, and if it's `-`, then it flips, and if has no variance then it's changed to neutral)

- because it's hard to keep track of variance position, it's good to know that the compiler does all the work for you

```
// variance checks by the compiler (postfix signs represent position classification):
abstract class Cat[-T, +U] {
  def meow[W-](volume: T-, listener: Cat[U+, T-]-): Cat[Cat[U+, T-]-, U+]+
}
// since T is always used in negative position and U in positive, the class type checks
```

437 - Lower bounds

- `Queue[T]` cannot be made covariant in `T` because `T` appears as a type of a parameter of the `enqueue` method, and that's a negative position
- there's still a way to solve that problem by generalizing `enqueue` by making it polymorphic (i.e. giving the method itself a type parameter) and using a **lower bound** for its type parameter:

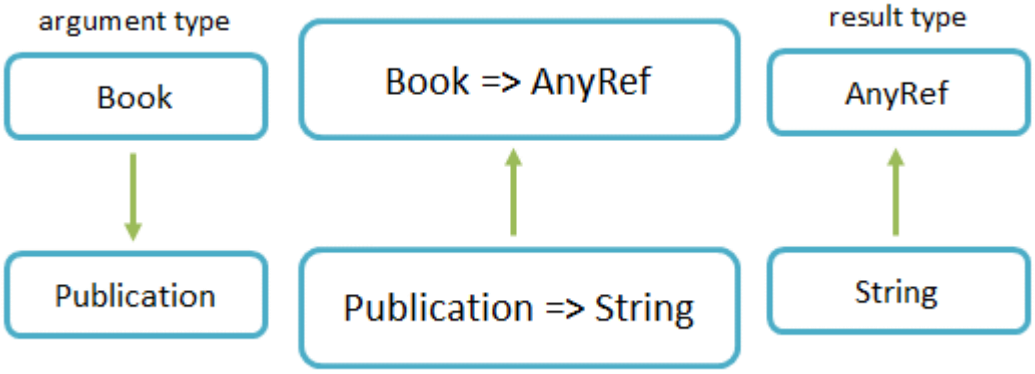
```
class Queue[+T](private val leading: List[T]), private val trailing: List[T]) {
  // defines T as the lower bound for U (U is required to be a supertype of T)
  def enqueue[U >: T](x: U) = // U is negative (flip) and T is positive (two flips)
    new Queue[U](leading, x :: trailing)
    // ...
}
// the param to 'enqueue' is now of type 'U'
// the method's return type is now 'Queue[U]', instead of 'Queue[T]'
// imagine e.g. class Fruit with two subclasses, Apple and Orange
// With the new definition of class Queue, it is possible to append an Orange to a
// Queue[Apple] and the result will be of type Queue[Fruit]
```

438 - Contravariance

- **Liskov Substitution Principle** says that it is safe to assume that a type `T` is a subtype of a type `U` if you can substitute a value of type `T` wherever a value of type `U` is required
- the principle holds if `T` supports the same operations as `U` and all of `T's` operations require less and provide more than the corresponding operations in `U`

```
// example of Contravariance of a function parameter
class Publication(val title: String)
class Book(title: String) extends Publication(title)

object Library {
  val books: Set[Book] =
    Set(
      new Book("Programming in Scala"),
      new Book("Walden")
    )

  def printBookList(info: Book => AnyRef) {    // requires function from Book to AnyRef
    for (book <- books) println(info(book))    // always passes a Book to a function
  }
}

object Customer extends Application {
  def getTitle(p: Publication): String = p.title    // accesses only Publication
```

```
    Library.printBookList(getTitle)    // provides function from Publication to String
}
// any method declared in Publication is also available on its subclass Book
// Publication => String is a subtype of Book => AnyRef
```



- because the result type of a `Function1` is defined as ***covariant***, the inheritance relationship of the two result types, shown at the right of the image, is in the same direction as that of the two functions shown in the center
- because the parameter type of a `Function1` is defined as ***contravariant***, the inheritance relationship of the two parameter types, shown at the left of the image, is in the opposite direction as that of the two functions

441 - Object private data

- object or class components that are declared as `private[this]`
- may be accessed only from within their containing object, in which they are defined
- accesses to vars from the same object do not cause problems with variance
- variance rules can be broken by having a reference to a containing object that has a statically weaker type than the type the object was defined with
- for object private values, this is not possible
- variance checking has a special case for object private definitions, which is that such definitions are omitted when checking correctness of variance positions

```
/*
 * Purely functional Queue that performs at most one trailing
 * to leading adjustment for any sequence of head operations
 * Yes, it has reassignable fields, but they are private,
 * thus invisible to any client using the class
 */
class CovariantQueue[+T] private (
      private[this] var leading: List[T],    // object private vars
      private[this] var trailing: List[T]
      // without [this]:
      // error: covariant type T occurs in contravariant position
      // in type List[T] of parameter of setter leading_=
   ) {
   private def mirror() =
      if (leading.isEmpty) {
         while (!trailing.isEmpty) {
```

```
        leading = trailing.head :: leading
        trailing = trailing.tail
      }
    }

  def head: T = {
    mirror()
    leading.head
  }

  def tail: CovariantQueue[T] = {
    mirror()
    new CovariantQueue(leading.tail, trailing)
  }

  def enqueue[U >: T](x: U) =
    new CovariantQueue[U](leading, x :: trailing)

}
```

443 - Upper bounds

- with the `T <: Ordered[T]` you indicate that the type parameter `T` has an upper bound `Ordered[T]`, which means that the passed element's type must be a subtype of `Ordered`
- used e.g. to require that the passed type mixes in a trait (i.e. is a subtype of trait)

```
// requires that passed list type mixes in Ordered trait
def orderedMergeSort[T <: Ordered[T]](xs: List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (x < y) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }
  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
    merge(orderedMergeSort(ys), orderedMergeSort(zs))
  }
}

// this is not a most general way to implement mergeSort, you cannot pass e.g. List[Int]
// that's achieved with 'implicit parameters' and 'view bounds' (section 21.6)
```

Abstract Members

- a member of a class or trait is `abstract` if the member does not have a complete definition in the class
- abstract members are intended to be implemented by subclasses
- in Scala, besides methods, you can declare abstract fields and even abstract types as members of classes and traits (vals, vars, methods and types)

- a concrete implementation needs to fill in definitions for each of its abstract members:

```
// declaration of all four types of abstract members
trait Abstract {
  type T
  def transform(x: T): T
  val initial: T
  var current: T
}

// the concrete implementation of four type of abstract members
class Concrete extends Abstract {
  type T = String    // defines type 'T' as an alias of type 'String'
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
```

448 - Type members

- **abstract types** are always members of some class or trait
- traits are abstract by definition
- a **non-abstract type member** is a way to define a new name (alias) for a type
- one reason to use a type member is to define a short, descriptive alias for a type whose real name is more verbose or less obvious in meaning (helps clarify the code)
- the other main use of type members is to declare abstract types that must be defined in subclasses

449 - Abstract vals

- have a form like `val initial: String`
- `val` is given a name and a type, but not its value
- use it when you know that each instance of the class will have an unchangeable value, but you don't know what that value will be
- its concrete implementation must be a `val` (may not be `var` or `def`)
- guaranteed to return always the same value, unlike methods, which could be implemented by a concrete method that returns a different value every time it's called
- **abstract method declarations** may be implemented by both, concrete method and concrete `val` definitions

450 - Abstract vars

- implicitly declare abstract getters and setters, just like non-abstract `vars` do
- reassignable field is not created

```
trait AbstractTime {
  var hour: Int
  var minute: Int
}

// gets expanded to:
trait AbstractTime {
  def hour: Int
  def hour_=(x: Int)
  def minute: Int
```

```
  def minute_=(x: Int)
}
```

451 - Initializing abstract vals

- abstract vals sometimes play a role of superclass parameters, i.e. they let you provide details in a subclass that are missing in a superclass
- that is particularly important for ***traits***, because they don't have a constructor to which you could pass parameters

```
// instead of class with two class parameters
trait RationalTrait {
  val numerArg: Int    // 0 until mixed in
  val denomArg: Int
}

def main(args: Array[String]): Unit = {
  // example implementation of two abstract vals
  // yields an instance of an anonymous class
  // which mixes in the trait and is defined by the body
  new RationalTrait {
    val numerArg = 1    // set to 1 as part of the initialization of the anonymous class
    val denomArg = 2    // but the anonymous class is initialized after the RationalTrait
  }                       // in the meantime, vals are set to their type's default values
}
```

- a class parameter argument is evaluated **before** it is passed to the class constructor (unless it's a by-name parameter)
- an implementing `val` definition in a subclass is evaluated only **after** the superclass has been initialized
- e.g. class parameters of `Rational(expr1, expr2)` are evaluated just before instantiation of the `Rational` object, but `RationalTrait`'s vals are evaluated as part of the initialization of the **anonymous class**

```
trait ProblematicRationalTrait {
  val numerArg: Int    // initialized once an anonymous class is created
  val denomArg: Int    // which happens after the trait is initialized
  require(denomArg != 0)    // throws "requirement failed" exception
  private val g = gcd(numerArg, denomArg)
  val numer = numerArg / g
  val denom = denomArg / g

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a
    else gcd(b, a % b)

  override def toString = numer + "/" + denom
}

// when you execute this, 'require' fails, since Int vals are 0 until
val x = 2
val fun = new ProblematicRationalTrait {
```

```
  val numerArg = 1 * x
  val denomArg = 2 * x
}
```

453 - Pre-initialized fields

- let you initialize a field of a subclass before the superclass is called
- achieved by putting field definition in braces before superclass constructor call:

```
// anonymous class creation:
new {
  val numerArg = 1 * x
  val denomArg = 2 * x
} with ProblematicRationalTrait

// object definition:
object twoThirds extends {
  val numerArg = 2
  val denomArg = 3
} with ProblematicRationalTrait

// subclass definition:
class RationalClass(n: Int, d: Int) extends {
  val numerArg = n
  val denomArg = d
} with ProblematicRationalTrait {
  def + (that: RationalClass) = new RationalClass(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )
}
// in all cases initialization section comes before the trait is mentioned
```

- because pre-initialized fields are initialized before the superclass constructor is called, their initializers cannot refer to the object that's being constructed
- so, if such an object refers to `this`, the reference goes to the object containing the class or object that's being constructed, not the constructed object itself:

```
object AbsRat {
  // ...
  val rat = new {
    val numerArg = 1
    val denomArg = this.numerArg * 2    // value numerArg is not member of object AbsRat
  } with ProblematicRationalTrait
  // ...
```

455 - Lazy vals

- evaluated the first time the val is used
- never evaluated more than once (the result of first time evaluation is stored in val)
- Scala objects are also initialized on demand, in fact an object definition can be thought of as a shorthand definition of a lazy val with an anonymous class that describes the object's contents

- since lazy vals get executed on demand, their textual order is not important when determining the order of their initialization
- in the presence of side effects (i.e. when our code produces or is affected by mutations), initialization order starts to matter and then it can be difficult to determine the actual order, which is why lazy vals are an ideal complement to functional objects

```
trait LazyRationalTrait {
   val numerArg: Int
   val denomArg: Int
   lazy val numer = numerArg / g
   lazy val denom = denomArg / g
   override def toString = numer + "/" + denom

   private lazy val g = {
      require(denomArg != 0)
      gcd(numerArg, denomArg)
   }

   private def gcd(a: Int, b: Int): Int =
      if (b == 0) a else gcd(b, a % b)
}

// using LazyRationalTrait from the interpreter:
val x = 2
new LazyRationalTrait {
   val numerArg = 1 * x
   val denomArg = 2 * x
}
// res2: LazyRationalTrait = 1/2

// 1. - fresh instance of LazyRationalTrait gets created and the initialization code
//      of LazyRationalTrait is run (fields are not initialized)
// 2. - the primary constructor of the anonymous subclass is executed (expression 'new')
//    - this involves the initialization of 'numerArg' with 2 and 'denomArg' with 4
// 3. - the 'toString' method is invoked on the constructed object (by the interpreter)
// 4. - the 'numer' field is accessed for the first time, by the 'toString' method
// 5. - the initializer of 'numer' accesses the private field 'g', so 'g' is evaluated
//    - the evaluation of 'g' accesses 'numerArg' and 'denomArg' (from step 2)
// 6. - the 'toString' method accesses the value of 'denom', which causes its evaluation
//    - that evaluation accesses the values of 'denomArg' and 'g'
//    - the initializer of the 'g' field is not re-evaluated (it's a 'val', not 'def')
// 7. - finally, the resulting string "1/2" is constructed and printed
```

459 - Abstract types

- used as a placeholder for a type that will be defined further down the hierarchy

```
// the type of food cannot be determined at the 'Animal' level, every subclass defines it
class Food
abstract class Animal {
   type SuitableFood <: Food    // upper bound is 'Food' (requires subclass of 'Food')
   def eat(food: SuitableFood)
}

class Grass extends Food
class Cow extends Animal {
```

```
    type SuitableFood = Grass    // 'Cow' fixes its 'SuitableFood' to be 'Grass'
                                 // 'SuitableFood' becomes alias for class 'Grass'
    override def eat(food: Grass) {}    // concrete method for this kind of 'Food'
}
```

461 - Path-dependent types

- objects in Scala can have types as members (e.g. any instance of 'Cow' will have type 'SuitableFood' as its member)
- e.g. `milka.SuitableFood` means "the type 'SuitableFood' that is a member of the object referenced from 'milka'", or "the type of 'Food' that suitable for 'milka'"
- a type like `milka.SuitableFood` is called a **path-dependent type**, where the word "path" means "the reference to an object"
- **path** can be a single name, such as 'milka', or a longer access path, like `farm.barn.milka.SuitableFood`, where path components are variables (or singleton object names)that refer to objects

```
class DogFood extends Food
class Dog extends Animal {
    type SuitableFood = DogFood
    override def eat(food: DogFood) {}
}

val milka = new Cow
val lassie = new Dog
lassie eat (new milka.SuitableFood)    // error: type mismatch; found: Grass, required: DogFood

// 'SuitableFood' types of two 'Dog's both point to the same type, 'DogFood'
val mickey = new Dog
lassie eat (new mickey.SuitableFood)    // OK
```

- although path-dependent types resemble Java's inner classes, there is a crucial difference:
- a path-dependent type names an outer **object**, whereas an inner class type name an outer class

```
class Outer {
    class Inner
}

// the inner class is addressed 'Outer#Inner', instead of Java's 'Outer.Inner'
// in Scala, '.' notation syntax is reserved for objects

val out1 = new Outer
val out2 = new Outer

out1.Inner    // path-dependent type
out2.Inner    // path-dependent type (different one)
// both types are subtypes of 'Outer#Inner', which represents the 'Inner' class with an
// arbitrary outer object of type 'Outer'
// by contrast, 'out1.Inner' refers to the 'Inner' class with a specific outer object
// likewise, type 'out2.Inner' refers to the 'Inner' class with a different, specific
// outer object (the one referenced from 'out2')
```

- the same as in Java, inner class instances hold a reference to an enclosing outer class instance, which allows an inner class to access members of its outer class
- thus, you cannot instantiate inner class without in some way specifying outer class instance
- one way to do this is to instantiate the inner class inside the body of the outer class (in this case, the current outer class instance is used - 'this')
- the other way is to use a path-dependent type:

```
new out1.Inner    // since 'out1' is a reference to a specific outer object
```

- the resulting inner object will contain a reference to its outer object ('out1')
- by contrast, because the type `Outer#Inner` does not name any specific instance of `Outer`, you can't instantiate it:

```
new Outer#Inner    // error: Outer is not a legal prefix for a constructor
```

464 - Structural subtyping with Refinement types

- when one class inherits from the other, the first one is said to be a **nominal subtype** of the other one (explicit subtype, by name)
- Scala additionally supports **structural subtyping**, where you get a subtyping relationship simply because two types have the same members
- ***structural subtyping*** is expressed using **refinement types**
- it is recommended that the ***nominal subtyping*** is used wherever it can, because ***structural subtyping*** can be more flexible than needed (e.g. a Graph and a Cowboy can `draw()`, but you'd rather get a compilation error than call graphical draw on cowboy)

```
// sometimes there is no more to a type than its members
// e.g. if you wanted to define 'Pasture' class that can contain animals that eat 'Grass'
// one could define a trait 'AnimalThatEatsGrass' and mix it in classes, where applicable
// but that would be verbose, since 'Cow' already declares it's an animal that eats grass
// and with the trait, it again declares that it's an 'AnimalThatEatsGrass'
// instead, you can use a 'refinement type', and to do that
// you write the base type, Animal, followed by a sequence of members in curly braces
// which are used to further refine the types of members from the base class
// so here is how to write the type "animal that eats grass":
Animal { type SuitableFood = Grass }

// and given this type, you can write the pasture:
class Pasture {
  var animals: List[Animal { type SuitableFood = Grass }] = Nil
  // ...
}
```

- another application of structural subtyping is grouping together a number of classes that were written by someone else
- for example, to generalize the loan pattern from page 216, which worked for only for type `PrintWriter`, to work with any type with a `close` method:

```
// the first try:
```

```
def using[T, S](obj: T)(operation: T => S) = {    // operation from any to any type
   val result = operation(obj)
   obj.close()  // type error: 'T' can be any type and 'AnyRef' doesn't have 'close' method
   result
}

// the proper implementation:
def using[T <: { def close(): Unit}, S](obj: T)(operation: T => S) = {
   // upper bound of 'T' is the structural type '{def close(): Unit}'
   // which means: "any subtype of AnyRef that has a 'close' method"
   val result = operation(obj)
   obj.close()
   result
}
```

- **Structural type** is a refinement type where the refinements are for members that are not in the base type

466 - Enumerations

- Scala's ***enumerations*** are not a built-in construct
- defined in `scala.Enumaration`
- to create a new enumeration, you define an object that extends `scala.Enumeration`

```
object Color extends Enumeration {
   val Red, Green, Blue = Value    // 'Value' is an inner class of 'Enumeration'
}
// 'Value' is also a method of Enumeration that returns a new instance of 'Value' class
// so 'Color.Red' is of type 'Color.Value' and so is any other enum value in object Color

// Color object definition provides 3 values: 'Color.Red', 'Color.Green' and 'Color.Blue'

// to import everything from Color:
import Color._    // and then just use 'Red', 'Green' and 'Blue', without the object name
```

- `Color.Value` is a ***path-dependent*** type, with `Color` being the path and `Value` being the dependent type
- it's a completely new type, different from all other types
- you can associate names with enumeration values by using a different overloaded variant of the `Value` method:

```
object Direction extends Enumeration {
   val Left = Value("Left")
   val Right = Value("Right")
}

// to iterate over values:
for (d <- Direction.values) print(d + " ")    // Left Right

// you can get the number of enumeration value by its 'id' method (zero-based):
Direction.Right.id    // Int = 1

// also:
```

```
Direction(0)    // Direction.Value = Left
```

Implicit Conversions and Parameters

- used when working with two bodies of code that were developed separately, thus each may have its own way to represent the same concept
- **implicits** help by reducing the number of explicit conversions one has to write

```
// using Swing without implicit conversions (a lot of boilerplate):
val button = new JButton
button.addActionListener(
  new ActionListener {
    def actionPerformed(event: ActionEvent) {
      println("pressed!")
    }
  }
)

// Scala-friendly version:
button.addActionListener(    // type mismatch (passing function instead of ActionListener)
  (_: ActionEvent) => println("pressed!")
)

// first step is to define implicit conversion from function to action listener:
implicit def function2ActionListener(f: ActionEvent => Unit) =
  new ActionListener {
    def actionPerformed(event: ActionEvent) = f(event)
  }

// now we can write:
button.addActionListener(
  function2ActionListener(
    (_: ActionEvent) => println("pressed!")
  )
)

// which is already much better than the inner class version
// because 'function2ActionListener' is marked as 'implicit', it can be left out:
button.AddActionListener(    // now this works!
  (_: ActionEvent) => println("pressed!")
)
```

- how implicits work:
- the compiler first tries to compile it as is, but it sees a type error
- it looks for an implicit conversion that can repair the problem
- it finds 'function2ActionListener'
- it tries to use it as a conversion method, sees that it works and moves on

482 - Rules for implicits

- **implicit definitions** are definitions that the compiler is allowed to insert into a program in order to fix a type error
- you can use `implicit` to mark any variable, function or object definition

Implicit conversions are governed by the following general rules

- **Marking rule:** Only definitions marked `implicit` are used
- **Scope rule:** An inserted implicit conversion must be in scope as a single identifier, or be associated with the conversion's source or target type
- ***single identifier*** means that the compiler will not insert a conversion of the form `someVariable.convert`
- it is common for libraries to include a `Preamble` object that contains useful implicits, which allows the code that uses a library to do a single `import Preamble._`
- there's one exception to single identifier rule: the compiler will also look for implicit definitions in the companion objects of both, source and target types
- when implicit is placed in a companion object of some type, it is said that the conversion is **associated** to that type
- ***modular code reasoning:*** when you read a source file, the only things you need to consider in other source files are those that are either imported or explicitly referenced through a fully qualified name
- **One-at-a-time rule: Only one implicit is tried**
- for sanity's sake, the compiler does not insert further implicits when it's already in the process of trying another implicit, e.g. `convert1(convert2(x)) + y`
- that would cause compile time to increase dramatically on erroneous code and would increase the difference between what the programmer writes and what the program does
- it is possible to circumvent this rule by having implicits take implicit params
- **Explicits-First rule:**
- the compiler will not change code that already works
- a consequence of this rule is that you can make trade offs between verbose (explicits) and terse (implicits) code

484 - Naming an implicit conversion

- implicit conversions can have arbitrary names
- the name matters only in two situations:
- if you want to write it explicitly in a method application
- for determining which implicits are available in a program

```
// to determine which implicits will be used:
object MyConversions {
  implicit def stringWrapper(s: String): IndexedSeq[Char] = ...
  implicit def intToString(x: Int): String = ...
}

// you can achieve that your code uses only 'stringWrapper' and not 'intToString':
import MyConversions.stringWrapper   // possible only because implicit has a name
// ... code making use of 'stringWrapper'
```

485 - Where implicits are tried

- there are 3 places where implicits are used:
  1 conversions to an expected type (use one type where the other is expected)
  2 conversions of the receiver of a selection (adapts receiver of a method call)
  3 implicit parameters

485 - Implicit conversion to an expected type

- whenever the compiler sees an `X`, but needs a `Y`, it will look for an implicit function that converts `X` to `Y`

```
val i: Int = 3.5    // type mismatch (loss of precision)

// however, if you define implicit conversion:
implicit def doubleToInt(x: Double) = x.toInt

val i: Int = 3.5    // i: Int = 3

/*
 * 1. the compiler sees a 'Double' 3.5 in a context where it requires an 'Int'
 * 2. before giving up and reporting 'type mismatch', it searches for a suitable implicit
 * 3. it finds 'doubleToInt', and wraps 3.5 in the 'doubleToInt' function call
 */
```

486 - Converting the receiver

- implicits are applied on an object on which a method is invoked
- has 2 main uses: allows smoother integration of a new class into an existing class hierarchy and second, they support writing DSLs withing the Scala language
- how it works:
- you write down `obj.doIt` where `obj` doesn't have a member named `doIt`
- before giving up, compiler tries to insert conversions that apply to the `obj`, converting it to a type that has a `doIt` member

```
// use an instance of one type as if it was an instance of some other type
class Rational(n: Int, d: Int) {    // existing class (from page 155)
   // ...
   def + (that: Rational): Rational = ...
   def + (that: Int): Rational = ...
}

// the two overloaded methods allow you to:
val oneHalf = new Rational(1, 2)    // Rational = 1/2
oneHalf + oneHalf    // Rational = 1/1
oneHalf + 1    // Rational = 3/2

// but:
1 + oneHalf    // error: overloaded method value + (of Int) cannot bi applied to Rational

// so:
implicit def intToRational(x: Int) = new Rational(x, 1)

1 + oneHalf    // Rational = 3/2

/*
 * 1. the compiler first tries to type check the expression '1 + oneHalf' as it is
 * 2. this fails because none of Int's '+' methods takes a 'Rational' argument
 * 3. compiler searches for an implicit conversion from 'Int' to another type
 *    that has a '+' method which can be applied to a 'Rational'
 * 4. it finds 'intToRational' and wraps 1 in the 'intToRational' call:
 */
intToRational(1) + oneHalf
```

489 - Simulating new syntax

- the major use of implicit conversions is to simulate adding new syntax

```
// you can make a map using syntax:
Map(1 -> "one", 2 -> "two")    // what is '->'

// '->' is a method of the class 'ArrowAssoc' (defined in 'scala.Predef' preamble)
// preamble also defines an implicit conversion from 'Any' to 'ArrayAssoc' so that the
// '->' method can be found:

package scala
object Predef {
   class ArrowAssoc[A](x: A) {
      def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
   }

   implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)

   // ...
}
```

- that is called a **rich wrapper pattern**, which is common in libraries that provide syntax-like extensions to the language
- classes named 'RichSomething' (e.g. 'RichInt' or 'RichBoolean') are likely using implicits to add the syntax-like methods to type 'Something'

489 - Implicit parameters

- compiler can also insert implicits within argument lists, e.g. replacing `someCall(a)` with `someCall(a)(b)` or `new SomeClass(a)` with `new SomeClass(a)(b)`, thereby adding a missing parameter list to complete a function call
- it is the entire last curried parameter that's supplied, not just the last parameter, e.g. compiler might replace `aCall(a)` with `aCall(a)(b, c, d)`
- for this to work, not just that the inserted identifiers (such as b, c and d) must be marked `implicit` where they are defined, but also the last parameter list in `aCall`'s definition must be marked `implicit`:

```
// suppose you have a class which encapsulates a user's preferred shell prompt string:
class PreferredPrompt(val preference: String)

object Greeter {
   def greet
      (name: String)    // first param list
      (implicit prompt: PreferredPrompt) {    // implicit applies to the entire param list
         println("Welcome, " + name)
         println(prompt.preference)
   }
}

object Prompt {    // dummy - just hosting 'main'
   def main(args: Array[String]): Unit = {
      val bobsPrompt = new PreferredPrompt("relax> ")
      Greeter.greet("Bob")(bobsPrompt)    // explicit prompt
```

```
    implicit val prompt = new PreferredPrompt("Yes, master> ")   // implicit identifier
    Greeter.greet("Joe")   // implicit prompt
  }
}
```

- example with multiple parameters in the last parameter list:

```
class PreferredPrompt(val preference: String)
class PreferredDrink(val preference: String)

object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt, drink: PreferredDrink) {
    println("Welcome, " + name + ". The system is ready.")
    print("But while you work, ")
    println("why not enjoy a cup of " + drink.preference + "?")
    println(prompt.preference)
  }
}

object Prompt {    // dummy - just hosting 'main'
  def main(args: Array[String]): Unit = {
    val bobsPrompt = new PreferredPrompt("relax> ")
    val bobsDrink = new PreferredDrink("travarica")
    Greeter.greet("Bob")(bobsPrompt, bobsDrink)   // all explicit

    implicit val prompt = new PreferredPrompt("Yes, master> ")
    implicit val drink = new PreferredDrink("rakija")
    Greeter.greet("Joe")
  }
}
```

- implicit parameters are most often used to provide information about a type mentioned explicitly in the earlier parameter list (like ***type classes*** in Haskell):

```
// the weakness of this method is that you cannot use it to sort list of Ints
// because it requires that 'T' is a subtype of 'Ordered[T]', which Int isn't
def maxListUpBound[T <: Ordered[T]](elements: List[T]): T =
  elements match {
    case List() => throw new IllegalArgumentException("empty")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListUpBound(rest)
      if (x > maxRest) x
      else maxRest
  }

// to remedy the weakness, we could add an extra argument
// that converts 'T' to 'Ordered[T]' (i.e. provides info on how to order 'T's)
def maxListImpParm[T](elements: List[T])(implicit ordered: T => Ordered[T]): T =
  elements match {
    case List() => throw new IllegalArgumentException("empty")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListImpParm(rest)(ordered)
```

```
        if (ordered(x) > maxRest) x
        else maxRest
    }

// because patter is so common, the standard library provides implicit 'orderer'
// methods for many common types, which is why you can use 'maxListImpParm' with:
maxListImpParm(List(1, 5, 10, 3))    // compiler inserts 'orderer' function for Ints
maxListImpParm(List(1.5, 5.2, 10.7, 3.22323))    // for Doubles
maxListImpParm(List("one", "two", "three"))       // for String

/*
 * Because elements must always be provided explicitly in any invocation of
 * maxListImpParm, the compiler will know T at compile time, and can therefore
 * determine whether an implicit definition of type T => Ordered[T] is in
 * scope. If so, it can pass in the second parameter list, 'orderer', implicitly.
 */
```

495 - A style rule for implicit parameters

- it is best to use a custom named type in the types of implicit parameters (e.g. in the `Prompt` example, the type of `prompt` and `drink` was not `String`, but `PreferredPrompt` and `PreferredDrink`)
- use at least one role-determining name within the type of an implicit parameter (in our case `Ordered`)

495 - View bounds

- when you use `implicit` on a parameter, then not only will the compiler try to supply that parameter with an implicit value, but it will also use that parameter as an available implicit in the body of the method:

```
def maxList[T](elements: List[T])(implicit orderer: T => Ordered[T]): T =
  elements match {
    case List() => throw new IllegalArgumentException("empty")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)    // '(orderer)' is implicit
      if (x > maxRest) x             // 'orderer(x)' is implicit
      else maxRest
  }

/*
 * 1. compiler sees that types don't match (e.g. 'x' of type 'T' doesn't have '>' method)
 * 2. compiler looks for implicit conversions to repair the code
 * 3. it finds 'orderer' in scope and converts the code to 'orderer(x) > maxRest'
 * 4. it also converts 'maxList(rest)' to 'maxList(rest)(orderer)'
 * 5. after these two implicit insertions the code fully type checks
 *
 * All that happens without a single mention of the 'orderer' parameter in the body, thus
 * all uses of 'orderer' are implicit
 */
```

- because this pattern is so common, Scala lets you leave out the name of this parameter and shorten the method header by using a **view bound:**

```
def maxList[T <% Ordered[T]](elements: List[T]): T =
```

```
elements match {
  case List() => throw new IllegalArgumentException("empty")
  case List(x) => x
  case x :: rest =>
    val maxRest = maxList(rest)    // '(orderer)' is implicit
    if (x > maxRest) x             // 'orderer(x)' is implicit
    else maxRest
}
```

- you can think of `T <% Ordered[T]` as saying "I can use any T, so long as T can be treated as an Ordered[T]", which is different from "T is an Ordered[T]", as **upper bound**, `T <: Ordered[T]`, would say
- so even though class `Int` is not a subtype of `Ordered[Int]`, we can still pass a `List[Int]` to `maxList`, so long as an implicit conversion from `Int` to `Ordered[Int]` is available
- if type `T` happens to already be an `Ordered[T]`, you can still pass a `List[T]` to `maxList`because the compiler will use an implicit **identity function**, declared in `Predef`:

```
implicit def identity[A](x: A): A = x   // simply returns received object
```

498 - When multiple conversions apply

- when multiple implicit conversions are in scope, compiler chooses the most specific one (e.g. if one of the conversions takes `String` and the other takes `Any`, the compiler will choose the one that takes a `String`)
- one implicit conversion is **more specific** than the other if one of the following applies:
- the argument type of the former is a subtype of the latter's
- both conversions are methods and the enclosing class of the former extends the enclosing class of the latter one

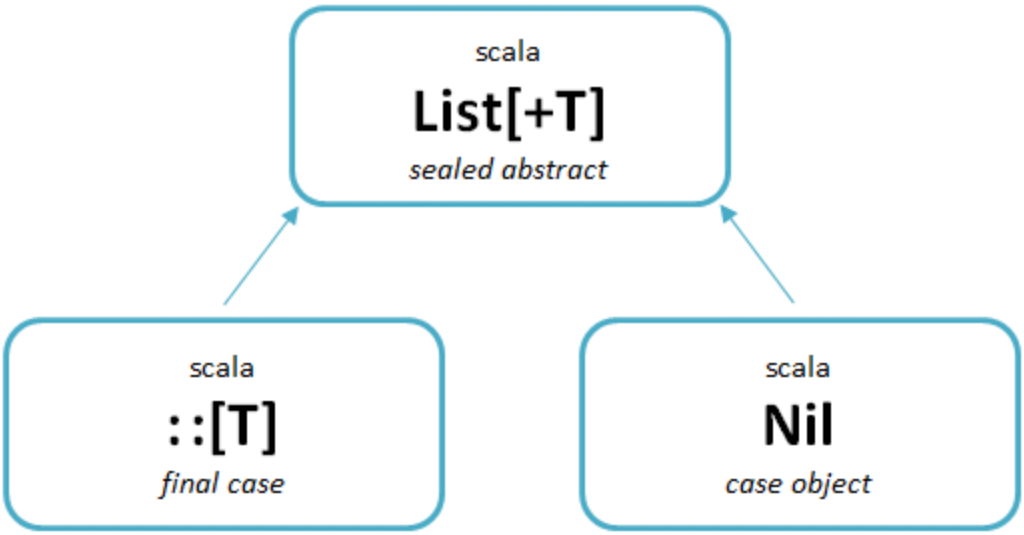
501 - Debugging implicits

- when you wonder why the compiler did not find an implicit conversion that you think should have been applied, it helps to write the conversion explicitly, which would possibly produce an error message so you'll know the reason why it was not applied
- if inserting the conversion explicitly make the error go away, then you know that insertion was prevented by one of the rules (often Scope rule)
- `-Xprint:typer` option tells the compiler to show what the code looks like after all implicit conversions have been added by the type checker
- implicits can make code confusing if used too frequently, thus, before writing a new implicit conversion, first try to achieve the same effect using inheritance, mixin composition or method overloading

Implementing Lists

503 - The List class in principle

- lists are not built-in as a language construct in Scala, they are defined by an abstract class `scala.List`, which comes with 2 subclasses, `Nil` and `::`




```
package scala
abstract class List[+T] {    // you can assign 'List[Int]' to var of type 'List[Any]'

   // 3 main methods are abstract in class 'List', and concrete in classes 'Nil' and '::'
   def isEmpty: Boolean
   def head: T
   def tail: List[T]
}
```

- **The `Nil` object**
- defines an empty list
- inherits from type `List[Nothing]`
- because of covariance, `Nil` is compatible with every instance of the `List` type

```
case object Nil extends List[Nothing] {
   override def isEmpty = true
   def head: Nothing = throw new NoSuchElementException("head of empty list")
   def tail: List[Nothing] = throw new NoSuchElementException("tail of empty list")
}
```

- **The `::` object**
- pronounced **cons**, represents non-empty lists
- the pattern `x :: xs` is treated as `::(x, xs)`, which is treated as `xs.::(x)`

```
// idealized implementation (the real deal on page 511)
final case class ::[T](hd: T, tl: List[T]) extends List[T] {
   def head = hd
   def tail = tl
   override def isEmpty: Boolean = false
}

// since definitions of 'head' and 'tail' simply return the corresponding param, we can
```

```
// write the code so that it directly uses the parameters as implementations of the
// abstract methods 'head' and 'tail' that were inherited from class 'List'
final case class ::[T](head: T, tail: List[T]) extends List[T] {
  override def isEmpty: Boolean = false
}
// this works because every 'case class' param is implicitly also a field
// as if param declaration was prefixed with the 'val' keyword
```

- **Some more methods**
- all other `List` methods can be elegantly written using the basic three, e.g.:

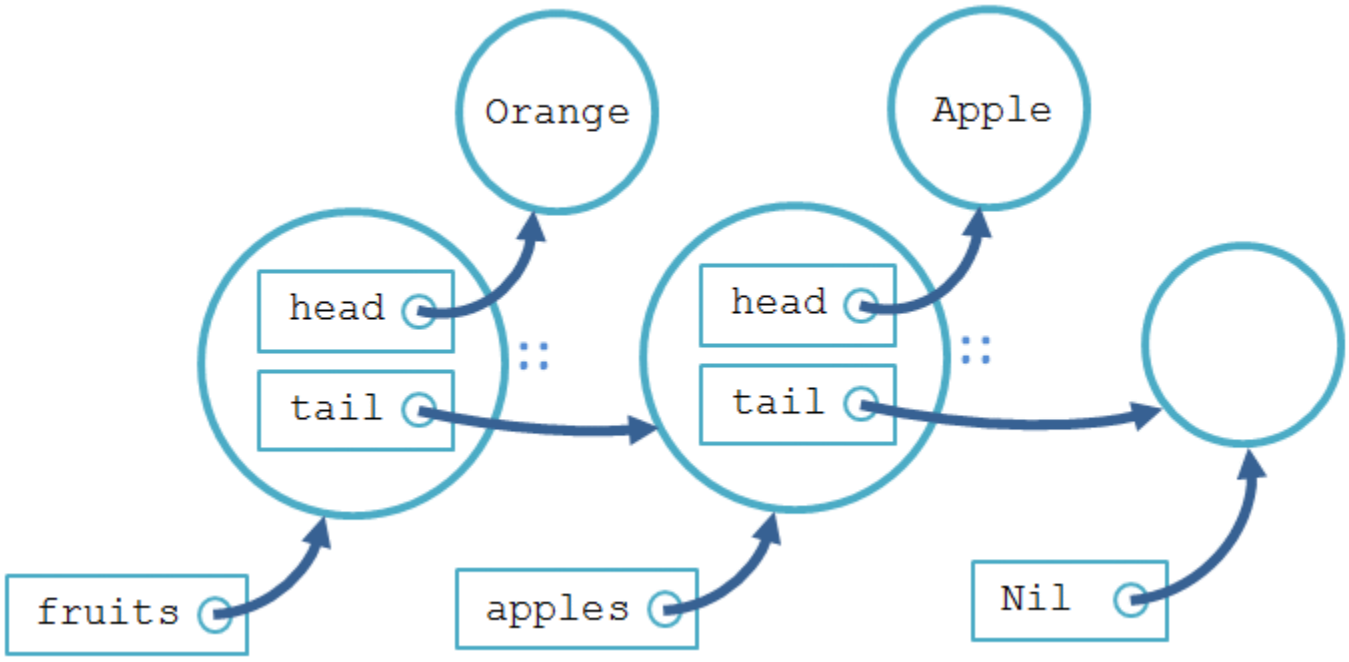
```
def length: Int =
  if (isEmpty) 0 else 1 + tail.length

def drop(n: Int): List[T] =
  if (isEmpty) Nil
  else if (n <= 0) this
  else tail.drop(n - 1)

def map[U](f: T => U): List[U] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)
```

507 - List construction

- `::` method should take an element value and yield a new list

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit

val apples = new Apple :: Nil          // apples: List[Apple]
val fruits = new Orange :: apples    // fruits: List[Fruit] - most precise common supertype

// this flexibility is obtained by defining the 'cons' method as:
def ::[U >: T](x: U): List[U] = new scala.::(x, this)

// the method is itself polymorphic:
// 'U' is constrained to be a supertype of the list element type 'T'
```

- when the code above is executed the result type is widened as necessary to include the types of all list elements



- first, the type parameter `U` of `::` is instantiated to `Fruit`
- the lower-bound constraint of `U` is satisfied, because the list `apples` has type `List[Apple]` and `Fruit` is supertype of `Apple`
- the argument to the `::` is `new Orange`, which conforms to type `Fruit`
- therefore, the method is type-correct with result type `List[Fruit]`

```
def :::[U >: T](prefix: MyList[U]): MyList[U] =
   if (prefix.isEmpty) this
   else prefix.head :: prefix.tail ::: this

// the infix operations can be expanded to equivalent method calls:
prefix.head :: prefix.tail ::: this
   // equals (because '::' and ':::' are right-associative)
prefix.head :: (prefix.tail ::: this)
   // equals (because '::' binds to the right)
(prefix.tail ::: this).::(prefix.head)
   // equals (because ':::' binds to the right)
this.:::(prefix.tail).::(prefix.head)
```

509 - The `ListBuffer` class

- the typical access pattern for a list is recursive, e.g. to increment every element without using `map`:

```
def incAll(xs: List[Int]): List[Int] = xs match {
   case List() => List()
   case x :: xs1 => x + 1 :: incAll(xs1)   // not tail recursive (call is inside '::')
}
```

- since the function is not tail recursive, each call allocates a new stack frame

- this means that you cannot use `incAll` on list with more than 30k to 50k elements
- **list buffers** let you accumulate the elements of a list
- **ListBuffer** is a class in package `scala.collection.mutable`

```
// increment all elements of a list using 'ListBuffer':
def incAll(xs: List[Int]): List[Int] = {
  val buf = new ListBuffer[Int]
  for (x <- xs) buf += x + 1
  buf.toList
}
// both '+=' and 'toList' take constant time
```

511 - The `List` class in practice

- most methods in the real implementation of class `List` avoid recursion (even if it's tail call optimized) and use loops with list buffers instead

```
// the real implementation of 'map' method:
final override def map[U](f: T => U): List[U] = {
  val b = new ListBuffer[U]
  var these = this
  while (!these.isEmpty) {    // highly efficient
    b += f(these.head)
    these = these.tail
  }
  b.toList
}
```

- a tail recursive implementation would be similarly efficient, but a general recursive implementation would be slower and less scalable

```
// the real implementation of '::' method:
final case class ::[U](hd: U, private[scala] var tl: List[U]) extends List[U] {
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}

// 'tl' is a 'var' - possible to modify the tail of a list after it's constructed
// 'private[scala]' - accessible only from within 'scala' package
// the client code outside 'scala' package can neither read nor write 'tl'
// 'ListBuffer', since it is contained in subpackage of 'scala', can access 'tl' field
```

- the elements of a list buffer are represented as a list and appending new elements involves modification of 'tl' field of the last '::' cell in that list

```
// 'ListBuffer' implementation:
package scala.collection.mutable
final class ListBuffer[T] extends Buffer[T] {
  private var start: List[T] = Nil
  private var last0: ::[T] = _
```

```
    private var exported: Boolean = false
    // ...

// start    - points to the list of all elements stored in the buffer
// last0    - points to the last :: cell in that list
// exported - indicates whether the buffer has been turned into a list using 'toList'

override def toList: List[T] = {
   exported = !start.isEmpty
   start
}
// very efficient, since it doesn't copy the list which is stored in list buffer

// once a list is returned from 'toList', it must be immutable, but appending to the
// 'last0' will modify the 'start' list
// to maintain correctness, we work on a fresh list instead:
override def += (x: T) {
   if (exported) copy()    // maintain correctness
   if (start.isEmpty) {
      last 0 = new scala.::(x, Nil)
      start = last0
   } else {
      val last1 = last0
      last0 = new scala.::(x, Nil)
      last1.tl = last0
   }
}

// most use cases add elements incrementally and then do one 'toList' call at the end
```

513 - Functional on the outside

- lists are purely functional on the outside, but have an imperative implementation using list buffers on the inside
- a typical strategy in Scala programming is to combine purity with efficiency by carefully delimiting the effects of impure operations
- Scala opts for pervasive sharing and no mutation for lists
- `ListBuffer` still allows you to build up lists imperatively and incrementally

For Expressions Revisited

- all `for` expressions that `yield` a result are translated by the compiler into combination of higher-order methods `map`, `flatMap` and `withFilter`
- all `for` expressions without `yield` are translated into combination of `withFilter` and `foreach`

518 - For expressions

- generally, a `for` expression is of the form `for (seq) yield expr`, where `seq` is a sequence of **generators**, **definitions** and **filters** with semicolon between successive elements

```
for (p <- persons; n = p.name; if (n startsWith "To")) yield n

// if you replace parentheses with curly braces, semicolons become optional:
for {
   p <- persons                 // generator
```

```
  n = p.name                  // definition (has the same effect as 'val' definition)
  if (n startsWith "To")   // filter
} yield n
```

519 - The n-queens problem

- a particularly suitable application area of for expressions are combinatorial puzzles
- an example of such a puzzle is the n-queens problem, where 'n' queens are supposed to be placed on a 'n x n' board, each queen in its own row, such that no two queens check one another
- the goal is to find all possible solutions that match the given requirements:

```
def queens(n: Int): List[List[(Int, Int)]] = {
  def placeQueens(k: Int): List[List[(Int, Int)]] =
    if (k == 0)
      List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
      } yield queen :: queens

  placeQueens(n)
}

def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._2 == q2._2 ||   // in the same column
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs   // in diagonal


def printSolutions(tbls: List[List[(Int, Int)]]) = {
  def printSolution(row: List[(Int, Int)]) = {
    val len = row.head._1
    println("_" * (len * 2))
    val tbl =
      for {
        pos <- row.reverse
        col <- 1 to len
        pipe = if (col == 1) "|" else ""
        mark = if (col == pos._2) "Q|" else "_|"
        nl = if (col == len) "\n" else ""
      } print(pipe + mark + nl)
  }
  for (tbl <- tbls) printSolution(tbl)
}
```

522 - Querying with `for` expressions

- the `for` notation is essentially equivalent to common operations of database query languages

```scala
// a db table 'books' might be represented as a list of books:
case class Book(title: String, authors: String*)

val books = List[Book] =
  List(
    Book(
      "Essential JavaScript design patterns", "Addi Osmani"),
    Book(
      "Developing backbone.js applications", "Addi Osmani"),
    Book(
      "Effective JavaScript", "Dave Herman"),
    Book(
      "JavaScript: The good parts", "Douglas Crockford"),
    Book(
      "AngularJS", "Brad Green", "Shyam Seshadri"),
    Book(
      "Taming text", "Grant S. Ingersoll", "Thomas S. Morton", "Andrew L. Farris"),
    Book(
      "Graph Databases", "Ian Robinson", "Jim Webber", "Emil Eifrem"),
    Book(
      "Node.js in action", "Mike Cantelon", "TJ Holowaychuk", "Nathan Rajlich"),
    Book(
      "ClojureScript up and running", "Stuart Sierra", "Luke VanderHart")
  )

// to find the titles of all books whose author's first name starts with "A":
val aAuthors =
  for (b <- books; a <- b.authors if a startsWith "A")
  yield b.title

// to find the titles of all books that have "JavaScript" in title
val js =
  for (b <- books if (b.title indexOf "JavaScript") >= 0)
  yield b.title

// to find the names of all authors that have written at least 2 books
val two =
  for (b1 <- books;
       b2 <- books if b1 != b2;
       a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
  yield a1

def removeDuplicates[A](xs: List[A]): List[A] = {
  if (xs.isEmpty) xs
  else {
    // take head and compare with all in tail
    // then repeat the same thing with tail
    xs.head :: removeDuplicates(
      for (x <- xs.tail if x != xs.head) yield x
    )

   // the same thing with filter
   // remove from tail if element equals head
    xs.head :: removeDuplicates(
      xs.tail filter (x => x != xs.head)
    )
  }
```

```
}
```

524 - Translation of `for` expressions

- how the compiler translates `for` expressions to higher-order function calls
- **Translating `for` expressions with one generator**

```
// a simple 'for' expression:
for (x <- expr1) yield expr2
// is translated to:
expr1.map(x => expr2)
```

- **Translating `for` expressions starting with a generator and a filter**

```
// a 'for' expression that combine a leading generator with some other elements:
for (x <- expr1 if expr2) yield expr3
// is translated to 'for' with one less element:
for (x <- expr1 withFilter (x => expr2)) yield expr3
// and then to:
expr1 withFilter (x => expr2) map (x => expr3)

// the same translation scheme applies if there are further elements following the filter
// if 'seq' is an arbitrary sequence of generators, definitions and filters, then:
for (x <- expr1 if expr2; seq) yield expr3
// is translated to:
for (x <- expr1 withFilter expr2; seq) yield expr3
// and then translation continues with the second expression which is shorter by one elem
```

- **Translating `for` expressions starting with two generators**

```
for (x <- expr1; y <- expr2; seq) yield expr3
// is translated to:
expr1.flatMap(x => for (y <- expr2; seq) yield expr3)
// the inner 'for' expression is also translated with the same rules

// the previous "query" example:
for (b1 <- books; b2 <- books if b1 != b2;
      a1 <- b1.authors; a2 <- b2.authors if a1 == a1)
yield a1

// is translated to:
books flatMap (b1 =>
  books withFilter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
      b2.authors withFilter (a2 => a1 == a2) map (a2 =>
        a1))))
```

- these were all examples where generators bind simple variables (as opposed to patterns) and with no definitions
- **Translating patterns in generators**
- the translation scheme becomes more complicated if the left hand side of generator is a pattern ('pat')

```
// if instead if simple variable tuple appears:
for ((x1, ..., xn) <- expr1) yield expr2
// translates to:
expr1.map { case (x1, ..., xn) => expr2 }

// if a single pattern is involved:
for (pat <- expr1) yield expr2
// translates to:
expr1 withFilter {
  case pat => true
  case _ => false
} map {
  case pat => expr2
}

// the generated items are first filtered with pattern matching
// and only the remaining ones are mapped
// so it's guaranteed that a pattern-matching generator will never throw a 'MatchError'
```

- **Translating definitions**

```
// if a 'for' expression contains embedded definitions:
for (x <- expr1; y = expr2; seq) yield expr3
// is translated to:
for ((x, y) <- for (x <- expr1) yield (x, expr2); seq)
yield expr3

// 'expr2' is evaluated every time a new 'x' value is generated
// which is necessary because 'expr2' might depend on 'x'
// so it's not a good idea to have definitions in 'for' expressions that do not refer
// to variables bound by some preceding generator, because reevaluating such
// expressions is wasteful, e.g. instead of:
for (x <- 1 to 1000; y = expensiveComputationNotInvolvingX)
yield x * y
// it's better to write:
val y = expensiveComputationNotInvolvingX
for (x <- 1 to 1000) yield x * y
```

- **Translating `for` loops**
- the translation of `for` loops that perform a side effect without returning anything is similar, but simpler than `for` expressions

```
// wherever the previous translations used 'map' or 'flatMap', we use 'foreach':
for (x <- expr1) body
// translates to:
expr1 foreach (x => body)

// or slightly larger example:
for (x <- expr1; if expr2; y <- expr3) body
// translates to:
expr1 withFilter (x => expr2) foreach (x =>
  expr3 foreach (y => body))

// for example, summing up all elements of a matrix represented as list of lists:
```

```
var sum = 0
for (xs <- xss; x <- xs) sum += x
// is translated into two nested 'foreach' applications:
var sum = 0
xss foreach (xs =>
  xs foreach (x =>
    sum += x))
```

528 - Going the other way

- every application of a `map`, `flatMap` and `withFilter` can be represented as a `for` expression

```
object ReversedTranslationDemo {

  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x

  def main(args: Array[String]): Unit = {
    val xs = List(1, 2, 3, 4)
    def f1 = (x: Int) => x + 1
    def f2 = (x: Int) => (x. to (x + 1)).toList
    def f3 = (x: Int) => x % 2 == 0

    val mapped = map(xs, f1)
    val flatmapped = flatMap(xs, f2)
    val filtered = filter(xs, f3)

    println(mapped)      // List(2, 3, 4, 5)
    println(flatmapped)  // List(1, 2, 2, 3, 3, 4, 4, 5)
    println(filtered)    // List(2, 4)
  }
}
```

529 - Generalizing `for`

- it is possible to apply `for` notation to every type that defines `map`, `flatMap`, `withFilter` or `foreach`
- if a type defines:
- just `map`, it allows `for` expressions with a single generator
- `map` and `flatMap` - more than one generator
- `foreach` - it allows `for loops` (with single and multiple generators)
- `withFilter` - it allows filter expressions starting with an `if` inside `for`
- the translation happens before type checking, which allows maximal flexibility, because it is only required that the result of expansion type checks
- Scala defines no typing rules for `for` expressions and doesn't require that methods `map`, `flatMap`, `withFilter` and `foreach` to have any particular type signatures
- nevertheless, there is a typical setup that captures most common intention of the higher order methods to which `for` expressions translate:

```
// a class that would be used for a collection 'C' of elements 'A':
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def withFilter(p: A => Boolean): C[A]
  def foreach(b: A => Unit): Unit
}
```

The Scala Collections API

534 - Mutable and immutable collections

- there are 4 packages with collection types:
- **scala.collection** - may be changed by other parties in run time
- **scala.collection.immutable** - collection changes in place
- **scala.collection.mutable** - guaranteed to be immutable for everyone
- **scala.collection.generic** - provide building blocks for implementing collections
- typically, collection classes defer the implementation of some of their operations to classes in `generic`

535 - Collections consistency

```
// the most important collection classes:
Traversable
  Iterable
    Seq
      IndexedSeq
        Vector
        ResizableArray
        GenericArray
      LinearSeq
        MutableList
        List
        Stream
      Buffer
        ListBuffer
        ArrayBuffer
    Set
      SortedSet
        TreeSet
      HashSet (mutable)
      LinkedHashSet
      HashSet (immutable)
      BitSet
      EmptySet, Set1, Set2, Set3, Set4
    Map
      SortedMap
        TreeMap
      HashMap (mutable)
      LinkedHashMap (mutable)
      HashMap (immutable)
      EmptyMap, Map1, Map2, Map3, Map4
```

- there is a quite a bit commonality shared by all these classes, e.g. every kind of collection can be created by the same uniform syntax:

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25)
Set(Color.Red, Color.Green, Color.Blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
```

537 - Trait `Traversable`

- on top of the collection hierarchy
- its only ***abstract*** operation is `foreach`:

```
def foreach[U](f: Elem => U)    // 'U` - arbitrary result type
```

- collection classes that mix in `Traversable`, just need to implement the `foreach` method, all other methods can be inherited from `Traversable`
- `foreach` is supposed to traverse all elements and apply a given operation, `f`, to each element
- `f` is invoked only because of its side effects (result of `f` is discarded)

The following table lists all concrete methods of `Traversable`:

- **Abstract method**
- `xs foreach f` Executes function f for every element of xs

- **Addition**
- `xs ++ ys` A collection consisting of the elements of both xs and ys

- **Maps**
- `xs map f` The collection obtained from applying f to every element of xs
- `xs flatMap f` The collection obtained by applying f to every element of xs and concatenating the results
- `xs collect f` The collection obtained by applying partial function f to every element in xs for which it is defined and collecting the results

- **Conversions**
- `xs.toArray` Converts the collection to an array
- `xs.toList` Converts the collection to a list
- `xs.toIterable` Converts the collection to an iterable
- `xs.toSeq` Converts the collection to a sequence
- `xs.toIndexedSeq` Converts the collection to an indexed sequence
- `xs.toStream` Converts the collection to a stream (a lazily computed sequence)
- `xs.toSet` Converts the collection to a set
- `xs.toMap` Converts the collection of key/value pairs to a map

- **Copying**
- `xs copyToBuffer buf` Copies all elements to buffer 'buf'
- `xs copyToArray(arr, s, len)` Copies at most 'len' elements of 'arr', starting at 's'

- **Element retrieval**
- `xs.head` Retrieves the first element of the collection
- `xs.headOption` The first element of xs in an option value, or None if xs is empty
- `xs.last` The last element of the collection (or some elem. if no order)
- `xs.lastOption` The last element of xs in an option value, or None if xs is empty
- `xs find p` An option containing the first element in xs that satisfies p

- **Subcollections**
- `xs.tail` Returns the rest of the collection (except xs.head)
- `xs.init` The rest of the collection except xs.last
- `xs slice (from, to)` Collection of elements from 'from', up to and excluding 'to'
- `xs take n` First n elements (or some elements if no order is defined)
- `xs drop n` The rest of collection (except xs take n)
- `xs takeWhile p` The longest prefix of elements that satisfy p
- `xs dropWhile p` The collection without prefix that satisfies p
- `xs filter p` The collection of all elements that satisfy p
- `xs withFilter p` A non-strict filter
- `xs filterNot p` The collection of all elements that do not satisfy p

- **Subdivisions**
- `xs splitAt n` Splits xs returning pair of collections (xs take n, xs drop n)
- `xs span p` Splits xs returning (xs takeWhile p, xs dropWhile p)
- `xs partition p` Splits on (xs filter p, xs filterNot p)
- `xs groupBy f` Partitions xs into a map of collections according to function f

- **Element conditions**
- `xs forall p` A boolean indicating whether all elements satisfy p
- `xs exists p` A boolean indicating whether p holds for at least one element
- `xs count p` The number of elements in xs that satisfy the predicate p

- **Folds**
- `(z /: xs)(op)` Applies operation op between successive elements, going left to right, starting with z
- `(xs :\ z)(op)` Applies operation op between successive elements, going right to left, starting with z
- `xs.foldLeft(z)(op)` Same as (z /: xs)(op)
- `xs.foldRight(z)(op)` Same as (xs :\ z)(op)
- `xs reduceLeft op` Applies binary operation op between successive elements of non-empty collection xs, going left to right
- `xs reduceRight op` Applies binary operation op between successive elements of non-empty collection xs, going right to left

- **Specific folds**
- `xs.sum` The sum of all numeric element values of xs
- `xs.product` The product of all numeric element values of xs
- `xs.min` The minimum of the ordered element values of xs
- `xs.max` The maximum of the ordered element values of xs

- **Strings**

- `xs addString (b, start, sep, end)` Adds a string to StringBuilder b that allows all elems between sep enclosed in strings start and end (start, sep and end are all optional)
- `xs mkString (start, sep, end)` Converts the collection to a string that shows all elems between sep enclosed in strings start and end (start, sep and end are optional)
- `xs.stringPrefix` The collection name returned from xs.toString

- **Views**
- `xs.view` Produces a view over xs
- `xs view (from, to)` Produces a view that represents elems in some index range

542 - Trait `Iterable`

- all methods are defined in terms of an abstract method `iterator`, which yields the collection's elements one by one
- the `foreach` method from trait `Traversable` is implemented in `Iterable`:

```
// the actual implementation
def foreach[U](f: Elem => U): Unit = {
  val it = iterator
  while (it.hasNext) f(it.next())
}
```

- many subclasses of `Iterable` override this standard implementation, because they can provide more efficient implementation (performance matters, since it is the basis for all operations in `Traversable`)
- two more methods exist in `Iterable` that return iterators `grouped`** and **`sliding` (they return subsequences of elements, whose maximal size is given as an argument)
- `grouped` chunks its elements into increments, whereas `sliding` yields a sliding window over the elements:

```
val xs = List(1, 2, 3, 4, 5)
val git = xs grouped 3   // returns non-empty Iterator[List[Int]]
git.next()   // List(1, 2, 3)
git.next()   // List(4, 5)

val sit = xs sliding 3   // returns non-empty Iterator[List[Int]]
sit.next()   // List(1, 2, 3)
sit.next()   // List(2, 3, 4)
sit.next()   // List(3, 4, 5)
```

The summary of operations in trait `Iterable`:

- **Abstract method**
- `xs.iterator` Iterator that yields every element in the same order as `foreach`

- **Other iterators**
- `xs grouped size` Iterator that yields fixed-size chunks of the collection
- `xs sliding size` Iterator that yields a sliding fixed-size window of elements

- **Subcollections**
- `xs takeRight n` Collection consisting of last n elems of xs (or arbitrary)
- `xs dropRight n` The rest of the collection except `xs takeRight n`

- **Zippers**
- `xs zip ys` An iterable of pairs of corresponding elems from xs and ys
- `xs zipAll (ys, x, y)` An iterable of pairs, where shorter sequence is extended to match the longer one by appending elements x or y
- `xs.zipWithIndex` An iterable of pairs from xs with their indices

- **Comparison**
- `xs sameElement ys` Tests whether xs and ys have same elements in the same order

544 - Why both `Traversable` and `Iterable`

- often times it's easier or more efficient to implement `foreach` than `iterator`:

```
// a class hierarchy for binary trees that have integers at the leaves:
sealed abstract class PlainTree
case class Branch(left: Tree, right: Tree) extends Tree
case class Node(elem: Int) extends Tree

// now assume you want to make trees traversable:
sealed abstract class Tree extends Traversable[Int] {    // O(N + N-1)

   def foreach[U](f: Int => U) = this match {
      case Node(elem) => f(elem)
      case Branch(l, r) => l foreach f; r foreach f
   }
}

// less efficient IterTree, to make iterable
sealed abstract class IterTree extends Iterable[Int] {
   def iterator: Iterator[Int] = this match {
      case IterNode(elem) => Iterator.single(elem)
      case IterBranch(l, r) => l.iterator ++ r.iterator
   }
}

case class IterBranch(left: IterTree, right: IterTree) extends IterTree
case class IterNode(elem: Int) extends IterTree

// IterTree is much less efficient since iterator concatenation method ++
// makes the traversal O(N log(N))
```

546 - Subcategories of `Iterable`

- in the Scala inheritance hierarchy, bellow `Iterable`, there are three traits: `Seq`, `Set` and `Map`
- the common characteristic is that they all implement the `PartialFunction` trait, with its `apply`and `isDefinedAt` methods
- for sequences, `apply` is positional indexing (elems are numbered from 0):

```
Seq(1, 2, 3)(1) == 2
```

- for sets, `apply` is a membership test:

```
Set('a', 'b', 'c')('b') == true
```

- for maps, `apply` is a selection:

```
Map('a' -> 1, 'b' -> 10, 'c' -> 100)('b') == 10
```

546 - The sequence traits `Seq`, `IndexedSeq` and `LinearSeq`

- **`seq`** trait represents a kind of `iterable` that has a `length` and whose elements have fixed index positions, starting from `0` up to `length - 1`
- the `update` method is only available on mutable sequences, since it changes the sequence in place
- the `updated` method always returns a new sequence and it is available on all sequences
- each `Seq` trait has two subtraits, `LinearSeq` and `IndexedSeq`, which do not add any new operations, but each offers different performance characteristics
- a linear sequence (e.g. `List` or `Stream`) has efficient `head` and `tail` operations
- an indexed sequence (e.g. `Array` or `ArrayBuffer`) has efficient `apply`, `length` and (if mutable) `update` operations
- the `Vector` class provides an interesting compromise between indexed and linear access, since it has both effectively constant time indexing overhead and constant time linear access overhead

Operations in trait `Seq`:

- **Indexing and length**
- `xs(i)` (or `xs apply i`) The element of xs at index i
- `xs isDefinedAt i` Tests whether i is contained in xs.indices
- `xs.length` The length of the sequence (same as `size`)
- `xs.lengthCompare ys` Returns -1 if xs is shorter than ys, +1 if it's longer, and 0 if they have the same length. Works even if one of sequences is infinite
- `xs.indices` The index range of xs, extending from 0 to xs.length - 1

- **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` The first index of xs that begins the ys sequence
- `xs lastIndexOfSlice ys` The last index of xs that begins the ys sequence
- `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` The length of the longest prefix in xs that all satisfy p

- **Additions**
- `xs += xs` A new sequence consisting of x prepended to xs
- `xs :+ x` A new sequence consisting of x appended to xs
- `xs padTo (len, x)` The sequence resulting from appending the value x to xs until length len is reached

- **Updates**

- `xs patch (i, ys, r)` The sequence resulting from replacing r elements of xs starting with i by the patch ys
- `xs updated (i, x)` A copy of xs with the element at index i replaced with x
- `xs(i) = x` (or xs.update(i, x) - available only for mutable.Seqs) Changes the element of xs at index i to y

- **Sorting**
- `xs.sorted` A new sequence obtained by sorting xs using the standard ordering of the element type of xs
- `xs sortWith lessThan` A new sequence obtained by sorting xs using lessThan as comparison operation
- `xs sortBy f` A new sequence obtained by sorting xs in a way that the function f is first applied to two elements and then results are compared

- **Reversals**
- `xs.reverse` A 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` A sequence obtained by mapping f over elements of xs in reverse order

- **Comparisons**
- `xs startsWith ys` Tests whether xs starts with sequence ys
- `xs endsWith ys` Tests whether xs ends with sequence ys
- `xs contains x` Tests whether xs has an element equal to x
- `xs containsSlice ys` Tests whether xs has a continuous subsequence ys
- `(xs corresponds ys)(p)` Tests whether corresponding elements of xs and ys satisfy the binary predicate p

- **Multiset operations**
- `xs intersect ys` The multi-set intersection of xs and ys that preserves the order of elements in xs
- `xs diff ys` The multi-set difference of xs and ys that preserves the order of elements in xs
- `xs union ys` (or xs ++ ys) Multiset union
- `xs.distinct` A subsequence of xs that contains no duplicates

550 - Buffers

- buffers allow not only updates of existing elements, but also element insertions, removals, and efficient additions of new elements at the end of the buffer
- buffers support element addition at the end and at the front, element insertions and element removal
- two most common buffer implementations are `ListBuffer` and `ArrayBuffer`

Operations in trait `Buffer`:

- **Additions**
- `buf += x` Appends element x to buffer buf and returns buf
- `buf += (x, y)` Appends given elements to buf
- `buf ++= xs` Appends all elements in xs to buf
- `x +=: buf` Prepends element x to buf
- `xs ++=: buf` Prepends all elements in xs to buf
- `buf insert (i, x)` Inserts element x at index i in buf
- `buf insertAll (i, xs)` Inserts all elements in xs at index i in buf

- **Removals**
- `buf -= x` Removes element x from buffer buf
- `buf remove i` Removes element at index i from buf
- `buf remove (i, n)` Removes n elements starting at index i from buf
- `buf trimStart n` Removes first n elements from buf
- `buf trimEnd n` Removes last n elements from buf
- `buf.clear()` Removes all elements from buf

- **Cloning**
- `buf.clone` A new buffer with the same elements as buf

551 - Sets

- Sets are iterables that contain no duplicate elements

Operations in trait `Set`:

- **Tests**
- `xs contains x` Tests whether x is an element of xs
- `xs(x)` Same as `xs contains x`
- `xs subsetOf ys` Tests whether xs is a subset of ys

- **Additions**
- `xs + x` The set containing all elements of xs as well as x
- `xs + (x, y, z)` The set containing all elements of xs as well as x, y and z
- `xs ++ ys` The set containing all elements of xs and of ys

- **Removals**
- `xs - x` The set containing all elements of xs except x
- `xs - (x, y, z)` The set containing all elements of xs except x, y and z
- `xs -- ys` The set containing all elements of xs except elements of ys
- `xs.empty` An empty set of the same class as xs

- **Binary operations**
- `xs & ys` The set intersection of xs and ys
- `xs intersect ys` Same as `xs & ys`
- `xs | ys` The set union of xs and ys
- `xs union ys` Same as `xs | ys`
- `xs &~ ys` The set difference of xs and ys
- `xs diff ys` Same as `xs &~ ys`

Operations in trait `mutable.Set`:

- **Additions**
- `xs += x` Adds x to xs as a side effect and returns xs
- `xs += (x, y, z)` Adds x, y and z to set xs and returns xs
- `xs ++= ys` Adds elements of ys to xs and returns xs
- `xs add x` Adds x to xs and returns true if x was not previously contained in the set, false if it was already in the set

- **Removals**
- `xs -= x` Removes x from xs and returns xs

- `xs -= (x, y, z)` Removes x, y and z from xs and returns xs
- `xs --= ys` Removes all elements from xs that are in ys and returns xs
- `xs remove x` Removes x from xs and returns true if x was previously contained in the set or false if it wasn't
- `xs retain p` Keeps only those elements in xs that satisfy predicate p
- `xs.clear()` Removes all elements from xs

- **Update**
- `xs(x) = b` (or `xs.update(x, b)`) If boolean argument b is true, adds x to xs, otherwise removes x from xs

- **Cloning**
- `xs.clone` A new mutable set with the same elements as xs
- mutable set also has `+`, `++`, `-` and `--` methods, but they are rarely used because they involve copying the set
- the current default implementation of a mutable set uses a hash table to store the set's elements

- the default implementation of an immutable set uses a representation that adapts to the number of element of the set:

- empty set is represented as a singleton
- sets of up to four elements are represented by a single object with elems as fields
- beyond 4 elements, immutable sets are represented as **hash tries**
- this decision results in more compact and efficient small (up to 4) immutable sets (compared to small mutable sets)

556 - Sorted sets

- `SortedSet` is a subtrait of `Set` in which elements are traversed in sorted order, regardless of the order in which elements were added to the set
- the default representation of a `SortedSet` is an ordered binary tree, maintaining the invariant that all elements in the left subtree of any node are smaller than all elements in the right subtree (thus, simple, in-order traversal yields elements in the ascending order)
- `immutable.TreeSet` uses a red-black tree implementation to maintain that order and at the same time keep the tree balanced

```
// to create an empty tree set, we may want to first specify the desired ordering:
val myOrdering = Ordering.fromLessThan[String](_ > _)    // scala.math.Ordering[String]

// then, to create an empty tree set with that ordering:
import scala.collection.immutable.TreeSet
TreeSet.empty(myOrdering)    // TreeSet()

// or we can leave out the ordering, but give an element type of the empty set
// in which case the default ordering will be used (ascending - (_ < _)):
val set = TreeSet.empty[String]

// if you make new sets from a tree set (e.g. by concatenation or filtering)
// in the new set, the elements will stay in the same order:
val numbers = set + ("one", "four", "eight")    // TreeSet(eight, four, one)

// sorted sets also support ranges of elements (including start and up to end, excluded):
```

```
numbers range ("eight", "one")   // TreeSet(eight, four)

// they also support 'from' method, which returns elements >= to argument received:
numbers from "four"   // TreeSet(four, one)
```

557 - Bit sets

- sets of non-negative integer elements, that are implemented as one or more words of packed bits
- internal representation uses an array of `Longs`, where the first long covers elements from 0 to 63, the second from 64 to 127, and so on
- for every long, each of its 64 bits is set to 1 if the corresponding element is contained in the set and otherwise it contains zero (the size of a bit set depends on the largest integer that's stored in it)
- if N is the largest integer, then the size of the set is N/64 Long words, or N/8 bytes, plus a small number of extra bytes that carry status information
- hence, bit sets are convenient for storing many small elements
- another advantage of bit sets is that operations `contains`, `+=` and `-=` are extremely efficient

557 - Maps

- `Iterables` of pairs of keys and values (mappings, associations)

Operations in trait `Map`:

- **Lookups**
- `ms get k` The value associated with key 'k' as an 'option', or 'None' if 'k' is not found
- `ms(k)` (or `ms apply k`) The value associated with key 'k', or a thrown exception if not found
- `ms getOrElse (k, d)` The value associated with key 'k', or the default value 'd' if not found
- `ms contains k` Tests whether 'ms' contains a mapping for key 'k'
- `ms isDefinedAt k` Same as `contains`

- **Additions and updates**
- `ms + (k -> v)` The map containing 'ms' and the mapping 'k -> v'
- `ms + (k -> v, l -> w)` The map containing 'ms' and given key value pairs
- `ms ++ kvs` The map containing 'ms' and all key value pairs of 'kvs'
- `ms updated (k, v)` Same as `ms + (k -> v)`

- **Removals**
- `ms - k` The map containing 'ms' except for any mapping of key 'k'
- `ms - (k, l, m)` The map containing 'ms' except for any mappings with the given keys
- `ms -- ks` The map containing 'ms' except for any mapping with a key in 'ks'

- **Subcollections**
- `ms.keys` An iterable containing each key of 'ms'
- `ms.keySet` A set containing each key in 'ms'
- `ms.keysIterator` An iterator yielding each key in ms
- `ms.values` An iterable containing each value associated with a key in 'ms'
- `ms.valuesIterator` An iterator yielding each value associated with a key in 'ms'

- **Transformation**
- `ms filterKeys p` A map view containing only those mappings in 'ms' where the key satisfies predicate 'p'

- `ms mapValues f` A map view resulting from applying function 'f' to each value associated with a key in 'ms'

Operations in trait `mutable.Map`:

- **Additions and updates**
- `ms(k) = v` (or `ms.update(k, v)`) Adds 'k -> v' as a side effect, overwriting any previous mapping of 'k'
- `ms += (k -> v)` Adds mapping 'k -> v' and returns the altered 'ms'
- `ms += (k -> v, l -> w)` Adds the given mappings to 'ms' and returns 'ms'
- `ms ++= kvs` Adds all mappings in 'kvs' to 'ms' and returns 'ms'
- `ms put (k, v)` Adds mapping 'k -> v' and returns any value previously associated with 'k' as an 'option'
- `ms getOrElseUpdate (k, d)` If key 'k' is defined, returns its value. Otherwise updates 'ms' with the mapping 'k -> d' and returns 'd'

- **Removals**
- `ms -= k` Removes mapping with key 'k' and returns 'ms'
- `ms -= (k, l, m)` Removes mappings with the given keys and returns 'ms'
- `ms --= ks` Removes all keys contained in 'ks' and returns 'ms'
- `ms remove k` Removes any mapping with key 'k' and returns any value previously associated with 'k' as an 'option'
- `ms retain p` Keeps only those mappings that have a key in satisfying predicate 'p'
- `ms clear()` Removes all mappings from 'ms'

- **Transformation and cloning**
- `ms transform f` Transforms all associated values in 'ms' with function 'f'
- `ms.clone` Returns a new mutable map with the same mappings as 'ms'
- same as with sets, mutable maps also support `+`, `-` and `updated`, but they are also rarely used since they involve copying of the mutable map

```
// getOrElseUpdate is useful for accessing maps that act as caches:
// if you were to have an expensive operation triggered by invoking a function 'f':
def f(x: String) = {
  println("taking my time slow."); Thread.sleep(100)
  x.reverse
}

// assume further that 'f' has no side-effects, so invoking it again with the same
// argument will always yield the same result
// in that case, you could save time by storing previously computed bindings of
// arguments and results of 'f' in a map, and only computing the result of 'f' if
// a result of an argument was not found there:
val cache = collection.mutable.Map[String, String]()   // Map()

// the more efficient version of function 'f':
def cachedF(s: String) = cache.getOrElseUpdate(s, f(s))

cachedF("ijk")
// taking my time
// String = kji

cachedF("ijk")
```

```
// String = kji

// the second argument to 'getOrElseUpdate' is "by-name", so the computation of f("ijk")
// is only performed if 'getOrElseUpdate' requires the value of its second argument
// which happens only if its first argument is not found in the map

// the alternative is to implement 'cachedF' directly, using just basic map operations
// but that would've taken more code:
def cachedF(arg: String) = cache get arg match {
  case Some(result) => result
  case None =>
    val result = f(arg)
    cache(arg) = result
    result
}
```

562 - Synchronized sets and maps

- if you need a thread-safe map, you could mix the `SynchronizedMap` trait into a map implementation, e.g. `HashMap`

```
import scala.collection.mutable.{Map, SynchronizedMap, HashMap}

object MapMaker {

  def makeMap(): Map[String, String] = {

    // a synthetic subclass of HashMap that mixes in SynchronizedMap trait
    // is generated, an instance of it is created and then returned
    new HashMap[String, String] with SynchronizedMap[String, String] {

      // if you ask a map to give you a key that doesn't exist you'll get
      // NoSuchElementException, but if you override 'default', you'll get
      // a value returned by 'default' method
      override def default(key: String) = "Why?"
    }
  }

  def main(args: Array[String]): Unit = {
    // because our map mixes in SynchronizedMap it may be used
    // by multiple threads at the same time
    val capital = makeMap
    capital ++= List("US" -> "Washington", "Croatia" -> "Zagreb")

    println(capital("Croatia"))       // Zagreb
    println(capital("New Zealand"))   // Why?

    capital += ("New Zealand" -> "Wellington")

    println(capital("New Zealand"))   // Wellington
  }
}
```

- regardless of `Synchronized` collections, you're encouraged to use immutable collections with Scala actors instead

564 - Concrete immutable collection classes

- **Lists**
- finite immutable sequences that provide constant time access to their first element and they have a constant time ***cons*** operation for adding a new element to the front
- most other operations take linear time (e.g. accessing "non-head" elements)

- **Streams**
- like a list, except that its elements are computed lazily
- because of it laziness, a stream can be infinitely long (only requested elements are computed)
- they have the same performance characteristics as lists
- whereas lists are constructed with the `::` operator, streams are constructed with `#::`:

```
val str = 1 #:: 2 #:: 3 #:: Stream.empty    // Stream[Int] = Stream(1, ?)

// the head of the stream is '1', and the tail has '2' and '3'
// the tail is not printed because it hasn't been computed yet
// 'toString' method of a stream is careful not to force any extra evaluation

// computing a Fibonacci sequence starting with the given two numbers
def fibFrom(a: Int, b: Int): Stream[Int] =
  a #:: fibFrom(b, a + b)

// if the function used '::' instead of '#::', it would cause an infinite recursion
// since it uses a stream, the right hand side is not evaluated until it is requested
val fibs = fibFrom(1, 2).take(7)    // Stream[Int] = Stream(1, ?)
fibs.toList                                 // List(1, 1, 2, 3, 5, 8, 13)
```

- **Vectors**
- introduced in Scala 2.8
- provide efficient access to elements beyond the head
- access to any element take "effectively constant time" (larger constant than list's head or array's element, but constant nonetheless)

```
// used as any other sequence:
val vec1 = scala.collection.immutable.Vector.empty    // Vector[Nothing] = Vector()
val vec2 = vec1 :+ 1 :+ 2    // Vector[Int] = Vector(1, 2)
val vec3 = 100 +: vec2       // Vector[Int] = Vector(100, 1, 2)
val third = vec3(0)          // Int = 100
```

- vectors are represented as broad, shallow trees, where every tree node contains up to 32 elements of the vector or up to 32 other tree nodes
- so vectors with up to 32 elements can be represented in a single tree node
- vectors with up to 32 * 32 (1024) elements can be represented with a single indirection
- 2^15 (approx 32.77k) elements can be stored within two hops from the root
- 2^20 (approx 1M) - 3 hops, 2^25 (approx 33.5M) - 4 hops, 2^30 (approx 1B) - 5 hops
- so for all vectors of up to 1.074B elements, an element selection involves up to five primitive array selections (thus constant time)

```
// we cannot change an element of a vector in place
// but we can use 'updated' method which returns a new vector that differs from
```

```
// the original vector only in a single element:
val vec = Vector(1, 2, 3)
vec updated (2, 4)    // Vector[Int] = Vector(1, 2, 4)
println(vec)             // Vector[Int] = Vector(1, 2, 3)
```

- like selection, functional vector updates also take "effectively constant time"
- implementation, to update an element in the middle of a vector, copies the node that contains the element and all nodes that point to it, starting from the root (so it creates between 1 and 5 nodes that each contain up to 32 elements or subtrees)
- that is certainly more expensive than in-place update of a mutable array, but it's still a lot cheaper than copying the whole vector
- because of this characteristics, vectors strike a good balance between fast random selections and fast random functional updates and are, thus, the current default implementation of immutable indexed sequences:

```
scala.collection.immutable.IndexedSeq(1, 2, 3)    // IndexedSeq[Int] = Vector(1, 2, 3)
```

- **Immutable stacks**
- `push`, `pop` and `top` all take constant time
- rarely used because their functionality is subsumed by lists

```
val stack = scala.collection.immutable.Stack.empty    // Stack[Nothing] = Stack()
val hasOne = stack.push(1)    // Stack[Int] = Stack(1)
stack          // Stack()
hasOne.top    // Int = 1
hasOne.pop    // Stack()  - returns the stack, not the popped element (like list's tail)
```

- **Immutable queues**

```
val empty = scala.collection.immutable.Queue[Int]()    // Queue[Int] = Queue()
val hasOne = empty.enqueue(1)                                  // Queue[Int] = Queue(1)
val has123 = hasOne.enqueue(List(2, 3))                  // Queue[Int] = Queue(1, 2, 3)
val (element, has23) = has123.dequeue    // element: Int = 1; has23: Queue(2, 3)
// dequeue returns a pair consisting of the element removed and the rest of the queue
```

- **Ranges**
- ordered sequence of integers that are equally spaced apart
- represented in constant space, since they can be defined by just 3 numbers: start, end and step, thus making most operations on ranges extremely fast
- to create a range, use predefined methods `to` and `by`:

```
1 to 3    // immutable.Range.Inclusive with immutable.Range.ByOne = Range(1, 2, 3)
5 to 14 by 3    // immutable.Range = Range(5, 8, 11, 14)
1 until 3    // immutable.Range = Range(1, 2)
```

- **Hash tries**

- a standard way to implement immutable sets and maps efficiently
- represented similar to vectors, in that they are also trees where every node has 32 elements or subtrees, but selection is done based on a hash code
- e.g. to find a given key in a map, we use the lowest five bits of the hash code of the key to select the first subtree, the next five bits for the next subtree, and so on
- selection stops once all elements stored in a node have hash codes that differ from each other in the bits that are selected so far, thus not all the bits of the hash code are always used
- strike a nice balance between reasonably fast lookups and reasonably efficient functional insertions (`+`) and deletions (`-`)
- sets and maps that contain less than five elements are stored as single objects that just contain the elements as fields

- **Red-black trees**
- a form of balanced binary trees where some nodes are designated "red" and others "black"
- like any other balanced binary tree, operations on them take ***log(n)***
- `TreeSet` and `TreeMap` use a red-black tree internally

```
val set = collection.immutable.TreeSet.empty[Int]    // TreeSet[Int] = TreeSet()
set + 1 + 3 + 3    // TreeSet[Int] = TreeSet(1, 3)
```

- they are also the standard implementation of `SortedSet`, because they provide efficient iterator that returns all elements of the set in sorted order

- **Immutable bit sets**
- represent collections of small integers as the bits of a larger integer, e.g. the bit set containing `3, 2 and 0` would be represented as integer `1101` in binary, which is `13` in decimal
- internally, they use an array of 64-bit Longs, where the first Long in the array is for integers 0 through 63, second for 64 to 127, and so on, thus they are very compact as long as the largest integer in the set is less than a few hundred or so
- testing for inclusion takes constant time
- adding an item to a set takes time proportional to the number of Longs in the array

```
val bits = scala.collection.immutable.BitSet.empty    // BitSet = BitSet()
val moreBits = bits + 3 + 4 + 4    // BitSet = BitSet(3, 4)
moreBits(3)    // true
moreBits(0)    // false
```

- **List maps**
- represents a map as a linked list of key-value pairs
- in general, operations might have to iterate through the entire list, thus taking time linear in the size of the map
- rarely used, since regular immutable maps are almost always faster
- the one exception is a case when a map is constructed in such a way that the first elements in the list are selected much more often than the other elements

```
val map = collection.immutable.ListMap(1 -> "one", 2 -> "two")
// immutable.ListMap[Int, java.lang.String] = Map((1, one), (2, two))
```

```
map(2)    // java.lang.String = two
```

571 - Concrete mutable collection classes

- **Array buffers**
- holds an array and a size
- most operations have the same speed as arrays' counterparts, because the operations simply access and modify the underlying array
- additionally, they support efficient addition of elements to the end, which take "amortized constant time", thus making them useful for building a large collections, as long as new items are always added to the end:

```
val buf = collection.mutable.ArrayBuffer.empty[Int]  // ArrayBuffer[Int] = ArrayBuffer()
buf += 1         // buf.type = ArrayBuffer(1)
buf += 10        // buf.type = ArrayBuffer(1, 10)
buf.toArray    // Array[Int] = Array(1, 10)

// buf.type - a singleton type
// means that the variable holds exactly the object referred to by buf
```

- **List buffers**
- like an array buffer, except that it uses a linked list internally instead of array
- if you plan to convert the buffer to a list once it's built up, use a list buffer instead of an array buffer

```
val buf = collection.mutable.ListBuffer.empty[Int]    // ListBuffer[Int] = ListBuffer()
buf += 1       // buf.type = ListBuffer(1)
buf += 10      // buf.type = ListBuffer(1, 10)
buf.toList    // List[Int] = List(1, 10)
```

- **String builders**
- imported in the default namespace, so they may be created: `new StringBuilder`

```
val buf = new StringBuilder    // StringBuilder = StringBuilder()
buf += 'a'           // StringBuilder(a)
buf ++= "bcdef"    // StringBuilder(a, b, c, d, e, f)
buf.toString       // String = abcdef
```

- **Linked lists**
- mutable sequences that consist of nodes that are linked with `next` pointers
- in most languages empty linked list would be represented with `null`, but in Scala, which makes even empty sequences support all sequence methods, empty linked list is encoded in a special way. Their `next` pointer points back to the node itself

```
// otherwise this would happen:
LinkedList.empty.isEmpty  // in Java something like this would throw NullPointerException
```

- **Double linked lists**

- the same as single linked lists, except that besides `next`, they have another mutable field, `prev`, which points to the element preceding the current node
- the main benefit of that additional link is that it makes element removal very fast

- **Mutable lists**
- a `MutableList` consists of a single linked list together with a pointer that refers to the terminal empty node of that list, which makes list append operation take constant time, because it avoids having to traverse the list in search for its terminal node
- it is the standard implementation of `mutable.LinearSeq`

- **Queues**
- instead of immutable queue's `enqueue` method, we use `+=` and `++=` to append
- also, on mutable queue, `dequeue` method just removes the head element from the queue and returns it

```
val q = new scala.collection.mutable.Queue[String]    // Queue[String] = Queue()
q += "a"                    // q.type = Queue(a)
q ++= List("b", "c")    // q.type = Queue(a, b, c)
q                                 // mutable.Queue[String] = Queue(a, b, c)
q.dequeue                  // String = a
q                                 // mutable.Queue[String] = Queue(b, c)
```

- **Array sequences**
- `ArraySeq` is a mutable sequence of fixed size, implemented as `Array[AnyRef]`
- used for its performance characteristics (array), when you want to create generic instances of a sequence, but do not know the type of elements and do not have a `ClassManifest` to provide at runtime

- **Stacks**
- works exactly the same as the immutable version, except that modifications happen in place

```
val stack = new scala.collection.mutable.Stack[Int]    // mutable.Stack[Int] = Stack()
stack.push(1)    // stack.type = Stack(1)
stack                 // mutable.Stack[Int] = Stack(1)
stack.push(2)    // stack.type = Stack(2, 1)
stack                 // mutable.Stack[Int] = Stack(2, 1)
stack.top          // Int = 2
stack                 // mutable.Stack[Int] = Stack(2, 1)
stack.pop          // Int = 2
stack                 // mutable.Stack[Int] = Stack(1)
```

- **Array stacks**
- an alternative implementation of a mutable stack, which is backed by an `Array` that gets resized as needed
- provides fast indexing and is slightly more efficient than a normal mutable stack

- **Hash tables**
- stores its elements in an underlying array, placing each item at a position in the array determined by the hash code of that item
- element addition takes constant time, so long as there isn't already another element in the array that has the same hash code

- since it's very fast (as long as the objects placed in them have a good distribution of hash codes), the default mutable map and set types are based on hash tables

```
val map = collection.mutable.HashMap.empty[Int, String]  // HashMap[Int, String] = Map()
map += (5 -> "html5")    // map.type = Map((5, html5))
map += (3 -> "css3")     // map.type = Map((5, html5), (3, css3))
map(5)                   // html5
map contains 2           // false
```

- iteration over a hash table is not guaranteed to occur in any particular order, it simply iterates through the underlying array in whichever order it happens to be
- to get a guaranteed iteration order, use a linked hash map or set instead, which is just like a regular hash map or set, except that it also includes a linked list of the elements in the order in which they were added

- **Weak hash maps**
- a special kind of hash map in which the garbage collector does not follow links from the map to the keys stored in it, which means that a key and its associated value will disappear from the map if there is no other reference to that key
- useful for tasks such as caching, where you want to reuse an expensive function's result if the function is called again on the same key
- if keys and function results were stored in a regular hash map, the map could grow without bounds, since no key would ever become eligible for garbage collection
- in `WeakHashMap`, as soon as a key object becomes unreachable, its entry is removed from the weak hash map
- implemented as a wrapper of `java.util.WeakHashMap`

- **Concurrent maps**
- can be safely accessed by several threads at once
- `ConcurrentMap` is a trait in collections library, whose current implementation is Java's `java.util.concurrent.ConcurrentMap`, which can be automatically converted to Scala map using the standard Java/Scala collection conversion

Operations in trait `ConcurrentMap`:

- `m putIfAbsent (k, v)` Adds key/value binding 'k -> v' unless 'k' exists in 'm'
- `m remove (k, v)` Removes entry for 'k' if it is currently mapped to 'v'
- `m replace (k, old, new)` Replaces value of key 'k' to 'new', if it is set to 'old'
- `m replace (k, v)` Replaces value of 'k' with 'v' if it was previously bound to some value

- **Mutable bit sets**
- slightly more efficient at updating than immutable ones, because they don't have to copy around Longs that haven't changed

```
val bits = scala.collection.mutable.BitSet.empty   // mutable.BitSet = BitSet()
bits += 1   // bits.type = BitSet(1)
bits += 3   // bits.type = BitSet(1, 3)
bits        // mutable.BitSet = BitSet(1, 3)
```

578 - Arrays

- a special kind of collection
- Scala arrays correspond one-to-one to Java arrays (e.g. `Array[Int]` is represented as `int[]`), but at the same time they offer much more, Scala arrays:
- can be generic
- are compatible with sequences (you can pass `Array[T]` where `Seq[T]` is required)
- support all sequence operations:

```
val a1 = Array(1, 2, 3)              // Array[Int] = Array(1, 2, 3)
val a2 = a1 map (_ * 3)              // Array[Int] = Array(3, 6, 9)
val a3 = a2 filter (_ % 2 != 0)      // Array[Int] = Array(3, 9)
val a4 = a3.reverse                  // Array[Int] = Array(9, 3)
```

- all this is possible because of systematic use of implicit conversions in the implementation
- representation of native array is not a subtype of `Seq`, instead there is implicit ***wrapping***conversion between arrays and instances of `scala.collection.mutable.WrappedArray`, which is a subclass of `Seq`:

```
val seq: Seq[Int] = a1               // Seq[Int] = WrappedArray(1, 2, 3)
val a4: Array[Int] = seq.toArray     // Array[Int] = Array(1, 2, 3)
a1 eq a4                             // true
```

- there is another implicit conversion that gets applied to arrays, but this one does not turn arrays into sequences, it simply ***adds*** all sequence methods to it
- ***adding*** means that the array is wrapped in another object, of type `ArrayOps`, which is typically short-lived (usually inaccessible after the call to the sequence method). Modern VMs often avoid creating this object entirely

```
// the difference between two implicit conversions:
val seq: Seq[Int] = a1    // Seq[Int] = WrappedArray(1, 2, 3)
seq.reverse               // Seq[Int] = WrappedArray(3, 2, 1)
val ops: collection.mutable.ArrayOps[Int] = a1    // mutable.ArrayOps[Int] = [I(1, 2, 3)]
ops.reverse               // Array[Int] = Array(3, 2, 1)

// calling reverse on 'seq', which is a 'WrappedArray', gives again a 'WrappedArray'
// that's logical because wrapped arrays are 'Seqs' and calling reverse on any 'Seq'
// will give again a 'Seq'
// calling 'reverse' on the 'ArrayOps' results in an 'Array', not a 'Seq'
// this was only demonstration, you'd never define a value of class 'ArrayOps'
// you'd simply call a 'seq' method on an array:
a1.reverse    // Array[Int] = Array(3, 2, 1)
```

- this raises the one question, though, how the compiler picked `ArrayOps` (`intArrayOps`, to be more precise) over the other implicit conversion, to `WrappedArray`, since both conversions map an array to a type that supports a `reverse` method?
- the two implicit conversions are prioritized, and the `ArrayOps` conversion has the higher priority, since it is defined in the `Predef` object, whereas the other is defined in a class `scala.LowPriorityImplicits`, which is a superclass of `Predef`

- implicits in subclasses and subobjects take precedence over implicits in base classes

What's the story on generic arrays?

- in Java, you cannot write `T[]`, how then Scala's `Array[T]` is represented?
- a generic array could be at runtime any of Java's primitive array types, or it could be an array of objects and the only common runtime type encompassing all that is `AnyRef`, so that's the type Scala compiler maps `Array[T]` to
- at runtime, when an element of an array of type `Array[T]` is accessed or updated, there is a sequence of type tests that determine the actual array type, followed by the correct array operation on the Java array
- since these tests slow down operations a bit, you can expect access to generic arrays to be 3 to 4 times slower than to primitive or object arrays

- representing a generic array type is not enough, there must also be a way to ***create*** generic arrays, which is an even harder problem:

```
// This is not enough - doesn't compile!
def evenElems[T](xs: Vector[T]): Array[T] = {
  // this could be e.g. an Array[Int], or an Array[Boolean]
  // or an array of some of the Java primitive, or an array of some reference type
  // which all have different runtime representations
  val arr = new Array[T]((xs.length + 1) / 2)   // cannot find a class tag for type T
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}

// the reason why Scala runtime cannot pick the type is type erasure
// the actual type that corresponds to the type T is erased at runtime

// it is required that you provide a runtime type hint to the compiler
// this hint takes the form of a 'class manifest' of type 'scala.reflect.ClassManifest'
// a class manifest is a type descriptor object that describes what the top-level class
// of a type is
// there's also a full manifest (scala.reflect.Manifest), that describes all aspects
// of a type, but for array creation, only a class manifest is needed

// the compiler will generate code to construct and pass class manifests automatically
// if you demand a class manifest as an implicit parameter:
def evenElems[T](xs: Vector[T])(implicit m: ClassManifest[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}   //> evenElems: [T](xs: Vector[T])(implicit m: ClassManifest[T])Array[T]

// or written shorter, with type 'context bound':
def evenElems[T: ClassManifest](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}   //> evenElems: [T](xs: Vector[T])(implicit evidence$1: ClassManifest[T])Array[T]

// the two versions of 'evenElems' are exactly the same
// when the 'Array[T]' is constructed, the compiler looks for a class manifest for
```

```
// the type parameter 'T', that is, it looks for an implicit value of type
// 'ClassManifest[T]' and if such a value is found, the manifest is used to construct
// the right kind of array
evenElems(Vector(1, 2, 3, 4, 5))    //Array[Int] = Array(1, 3, 5)
evenElems(Vector("compiler", "of Scala", "is", "not", "a bitch"))

// compiler automatically constructed a class manifest for the element type
// and passed it to the implicit parameter of 'evenElems'

// compiler can do that for all the concrete types, but not if the argument is itself
// another type parameter without its class manifest:
def wrap[U](xs: Vector[U]) = evenElems(xs)    // No ClassManifest available for U
// not enough arguments for method evenElems
// unspecified value parameter evidence$1

def wrap[U: ClassManifest](xs: Vector[U]) = evenElems(xs)
// wrap: [U](xs: Vector[U])(implicit evidence$1: ClassManifest[U])Array[U]

// the context bound in the definition of 'U' is just a shorthand for an implicit
// parameter named here 'evidence$1' of type 'ClassManifest[U]'
```

- so generic array creation demands class manifests
- whenever you create an array of type parameter 'T', you also need to provide an implicit class manifest for 'T'
- the easiest way to do that is to declare the type parameter with a 'ClassManifest' context bound

583 - Strings

- like arrays, strings are not directly sequences, but they can be converted to them

```
val str = "hell0"           // java.lang.String = hello
str.reverse                  // 0lleh
str.map(_.toUpper)          // HELL0
str drop 3                   // l0
str slice (1, 4)            // ell
val s: Seq[Char] = str    // Seq[Char] = WrappedString(h, e, l, l, 0)

// these operations are supported by two implicit conversions:
// low-priority conversion to 'WrappedString', a subclass of 'immutable.IndexedSeq'
//   which was applied in the last line above
// high-priority conversion to 'StringOps' object, which adds all immutable seq methods
//   which was applied to support 'reverse', 'map', 'drop' and 'slice'
```

548 - Performance characteristics

- different collection types have different performance characteristics, which is often the primary reason for picking one over another

Performance characteristics of some common operations on collections:

```
/*
The meaning of symbols:
  C     the operation takes (fast) constant time
  eC    effectively constant time (depends on assumptions, e.g. hash key distribution)
  aC    amortized constant time (in average, but some invocations might take longer)
  Log   time proportional to the logarithm of the collection size
```

```
  L      linear time (proportional to the collection size)
  -      the operation is not supported
*/
```

Performance characteristics of sequence types:

```
/*
                head  tail  apply  update  prepend  append  insert
 immutable
  List          C     C     L      L       C        L       -
  Stream        C     C     L      L       C        L       -
  Vector        eC    eC    eC     eC      eC       eC      -
  Stack         C     C     L      L       C        L       -
  Queue         aC    aC    L      L       L        C       -
  Range         C     C     C      -       -        -       -
  String        C     L     C      L       L        L       -
 mutable
  ArrayBuffer   C     L     C      C       L        aC      L
  ListBuffer    C     L     L      L       C        C       L
  StringBuilder C     L     C      C       L        aC      L
  MutableList   C     L     L      L       C        C       L
  Queue         C     L     L      L       C        C       L
  ArraySeq      C     L     C      C       -        -       -
  Stack         C     L     L      L       C        L       L
  ArrayStack    C     L     C      C       aC       L       L
  Array         C     L     C      C       -        -       -
*/
```

Performance characteristics of sets and maps:

```
/*
                lookup   add   remove   min
 immutable
  HashSet/HashMap   eC     eC    eC       L
  TreeSet/TreeMap   Log    Log   Log      Log
  BitSet            C      L     L       eCa
  ListMap           L      L     L       L
 mutable
  HashSet/HashMap   eC     eC    eC       L
  WeakHashMap       eC     eC    eC       L
  BitSet            C      aC    C       eCa

 eCa - assumption that bits are densely packed
*/
```

585 - Equality

- the collection libraries have a uniform approach to equality and hashing
- when checking equality, Scala first divides collections into sets, maps and sequences (collections of different categories are always unequal, even if they contain the same elements)
- withing a category, collections are equal only if they have the same elements (for sequences, elements must be in the same order), e.g. `List(1, 2, 3) == Vector(1, 2, 3)`
- for equality check, it's irrelevant whether a collection is mutable of immutable
- you have to be careful not to use mutable collections as a key in a hash map:

```
import collection.mutable.{HashMap, ArrayBuffer}
val buf = ArrayBuffer(1, 2, 3)   // mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)
val map = HashMap(buf -> 3)
// mutable.HashMap[mutable.ArrayBuffer[Int], Int] = Map((ArrayBuffer(1, 2, 3), 3))
map(buf)   // 3
buf(0) += 1
map(buf)   // java.util.NoSuchElementException: key not found: ArrayBuffer(2, 2, 3)
```

587 - Views

- methods that construct new collections are called **transformers**, because they take at least one collection as their receiver object and produce another collection (e.g. `map`, `filter`, `++`)
- transformers can be implemented in two principal ways, **strict** and **non-strict (lazy)**
- a strict transformers construct a new collection with all of its elements, whereas lazy transformers construct only a proxy for the result collection, where its elements are constructed on demand:

```
def lazyMap[T, U](coll: Iterable[T], f: T => U) =
  new Iterable[U] {
    def iterator = coll.iterator map f
  }

// lazyMap constructs a new 'Iterable' without stepping through all elements of the
// given collection
// the given function 'f' is instead applied to the elements of the new collection's
// 'iterator' as they are demanded
```

- Scala collections are by default strict in all their transformers, except `Stream`, which implements all its transformer methods lazily
- there is a systematic way to turn every collection into a lazy one and vice versa, which is base on collection views
- a **view** is a special kind of collection that represents some base collection, but implements all its transformers lazily
- to go from a collection to its view, you use the collection's `view` method
- to get back from a view to a strict collection, you use the `force` method

```
val v = Vector(1 to 5: _*)   // immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
v map (_ + 1) map (_ * 2)    // immutable.Vector[Int] = Vector(4, 6, 8, 10, 12)

// a note about vector creation:
// if we had created the vector like this, we would've get 'Range':
val v = Vector(1 to 5)   // Vector[immutable.Range.Inclusive] = Vector(Range(1,2,3,4,5))

// the expression 'v map (_ + 1)' constructs a new vector that is then transformed
// into a third vector by the second 'map' expression

// we could've used a single 'map' with the composition of the two functions,
// but that often isn't possible, since the code resides in different modules
// a more general way to avoid the intermediate results is by first turning the
// vector into a view, applying transformations to it, and forcing the view to a vector
(v.view map (_ + 1) map (_ * 2)).force

// or one by one:
```

```
val vv = v.view    // collection.SeqView[Int, Vector[Int]] = SeqView(...)
// the 'v.view' gives us a 'SeqView', i.e. a lazily evaluated 'Seq'
// the type 'SeqView' has two type parameters, 'Int' shows the type of view's elems
// and the 'Vector[Int]' shows the type constructor we get back when forcing the view

// applying the first map to the view gives us:
val resInter = vv map (_ + 1)    // SeqView[Int,Seq[_]] = SeqViewM(...)
// 'SeqView(...)' is in essence a wrapper that records the fact that a 'map' with
// function (_ + 1) needs to be applied on the vector 'v'
// it does not apply that 'map' until the view is 'forced'
// the "M" after 'SeqView' is an indication that the view encapsulates a 'map' operation
// other letters indicate other delayed operations, "S" for 'slice', "R" for 'reverse'

// we now apply the second 'map' to the last result:
val res = resInter map (_ * 2)    // SeqView[Int, Seq[_]] = SeqViewMM(...)
// we now get a 'SeqView' that contains two map operations, so it prints with double "M"

// finally, forcing the last result gives:
res.force    // Seq[Int] = Vector(4, 6, 8, 10, 12)

// both stored functions get applied as part of the execution of the 'force' operation
// that way, no intermediate data structure is needed

// one detail to note is that the static type of the final result is a 'Seq',
// not a 'Vector'
// tracing the types back we see that as soon as the first delayed 'map' was applied,
// the result had static type 'SeqViewM[Int, Seq[_]]', that is, the knowledge that
// the view was applied to the specific sequence type 'Vector' got lost
// the implementation of a view, for any particular class, requires quite a bit of code,
// so the Scala collection libraries provide view mostly only for general collection
// types, not for specific implementations (exception is 'Array': applying delayed
// operations on array will again give results with static type 'Array')
```

- there are two reasons why you might want to consider using views, the first, obviously, performance and the second:

```
// the problem of finding the first palindrome in a list of words:
def isPalindrome(x: String) = x == x.reverse
def findPalindrome(s: Seq[String]) = s find isPalindrome
// if 'words' is a previously defined (very long) list of words:
findPalindrome(words take 1000000)
// this always constructs an intermediary sequence consisting of million words
// so, if the first word is a palindrome, this would copy 999 999 words into the
// intermediary result without being inspected at all afterwards

// with views:
findPalindrome(words.view take 1000000)
// this would only construct a single lightweight view object

// ##### views over mutable sequences: #####
// many transformer functions on such views provide a window into the original sequence
// that can then be used to update selectively some elements of that sequence:
val arr = (0 to 4).toArray    // Array[Int] = Array(0, 1, 2, 3, 4)

// we can create a subwindow into that array by creating a slice of a view of the array:
val subarr = arr.view.slice(2, 5)  //IndexedSeqView[Int,Array[Int]] = IndexedSeqViewS(...)
// this gives a view which refers to elements at position 2 through 4 of the array 'arr'
```

```
// the view does not copy these elements, it simply provides a reference to them

// now assume you have a method that modifies some elements of a sequence
// e.g. the 'negate' method would negate all elements of the sequence it receives:
def negate(xs: collection.mutable.Seq[Int]) =
  for (i <- 0 until xs.length) xs(i) = -xs(i)

// if you wanted to negate elements from positions 2 through 4:
negate(subarr)
arr    // Array[Int] = Array(0, 1, -2, -3, -4, 5)
// 'negate' changed all elements which were a slice of the elements of 'arr'
```

- for smaller collections, the added overhead of forming and applying closures in views is often greater than the gain from avoiding the intermediary data structures
- evaluation in views can be very confusing if the delayed operation have side effects
- it is recommended that you use views either in purely functional code, where the collection transformations do not have side effects, or that you apply them over mutable collections where all modifications are done explicitly

593 - Iterators

- a way to access elements of a collection one by one
- a call to `it.next()` returns the next element and advances the state of the iterator
- if there are no more elements, `next` throws `NoSuchElementException`
- to avoid that, we use `hasNext` method

```
// the most straightforward way to step through the elements returned by an iterator:
while (it.hasNext)
  println(it.next())

// iterators provide analogues of most of the methods of 'Traversable', 'Iterable'
// and 'Seq' traits:
it foreach println

// for can be used instead of 'foreach', 'map', 'filter' and 'flatMap':
for (elem <- it) println(elem)
```

- there is an important difference between `foreach` on iterators and the same method on traversable collections: when called on an iterator, `foreach` will leave the iterator at its end when it's done (calling `next` yields 'NoSuchElementException'), but when called on a collection, it leaves the number of elements in the collection unchanged
- the other operations that `Iterator` has in common with `Traversable` all have the same property of leaving the iterator at its end when done iterating:

```
val it = Iterator("a", "b", "is a", "b")   // Iterator[String] = non-empty iterator
val res = it.map(_.length)   // Iterator[Int] = non-empty iterator
res foreach print   // 1141
it.next()   // java.util.NoSuchElementException: next on empty iterator

// a method that finds the first word in an iterator that has at least two characters:
val it = Iterator("a", "member", "of", "words")   // Iterator[String] = non-empty iterator
```

```
val it2 = it dropWhile (_.length < 2)   // Iterator[String] = non-empty iterator
it2.next   // String = member
it2.next   // String = of
it.next    // String = words

// there's only one standard operation, 'duplicate', that allows reuse of an iterator:
val (it1, it2) = it.duplicate
// 'duplicate' returns a pair of iterators that work independently
// the original iterator 'it' is advanced to its end by the 'duplicate' operation
```

- iterators behave like collections **if you never access an iterator again after invoking a method on it**
- Scala makes this explicit, by providing an abstraction called `TraversableOnce`, which is a common supertrait of `Traversable` and `Iterator`
- `TraversableOnce` object can be traversed using `foreach`, but the state of that object after the traversal is not specified
- if the `TraversableOnce` object is an `Iterator`, it will be at its end, but if it's a `Traversable`, it will still exist as before
- a common use case for `TraversableOnce` is to use it as an argument type for methods that can take either an iterator or traversable, e.g. appending method `++` in trait `Traversable`, which takes a `TraversableOnce` parameter, so you can append elements coming from either an iterator or a traversable collection

All operations in trait `Iterator`:

- **Abstract methods**
- `it.next()` Returns the next element and advances iter past it
- `it.hasNext` Returns 'true' if 'it' can return another element

- **Variations**
- `it.buffered` A buffered iter returning all elements of 'it'
- `it grouped size` An iter that yields elems returned by 'it' in fixed-sized sequence chunks
- `xs sliding size` An iter that yields elems returned by 'it' in sequences representing a sliding fixed-sized window

- **Copying**
- `it copyToBuffer buf` Copies all elems returned by 'it' to buffer 'buf'
- `it copyToArray(arr, s, l)` Copies at most 'l' elems returned by 'it' to array 'arr' starting at index 's' (last 2 args are optional)

- **Duplication**
- `it.duplicate` A pair of iters that each independently return all elements of 'it'

- **Additions**
- `it ++ jt` An iter returning all elems returned by 'it' followed by all elems returned by 'jt'
- `it padTo (len, x)` An iter that returns all elems of 'it' followed by copies of 'x' until length 'len' elems are returned overall

- **Maps**
- `it map f` The iter obtained from applying 'f' to every elem

- `it flatMap f` The iter obtained from applying the iter-valued function 'f' to every elem and appending the result
- `it collect f` The iter obtained from applying the partial function 'f' to every elem for which it is defined and collecting the results

- **Conversions**
- `it.toArray` Collects the elements returned by 'it' in an array
- `it.toList` Collects the elements returned by 'it' in a list
- `it.toIterable` Collects the elements returned by 'it' in an iterable
- `it.toSeq` Collects the elements returned by 'it' in a sequence
- `it.toIndexedSeq` Collects the elements returned by 'it' in an indexed sequence
- `it.toStream` Collects the elements returned by 'it' in a stream
- `it.toSet` Collects the elements returned by 'it' in a set
- `it.toMap` Collects the key/value pairs returned by 'it' in a map

- **Size info**
- `it.isEmpty` Tests whether 'it' is empty (opposite of 'hasNext')
- `it.nonEmpty` Tests whether the collection contains elems (alias of hasNext)
- `it.size` The number of elems returned by 'it' (waists 'it')
- `it.length` Same as 'it.size'
- `it.hasDefiniteSize` Returns true if 'it' is known to return finitely many elems

- **Element retrieval index search**
- `it find p` An option containing the first elem that satisfies 'p', or 'None' if no element qualifies (advances 'it' to just after the elem or to end)
- `it indexOf x` The index of the first elem returned by 'it' that equals 'x' (advances past the position of 'x')
- `it indexWhere p` The index of the first elem that satisfies 'p' (advances 'it' past the position of that elem)

- **Subiterators**
- `it take n` An iter returning the first 'n' elems ('it' advances past n'th elem, or its end)
- `it drop n` The iter that starts with the (n + 1)'th elem (advances 'it' to that same position)
- `it slice (m, n)` The iter that returns a slice of the elems of 'it', starting with the m'th and ending before n'th
- `it takeWhile p` An iter returning elems from 'it' as long as 'p' is true
- `it dropWhile p` An iter skipping elems from 'it' as long as 'p' is true, and returning the remainder
- `it filter p` An iter returning all elems from 'it' that satisfy 'p'
- `it withFilter p` Same as 'filter' (needed so that iters can be used in 'for' expressions)
- `it filterNot p` An iter returning all elems from 'it' that don't satisfy 'p'

- **Subdivisions**
- `it partition p` Splits 'it' into a pair of two iters, based on whether elems satisfy 'p'

- **Element conditions**
- `it forall p` A boolean indicating whether 'p' holds for all elems
- `it exists p` A boolean indicating whether 'p' holds for some element
- `it count p` The number of elems that satisfy predicate 'p'

- **Folds**
- `(z /: it)(op)` Applies binary operation 'op' between successive elems, going left to right, starting with 'z'
- `(z :\ it)(op)` Applies binary operation 'op' between successive elems, going right to left, starting with 'z'
- `it.foldLeft(z)(op)` Same as `(z /: it)(op)`
- `it.foldRight(z)(op)` Same as `(z :\ it)(op)`
- `it reduceLeft op` Applies binary operation 'op' between successive elems returned by non-empty iter 'it', going left to right
- `it reduceRight op` Applies binary operation 'op' between successive elems returned by non-empty iter 'it', going right to left

- **Specific folds**
- `it.sum` The sum of the numeric elem values returned by 'it'
- `it.product` The product of the numeric elem values returned by 'it'
- `it.min` The minimum of the ordered elem values returned by 'it'
- `it.max` The maximum of the ordered elem values returned by 'it'

- **Zippers**
- `it zip jt` Iter of pairs of corresponding elems from 'it' and 'jt'
- `it zipAll (jt, x, y)` Iter of pairs of corresponding elems from 'it' and 'jt', where the shorter iter is extended to match the longer one, by appending elems x or y
- `it.zipWithIndex` Iter of pairs of elems returned from 'it' with their indices

- **Update**
- `it patch (i, jt, r)` Iter resulting from 'it' by replacing 'r' elems starting with 'i', by the patch iter 'jt'

- **Comparison**
- `it sameElements jt` A test whether iters 'it' and 'jt' return the same elems in the same order (at least one of two iters ends up advancing to its end)

- **Strings**
- `it addString (b, start, sep, end)` Adds a string to 'StringBuilder b' that shows all elems of 'it' between separators 'sep', enclosed in strings 'start' and 'end' ('start', 'sep', 'end' are all optional)
- `it mkString (start, sep, end)` Converts the collection to a string that shows all elems of 'it' between separators 'sep', enclosed in strings 'start' and 'end' ('start', 'sep', 'end' are all optional)

600 - Buffered iterators

- iterators that can "look ahead" so you can inspect the next element to be returned, without advancing past that element
- every iterator can be converted to a buffered iterator by calling its `buffered` method

```
// skip leading empty strings in an iterator (advances past the first non-empty elem)
def skipEmptyWordsNOT(it: Iterator[String]) {
  while (it.next().isEmpty) {}
}

// with buffered iterator (instance of trait 'BufferedIterator'):
def skipEmptyWords(it: BufferedIterator[String]) =
```

```
   while (it.head.isEmpty) { it.next() }

// converting to buffered:
val it = Iterator(1, 2, 3, 4)    // Iterator[Int] = non-empty iterator
val bit = it.buffered   // java.lang.Object with BufferedIterator[Int] = non-empty iter
bit.head      // Int = 1
bit.next()    // Int = 1
bit.next()    // Int = 2
```

601 - Creating collections from scratch

- as with lists, `List(1, 2)` and maps, `Map('a' -> 1, 'b' -> 2)`, you can create any collection type by appending list of elements in parentheses to a collection name, which is, under the cover, a call to the `apply` method of some object:

```
Traversable()                    // empty traversable object
List()                           // empty list
List(1.0, 2.0)                   // list with two elements
Vector(1.0, 2.0)                 // vector with two elements
Iterator(1, 2, 3)                // iterator returning three integers
Set(dog, cat, bird)              // a set of three animals
HashSet(dog, cat)                // a hash set of two animal
Map('a' -> 8, 'b' -> 0 )    // a map from characters to integers

// under the cover:
List(1.0, 2.0)
// expands to
List.apply(1.0, 2.0)
// where 'List' is the companion object of the 'List' class, which takes an arbitrary
// number of arguments and constructs a list from them

// every collection class has a companion object with 'apply' method
// no matter if a collection is a concrete class or a trait

// if it's a trait, calling apply will product some default implementation of the trait:
Traversable(1, 2, 3)                // Traversable[Int] = List(1, 2, 3)
mutable.Traversable(1, 2, 3)    // mutable.Traversable[Int] = ArrayBuffer(1, 2, 3)

// besides 'apply', every collection companion object also defines 'empty'
```

Factory methods for sequences:

- `S.empty` The empty sequence
- `S(x, y, z)` A sequence consisting of elements x, y and z
- `S.concat(xs, ys, zs)` The sequence obtained by concatenating elems of xs, ys and zs
- `S.fill(n)(e)` A sequence of length 'n' where each elem is computed by expression 'e'
- `S.fill(m, n)(e)` A sequence of sequences of dimension 'm x n', where each elem is computed by expression 'e'
- `S.tabulate(n)(f)` A sequence of length 'n' where the elem at each index 'i' is computed by 'f(i)'
- `S.tabulate(m, n)(f)` A sequence of sequences of dimension 'm x n', where the elem at each index '(i, j)' is computed by 'f(i, j)'
- `S.range(start, end)` The sequence of integers 'start ... end - 1'

- `S.range(start, end, step)` The sequence of integers starting with 'start' and progressing by 'step' increments up to, and excluding 'end'
- `S.iterate(x, n)(f)` The sequence of length 'n' with elems 'x, f(x), f(f(x)), ...'

603 - Conversions between Java and Scala collections

- Scala offers implicit conversions between all major collection types in the `JavaConversions` object

```
// two-way conversions:
Iterator              <->        java.util.Iterator
Iterator              <->        java.util.Enumeration
Iterable              <->        java.util.Iterable
Iterable              <->        java.util.Collection
mutable.Buffer        <->        java.util.List
mutable.Set           <->        java.util.Set
mutable.Map           <->        java.util.Map

// to enable these automatic conversions:
import collection.JavaConversions._

import collection.mutable._
val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3)    // java.util.List[Int] = [1, 2, 3]
val buf: Seq[Int] = jul    // mutable.Seq[Int] = ArrayBuffer(1, 2, 3)
val m: java.util.Map[String, Int] = HashMap("a" -> 1, "ab" -> 2)
// java.util.Map[String, Int] = {ab=2, a=1}

// internally, these work by setting up a "wrapper" object that forwards all operations
// to the underlying collection object, so collections are never copied

// one-way conversion to Java types:
Seq                   ->         java.util.List
mutable.Seq           ->         java.util.List
Set                   ->         java.util.Set
Map                   ->         java.util.Map

// because Java does not distinguish between mutable and immutable collection in their
// type, a conversion from, say, 'immutable.List' will yield a 'java.util.List',
// on which all attempted mutations will throw an 'UnsupportedOperationException':
val jul: java.util.List[Int] = List(1, 2, 3)    // java.util.List[Int] = [1, 2, 3]
jul.add(8)    // java.lang.UnsupportedOperationException at java.util.AbstractList.add
```

The Architecture of Scala Collections

- the design approach was to implement most operations in collection "templates" that can be flexibly inherited from individual base classes and implementations

608 - Builders

- almost all collection operations are implemented in terms of ***traversals*** and ***builders***
- traversals are handled by `Traversable`'s `foreach` method, and building new collections is handled by instances of class `Builder`

```
// abbreviated outline of the 'Builder' class
package scala.collection.generic

class Builder[-Elem, +To] {
```

```
  def +=(elem: Elem): this.type
  def result(): To
  def clear()
  def mapResult(f: To => NewTo): Bulder[Elem, NewTo] = //...
}
```

- you can add an element to a builder with `b += x`, or more than one element with `b += (x, y)` or with `b ++= xs` (works same as for buffers, which are, in fact, enriched version of builders)
- the `result()` method returns a collection from a builder
- the state of builder is undefined after taking its result, but it can be reset into a new clean state using `clear()`
- builders are generic in both the element type, `Elem` and in the type `To`, of collections they return

```
// to use 'ArrayBuffer' to produce a builder that builds arrays:
val buf = new ArrayBuffer[Int]    // mutable.ArrayBuffer[Int] = ArrayBuffer()
val bldr = buf mapResult (_.toArray)    // mutable.Builder[Int, Array[Int]] = ArrayBuffer()

// the result value, 'bldr' is a builder that uses the array buffer to collect elems
// when a result is demanded from 'bldr', the result of 'buf' is computed, which
// yields the array buffer 'but' itself
// this array buffer is then mapped with '_.toArray' to an array
// so the end result is that 'bldr' is a builder for arrays
```

609 - Factoring out common operations

- Scala collection library avoids code duplication and achieves the ***same result type*** principle by using generic builders and traversals over collections in so-called **implementation traits**
- these traits are named with a **Like** suffix (e.g. `IndexedSeqLike` is the implementation trait for `IndexedSeq`)
- collection classes such as `IndexedSeq` or `Traversable` inherit all their concrete method implementations from these traits
- ***implementation traits*** have two type parameters instead of usual one for collections, because they parameterize not only over the collection's element type, but also over the collection's **representation type** (i.e. the type of the underlying collection, such as `Seq[I]` or `List[T]`)

```
// the header of the trait 'TraversableLike':
trait TraversableLike[+Elem, +Repr] { ... }
// type parameter 'Elem' - element type of traversable
// type parameter 'Repr' - representation type of elements
```

- there are no constraints on 'Repr', it might be instantiated to a type that is itself not a subtype of `Traversable` (that way classes like `String` and `Array` can still make use of all operations defined in a collection implementation trait)
- e.g. `filter` is implemented once for all collection classes in the trait `TraversableLike`:

```
// implementation of 'filter' in 'TraversableLike'
package scala.collection
```

```
class TraversableLike[+Elem, +Repr] {
  def newBuilder: Builder[Elem, Repr]   // deferred to concrete implementation classes
  def foreach[U](f: Elem => U)              // deferred

  def filter(p: Elem => Boolean): Repr = {
    val b = newBuilder    // first constructs a new builder for the representation type
    foreach { elem => if (p(elem)) b += elem }    // traverses all elems of the collection
                          // and if an elem satisfies 'p' adds it to the builder
    b.result    // finally, all elems collected in the builder are returned
                // as an instance of the 'Repr' collection type by calling 'result'
  }
}
```

- a bit more complicated is the `map` operation on collections, for example, if `f` is a function from `String` to `Int`, and `xs` is a `List[String]`, then `xs map f` should give a `List[Int]`, but if `xs` is an `Array[String]`, then the same expression should return `Array[Int]`
- how does Scala achieve that without duplicating implementations of the `map` methods in both `List` and `Array`?
- the `newBuilder` & `foreach` combination is not enough, since it only allows creation of new instances of the same collection type
- on top of that requirement, there's a problem that even the result type constructor of a function like `map` might depend in non trivial ways on the other argument types:

```
import collection.immutable.BitSet
val bits = BitSet(1, 2, 3)   // immutable.BitSet = BitSet(1, 2, 3)
bits map (_ * 2)                  // BitSet(2, 4, 6)
bits map (_.toFloat)            // immutable.Set[Float] = Set(1.0, 2.0, 3.0)

// if you map the doubling function over a bit set, you get another bit set back
// but if you map the function 'toFloat' over the same bit set, you get 'Set[Float]'
// because bit sets can only contain ints
```

- the map's result type depends on the type of function that's passed to it
- if the result type of that function stays int, the result will be bit set, but if the result type of the function argument is something else, the result is just a set

```
// the problem is, of course, not just with bit sets:
Map("a" -> 1, "b" -> 2) map { case (x, y) => (y, x)}  // Map[Int, String] = Map(1->a,2->b)
Map("a" -> 1, "b" -> 2) map { case (x, y) => y}  // Iterable[Int] = List(1, 2)

// second function maps key/value pair to integer, in which case we cannot form a map
// but we can still form an iterable, a supertrait of map
// every operation that's legal on iterable, must also be legal on a map
```

- Scala solves this problem with overloading that's provided by implicit parameters:

```
// implementation of 'map' in 'TraversableLike':
def map[B, That](p: Elem => B)(implicit bf: CanBuildFrom[B, That, This]): That = {
  val b = bf(this)
```

```
    for (x <- this) b += f(x)
    b.result
}

// where 'filter' used the 'newBuilder' method, 'map' uses a 'builder factory' that's
// passed as an additional implicit parameter of type 'CanBuildFrom':
package scala.collection.generic

trait CanBuildFrom[-From, -Elem, +To] {
    def apply(from: From): Builder[Elem, To]    // creates a new builder
}

// Elem - indicates the element type of the collection to be built
// To   - indicates the type of collection to be built
// From - indicates the type for which this builder factory applies

// e.g. BitSet's companion object would contain a builder factory of type
// CanBuildFrom[BitSet, Int, BitSet], which means that
// when operating on BitSet, you can construct another BitSet provided the type of the
// collection to build is 'Int'
// if this is not the case, you can always fall back to different, more general
// implicit builder factory implemented in mutable.Set's companion object:
CanBuildFrom[Set[_], A, Set[A]]

// which means that, when operating on an arbitrary set (Set[_]), you can build a set
// again, no matter what the element type 'A' is

// given various implicit instances of `CanBuildFrom`, you can rely on Scala's implicit
// resolution rules to pick the one that's appropriate and maximally specific

// but what about dynamic types?
val xs: Iterable[Int] = List(1, 2, 3)    // Iterable[Int] = List(1, 2, 3)
val ys = xs map (x => x * x)                    // Iterable[Int] = List(1, 4, 9)

// the static type 'ys' is iterable, as expected, but its dynamic type is still 'List'
// this is achieved by one more indirection. The 'apply' method in 'CanBuildFrom' is
// passed the source collection as argument
// all builder factories for generic traversables (except for leaf classes) forward the
// call to a method 'genericBuilder' of a collection, which in turn calls the builder
// that belongs to the collection in which it is defined

// so Scala uses static implicit resolution to resolve constraints on the types of 'map'
// and virtual dispatch to pick the best dynamic type that corresponds to these
// constraints
```

614 - Integrating new collections

- **Integrating sequences**

```
// sequence type for RNA strands (A, T, G, U)
abstract class Base
case object A extends Base
case object T extends Base
case object G extends Base
case object U extends Base

object Base {
    val fromInt: Int => Base = Array(A, T, G, U)
    val toInt: Base => Int = Map(A -> 0, T -> 1, G -> 2, U -> 3)
```

```
}

// RNA strands class, v1
import collection.IndexedSeqLike
import collection.mutable.{Builder, ArrayBuffer}
import collection.generic.CanBuildFrom

// RNA strands can be very long, so we're building our own collection to optimize
// since there are only 4 bases, a base can be uniquely identified with 2 bits
// so you can store 16 bases in an integer
// we'll create a specialized subclass of 'Seq[Base]'

// 'groups' represents packed bases (16 in each array elem, except maybe in last)
// 'length' specifies total number of bases on the array
// 'private', so clients cannot instantiate it with 'new' (hiding implementation)
final class RNA1 private (val groups: Array[Int], val length: Int)    // parametric fields
      extends IndexedSeq[Base] {    // 'IndexedSeq' has 'length' and 'apply' methods
   import RNA1._
   def apply(idx: Int): Base = {
      if (idx < 0 || length <= idx)
         throw new IndexOutOfBoundsException

      // extract int value from the 'groups', then extract 2-bit number
      Base.fromInt(groups(idx / N) >> (idx % N * S) & M)
   }
}

object RNA1 {
   private val S = 2                  // number of bits necessary to represent group
   private val N = 32                 // number of groups that fit into Int
   private val M = (1 << S) - 1    // bitmask to isolate a group (lowest S bits in a word)

   // converts given sequence of bases to instance of RNA1
   def fromSeq(buf: Seq[Base]): RNA1 = {
      val groups = new Array[Int]((buf.length + N - 1) / N)
      for (i <- 0 until buf.length)    // packs all the bases
         groups(i / N) |= Base.toInt(buf(i)) << (i % N * S)    // bitwise-or equals

      new RNA1(groups, buf.length)
   }
   def apply(bases: Base*) = fromSeq(bases)
}

// using RNA:
val xs = List(A, G, T, A)          // List[Product with Base] = List(A, G, T, A)
RNA1.fromSeq(xs)                        // RNA(A, G, T, A)
val rna1 = RNA1(A, U, G, G, T)   // RNA1(A, U, G, G, T)

println(rna1.length)                  // Int = 5
println(rna1.last)                     // Base = T
println(rna1.take(3))                 // IndexedSeq[Base] = Vector(A, U, G)

// the last line returns 'Vector', as the default implementation of 'IndexedSeq', since
// all we did in class 'RNA1' was extend 'IndexedSeq', which used its 'take' method,
// which doesn't know how to handle bases

// we might override method 'take':
def take(count: Int): RNA1 = RNA1.fromSeq(super.take(count))
// which would take care of 'take', but what about 'drop', 'filter' or 'init'?
```

```
// we would have to override over 50 methods on sequences that return a sequence

// there is a way, the 'RNA' class needs to inherit not only from 'IndexedSeq', but
// also from its implementation trait 'IndexedSeqLike':
final class RNA2 private (val groups: Array[Int], val length: Int)
      extends IndexedSeq[Base] with IndexedSeqLike[Base, RNA2] {
   import RNA2._
   override def newBuilder: Builder[Base, RNA2] =
      new ArrayBuffer[Base] mapResult fromSeq

   def apply(idx: Int): Base = // ... same as before
}
// 'IndexedSeqLike' trait implements all concrete methods of 'IndexedSeq'
// the return type of methods like 'take', 'drop', 'filter' is the second type parameter
// passed to class 'IndexedSeqLike', 'RNA2'

// to be able to do this, 'IndexedSeqLike' uses the 'newBuilder' abstraction, which
// creates a builder of the right kind
// subclasses of trait 'IndexedSeqLike' have to override 'newBuilder' to return
// collections of their own kind ('Builder[Base, RNA2]', in case of RNA2 class)

// it first creates an 'ArrayBuffer', which is itself a 'Builder[Base, ArrayBuffer]'
// it then transforms the 'ArrayBuffer' builder to an 'RNA2' builder, by calling its
// 'mapResult' method
// 'mapResult' expects a transformation function from 'ArrayBuffer' to 'RNA2' as param
// the function we send is simply 'RNA2.fromSeq', which converts an arbitrary base
// sequence to an 'RNA2' value (array buffer is a kind of sequence, so 'fromSeq' works)
```

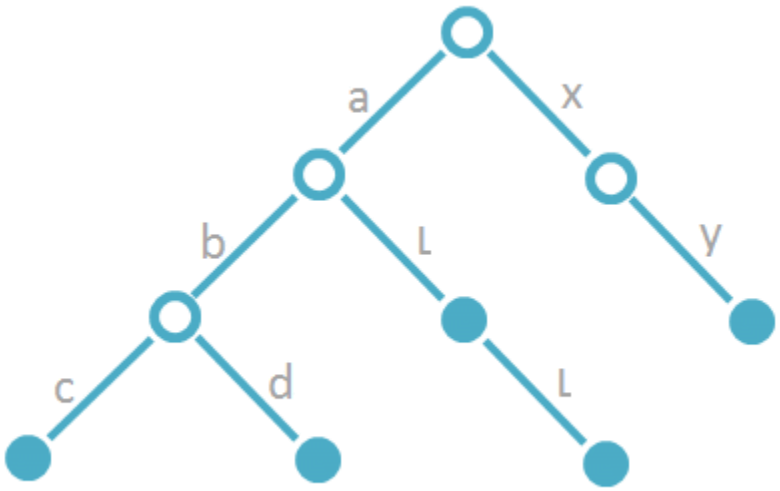
- there are methods that might return the same kind of collection, but with a different element type, e.g. `map`. If `s` is a `Seq[Int]`, and `f` is a function from `Int` to `String`, then `s.map(f)` would return a `Seq[String]`, meaning that the element type changes between the receiver and the result, but the type of collection stays the same
- there are a number of methods that behave the same as `map`, like `flatMap`, `collect`, and even the append, `++` method, which also may return a result of a different type, e.g. appending a list of `String` to a list of `Int` would give a list of `Any`
- we can accept the rule that mapping bases to bases over an `RNA` strand would yield again an `RNA` strand, but mapping bases to some other type necessarily results in a different type:

```
// mapping to the same type:
val rna = RNA(A, U, G, G, T)
rna map { case A => T case b => b}    // Vector(A, U, G, G, T)
rna ++ rna   // Vector(A, U, G, G, T, A, U, G, G, T)

// but mapping to some other type:
rna map Base.toInt    // IndexedSeq[Int] = Vector(0, 3, 2, 2, 1)
rna ++ List("ie", "eg")    // IndexedSeq[java.lang.Object] = Vector(A, U, G, G, T, ie, eg)

// to figure out a better way, first look at the signature of the 'map' method
// which is originally defined in class 'scala.collection.TraversableLike':
def map[B, That](f: A => B)
   (implicit cbf: CanBuildFrom[Repr, B, That]): That

// A    - type of elements of the collection
// Repr - type of the collection itself (gets passed to TraversableLike, IndexedSeqLike)
// B    - result type of the mapping function (elem type of the new collection)
```

```
// That - result type of 'map' (type of the new collection, that gets created)

// how is the type of 'That' determined?
// it is linked to the other types by an implicit parameter 'cbf',
// these cbf implicits are defined by the individual collection classes
// 'CanBuildFrom[Repr, B, That]' says: "Here is a way, given a collection of type 'From',
// to build with elements of type 'Elem' a collection of type 'To'"

// now it's clear, there was no 'CanBuildFrom' instance that creates 'RNA2' sequences,
// so the next best thing available was used, 'CanBuildFrom' of the companion object
// of the inherited trait 'IndexedSeq', which creates indexed seqs

// to address this, we need to define an implicit instance of 'CanBuildFrom' in the
// companion object of the 'RNA' class, which should be 'CanBuildFrom[RNA, Base, RNA]'
// which states that, given an 'RNA' and a new element type 'Base', we can build
// another collection which is again 'RNA':
final class RNA private (val groups: Array[Int], val length: Int)
      extends IndexedSeq[Base] with IndexedSeqLike[Base, RNA] {

   import RNA._
   // mandatory re-implementation of 'newBuilder' in 'IndexedSeq'
   override protected[this] def newBuilder: Builder[Base, RNA] = RNA.newBuilder
   // Mandatory implementation of 'apply' in 'IndexedSeq'
   def apply(idx: Int): Base = {
      if (idx < 0 || length <= idx)
         throw new IndexOutOfBoundsException

      Base.fromInt(groups(idx / N) >> (idx % N * S) & M)
   }

   // optional implementation of 'foreach' to make it more efficient
   // for every selected array elem it immediately applies given function to all bases
   // contained in it (as opposed to default 'foreach', which simply selects every i-th
   // elem using 'apply')
   override def foreach[U](f: Base => U): Unit = {
      var i = 0
      var b = 0
      while (i < length) {
         b = if (i % N == 0) groups(i / N) else b >>> S
         f(Base.fromInt(b & M))
         i += 1
      }
   }
}

object RNA {
   private val S = 2
   private val M = (1 << S) - 1
   private val N = 32 / S

   def fromSeq(buf: Seq[Base]): RNA = {
      val groups = new Array[Int]((buf.length + N - 1) / N)
      for (i <- 0 until buf.length)
         groups(i / N) |= Base.toInt(buf(i)) << (i % N * S)

      new RNA(groups, buf.length)
   }

   def apply(bases: Base*) = fromSeq(bases)
```

```
  // implementation moved here from the RNA class (only a call to this one left there)
  def newBuilder: Builder[Base, RNA] = new ArrayBuffer mapResult fromSeq

  //
  implicit def canBuildFrom: CanBuildFrom[RNA, Base, RNA] =
    new CanBuildFrom[RNA, Base, RNA] {
      // these two are useful for adapting the dynamic type of builder's return type
      // to be the same as the dynamic type of the receiver (not in play here - final)
      def apply(): Builder[Base, RNA] = newBuilder
      def apply(from: RNA): Builder[Base, RNA] = newBuilder
    }
}
```

- **Integrating new sets and maps**
- how to integrate new kind of map into the collection framework?
- e.g. a mutable map with `String` as the key type, by a **Patricia trie**, where "Patricia" stands for ***"Practical Algorithm To Retrieve Information Coded in Alphanumerics"***
- the idea is to store a set or a map as a tree where subsequent character in a search key uniquely determines a descendant tree
- e.g. Patricia trie that stores strings `abc`, `abd`, `al`, `all` and `xy` would look like the following image:



- to find a node corresponding to the string `abc`, you'd simply follow the subtree labeled "a", then proceed to subtree "b", to finally reach its subtree "c"
- if it's used as a map, the value associated with a key is stored in nodes that can be reached by that key
- if it's a set, you simply store a marker saying that the node is present in the set
- ***Patricia tries*** support very efficient lookups and updates
- another great feature is that they support selecting a subcollection by giving a prefix (e.g. to find a subcollection of all keys that start with an "a", you simply follow the "a" link from the root of the tree)

```
// PrefixMap implementation (Patricia trie based)
```

```
// by 'Prefix' we want to say that our map has method 'withPrefix', that selects
// a submap of all keys starting with a given prefix
import collection._
class PrefixMap[T]
      extends mutable.Map[String, T] with mutable.MapLike[String, T, PrefixMap[T]] {
        // inheriting 'MapLike' serves to get the right result type for transformations
        // such as 'filter'
    var suffixes: immutable.Map[Char, PrefixMap[T]] = Map.empty
    // immutable maps with small number of elems are more efficient than mutable maps
    var value: Option[T] = None

    //
    def get(s: String): Option[T] =
      if (s.isEmpty) value    // simply select the optional value stored in the root
      // try to select the submap corresponding to the first char or the string
      // if that yields a submap, follow up by looking up the remainder of the key
      // after its first char and if the key is not found return 'None'
      // When a 'flatMap' is applied to an optional value and a closure (which returns
      // an optional value), 'ov flatMap f' will succeed if both 'ov' and 'f' return a
      // defined value, otherwise it returns 'None'
      else suffixes get (s(0)) flatMap (_.get(s substring 1))

    def withPrefix(s: String): PrefixMap[T] =
      if (s.isEmpty) this
      else {
        val leading = s(0)
        suffixes get leading match {
          case None => suffixes = suffixes + (leading -> empty)
          case _ =>
        }
        suffixes(leading) withPrefix (s substring 1)
      }

    override def update(s: String, elem: T) =
      // first locates the key by calling 'withPrefix', creating submaps if not in tree
      withPrefix(s).value = Some(elem)

    override def remove(s: String): Option[T] =    // similar to 'get', only it
      if (s.isEmpty) { val prev = value; value = None; prev}    // first sets nodes to None
      else suffixes get (s(0)) flatMap (_.remove(s substring 1))

    // returns iterator that yields all key-value pairs from the map
    // If the map contains a defined value, 'Some(x)', in the value field at its root,
    // then '("", x)' is the first element returned from the iterator
    // The iterator needs to traverse the iterators of all submaps stored in the suffixes
    // field, but it needs to add a character in front of every key string returned by
    // those iterators
    // If 'm' is the submap reached from the root through a character 'chr', and '(s, v)'
    // is an element returned from 'm.iterator', then the root's iterator will return
    // '(chr +: s, v)' instead
    def iterator: Iterator[(String, T)] =
      // 'Option' values define an iterator method that returns either no element, if the
      // option value is 'None', or exactly one element, 'x', if the option is 'Some(x)'
      (for (v <- value.iterator) yield ("", v)) ++
      (for ((chr, m) <- suffixes.iterator;
            (s, v) <- m.iterator) yield (chr +: s, v))

// since maps and sets come with default builders, which are instances of 'MapBuilder',
// there's no need to implement the 'newBuilder' method
```

```
  def += (kv: (String, T)): this.type = { update(kv._1, kv._2); this }
  def -= (s: String): this.type = { remove(s); this }

  // to build the right kind of set or map, we need to start with an empty set or map of
  // this kind, thus the 'empty' method
  override def emtpy = new PrefixMap[T]
}


import scala.collection.mutable.{Builder, MapBuilder}
import scala.collection.generic.CanBuildFrom

// the main purpose of this object is to define some convenience factory methods
// and to define a 'CanBuildFrom' implicit to make typing work better
object PrefixMap extends {
  def empty[T] = new PrefixMap[T]
  // methods 'empty' and 'apply' allow us to write 'PrefixMap' literals
  def apply[T](kvs: (String, T)*): PrefixMap[T] = {
    val m: PrefixMap[T] = empty
    for (kv <- kvs) m += kv
    m
  }

  def newBuilder[T]: Builder[(String, T), PrefixMap[T]] =
    new MapBuilder[String, T, PrefixMap[T]](empty)

  // makes methods like 'map' return best possible type
  implicit def canBuildFrom[T]: CanBuildFrom[PrefixMap[_], (String, T), PrefixMap[T]] =
    new CanBuildFrom[PrefixMap[_], (String, T), PrefixMap[T]] {
      def apply(from: PrefixMap[_]) = newBuilder[T]
      def apply() = newBuilder[T]
    }
}

// made possible by methods 'empty' and 'apply':
val pm = PrefixMap("hello" -> 5, "hi" -> 2)    // PrefixMap[Int] = Map((hello. 5), (hi, 2))
val epm = PrefixMap.empty[String]              // PrefixMap[String] = Map()

// made possible by 'CanBuildFrom'
// consider mapping a function over the key-value pairs of a 'PrefixMap'
// as long as that function produces pairs of strings and some other type, the resulting
// collection will again be a 'PrefixMap'
// Without 'canBuildFrom' implicit, the result would have been a general mutable map:
pm map { case (k, v) => (k + "!", "x" * v) }
      // PrefixMap[String] = Map((hello!, xxxxx), (hi!, xx))
```

- to summarize, if you want to fully integrate a new collection class into the collection framework, you need to pay attention to the following:
- decide whether the collection should be mutable or immutable
- pick the right traits for the collection
- inherit from the right implementation trait to implement most collection operations
- if you want `map` and similar operations to return instances of your collection type, provide an implicit `CanBuildFrom` in your class's companion object

Extractors

- until now, constructor patterns were linked to ***case classes***, but sometimes you might want to write patterns like this without creating an associated case class, moreover, you may wish to be able to create your own kinds of patterns that are decoupled from an object's representation.
  Enter **extractors**

631 - An example: extracting email address

```
// given a string, we want to decide whether it's an email address, and it it is
// we want to extract user and domain parts of the address
// the traditional way:
def isEMail(s: String): Boolean
def domain(s: String): String
def user(s: String): String

// and then:
if (isEMail(s)) println(user(s) + " AT " + domain(s))
else println("wtf?")

// lets assume that we could match a string with a pattern:
EMail(user, domain)
// the pattern would match if the string contained an embedded '@' sign, in which case
// it would bind variable 'user' to the part of the string before it, and variable
// 'domain' to the part after it

// the previous expression could be written more clearly like this:
s match {
  case EMail(user, domain) => println(user + " AT " + domain)
  case _ => println("wtf?")
}

// to find two successive emails with the same user part:
ss match {
  case EMail(u1, d1) :: EMail(u2, d2) :: _ if (u1 == u2) => ...
  case _ => ...
}
```

632 - Extractors

- an **extractor** is an object that has a method called `unapply` as one of it members (whose purpose is to match a value and take it apart)
- often times, the extractor object also defines a dual method `apply` for building values, but that's not required

```
// extractor object for e-mail addresses:
object EMail {
  // the injection method (optional)
  def apply(user: String, domain: String) = user + "@" + domain

  // the extraction method (mandatory)
  def unapply(str: String): Option[(String, String)] = {
    // returns option type over pair of strings, since it must handle the case where
    // received param is not an email address
    val parts = str split "@"
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
```

```
// 'apply' is used to turn EMail into an object that can be applied to arguments in
// parentheses in the same way a method is applied, so you can write:
EMail("John", "epfl.ch")  // to construct the string "John@epfl.ch"

// to make this more explicit, we could also let 'EMail' inherit from function type:
object EMail extends ((String, String) => String) { ... }

// '(String, String) => String' is the same as 'Function2[String, String, String]',
// which declares an abstract 'apply' method that 'EMail' implements
// As a result of this inheritance declaration, we could pass 'EMail' to a method
// that expects a 'Function[String, String, String]'

unapply("John@epfl.ch") 'equals' Some("John", "epfl.ch")
unapply("John Doe") 'equals' None
// a side note: when passing a tuple to a function that takes a single argument, we
// can leave off one pair of parentheses, so instead of 'Some((user, domain))' we
// can write 'Some(user, domain)'

// whenever pattern matching encounters a pattern referring to an extractor object,
// it invokes  the extractor's 'unapply' method on the selector expression:
selectorString match { case EMail(user, domain) => ... }
// will be turned into the call:
EMail.unapply(selectorString)    // which returns either 'None' or 'Some(u, d)'

// in the 'None' case, the pattern doesn't match and the system tries another pattern
// or fails with a 'MatchError' exception


// in the previous example, 'selectorString' matched the argument type of 'unapply', but
// that is not necessary and it would also be possible to use the 'EMail' extractor
// to match selector expressions for more general types:
val x: Any = ...
x match { case EMail(user, domain) => ... }

// here, the pattern matcher will first check whether the given value 'x' conforms to
// 'String', the parameter type of 'unapply' method, and if it does, the value is cast
// to 'String' and pattern matching proceeds as normal
// If it doesn't conform, the pattern fails immediately
```

- in object `EMail`, the `apply` method is called **injection**, because it takes some arguments and yields an element of a given set (a set of strings that are email addresses, in this case)
- the `unapply` method is called **extraction**, because it takes an element of the same set and extracts some of its parts (the user and domain substrings, in this case)
- ***injection*** and ***extraction*** are often grouped together in one object, because then you can use the object's name for both a constructor and a pattern, which simulates the convention for pattern matching with case classes
- it is also possible to define an extraction in an object without a corresponding injection, when the object itself is called an **extractor**, regardless of whether or not it has an `apply` method

```
// if an injection method is included, it should be dual to the extraction method
// e.g. a call to:
EMail.unapply(EMail.apply(user, domain))
// should return
Some(user, domain)
```

```
// going in the other direction means running first the 'unapply' and then 'apply':
EMail.unapply(obj) match {
   case Some(u, d) => EMail.apply(u, d)
}
// where, if the match on 'obj' succeeds, we expect to get back that same object from
// the 'apply'
```

- the duality of `apply` and `unapply` is a good design principle, which is not enforced by Scala, of course, but it's recommended when designing extractors

635 - Patterns with zero or one variable

- to bind `n` variables, `unapply` returns an ***N-element tuple*** wrapped in a `Some`
- the case when a pattern binds just one variable is treated differently, since there's no one-tuple in Scala, so to return just one pattern element, the `unapply` method simply wraps the element itself in a `Some`:

```
// extractor object for strings that consist of a substring appearing twice in a row
object Twice {
   def apply(s: String): String = s + s
   def unapply(s: String): Option[String] = {
      val length = s.length / 2
      val half = s.substring(0, length)
      if (half == s.substring(length)) Some(half) else None
   }
}
```

- it is also possible that an extractor pattern does not bind any variables, in which case the corresponding `unapply` returns a boolean, `true` for success and `false` for failure:

```
// extractor object that characterizes strings consisting of all uppercase letters:
object UpperCase {
   def unapply(s: String): Boolean = s.toUpperCase == s
   // it would make no sense to define 'apply', because there's nothing to construct
}

// function that applies all previously defined extractors:
def userTwiceUpper(s: String) = s match {
   case EMail(Twice(x @ UpperCase()), domain) => "match: " + x + " in domain " + domain
   case _ => "no match"
   // UpperCase is written with parentheses since without them, the match would test for
   // equality with the object 'UpperCase'
   // 'x @ UpperCase()' associates variable 'x' with the pattern matched by 'UpperCase()'
}

userTwiceUpper("CANCAN@gmail.com")   // match: CAN in domain gmail.com
userTwiceUpper("CANCAM@gmail.com")   // no match
userTwiceUpper("cancan@gmail.com")   // no match
```

637 - Variable argument extractors

- if you don't know the number of element values in advance, the previous method is not flexible enough

```
// match on a string representing domain name and extract all domain parts
// in the end, we should be able to use it like this:
dom match {    // domains are in reverse order so it better fits sequence patterns
   case Domain("org", "acm") => println("acm.org")
   case Domain("com", "sun", "java") => println("java.sun.com")
   case Domain("net", _*) => println("a .net domain")
}

// a sequence wildcard pattern '_*', at the end of an argument list matches any remaining
// elements in a sequence, which is more useful if top level domains come first
// because then we can use wildcard to match sub-domains of arbitrary length
```

- the question of supporting ***vararg matching*** remains, since the `unapply` methods are not sufficient, because they return a fixed number of sub-elements in the success case
- to handle this case, Scala lets you define a different extraction method, specifically for ***vararg***matching, which is called `unapplySeq`

```
object Domain {
   // the injection (optional)
   def apply(parts: String*): String =
      parts.reverse.mkString(".")

   // the extraction (mandatory)
   // first splits on periods, then reverses and wraps in 'Some'
   def unapplySeq(whole: String): Option[Seq[String]] =    // must return 'Option[Seq[T]]'
      Some(whole.split("\\.").reverse)
}

// to search for an email address "luka.bonaci" in some ".hr" domain:
def isLukaBonaciInDotHr(s: String): Boolean = s match {
   case EMail("luka.bonaci", Domain("hr", _*)) => true
   case _ => false
}

// it's also possible to return some fixed elements from 'unapplySeq', together with the
// variable part, which is expressed by returning all elements in a tuple, where the
// variable part comes last, as usual
// e.g. extractor for emails where the domain part is already expanded into sequence:
object ExpandedEMail {
   // returns optional value of a pair (Tuple2), where the first part is the user, and
   // the second part is a sequence of names representing the domain
   def unapplySeq(email: String): Option[(String, Seq[String])] = {
      val parts = email split "@"
      if (parts.length == 2)
         Some(parts(0), parts(1).split("\\.").reverse)
      else
         None
   }
}

val s = "luka@support.epfl.hr"
val ExpandedEMail(name, topdom, subdoms @ _*) = s
// name: String = luka
// topdom: String = hr
// subdoms: Seq[String] = WrappedArray(epfl, support)
```

640 - Extractors and sequence patterns

- sequence patterns, such as `List()`, `List(x, y, _*)`, `Array(x, 0, _)` are implemented using extractors in the standard Scala library. E.g. patterns of the form `List(...)` are possible because the `scala.List` companion object is an extractor that defines `unapplySeq` method:

```
// 'List' companion object (similar for 'Array')
package scala
object List {
  def apply[T](elems: T*) = elems.toList    // lets you write e.g. 'List(1, 2)' or 'List()'
  // returns all elements as a sequence
  def unapplySeq[T](x: List[T]): Option[Set[T]] = Some(x)
}
```

641 - Extractors versus case classes

- even though case classes are very useful, they have a shortcoming of exposing the concrete representation of data, because the name of the class in a constructor pattern corresponds to the concrete representation type of the selector object
- if a match against `case C(...)` succeeds, you know that the selector expression is an instance of class `C`
- extractors break this link between data representation and patterns, by allowing patterns that have nothing to do with the data type the the object that's matched against
- this property is called **representation independence**, and it is known to be very important in open systems of large scale, because it allows you to change an implementation type used in a set of components without affecting clients of these components
- on the other hand, case classes have their advantages over extractors
- case classes are much easier to set up and define, thus requiring less code
- they usually lead to more efficient pattern matches than extractors, because compiler can optimize patterns over case classes much better than those over extractors, because the mechanisms of case classes are fixed, whereas an `unapply` or `unapplySeq` in an extractor could do almost anything
- if your case classes inherit from a `sealed` base class, the compiler will check your pattern matches for exhaustiveness and will complain if some combination of possible values is not covered by a set of patterns, which is not available with extractors
- if you're writing a closed application, you should prefer case classes, but if you need to expose an API to clients, extractors might be preferable

642 - Regular expressions

- a particularly useful application area of extractors, since they make it much nicer to interact with regular expressions library `scala.util.matching`

Forming regular expressions

- Scala inherits its regex syntax from Java, which in turn inherits it from Perl

```
import scala.util.matching.Regex
val Decimal = new Regex("(-)?(\\d+)(\\.\\d*)?")    // Regex = (-)?(\d+)(\.\d*)?

// like in Java, we need to escape backslashes, which can be painful to write and read
// Scala provides raw strings to help with that
// the difference between raw and normal string is that all characters in a raw string
// appear exactly as they are typed, so we can write:
```

```
val Decimal = new Regex("""(-)?(\d+)(\.\d*)?""") // Regex = (-)?(\d+)(\.\d*)?

// another, even shorter way:
val Decimal = """(-)?(\d+)(\.\d*)?""".r               // Regex = (-)?(\d+)(\.\d*)?

// appending '.r' to a string creates regular expression (method of 'StringOps')
```

Searching for regular expressions

```
val input = "for -1.0 to 99 by 3"
for (s <- Decimal findAllIn input)
  println(s + "    ")          // -1.0  99  3

Decimal findFirstIn input     // Option[String] = Some(-1.0)
Decimal findPrefixOf input    // Option[String] = None  (must be at the start of a string)
```

Extracting with regular expressions

- every regex defines an extractor, which is used to identify substrings that are matched by the groups of the regular expression:

```
// we could decompose a decimal number like this:
val Decimal(sign, integerpart, decimalpart) = "-1.23"
// sign: String = -
// integerpart: String = 1
// decimalpart: String = .23

// the 'Decimal' regex value defines 'unapplySeq', which matches every string that
// corresponds to the regex syntax for decimal numbers
// If the string matches, the parts that correspond to the 3 groups in the regex
// are returned as elements of the pattern and are then matched by the 3 pattern
// variables 'sign', 'integerpart' and 'decimalpart'
// If a group is missing, the element value is set to 'null':
val Decimal(sign, integerpart, decimalpart) = "1.0"
// sign: String = null     integerpart = 1     decimalpart = .0

// it is also possible to mix extractors with regular expression searches in a for
// expression:
for (Decimal(s, i, d) <- Decimal findAllIn input)
  println("sign: " + s + ", integer: " + i + ", decimal: " + d)
```

Annotations

- structured information added to program source code
- not valid Scala expressions
- may be added to any variable, method, expression, or other program element

647 - Why have annotations?

- a **meta-programming** tool (program that take other programs as input)
- the compiler understands just one feature, annotations, but it doesn't attach any meaning to individual annotations
- example use cases:
- a documentation generator instructed to document certain methods as deprecated

- a pretty printer instructed to skip over parts of the program that have been carefully hand formatted
- a checker for non-closed files instructed to ignore a particular file that has been manually verified to be closed
- a side-effects checker instructed to verify that a specified method has no side effects

648 - Syntax of annotations

- a typical use of an annotation looks like this:

```
@deprecated def bigMistake() = // ... applies to the entirety of the 'bigMistake' method

@deprecated class QuickAndDirty { /*...*/ }

(e: @unchecked) match {    // applied to expressions
   // non-exhaustive cases...
}

// annotations have a richer general form:
@annot(exp1, exp2, ...)    // parentheses are optional if annotation has no arguments
// where 'annot' specifies the class of annotation

// the precise form of the arguments you may give to annotation depends on the particular
// annotation class (compiler supports arbitrary expressions, as long as they type check)

// some annotation classes can make use of this, e.g. to let you refer to a variable:
@cool val normal = "Hello"
@coolerThan(normal) val fonzy = "Hot"
```

- internally, Scala represents an annotation as just a constructor call of an annotation class (replace `@` with `new` and you have a valid instance creation expr.)
- a somewhat tricky bit concerns annotations that take other annotations as arguments

```
// we cannot write an annotation directly as an argument to an annotation, because they
// are not valid expressions
// in such cases we must use 'new' instead of '@':
import annotation._
class strategy(arg: Annotation) extends Annotation
class delayed extends Annotation

@strategy(@delayed) def f(){}          // error: illegal start of simple expression

@strategy(new delayed) def f(){}    // f: ()Unit
```

650 - Standard annotations

- introduced for features that are used widely enough to alter the language specification, yet not fundamental enough to merit their own syntax

Deprecation

- used when there's a need to purge some classes or methods from the specification
- lets us gracefully remove a method or a class that turns out to be a mistake
- since we cannot simply delete a language element, because clients' code might stop working, we mark a class or a method as deprecated, by simply writing `@deprecated` before its declaration

- such an annotation will cause the compiler to emit deprecation warning whenever Scala code accesses the language element

```
@deprecated def bigMistake() = // ...

// if you provide a string as an argument, that string will be emitted along with warning
@deprecated("use newShinyMethod() instead")
def bigMistake() = // ...

// now, any caller will get a message like this:
$scalac -deprecation Deprecation2.scala
// Deprecation2.scala:33: warning: method bigMistake in object Deprecation2 is
// deprecated: use newShinyMethod() instead
//     bigMistake()
//     ^
// one warning found
```

Volatile fields

- concurrent programming does not mix well with shared mutable state, and for this reason, the focus of Scala's concurrency support is message passing and a minimum of shared mutable state
- the `@volatile` annotation helps in cases when developers use mutable state in their concurrent programs
- it informs the compiler that a variable will be used by multiple threads
- such variables are implemented so that reads and writes of the variable value is slower, but accesses from multiple threads behave more predictably
- we get the same behavior as if we marked the variable with `volatile` modifier in Java

Binary serialization

- a serialization framework lets us convert objects into streams of bytes and vice versa, which is useful if we want to save objects to disk or transfer over network
- Scala doesn't have its own serialization framework, instead, you should use a framework of the underlying platform
- Scala provides 3 annotations that are useful for a variety of frameworks, and also the compiler interprets these annotations in the Java way
- `@serializable` marks a class as serializable (all classes are considered non serializable by default)
- `@SerialVersionUID(n)` helps to deal with serializable classes future changes, by providing a serial number to a class version, which the framework stores in the generated byte stream. Later, when we reload that byte stream and try to convert it to an object, the framework checks that the current version of the class has the same version number as the version in the byte stream. If not, the framework refuses to load the old instances of the class. So, if you want to make a serialization-incompatible changes to the class, change the version number
- **`@transient`** works the same as Java `transient` field modifier (field is excluded from serialization when its parent object is serialized)

Automatic `get` and `set` methods

- Scala code normally doesn't need explicit getters and setters for fields, because Scala blends the calling syntax for field access and method invocation
- because some platform-specific frameworks do expect get and set methods, Scala provides `@scala.reflect.BeanProperty` annotation, which tells the compiler to automatically generate getters and setters

- it conforms to Java Bean standard, which means that a field named `crazy` will be supplemented with `getCrazy` and `setCrazy`, which will be available only after the compilation passes (so you cannot compile code that uses those methods in that same pass)

Tailrec

- annotation `@tailrec` is added to a method that needs to be tail recursive (e.g. if you expect that, without tail recursion optimization, it would recurse very deep)
- it is used to make sure that the compiler will perform tail recursion optimization
- if the optimization cannot be performed, you will get a warning with an explanation

Unchecked

- `@unchecked` is interpreted by the compiler during pattern matching, and it tells the compiler not to worry if the match expression left out some cases

Native methods

- `@native` informs the compiler that a method's implementation is supplied by the runtime rather than the Scala code
- compiler then toggles the appropriate flags in the output, and it leaves out to the developer to supply the implementation using a mechanism such as ***Java Native Interface*** (JNI)
- when using this annotation, a method body must be supplied, but it will not be emitted into the output:

```
@native def beginCountDown() {}
```

Working with XML

657 - XML literals

- Scala lets you type in XML as a literal, anywhere that en expression is valid:

```
scala> <a>
         This is old stuff.
         Who needs XML any more? <exclamation/>
       </a>
// res0: scala.xml.Elem =
// <a>
//   This is old stuff.
//   Who needs XML any more? <exclamation></exclamation>
// </a>
```

- the usual XML suspects:
- class `Elem` represents an XML element
- class `Node` is the abstract superclass of all XML node classes
- class `Text` is a node holding just unstructured text, from withing a tag
- class `NodeSeq` holds a sequence of nodes. Many methods in the XML library process node sequences in places you might expect them to process individual nodes (since `Node` extends `NodeSeq`, you can use those methods on individual nodes. You can think of an individual node as a one-element node sequence)
- code can be evaluated in the middle of an XML literal, by using curly braces:

```
<a>{"hello" + ", world"}</a>   // scala.xml.Elem = <a>hello, world</a>

// braces can include arbitrary Scala content, e.g. including further XML literals:
val yearMade = 1955   // Int = 1955
<a>{ if (yearMade < 2000) <old>{yearMade}</old>
      else xml.NodeSeq.Empty }
</a>
// scala.xml.Elem = <a> <old>1955</old> </a>

// if an expression inside a braces does not evaluate to an XML node (may evaluate to
// any Scala value), the result is converted to a string and inserted as a text node:
<a>{3 + 4}</a>   // scala.xml.Elem = <a>7</a>

// any '<', '>' and '&' characters in the text will be escaped:
<a>{"</a>potential security hole<a>"}</a>
// scala.xml.Elem = <a>&lt;/a&gt;potential security hole&lt;a&gt;</a>
```

659 - Serialization

```
// writing our own serializer that converts internal data structures to XML:
abstract class CCTherm {
  val description: String
  val yearMade: Int
  val dateObtained: String
  val bookPrice: Int        // in cents
  val purchasePrice: Int    // in cents
  val condition: Int        // 1 to 10

  override def toString = description
  def toXML =
    <cctherm>
      <description>{description}</description>
      <yearMade>{yearMade}</yearMade>
      <dateObtained>{dateObtained}</dateObtained>
      <bookPrice>{bookPrice}</bookPrice>
      <purchasePrice>{purchasePrice}</purchasePrice>
      <condition>{condition}</condition>
    </cctherm>
}

// usage:
val therm = new CCTherm {   // possible because Scala instantiates anonymous subclass
  val description = "The joy of Clojure"
  val yearMade = 2011
  val dateObtained = "24.08.2013"
  val bookPrice = 2400
  val purchasePrice = 2000
  val condition = 10
} // therm: CCTherm = The joy of Clojure

therm.toXML
// scala.xml.Elem =
// <cctherm> ...

// to include curly braces in the XML text just double them:
```

```
<a>{{brace yourself!}}</a>    // scala.xml.Elem = <a>{brace yourself!}</a>
```

661 - Taking XML apart

- there are 3 particularly useful methods, based on ***XPath***, that are used to deconstruct XML: `text`, `\` and `\\`, and `@`

Extracting text

- by calling the `text` method on any XML node, you retrieve all the text within that node, minus any element tags:

```
<a>Sounds <tag/> good</a>.text    // String = Sounds  good

// any encoded characters are decoded automatically:
<a> input ---&gt; output </a>.text    // String =  input ---> output
```

Extracting sub-elements

- to find a sub-element by tag name, simply call `\` with the name of the tag:

```
<a><b><c>hello</c></b></a> \ "b"    // scala.xml.NodeSeq = <b><c>hello</c></b>

// you can do a deep search and look through sub-sub-elements (searches all levels):
<a><b><c><d>hello</d></c></b></a>     \ "a"    // NodeSeq =
<a><b><c><d>hello</d></c></b></a>    \\ "a"    // NodeSeq = <a><b><c><d>hello</d></c></b></a>
<a><b><c><d>hello</d></c></b></a>     \ "b"    // NodeSeq = <b><c><d>hello</d></c></b>
<a><b><c><d>hello</d></c></b></a>    \\ "b"    // NodeSeq = <b><c><d>hello</d></c></b>
<a><b><c><d>hello</d></c></b></a>     \ "c"    // NodeSeq =
<a><b><c><d>hello</d></c></b></a>    \\ "c"    // NodeSeq = <c><d>hello</d></c>
<a><b><c><d>hello</d></c></b></a>     \ "d"    // NodeSeq =
<a><b><c><d>hello</d></c></b></a>    \\ "d"  //> NodeSeq = <d>hello</d>
```

Extracting attributes

- you can extract tag attributes using the same `\` and `\\` methods, by simply putting `@` sign before the attribute name:

```
val joe = <employee
  name="JR"
  rank="dev"
  serial="8"/>  // scala.xml.Elem = <employee rank="dev" name="JR" serial="8"></employee>

joe \ "@name"      // scala.xml.NodeSeq = JR
joe \ "@serial"    // scala.xml.NodeSeq = 8
```

663 - Deserialization

```
// with XML deconstruction methods, we can now write a parser from XML back to our
// internal data structure:
// To parse back a 'CCTherm' instance:
def fromXML(node: scala.xml.Node): CCTherm =
```

```
  new CCTherm {
    val description   = (node \ "description"   ).text
    val yearMade      = (node \ "yearMade"      ).text.toInt
    val dateObtained  = (node \ "dateObtained"  ).text
    val bookPrice     = (node \ "bookPrice"     ).text.toInt
    val purchasePrice = (node \ "purchasePrice").text.toInt
    val condition     = (node \ "condition"     ).text.toInt
  }

val th = fromXML (therm.toXML)   // CCTherm = The joy of Clojure
```

664 - Loading and saving

- the last part needed to write a data serializer is conversion between XML and stream of bytes
- to convert XML to string, all you need is `toString` method
- however, it's better to use a library routine and convert all the way to bytes. That way, the resulting XML can include a directive that specifies which character encoding was used. Otherwise, if you encode string to bytes yourself, you must keep track of the character encoding yourself
- to convert from XML to a file of bytes, you can use `XML.save` command:

```
val node = therm.toXML
scala.xml.XML.save("therm1.xml", node)

// to load XML from a file:
val load = scala.xml.XML.loadFile("therm1.xml")
// scala.xml.Elem = <cctherm>
//     <description>The joy of Clojure</description>
//     <yearMade>2011</yearMade>
//     <dateObtained>24.08.2013</dateObtained>
//     <bookPrice>2400</bookPrice>
//     <purchasePrice>2000</purchasePrice>
//     <condition>10</condition>
//   </cctherm>
val th1 = fromXML(load)   // CCTherm = The joy of Clojure
```

665 - Pattern matching on XML

- sometimes you may not know what kind of XML structure you're supposed to take apart
- an XML pattern looks just like an XML literal, but with one difference. If you insert escape `{}`, the code withing curly braces is not an expression but a pattern
- a pattern embedded in curlies can use the full Scala pattern language, including binding new variables, type tests and ignoring content using `_` and `_*` patterns:

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents}</a> => "It's ej " + contents   // looks for <a> with single subnode
    case <b>{contents}</b> => "It's bi " + contents   // loks
    case _ => "Is it something?"
  }
// expression 'case <a>{contents}</a>' looks for <a> with single sub node
// if found, it binds the content to variable named 'contents' and then evaluates the
// code to the right of the fat arrow
```

```
// usage:
proc(<a>apple</a>)     // String = It's ej apple
proc(<b>banana</b>)    // String = It's bi banana

proc(<a>a <em>red</em> apple</a>)   // String = Is it something?
proc(<a/>)                          // String = Is it something?
```

- if you want the function to match cases like the last two, you'll have to match against a sequence of nodes instead of a single one
- the pattern for any sequence is written `_*`:

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents @ _*}</a> => "It's an ej " + contents
    case <b>{contents @ _*}</b> => "It's a bi " + contents
    case _ => "Is it something else?"
  }
// so now:
proc(<a>a <em>red</em> apple</a>)   // It's an ej ArrayBuffer(a , <em>red</em>,  apple)
proc(<a/>)                          // It's an ej Array()
```

- XML patterns work very nicely with `for` expressions as a way to iterate through some parts of an XML tree, while ignoring other parts:

```
// if you wished to skip over the white space between records in the following XML:
val catalog =
  <catalog>
    <cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>
    <cctherm>
      <description>Sprite Boy</description>
      <yearMade>1964</yearMade>
      <dateObtained>April 28, 2003</dateObtained>
      <bookPrice>1695</bookPrice>
      <purchasePrice>595</purchasePrice>
      <condition>5</condition>
    </cctherm>
  </catalog>

catalog match {
  case <catalog>{therms @ _*}</catalog> =>
    for (therm <- therms)
      println("processing: " + (therm \ "description").text)
}
// processing:
// processing: hot dog #5
// processing:
```

```
// processing: Sprite Boy
// processing:

// there's actually 5 nodes inside '<catalog>', although it looks as if there were 2
// there's a whitespace before, after and between two elements

// to ignore the whitespace and process only subnodes inside a '<cctherm>' elem:
catalog match {
  case <catalog>{therms @_*}</catalog> =>
    for (therm @ <cctherm>{_*}</cctherm> <- therms)
      println("processing: " + (therm \ "description").text)
}
// processing: hot dog #5
// processing: Sprite Boy

// '<cctherm>{_*}</cctherm>' restricts matches only to 'cctherm' elements,
// ignoring whitespace
```

Modular Programming Using Objects

- packages and access modifiers enable you to organize a large program using packages as ***modules***, where a module is a smaller program piece with a well defined interface and a hidden implementation
- while packages are quite helpful, they are limited, because they don't provide a way to abstract, i.e. to reconfigure a package two different ways within the same program or to inherit between packages. A package always includes one precise list of contents, which stays the same until you change the code and recompile

670 - The problem

- being able to compile different modules that make up a system separately helps different teams work independently
- being able to unplug one implementation of a module and plug in another is useful, because it allows different configurations to be used in different contexts
- any technique that aims to provide solutions to these problems should:
- have a module construct that provides clean separation of interface and implementation
- be able to replace one module with another that has the same interface, without changing or recompiling the modules that depend on the replaced one
- be able to wire modules together (can be thought of as ***configuring the system***)
- one approach of solving this problem is **dependency injection**, a technique supported by Spring and Guice on the Java platform
- Spring, for example, essentially allows you to represent the interface of a module as a Java interface and implementation of a module as Java classes, where you can specify dependencies between modules and wire an application together via external XML configuration files
- with Scala you have some alternatives provided by the language itself, which enables you to solve "the problem" without using an external framework

671 - Application design

- the intention is to partition our application into a **domain layer** and an **application layer**
- in the ***domain layer*** we'll define **domain objects**, which will capture business concepts and rules, and encapsulate state that will be persisted to a database

- in the ***application layer*** we'll provide an **API** organized in terms of the services the application offers to clients (including the user interface layer) and implement these services by coordinating tasks and delegating the work to the objects of the ***domain layer***
- to be able to plug in real and mock versions of certain objects in each of these layers (e.g. to more easily write unit tests) we'll treat objects that we want to mock as modules

674 - Abstraction

- to avoid using hard link between modules we use abstract classes and then make modules inherit from these classes
- that way, we can avoid code duplication by providing all common operations in the abstract classes

677 - Splitting modules into traits

- when a module becomes too large to fit into a single file, we can use traits to split a module into separate files
- **self type** specifies the requirements on any concrete class the trait is mixed into. If you have a trait that is only ever used when mixed in with another trait or traits, then you can specify that those other traits should be assumed:

```
trait SimpleRecipes {
    this: SimpleFoods =>     // self type
        // presents a requirement to a class that mixes this trait in:
        // that it has to always be mixed in together with SimpleFoods

    object FruitSalad extends Recipe(
        "fruit salad",
        List(Apple, Pear),  // 'this.Pear' is in scope because of self type
        "Mix it all together."
    )
    def allRecipes = List(FruitSalad)
}
// this is safe because any concrete class that mixes in 'SimpleRecipes' must also mix
// in, i.e. be a subtype of 'SimpleFoods', so 'Pear' will always be a member
// Since abstract classes and traits cannot be instantiated with 'new', there is no risk
// that the 'this.Pear' reference will ever fail
```

See the whole app for the complete picture.

680 - Runtime linking

- modules can be linked together at runtime, and you can decide which modules will link to which depending on runtime computations:

```
// a simple program that chooses a database at runtime:
object GotApples {
    def main(args: Array[String]) {
        val db: Database =
            if(args(0) == "student")
                StudentDatabase
            else
                SimpleDatabase

        object browser extends Browser {
            val database = db
        }
```

```
    val apple = SimpleDatabase.foodNamed("Apple").get
    for (recipe <- browser.recipesUsing(apple))
      println(recipe)
  }
}
```

681 - Tracking module instances

- despite using the same code, the different browser and database modules really are separate modules, which means that each module has its own contents, including any nested classes.
  E.g. `FoodCategory` in `SimpleDatabase` is a different class from `FoodCategory` in `StudentDatabase`

```
val category = StudentDatabase.allCategories.head
// category: StudentDatabase.FoodCategory = FoodCategory(edible,List(FrozenFood))

SimpleBrowser.displayCategory(category)
// type mismatch:
// found   : StudentDatabase.FoodCategory
// required : SimpleDatabase.FoodCategory
```

- if you want all `FoodCategory` objects to be the same, move its definition outside of any class or trait
- sometimes you can encounter a situation where two types are the same but the compiler can't verify it. In such cases you can often fix the problem using **singleton types**:

```
// the type checker doesn't know that 'db' and 'browser.database' are of the same type:
object GotApples {
  // same definitions as before ...
  for (cat <- db.allCategories)
    browser.displayCategory(cat)
  // type mismatch
  // found   : db.FoodCategory
  // required : browser.database.FoodCategory

// to circumvent the problem, inform the type checker that they are the same object:
  object browser extends Browser {
    val database: db.type = db
  }
}
```

- ***singleton type*** is extremely specific and holds only one object, in this case, whichever object is referred to by `db`
- usually such types are to specific to be useful, which is why the compiler is reluctant to insert them automatically

Object Equality

684 - Equality in Scala

- the definition of equality is different in Scala and Java

- Java has two equality comparisons, the `==` operator, which is the natural equality for value types and object identity for reference types, and the `equals` method, which is user-defined canonical equality for reference types
- this convention is problematic, because the more natural symbol, `==`, does not always correspond to the natural notion of equality
- e.g. comparing two strings `x` and `y` using `x == y` might well yield `false` in Java, even if `x`and `y` have exactly the same characters, in the same order
- Scala also has object reference equality method, but it's not used much. That kind of equality, written `x eq y`, is true only if `x` and `y` refer to the same object
- the `==` equality is reserved in Scala for the ***natural*** equality of each type. For value types, it is value comparison, just like in Java, but for reference types, it is the same as `equals`
- that means that you can redefine the behavior of `==` for new types by overriding the `equals`method, which is always inherited from class `Any` (where it represents object reference equality)
- it is not possible to override `==` directly (final in class `Any`)

```
// Scala treats '==' as if it was defined like this in class 'Any':
final def == (that: Any): Boolean =
   if (null eq this) {null eq that}
   else {this equals that}
```

685 - Writing an equality method

- common pitfalls when overriding `equals`:
- defining `equals` with the wrong signature
- changing `equals` without also changing `hashCode`
- defining `equals` in terms of mutable fields
- failing to define `equals` as an equivalence relation

Defining `equals` with the wrong signature

```
class Point(val x: Int, val y: Int){ /*...*/ }

// wrong way to define equals:
def equals(other: Point): Boolean =
   this.x == other.x && this.y == other.y

// the problem is that it doesn't match the signature of 'equals' in class 'Any'
// simple comparisons work fine, but the trouble starts with Points in collections:
import scala.collection.mutable._
val p1, p2 = new Point(1, 2)
val coll = HashSet(p1)    // mutable.HashSet[Point] = Set(Point@79f23c)
coll contains p2          // false

// the problem is that our method 'equals' only overloads 'equals' in class 'Any'
// overloading in Scala and in Java is resolved by the static type of the argument,
// not the runtime type
// So as long as the static type of the argument is 'Point', our 'equals' is called
// but once a static argument is of type 'Any', the 'equals' in 'Any' is called instead
// And since this method has not been overridden, it still compares object references
// This is why 'p1 equals p2a' yields 'false', and why the 'contains' method in
// 'HashSet' returned 'false', since it operates on generic sets, it calls the generic
// 'equals' method in 'Object' instead of the overloaded one in 'Point'
```

```scala
// a better 'equals' (with correct type, but still not perfect):
override def equals(other: Any) = other match {
  case that: Point => this.x == that.x && this.y == that.y
  case _ => false
}

// a related pitfall is to define '==' with a wrong signature
// normally, if you try to override '==' with the correct signature, the compiler will
// give you an error because '==' is final

// the common error (two errors) is to try to override '==' with a wrong signature:
def ==(other: Point): Boolean = /* ... */
// in this case, the method is treated as an overloaded '==' so it compiles
```

Changing `equals` without also changing `hashCode`

- for ***hash-based collections***, element of the collection are put in ***hash buckets*** determined by their hash code
- the `contains` test first determines a hash bucket to look in and then compares the given element with all elements in that bucket, and if element equal to the one provided is not found, the method returns `false`
- the original `hashCode`, in `AnyRef`, calculates hash code by some transformation of the address of the object, so hash codes of `p1` and `p2` are different, even though the fields have the same values
- different hash codes result, with high probability, with different buckets in the set
- the root of the problem can be described with the contract, that the "better equals" violated:
- ***If two objects are equal according to the `equals` method, then calling the `hashCode` on each of the two objects must produce the same integer result***
- ***`hashCode` and `equals` should only be redefined together***
- ***`hashCode` may only depend on fields the `equals` depends on***

```scala
// the suitable definition of 'hashCode':
class Point(val x: Int, val y: Int) {
  override def hashCode = 41 * (41 + x) + y   // 41 is prime
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
// this 'hashCode' should give reasonable distribution of hash codes at a low cost
```

Defining `equals` in terms of mutable fields

```scala
// the slight variation of class Point:
class Point(var x: Int, var y: Int) {   // notice 'vars'
  override def hashCode = 41 * (41 + x) + y
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
```

- the `equals` and `hashCode` are now defined in terms of these mutable fields, so their results change when fields change, which can have strange effects once you put points in collections:

```
val p = new Point(1, 2)     // Point = Point@3c990add
val coll = HashSet(p)       // mutable.HashSet[Point] = Set(Point@3c990add)
coll contains p             // true

// now we change a field:
p.x += 1
coll contains p             // false
coll.iterator contains p    // true

// what happened is that now 'hashCode' evaluates to different value, so 'contains'
// looks for provided element in a wrong bucket
```

- if you need a comparison that takes the current state of an object into account, you should name the method differently, something other than `equals`, e.g. `equalContents`

Failing to define `equals` as an equivalence relation (same as Java)

- the contract of the `equals` method in `scala.Any`:
- **reflexive**: ***for any non-null value `x`, the expression `x.equals(x)` should return `true`***
- **symmetric**: ***for any non-null values `x` and `y`, the expression `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`***
- **transitive**: ***for any non-null values `x`, `y` and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`***
- **consistent**: ***for any non-null values `x` and `y`, multiple invocations of `x.equals(y)` should consistently return `true` or consistently return `false`, provided no information used in equals comparisons is modified***
- ***for any non-null value `x`, the expression `x.equals(null)` should return `false`***
- the "better equals" definition conforms to ***the contract***, but things become more complicated when inheritance is introduced:

```
object Color extends Enumeration {
   val Red, Orange, Yellow, Green, Blue, Indigo, Violet = Value
}

// non-symmetric equals:
class ColoredPoint(x: Int, y: Int, val color: Color.Value) extends Point(x, y) {
   override def equals(other: Any) = other match {
      case that: ColoredPoint => this.color == that.color && super.equals(that)
      case _ => false
   }
// doesn't need to override 'hashCode', since this 'equals' is stricter than point's
// so the contract for 'hashCode' is satisfied
// If two colored points are equal, they must have the same coordinates, so their hash
// codes are guaranteed to be equal as well
}

// the 'equals' contract becomes broken once points and color points are mixed:
val p = new Point(1, 2)                          // Point = Point@6aeee
val cp = new ColoredPoint(1, 2, Color.Red)    // ColoredPoint = ColoredPoint@6aeee
p equals cp       // true  - invokes p's equals (from class 'Point')
```

```
cp equals p       // false - invokes cp's equals (from class 'ColoredPoint')

// so the relation defined by 'equals' is not symmetric, which can have unexpected
// consequences for collections:
HashSet[Point](p) contains cp   // true
HashSet[Point](cp) contains p   // false
```

- to fix the symmetry problem, you can either make the relation more general or more strict
- making it more general means that a pair of two objects, `x` and `y`, is taken to be equal if either comparing `x` with `y` or comparing `y` with `x` yields true:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value) extends Point(x, y) {
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case that: Point =>   // special comparison for points
      that equals this
    case _ =>
      false
  }
}
// Now, both 'cp equals p' and 'p equals cp' result in 'true'
// However, the contract for equals is still broken. Now the problem is that the new
// relation is no longer transitive:
val redp = new ColoredPoint(1, 2, Color.Red)
val blup = new ColoredPoint(1, 2, Color.Blue)

// taken individually:
redp == p   // true
p == blup   // true

// but:
redp == blup   // false - OK, but transitivity is messed up
```

- making relation more general seems to lead to a dead end, so we'll try stricter
- one way of making `equals` stricter is to always treat object of different classes as not equal:

```
// technically valid, but still not perfect 'equals':
class Point(val x: Int, val y: Int) {
  override def hashCode = 41 * (41 + x) + y
  override def equals(other: Any) = other match {
    case that: Point =>
      this.x == that.x && this.y == that.y && this.getClass = that.getClass
    case _ => false
  }
}

// revert back to non-symmetric one (which is, with this version of Point, symmetric)
class ColoredPoint(x: Int, y: Int, val color: Color.Value) extends Point(x, y) {
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case _ => false
```

```
  }
}

// here, an instance of class 'Point' is equal to some other instance of the same class
// only if the objects have the same field values and same runtime class

// but still, an anonymous subclass of point, with the same field values:
val anonP = new Point(1, 1) { override val y = 2 }    // Point = $anon$1@34e0a

// is 'anonP' equal to 'p'? Logically it is, but technically it isn't, which is what we
// don't want in Scala
```

- to redefine equality on several levels of the class hierarchy while keeping its contract, we are required to redefine one more method, `canEqual`:

```
// signature:
def canEqual(other: Any): Boolean
```

- `canEqual` should return `true` if the provided object is an instance of the class in which `canEqual` is (re)defined
- as soon as a class redefines `equals`, it should also explicitly state that objects of this class are never equal to objects of some superclass that implements a different equality method
- `canEqual` is called from `equals` to make sure that the objects are comparable both ways

```
class Point(val x: Int, val y: Int) {
  override def hashCode = 41 * (41 + x) + y
  override def equals(other: Any) = other match {
    case that: Point =>
      (that canEqual this) && (this.x == that.x) && (this.y == that.y)
    case _ =>
      false
  }
  def canEqual(other: Any) = other.isInstanceOf[Point]    // all points can be equal
}

class ColoredPoint(x: Int, y: Int, val color: Color.Value) extends Point(x, y) {
  override def hashCode = 41 * super.hashCode + color.hashCode
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (that canEqual this) && super.equals(that) && this.color == that.color
    case _ =>
      false
  }
  override def canEqual(other: Any) = other isInstanceOf[ColoredPoint]
}

// now:
val p = new Point(1, 2)
val cp = new ColoredPoint(1, 2, Color.Indigo)
val anonP = new Point(1, 1) { override val y = 2 }
val coll = List(p)      // List[Point] = List(Point@4aa)
coll contains p         // true
coll contains cp        // false
```

```
coll contains anonP   // true
```

- if superclass `equals` defines and calls `canEqual`, then programmers who implement subclasses can decide whether or not their subclasses may be equal to instances of the superclass

698 - Defining equality for parameterized types

- when classes are parameterized, `equals` pattern matching scheme needs to be adapted

```
// binary tree with two implementations
trait Tree[+T] {
  def elem: T
  def left: Tree[T]
  def right: Tree[T]
}

object EmptyTree extends Tree[Nothing] {
  def elem = throw new NoSuchElementException("EmptyTree.elem")
  def left = throw new NoSuchElementException("EmptyTree.left")
  def right = throw new NoSuchElementException("EmptyTree.Right")
}

class Branch[+T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T]

// implementing 'equals' and 'hashCode':
// no need for class 'Tree' - subclasses redefine them
// no need for object 'EmptyTree' - AnyRef's implementations are fine, after all,
// any empty tree is only equal to itself, so reference equality is just what we need

// a 'Branch' value should only be equal to other 'Branch' values, and only if they
// have equal 'elem', 'left' and 'right' fields:
class Branch[+T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {
  override def equals(other: Any) = other match {
    case that: Branch[T] =>
      this.elem == that.elem &&
      this.left == that.left &&
      this.right == that.right
    case _ => false
  }
}
// compiling this code gives 'unchecked' warning:
//  warning: non variable type-argument T in type pattern is unchecked since it is
//  eliminated by erasure

// we saw this before, compiler can only check that the 'other' reference is some kind
// of 'Branch'. It cannot check that the element type of the tree is 'T'
// The reason for this is that element types of parameterized types are eliminated by
// the compiler's erasure phase, so they are not available for inspection at runtime
```

```
// fortunately, we don't even need to check that two branches have the same element
// types. It's quite possible that, in order to declare two branches as equal, all
// we need to do is compare their fields:
val b1 = new Branch[List[String]](Nil, EmptyTree, EmptyTree)
val b2 = new Branch[List[Int]](Nil, EmptyTree, EmptyTree)
b1 == b2   // true
// the result shows that the element types was not compared (otherwise would be 'false')
```

- there's only a small change needed in order to formulate `equals` that does not produce `unchecked` warning, instead of element type `T`, use a lower case letter, such as `t`:

```
case that: Branch[t] =>
  this.elem == that.elem &&
  this.left == that.left &&
  this.right == that.right

// the reason this works is that a type parameter in a pattern starting with a lower
// case letter represents an unknown type, hence the pattern match:
case that: Branch[t] =>
// will succeed for 'Branch' of any type

// it can also be replaced with the underscore
case that: Branch[_] =>    // equivalent to the previous case, with 't'
```

- the only thing that remains, for class `Branch`, is to define the other two methods, `hashCode` and `canEqual`

```
// the possible implementation of 'hashCode':
override def hashCode: Int =
  41 * (
    41 * (
      41 + elem.hashCode
    ) + left.hashCode
  ) + right.hashCode

// canEqual implementation:
def canEqual(other: Any) = other.isInstanceOf[Branch[_]]
// 'Branch[_]' is a shorthand for so-called 'existential type', which is roughly
// speaking a type with some unknown parts in it (next chapter)
// so even though technically the underscore stand for two different things in a match
// pattern and in a type parameter of a method call, in essence the meaning is the same:
// it lets you label something that is unknown
```

703 - Recipes for `equals` and `hashCode`

- `equals` recipe:

- **1.** if you're going to override equals in a non-final class, you should crate a `canEqual` method. If the inherited definition of `equals` is from `AnyRef` (not redefined higher up the class hierarchy), the definition of `canEqual` will be new, otherwise it will override a previous definition of a method with the same signature. The only exception to this requirement is for final classes that redefine the `equals` method inherited from `AnyRef`. For them, the subclass anomalies cannot arise.

Consequently, they need not define `canEqual`. The type of the object passed to `canEqual` should be `Any`:

```
def canEqual(other: Any): Boolean = /* ... */
```

- **2.** the `canEqual` method should yield `true` if the argument object is an instance of the current class (i.e. the class in which `canEqual` is defined), `false` otherwise:

```
other.isInstanceOf[Rational]
```

- **3.** in the `equals` method, make sure you declare the type of the object passed as `Any`:

```
override def equals(other: Any): Boolean = /* ... */
```

- **4.** write the body of the `equals` method as a single `match` expression. The selector of the `match`should be the object passed to `equals`:

```
other match {
  // ...
}
```

- **5.** the `match` expression should have two cases, the first should declare a typed pattern for the type of the class on which you're defining `equals`:

```
case that: Rational =>
```

- **6.** in the body of this, first ***case***, write an expression that logical-ands together the individual expressions that must be `true` for the objects to be equal. If the `equals` you're overriding is not that of `AnyRef`, you'll most likely want to include an invocation of the superclass's `equals`

```
super.equals(that) &&    // ...
```

if you're defining `equals` for a class that first introduced `canEqual`, you should invoke `canEqual` on the argument to the equality method, passing `this` as the argument:

```
(that canEqual this) &&    // ...
```

overriding definitions of `equals` should also include the `canEqual` invocation, unless they contain a call to `super.equals`. In the latter case, the `canEqual` test will already be done by the superclass call. For each field, relevant to equality, verify that the field in this object is equal to the corresponding field in the passed object:

```
numer == that.numer && denom == that.denom
```

- **7.** for the second ***case***, use a wildcard pattern that yields false:

```
case _ => false
```

- `hashCode` recipe:

- include in the calculation each field in your object that is used to determine equality in the `equals` method (_the relevant fields_)
- for each relevant field, no matter its type, you can calculate a hash code by invoking `hashCode` on it
- to calculate hash code for the entire object, add **41** to the first field's hash code, multiply that by 41, add the second field's hash code, multiply that by 41, add the third field's hash code, multiply that by 41, and continue until you've done this for all relevant fields:

```
// hash code calculation for object with 5 relevant fields:
override def hashCode: Int =
  41 * (
    41 * (
      41 * (
        41 * (
          41 + a.hashCode
        ) + b.hashCode
      ) + c.hashCode
    ) + d. hashCode
  ) + e.hashCode
```

- you can leave off the `hashCode` invocation on fields of type `Int`, `Short`, `Byte` and `Char` (their hash codes are their values):

```
override def hashCode: Int =
  41 * (
    41 + numer
  ) + denom
```

- the number 41 was selected because it is an odd prime (you could use another number, but it should be an odd prime to minimize the potential for information loss on overflow). The reason we add 41 to the innermost value is to reduce the likelihood that the first multiplication will result in zero, under the assumption that it is more likely the first field will be zero than -41 (there, in addition part, it could be any other non-zero integer, 41 was chosen just because of other 41s)
- if the `equals` invokes `super.equals(that)`, you should start your `hashCode` with an invocation of `super.hashCode`. For example, had `Rational`'s `equals` method invoked `super.equals(that)`, its `hashCode` would have been:

```
override def hashCode: Int =
  41 * (
    41 * (
      super.hashCode
```

```
    ) + numer
  ) + denom
```

- one thing to keep in mind when defining `hashCode` like this is that your hash code will only be as good as the hash codes you build it out of, namely the hash codes you obtain by calling `hashCode`on the relevant fields. Sometimes you may need to do something extra besides just calling `hashCode` on the field to get a useful hash code for that field. For example, if one of your fields is a collection, you probably want a hash code for that field that is base on all the elements contained in the collection. If the field is a `List`, `Set`, `Map` or ***tuple***, you can simply call `hashCode` on the field, since `equals` and `hashCode` are overridden in those classes to take into account the contained elements. However, the same is not true for arrays, which do not take their elements into account when calculating `hashCode`. Thus for an array, you should treat each element of the array like an individual field of your object, calling `hashCode` on each element explicitly, or passing the array to one of the `hashCode` methods in singleton object `java.util.Arrays`
- if you find that particular hash code calculation is harming the performance of your program, you can consider caching the hash code. If the object is immutable, you can calculate the hash code when the object is created and store it in a field. You can easily do this by overriding `hashCode` with a `val` instead of a `def`:

```
override val hashCode: Int =
  41 * (
    41 + numer
  ) + denom

// trades off memory for computation time, since now each instance will have one more
// field to hold
```

- given how difficult is to correctly implement an equality method, you might opt out to define your classes of comparable objects as case classes. That way, the Scala compiler will add `equals` and `hashCode` with the right properties automatically

Combining Scala and Java

- Scala code is often used in tandem with large Java programs and frameworks, which, from time to time, runs into a few common problems

710 - Using Scala from Java

- if you call Scala code from Java, it's useful to understand how the whole system works and how Scala code looks from a Java point of view

General rules

- Scala is implemented as a translation to standard Java bytecode
- as much as possible, Scala features map directly onto the equivalent Java features (e.g. classes, methods, strings, exceptions, method overloading all map directly to Java)
- there are Scala features that has its own design. Traits, for example, have no equivalent in Java. Generic types are handled completely differently. Scala encodes language features like these using some combination of structures Java does have

- for features that are mapped indirectly, encoding is not fixed and there is an ongoing effort to make translations as simple as possible

Value types

- a value type, like `Int` can be translated in two different ways. Whenever possible, compiler translates a Scala `Int` to a Java `int` to get better performance
- sometimes this is not possible, because the compiler is not sure whether it is translating an `Int` or some other data type (e.g. a `List[Any]` might hold only integers, but the compiler cannot be sure that's the case)
- in cases like this the compiler uses objects and relies on wrapper classes and autoboxing

Singleton objects

- translation of singleton objects uses a combination of static and instance methods
- for every singleton object, the compiler creates a Java class with a dollar sign added to the end (e.g. for singleton `App`, compiler produces `App$`), which has all the methods and fields of the singleton object
- the Java class also has a single static field named `MODULE$` which holds the one instance of the class that is created at runtime

```
// for Scala singleton:
object App {
  def main(args: Array[String]) {
    println("Hello, universe!")
  }
}

// compiler generates Java code:
public final class App$ extends java.lang.Object implements scala.ScalaObject {
  public static final App$ MODULE$;
  public static {};
  public App$();
  public void main(java.lang.String[]);
  public int $tag();
}

// an important special case is if you have a standalone singleton object, one which does
// not come with a class of the same name
// in that case, compiler creates a Java class named 'App' that has a static forwarder
// method for each method of the singleton:
public final class App extends java.lang.Object{
  public static final int $tag();
  public static final void main(java.lang.String[]);
}

// if you, on the other hand, have a class named 'App', Scala would create a
// corresponding Java 'App' class to hold the members of the 'App' class you defined
// It would not add any forwarding methods for the same-named singleton object, and
// Java code would have to access the singleton via the 'MODULE$' field
```

Traits as interfaces

- compiling any trait creates a Java interface of the same name, which is usable as a Java type, and it lets you call methods on Scala objects through variables of that type
- implementing a trait in Java is not practical, with one special case being important:

- if you make a Scala trait that includes only abstract methods, then that trait will be translated directly to a Java interface, with no other code to worry about

713 - Annotations

- when the compiler sees an annotation, it first processes it according to the general Scala rules, and then it does something extra for Java

Deprecation

- for any method or class marked `@deprecated`, the compiler adds Java's own deprecation annotation
- because of this, Java compiler can issue deprecation warnings when Java code accesses deprecated Scala methods

Volatile fields

- same as for deprecation, thus volatile fields in Scala behave exactly according to Java's semantics, and accesses to volatile fields are sequenced precisely according to the rules specified for volatile fields in Java memory model

Serialization

- all of the 3 Scala's standard serialization annotations are translated to Java equivalents
- a `@serializable` class has Java's `Serializable` interface added to it

```
// a '@SerialVersionUID(1234L)' is converted to the following Java definition:
private final static long SerialVersionUID = 1234L
```

- any variable marked `@transient` is given the Java `transient` modifier

Exceptions thrown

- Scala does not check that thrown exceptions are caught, that is, Scala has no equivalent to Java's `throws` declaration on methods
- all Scala methods are translated to Java methods that declare no thrown exceptions
- the reason it all still works is that the Java bytecode verifier does not check the declarations anyway. The Java compiler checks, but not the verifier
- the reason this feature is omitted from Scala is that the Java experience with it has not been purely positive, because annotating methods with `throws` clauses is a heavy burden for developers
- the result is that this often makes code less reliable (programmers often, in order to satisfy the compiler, either throw all they can or catch-and-release exceptions)
- sometimes, when interfacing with Java, you may need to write Scala code that has Java-friendly annotations describing which exceptions your methods may throw, e.g. each method in Java RMI is required to mention `java.io.RemoteException` in its `throws` clause, thus, if you wish to write a RMI interface as a Scala trait with abstract methods, you would need to list `RemoteException`in the `throws` clauses for those methods
- to accomplish this, all you have to do is mark your methods with `@throws` annotations:

```
// a method marked as throwing 'IOException'
import java.io._
class Reader(fname: String) {
  private val in = new BufferedReader(new FileReader(fname))

  @throws(classOf[IOException])
```

```
  def read() = in.read()
}
// and here is how it looks in Java:
public class Reader extends java.lang.Object implements
scala.ScalaObject{
  public Reader(java.lang.String);
  public int read() throws java.io.IOException;    // proper Java 'throws'
  public int $tag();
}
```

Java annotations

- existing annotations from Java frameworks can be used directly in Scala code
- any Java framework will see the annotations you write just as if you were writing Java
- a wide variety of Java packages use annotations, e.g. `JUnit`, which, from its version 4, allows you to use annotations to indicate which parts of your code are tests

```
// annotations required in some Java libraries can be used the same way in Scala:
import org.junit.Test
import org.junit.Assert.assertEquals

class SetTest {
  @Test    // this notation may be used instead of '@org.junit.Test', since we did import
  def testMultiAdd {
    val set = Set() + 1 + 2 + 3 + 1 + 2 + 3
    assertEquals(3, set.size)
  }
}
```

Writing your own annotations

- to make an annotation that is visible by Java reflection, you must use Java notation and compile with `javac`
- for this use case, writing the annotation in Scala does not seem helpful, so the standard compiler does not support it. The reason is that Scala support would inevitably fall short of the full possibilities of Java annotations, and further, Scala will probably one day have its own reflection, in which case you would want to access Scala annotations with Scala reflection

```
// this is Java
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Ignore{}
// after compiling the above with 'javac', we can use the annotations:
object Tests {
  @Ignore    // ignores the method even though its name starts with 'test'
  def testData = List(0, 1, -1, 5, -5)

  // test methods:
  def test1 {
    assert(testData == (testData.head :: testData.tail))
  }
  def test2 {
    assert(testData.contains(testData.head))
```

```
  }
}
```

- to see when these annotations are present you can use the Java reflection API:

```
for {
  method <- Tests.getClass.getMethods
  if method.getName.startsWith("test")
  if method.getAnnotation(classOf[Ignore]) == null
} {
  println("found a test method: " + method)
}

// this code in action:
$ javac Ignore.java
$ scalac Tests.scala
$ scalac FindTests.scala
$ scala FindTests
//> found a test method: public void Tests$.test2()
//> found a test method: public void Tests$.test1()
```

718 - Existential types

- all Java types have a Scala equivalent, which is necessary so that Scala code can access any Java class
- most of the time the translation is straightforward, `Pattern` in Java is `Pattern` in Scala, `Iterator<Component>` in Java is `Iterator[Component]` in Scala, but for some cases Scala types we mentioned so far are not enough
- for example, what can be done with Java wildcard type parameters such as `Iterator<?>` or `Iterator<? extends Component>`?
- also, what can be done with raw types like `Iterator`, where the type parameter is omitted?
- for wildcard and raw types Scala uses an extra kind of type called **existential type**
- existential types are fully supported part of the language, but in practice they are mainly used when accessing Java types from Scala (we are covering this mostly so that you can understand error messages when your Scala code accesses Java code)

```
// the general form of an existential type:
/* type */ forSome { /* declarations */ }

// the 'type' part is an arbitrary Scala type
// the 'declarations' part is a list of abstract 'vals' and types
```

- the interpretation is that the declared variables and types exist but are unknown, just like abstract members of a class
- the 'type' is then allowed to refer to the declared variables and types even though it is unknown what they refer to

```
// Java 'Iterator<?>' would be written in Scala as:
Iterator[T] forSome { type T }    // an iterator of T's for some type 'T'
```

```
// Java 'Iterator<? extends Component>' would be:
Iterator[T] forSome { type T <: Component }
// iterator of 'T' for some type 'T' that is a subtype of 'Component'

// there is a shorter way to write this examples with placeholder notation:
Iterator[_]
// means the same as
Iterator[T] forSome { type T }

// similarly:
Iterator[T] forSome { type T <: Component }
// is the same as:
Iterator[_ <: Component]
```

- ***placeholder syntax*** is similar to the placeholder syntax for function literals, where, if you use an underscore in place of an expression, then Scala creates a function literal for you. Here, when you use an underscore in place of a type, Scala makes an ***existential type*** for you, where each underscore becomes one type parameter in a `forSome` clause
- so if you use multiple underscores in the same type, you will get the effect of a `forSome` clause with multiple types in it

```
// existential types usage example:
// Java class with wildcards:
public class Wild {
  Collection<?> contents() {
    Collection<String> stuff = new Vector<String>();
    stuff.add("a");
    stuff.add("b");
    stuff.add("see");
    return stuff;
  }
}
// if you access the above class in Scala:
val contents = (new Wild).contents
// java.util.Collection[?0] for Some { type ?0 } = [a, b, see]
contents.size()   // Int = 3
```

722 - Using `synchronized`

- for compatibility's sake, Scala provides access to Java's concurrent primitives (`wait`, `notify`and `notifyAll` may all be called in Scala and have the same meaning as in Java)
- technically, Scala doesn't have the `synchronized` keyword, instead it has a predefined `synchronized` method that can be called like this:

```
var counter = 0
synchronized {
  // one thread at a time
  counter = counter + 1
}
```

722 - Compiling Scala and Java together

- usually, when you compile Scala code that depends on Java code, you first build the java code to class files and then you build the Scala code, putting the Java class files on the classpath
- this approach doesn't work if the Java code has references back to Scala code. In that case one side will have unsatisfied external references
- to support such builds, Scala allows compiling against Java source code as well as Java class files. All you have to do is put the Java source files on the command line as if they were scala files. Then, Scala compiler won't compile those Java files, but it will scan them to see what they contain
- so the right sequence is to first compile Scala using Java source files, and then compile Java code using Scala class files:

```
$ scalac -d bin InventoryAnalysis.scala InventoryItem.java Inventory.java
$ javac -cp bin -d bin Inventory.java InventoryItem.java InventoryManagement.java
$ scala -cp bin InventoryManagement
//> Most expensive item = sprocket($4.99)
```

Actors and Concurrency

- Java's native, thread-based concurrency support is hard to work with and prone for errors, especially when programs get large and complex
- Scala augments Java's concurrency by adding **actors**
- actors provide a concurrency model that is easier to work with and can help you avoid many of the difficulties of using Java's native concurrency model

724 - Java concurrency troubles

- Java's built-in threading model is based on shared data and locks
- each object is associated with a logical ***monitor***, which is used to control multi-threaded access to data. In this model, you decide what data will be shared by multiple threads and enclose those, shared sections of the code with `sinchronized` blocks or methods
- the problem with this model is that, in each point in a program, you must reason about what data you're modifying or accessing that might be modified or accessed by other threads. At each method call, you must reason about what locks it will try to hold, and convince yourself that it will not deadlock while trying to obtain those locks
- making things even worse, testing is not reliable with multi-threaded code. Since threads are non-deterministic, you might successfully test a program a thousand times, and yet still the program could go wrong
- with this model, you must get it right, i.e. avoid deadlocks and race conditions through reason alone

725 - Actors and message passing

- Scala's actors library addresses the fundamental problem by providing an alternative, **share-nothing**, message-passing model
- an **actor** is a thread-like entity that has a **mailbox** for receiving messages
- to implement an actor, you subclass `scala.actors.Actor` and implement the `act` method:

```
import scala.actors._
object Shakespeare extends Actor {
  def act() {
    for (i <- 1 to 2) {
      println("To be or not to be. Is that a question?")
      Thread.sleep(3000)
```

```
    }
  }
}

// you start an actor by invoking its 'start' method
Shakespeare.start()
// To be or not to be. Is that a question?
// To be or not to be. Is that a question?
```

- actors run completely independently from one another:

```
import scala.actors._
object Hamlet extends Actor {
  def act() {
    for (i <- 1 to 2) {
      println("Yes, that was a question.")
      Thread.sleep(3000)
    }
  }
}

Shakespeare.start()
Hamlet.start()
// To be or not to be. Is that a question?
// Yes, that was a question.
// To be or not to be. Is that a question?
// To be or not to be. Is that a question?
// Yes, that was a question.
// Yes, that was a question.
```

- you can also create an actor using a utility method named `actor` in object `scala.actors.Actor`:

```
import scala.actors.Actor._
// this actor starts immediately when it's defined (no need to call 'start()' method)
val desdemona = actor {
  for (i <- 1 to 3) {
    print("I rule! ")
    Thread.sleep(1000)
  }
}

// I rule! I rule! I rule!
```

- actors communicate by sending each other **messages**
- message is sent using `!` method:

```
Hamlet ! "hi there"
```

- nothing happens in this case, because `Hamlet` is too busy acting to process its messages, and so the `"hi there"` message sits in its mailbox unread
- an actor can wait for new messages in its mailbox:

```
val parrot = actor {
  while (true) {
    receive {
      case msg => println(msg)
    }
  }
}
```

- when an actor sends a message, it does not block, and when an actor receives a message, it is not interrupted. The message waits in the receiving actor's mailbox until the actor calls `receive`, passing in a partial function
- remember? ***Partial function*** is not a full function, i.e. it might not be defined over all input values. ***Partial function literal*** is expressed as a series of `match` cases, so it looks like a `match`expression without the `match` keyword. It is actually an instance of the `PartialFunction` trait)
- in addition to `apply` method that takes one argument, a partial function offers `isDefinedAt`method (which also takes one argument) that returns `true` if the partial function can ***"handle"*** the passed value, which then means that the value is safe to pass to `apply`. If `isDefinedAt` returns `false` and that value is passed to `apply`, then `apply` throws an exception
- an actor will only process messages matching one of the cases in the partial function passed to `receive`
- for each message in the mailbox, `receive` first invokes `isDefinedAt` on the passed partial function to determine whether it has a case that will match and handle the message. The `receive`method then chooses the first message in the mailbox for which `isDefinedAt` returns `true`, and pass that message to the partial function's `apply` method, which then handles the message. If the mailbox contains no message for which `isDefinedAt` returns `true`, the actor on which `receive`was invoked will block until a matching message arrives

```
// actor that only handles ints:
val intParrot = actor {
  receive {
    case x: Int => println("got int: " + x)
  }
}

intParrot ! "hello"    // silently ignores it
intParrot ! math.Pi    // silently ignores it
intParrot ! 8          // got int: 8
```

729 - Treating native threads as actors

- the actor subsystem manages one or more native threads for its own use. So long as you work with an explicit actor that you define, you don't need to think much about how they map to threads
- the other direction is also supported, every native thread is also usable as an actor, however, you cannot use `Thread.currentThread` directly, because it does not have the necessary methods. Instead, you should use `Actor.self` if you want to view the current thread as an actor
- this facility is especially useful for debugging actors from the interactive shell:

```
import scala.actors.Actor._
self ! "hello"
self.receive { case x => x }   // Any = hello
// the `receive` method returns the value computed by the partial function passed to it,
// and in this case, the partial function returns the message itself, so the received
// message ends up being printed out by the interpreter
```

- if you use `receive` in the interpreter, the `receive` will block the shell until a message arrives (in case of `self.receive` this could mean waiting forever), thus, when using this technique, it's better to use a variant called `receiveWithin`, which allows you to specify a timeout in milliseconds:

```
self.receiveWithin(1000) { case x => x }   // wait for 1s
// Any = TIMEOUT

// 'receiveWithin' processes mailbox message or waits until for new messages for
// specified number of millis
```

730 - Better performance through thread reuse: `react`

- threads are not cheap! They consume enough memory that typical Java VMs, which normally hosts millions of objects, can have only thousands of threads. Moreover, switching threads often takes hundreds if not thousands of processor cycles
- to help you conserve threads, Scala provides an alternative the the usual `receive` method called `react`. Like `receive`, `react` takes a partial function, but unlike `receive` it does not return after it finds and processes a message. Its result type is `Nothing`. It evaluates the message handler and that's it (throws an exception that's caught behind the scenes)
- since `react` doesn't need to return, the implementation doesn't need to preserve the call stack of the current thread. Instead, the library can reuse the current thread for the next actor that wakes up. This approach is so effective that if every actor in a program used `react`, only a single thread would be used (Scala utilizes all processor cores that it can)

In practice, programs need at least a few `receive`s, but you should try to use `react` whenever possible

- because `react` does not return, the message handler you pass it must now both, process that message and arrange to do all of the actor's remaining work. A common way to do this is to have a top-level work method (such as `act` itself), that the message handler calls when it finishes

```
object NameResolver extends Actor {
  import java.net.{InetAddress, UnknownHostException}

  def act() {
    react {
      case (name: String, actor: Actor) =>
        actor ! getIp(name)
        act()
      case "EXIT" =>
        println("Name resolver over and out.")
        // quit
      case msg =>
        println("Unhandled message: " + msg)
```

```
        act()
      }
    }

    def getIp(name: String): Option[InetAddress] = {
      try {
        Some(InetAddress.getByName(name))
      } catch {
        case _: UnknownHostException => None
      }
    }
}

NameResolver.start()    // scala.actors.Actor = NameResolver$@2d325da0
NameResolver ! ("www.scala-lang.org", self)
self.receiveWithin(0) { case x => x }  // Any = Some(www.google.com/83.139.106.223)
```

- this coding pattern is so common with event-base actors, that there's a special support in the library for it. The `Actor.loop` executes a block of code repeatedly, even if the code calls `react`:

```
def act() {
  loop {
    react {
      case (name: String, actor: Actor) =>
        actor ! getIp(name)
      case msg =>
        println("Unhandled message: " + msg)
    }
  }
}
```

734 - How `react` works

- it has a return type of `Nothing` and it never returns normally, i.e. always ends with an exception
- conceptually, when you call `start` on an actor, the `start` method will in some way arrange things so that some thread will eventually call `act` on that actor. If the `act` method invokes `react`, the `react` method looks in the actor's mailbox for a message that the passed partial function can handle
- `receive` does this the same way, passes candidate messages to the partial function's `isDefinedAt` method
- if it finds a message that can be handled, `react` will schedule the handling of that message for later execution and throw an exception
- if it doesn't find one, it will place the actor in ***cold storage*** to be resurrected if and when it gets more messages in its mailbox, and then throw an exception
- in either case, `react` will complete abruptly with this exception and so will `act`
- the thread that invoked `act` will catch the exception, forget about the actor, and move on to do other things
- this explains why, if you want `react` to handle more than the first message, you have to call `act` again from inside your partial function, or use some other means to get `react` invoked again

733 - Good actors style

Actors should not block

- while an actor blocks, some other actor might make a request on it, and since it's blocked, it won't even notice the request. In the worst case, even a deadlock may happen, with multiple actors blocked as they each wait for some other blocked actor to respond
- instead of blocking, an actor should arrange for some message to arrive, designating that an action is ready to be taken. This rearrangement will often require help of other actors. E.g. instead of calling `Thread.sleep(time)`, you could create a helper actor that sleeps and then sends a message back when enough time has elapsed:

```
actor {
  Thread.sleep(time)
  mainActor ! "wakeup!"
}
// this actor blocks, but since it'll never receive a message that's OK
```

- the main actor remains available to answer new requests
- the `emoteLater` method demonstrates the use of this idiom. It creates a new actor that will do the `sleep` so that the main actor does not block
- to ensure that the actor will send the `Emote` message to the correct actor, it is prudent to evaluate `self` in the scope of the main actor instead of the scope of the helper actor
- because this actor does not block in `sleep`, it can continue to do other work while waiting for its next time to emote

```
// actor that uses helper actor to avoid blocking itself (echoes messages while waiting)
val sillyActor2 = actor {
  def emoteLater() {
    val mainActor = self
    actor {
      Thread.sleep(1000)
      mainActor ! "Emote"
    }
  }
  var emoted = 0
  emoteLater()

  loop {
    react {
      case "Emote" =>
        println("I'm really acting!")
        emoted += 1
        if (emoted < 5)
          emoteLater()
      case msg =>
        println("Received: " + msg)
    }
  }
}
```

Communicate with actors only via messages

- the key way the actors model addresses the difficulties of the shared data and locks is by providing a safe space, the actor's `act` method, where you can think sequentially, i.e. actors allow you to write a

multi-threaded program as a bunch of independent single-threaded programs that communicate with each other via asynchronous messages. So, all this works only if you abide by this simple rule, that the messages are the only way you let your actors communicate

- nevertheless, Scala actors library gives you the choice of using both message passing and shared data & locks in the same program. A good example is multiple actors sharing a reference to a `ConcurrentHashMap` (instead of using a single map owner actor and sending async messages to it) and alter the map synchronously - given that CHM is already implemented in Java concurrency library, thus it's guaranteed to be safe

Prefer immutable messages

- since actors model provides a single-threaded environment inside each actor's `act` method, we don't need to worry about whether the objects are thread-safe
- this is why the actors model is called ***share-nothing*** model
- there's one exception in the ***share-nothing*** rule: the data inside objects used to send messages between actors is ***shared*** by multiple actors. As a result, you ***do*** have to worry about whether message object are thread safe, and in general, they should be
- the best way to ensure that message objects are thread-safe is to only use immutable objects for messages (remember, object is immutable if it's an instance of any class that has only `val` fields, which themselves refer only to immutable objects)
- the easy way to define such message classes is to use ***case classes*** (so long as you don't explicitly add `var` field to it and ensure `val` fields are all immutable types)
- of course, for message instances you can also use a regular immutable class you define, or instances of many immutable classes provided by the Scala API, such as tuples, strings, lists, immutable sets and maps, and so on
- if an actor sends a mutable, unsynchronized object as a message, and never reads or writes that object thereafter, it would work, but it's just asking for trouble
- in general, it's recommended that you arrange your data such that every unsynchronized, mutable object is ***owned***, and therefore accessed by only one actor. You can arrange for objects to be transferred from one actor to another, but you need to make sure that, at any given time, it's clear which actor owns the object
- i.e. when you design an actor-based system, you need to decide which parts of ***mutable*** memory are assigned to which actor. All other actors should use this object through its owner actor, by passing messages to it
- if you happen to have a mutable object that you have to to send to another actor, you should make and send a copy of it instead
- but while you're at it, you may want to make it immutable. For example, arrays are mutable and unsynchronized. Any array you use should be accessed by one actor at a time. If you want to continue using it, as well as send it to another actor, you should send a copy. If the array holds only immutable objects, you can make a copy with `arr.clone`, but you should consider using `arr.toList` and send the resulting immutable list instead

Make messages self-contained

- when you return a value from a method, the caller is in a good position to remember what it was doing before it called the method. It can take the response value and then continue doing whatever it was doing
- with actors, things are little trickier. When one actor makes a request to another, the response might not come for a long time. The calling actor should not block, but should continue to do any other work it can while it waits for the response. A difficulty then, is interpreting the response when it finally does come back. How can actor remember what it was doing when it made the request?

- one way to simplify the logic of an actors program is to include redundant information in the messages. If the request is an immutable object, you can cheaply include a reference to the request in the return value. For example, the IP-lookup actor would be better if it returned the host name in addition to the IP address. It would make this actor slightly longer, but it should simplify the logic of any actor making requests on it:

```
def act() {
  loop {
    react {
      case (name: String, actor: Actor) =>
        actor ! (name, getIp(name))
    }
  }
}
```

- another way to increase redundancy in the messages is to make a case class for each kind of message. That makes an actors program much easier to understand. Imagine a programmer looking as a send of a string, for example:

```
lookerUpper ! ("www.scala-lang.org", self)
```

- it can be difficult to figure out which actors in the code might respond. It is much easier if the code looks like this:

```
case class LookupIP(hostname: String, requester: Actor)
lookerUpper ! LookupIP("www.scala-lang.org", self)
// now the programmer can search for 'LookupIP' in the source, probably finding
// very few possible responders
// updated name-resolving actor that uses case classes instead of tuples as messages:
import scala.actors.Actor._
import java.net.{InetAddress, UnknownHostException}

case class LookupIP(name: String, respondTo: Actor)
case class LookupResult(
  name: String,
  address: Option[InetAddress]
)

object NameResolver2 extends Actor {
  def act() {
    loop {
      react {
        case LookupIP(name, actor) =>
          actor ! LookupResult(name, getIp(name))
      }
    }
  }

  def getIp(name: String): Option[InetAddress] = {
    // same as before
    try {
      Some(InetAddress.getByName(name))
    } catch {
```

```
        case _: UnknownHostException => None
      }
    }
}

NameResolver2.start()    // scala.actors.Actor = NameResolver2$@7aa9a046
NameResolver2 ! LookupIP("www.gmail.com", self)
self.receiveWithin(0) { case x => x }
// Any = LookupResult(www.gmail.com,Some(www.gmail.com/173.194.70.17))
```

740 - Actors in action: Parallel discrete event simulation

- example of actors model on discrete event simulation code from chapter 18, changed so that events run in parallel
- the key idea is to make each simulated object an actor
- it is likely that there will be some common behavior between different simulated objects, so it makes sense to define a trait that can be mixed in any class to make it a simulated object:

```
trait Simulant extends Actor
class Wire extends Simulant
```

- we must decide how to make the simulation participants synchronized with the simulated time. E.g. participant A should not race ahead and process an event at time tick 100 until all other actors have finished with time tick 99
- we'll make such that no simulant should process events for time `n` until all other simulants are finished with time `n - 1`
- but how does a simulant know when it's safe to move forward? We'll use a "clock" actor that will keep track of the current time and tell simulants when it's time to move forward. To keep the clock from moving forward before all simulants are ready, the clock will ping actors at carefully chosen time to make sure they have received and processed all messages for the current time tick. Clock will send `Ping` and simulants will respond with `Pong` message stating they are ready for the clock to move forward:

```
case class Ping(time: Int)    // sender is always the "clock"
case class Pong(time: Int, from: Actor)    // 'time' and 'from' add redundancy
```

- simulants should not respond to a `Ping` until they have finished all the work for that tick, but how do they know their work is done?
- we'll add two more constraints. First, that simulants never send each other messages directly, but only schedule events on one another using 'simulation agenda'. Second, they never post events for the current time tick, only for times at least one tick into the future
- there will be a single agenda of work items, which will be held by the clock actor. That way, the clock can wait to send out pings until it has sent out requests for all work items of the current tick. Actors then know that, whenever they receive a `Ping`, they have already received from the clock all work items that need to happen at the current tick, thus it's safe that, when an actor receives a `Ping` to immediately send back a `Pong`, because no more work will be arriving during the current tick

```
class Clock extends Actor {
  private var running = false
  private var currentTime = 0
  private var agenda: List[WorkItem] = List()
}
```

- the final design issue is how a simulation is set up to begin with. A natural approach is to create the simulation with the clock stopped, add all the simulants, connect them all together, and then start the clock
- how do we know when the simulation is fully assembled so we can start the clock? We'll avoid sending messages to actors while setting the simulation up. The resulting code pattern will be that we use regular method calls to set up simulation, and actor messages while the simulation is running

```
// WorkItem will be designed slightly differently than in chapter 18:
case class WorkItem(time: Int, msg: Any, target: Actor)
// action is now comprised of a message and an actor (to whom that message is sent)

// afterDelay method becomes AfterDelay message that is sent to the clock:
case class AfterDelay(delay: Int, msg: Any, target: Actor)

// messages requesting simulation start and stop:
case object Start
case object Stop
```

Implementing the simulation framework

- the `Clock` class and the `Simulant` trait need to be implemented

```
class Clock extends Actor {
  private var running = false
  private var currentTime = 0
  private var agenda: List[WorkItem] = List()
  private var allSimulants: List[Actor] = List()
  private var busySimulants: Set[Actor] = Set.empty

  def add(sim: Simulant) {    // we must be able to add simulants to the clock
    allSimulants = sim :: allSimulants
  }


  def act() {
    loop {
      if (running && busySimulants.isEmpty)
        // advance clock once all simulants of the current time tick finish
        advance()
      reactToOneMessage()
    }
  }


  def advance() {
    if (agenda.isEmpty && currentTime > 0) {
      // stop simulation if agenda is clear and we're not setting up
      println("** Agenda empty. Clock exiting at time " + currentTime + ".")
```

```
        self ! Stop
        return
      }
      currentTime += 1
      println("Advancing to time " + currentTime)
      processCurrentEvents()
      for (sim <- allSimulants)   // ping all simulants
        sim ! Ping(currentTime)

      busySimulants = Set.empty ++ allSimulants   // add all simulants to busy set
    }


    private def processCurrentEvents() {
      // take items that need to occur at the current time
      val todoNow = agenda.takeWhile(_.time <= currentTime)
      // remove from agenda the items we just took
      agenda = agenda.drop(todoNow.length)

      for (WorkItem(time, msg, target) <- todoNow) {
        assert(time == currentTime)
        target ! msg   // send all current items to appropriate actors
      }
    }

    def reactToOneMessage() {
      react {
        case AfterDelay(delay, msg, target) =>
          val item = WorkItem(currentTime + delay, msg, target)
          agenda = insert(agenda, item)
        case Pong(time, sim) =>
          assert(time == currentTime)
          assert(busySimulants contains sim)
          busySimulants -= sim
        case Start => running = true
        case Stop =>
          for (sim <- allSimulants)
            sim ! Stop
          exit()
      }
    }

    private def insert(ag: List[WorkItem], item: WorkItem): List[WorkItem] = {
      // when you find position, append item and rest of ag to "heads"
      if(ag.isEmpty || item.time < ag.head.time) item :: ag
      // prepend head until you find position so it's sorted by time
      else ag.head :: insert(ag.tail, item)
    }
}
```

- boiled down, a `Simulant` is any actor that understands and cooperates with the simulation messages `Stop` and `Ping`:

```
trait Simulant extends Actor {
  val clock: Clock
  def handleSimMessage(msg: Any)
```

```
  def simStarting() { }   // not abstract

  def act() {
    loop {
      react {
        case Stop => exit()
        case Ping(time) =>
          // allows subclasses to do something before they respond with 'Pong'
          if (time == 1) simStarting()   // when simulation starts running
          clock ! Pong(time, self)
        case msg => handleSimMessage(msg)
      }
    }
  }
  start()   // starts immediately, it won't do nothing until clock receives 'Send' msg
}
```

Implementing a circuit simulation

```
class Circuit {
  // consists of:
  // simulation messages            // delay constants
  // Wire and Gate classes and methods  // misc. utility methods
  val clock = new Clock
  val WireDelay = 1
  val InverterDelay = 2
  val OrGateDelay = 3
  val AndGateDelay = 3

  // message types
  case class SetSignal(sig: Boolean)
  case class SignalChanged(wire: Wire, sig: Boolean)


  // simulant that has a current state signal and a list of gates observing this state
  class Wire(name: String, init: Boolean) extends Simulant {
    def this(name: String) { this(name, false)}
    def this() { this("unnamed") }

    val clock = Circuit.this.clock   // uses the circuit's clock
    clock.add(this)   // adds itself to the clock

    private var sigVal = init
    private var observers: List[Actor] = List()

    // process messages from the clock
    def handleSimMessage(msg: Any) {
      msg match {
        // the only message type a wire can receive
        case SetSignal(s) =>
          if (s != sigVal) {
            sigVal = s
            signalObservers()
          }
      }
    }
```

```
    // propagate changes to any gates watching the wire
    def signalObservers() {
      for (obs <- observers)
        clock ! AfterDelay(
          WireDelay,
          SignalChanged(this, sigVal),
          obs)
    }

    // we need to send the initial signal state to observers, as simulation starts
    override def simStarting() { signalObservers() }

    // a wire needs a method for connecting new gates
    def addObserver(obs: Actor) {
      observers = obs :: observers
    }

    // nice string representation
    override def toString = "Wire(" + name + ")"
}


// since gates 'And', 'Or' and 'Not' have a lot in common:
abstract class Gate(in1: Wire, in2: Wire, out: Wire) extends Simulant {
    val delay: Int
    def computeOutput(s1: Boolean, s2: Boolean): Boolean

    // since it mixes in 'Simulant' it's required to specify the clock
    val clock = Circuit.this.clock
    clock.add(this)

    // connect the gate to input wires
    in1.addObserver(this)
    in2.addObserver(this)

    // wires' state
    var s1, s2 = false

    // handle incoming messages that the clock sends
    def handleSimMessage(msg: Any) {
      msg match {
        case SignalChanged(w, sig) =>
          // change the state of wire
          if (w == in1)
            s1 = sig
          if (w == in2)
            s2 = sig

          // send message to output wire that the signal state changed
          clock ! AfterDelay(delay,
            SetSignal(computeOutput(s1, s2)),
            out)
      }
    }

}


def orGate(in1: Wire, in2: Wire, output: Wire) =
```

```
    new Gate(in1, in2, output) {
      val delay = OrGateDelay
      def computeOutput(s1: Boolean, s2: Boolean) = s1 || s2
    }

  def andGate(in1: Wire, in2: Wire, output: Wire) =
    new Gate(in1, in2, output) {
      val delay = AndGateDelay
      def computeOutput(s1: Boolean, s2: Boolean) = s1 && s2
    }

  private object DummyWire extends Wire("dummy")    // to simplify things
  // so we can define abstract gate as a class with 2 input wires

  def inverter(input: Wire, output: Wire) =
    new Gate(input, DummyWire, output) {    // inverter, of course, uses one input wire
      val delay = InverterDelay
      def computeOutput(s1: Boolean, ignored: Boolean) = !s1
    }


  // print out state changes of wires
  def probe(wire: Wire) = new Simulant {
    val clock = Circuit.this.clock
    clock.add(this)
    wire.addObserver(this)
    def handleSimMessage(msg: Any) {
      msg match {
        case SignalChanged(w, s) =>
          println("signal " + w + " changed to " + s)
      }
    }
  }


  // to allow clients to start simulation
  def start() { clock ! Start }

}
```

Combinator Parsing

- occasionally, you may need to process a small, special purpose language, e.g. read configuration files and make them easier to modify by hand than XML. Or e.g. you want to support an input language in your program, such as search terms with boolean operators
- whatever the reason, because you need to convert the input language into some data structure your software can process, you're going to need a ***parser***
- instead of using a standalone ***domain specific language***, we'll use an ***internal DSL***, which will consist of a library of **parser combinators**, i.e. functions and operators defined in Scala that will server as building blocks for parser, which will map one to one to the constructions of a context-free grammar, to make them easy to understand

760 - Example: Arithmetic expressions

- if you wanted to construct a parser for arithmetic expressions consisting of floating-point numbers, parentheses and the binary operators `+`, `-`, `*` and `/`, the first step would be to write down a grammar for the language to be parsed:

```
   expr ::= term {"+" term | "-" term}
   term ::= factor {"*" factor | "/" factor}
factor ::= floatingPointNumber | "(" expr ")"

// | denotes alternative productions, and {} denote repetition (zero or more times)
// [] denote an optional occurrence (not mentioned in grammar above)

/*
 - every expression is a term, which can be followed by a sequence of + or - operators
   and further terms
 - a term is a factor, possibly followed by a sequence of * or / operators and further
   factors
 - a factor is either a numeric literal or an expression in parentheses
*/

// actual code of 'Arith' class that consists of 3 parsers specified above in grammar:
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
   def expr: Parser[Any] = term~rep("+"~term | "-"~term).
   def term: Parser[Any] = factor~rep("*"~factor | "/"~factor).
   def factor: Parser[Any] = floatingPointNumber | "("~expr~")".
}

// 'JavaTokenParsers' is a trait that provides basics for writing a parser and also
// provides some primitive parsers that recognize some word classes: identifiers, string
// literals and numbers
// 'floatingPointNumber' is a primitive parser inherited from the trait

/*
 In order to translate grammar into source code:
  - every production becomes a method, so you need to prefix it with 'def'
  - the result type of each method is 'Parser[Any]' so you need to change ::= to
    ': Parser[Any]'
  - insert explicit operator ~ between every two consecutive symbols
  - repetition is expressed with 'rep()' instead of {}. Option is 'opt()' instead of []
  - the period at the end of each production is omitted (use a semicolon if you want)
*/
```

762 - Running the parser

- you can exercise the parser with the following small program:

```
object ParseExpr extends Arith {
   def main(args: Array[String]) {
      val arg = "2 * (3 + 7)"
      println("input: " + arg)
      println(parseAll(expr, arg))
   }
}
// 'parseAll(expr, input)' applies the parser 'expr' to the given 'input'
// there's also a method 'parse', which parses an input prefix
```

```
// input: 2 * (3 + 7)
// [1.12] parsed: ((2~List((*~(((~((3~List())~List((+~(7~List())))))~)))))~List())

// the output tells you that parser successfully analyzed the input string up to
// position [1.12], first row, 12th column (the whole input string)

// trying to parse an illegal expression:
val arg = "2 * (3 + 7))"
// input: 2 * (3 + 7))
// [1.12] failure: string matching regex `\z' expected but `)' found
```

763 - Basic regular expression parsers

- `floatingPointNumber` parser recognizes a floating point number in the Java format
- if you need to parse numbers in a format that's different from Java's, you can use a ***regular expression parser***
- you can use any regular expression as a parser. It parses all strings it matches. Its result is the parsed string, e.g:

```
// regex that describes all Java identifiers:
object MyParsers extends RegexParsers {    // trait
   val ident: Parser[String] = """[a-zA-Z_]\w*""".r
}
```

- Scala's parsing combinators are arranged in a hierarchy of traits, which are all contained in package `scala.util.parsing.combinator`
- the top level trait is `Parsers`, which defines a very general parsing framework
- one level bellow is trait `RegexParsers`, which requires that the input is a sequence of characters and provides a framework for regex parsing
- more specialized is trait `JavaTokenParsers`, which implements parsers for basic classes of tokens as they are defined in Java

764 - JSON parser

```
// the grammar that describes the syntax of JSON:
   value ::= obj | arr | stringLiteral | floatingPointNumber | "null" | "true" | "false".
     obj ::= "{" [members] "}".
     arr ::= "[" [values] "]".
members ::= member {"," member}.
 member ::= stringLiteral ":" value.
 values ::= value {"," value}.

/*
  value  - an object, array, string, number or one of reserved words: null, true, false
  object - possibly empty sequence of members separated by commas and enclosed in braces
  member - a string-value pair, where string and value are separated by colon
  array  - a sequence of values separated by commas, enclosed in square brackets
*/

// grammar translated to source code:
import scala.util.parsing.combinator._
class JSON extends JavaTokenParsers {
   def value:    Parser[Any] = obj | arr | stringLiteral |
```

```
                                floatingPointNumber | "null" | "true" | "false"
  def obj:    Parser[Any] = "{"~repsep(member, ","~)~"}"
  def arr:    Parser[Any] = "["~repsep(value, ","~)~"]"
  def member: Parser[Any] = stringLiteral~":"~value
}

// 'repsep' - combinator that parses a possibly empty sequence of terms, separated by
//         a given separator string

// we'll read from a file:
import java.io.FileReader
object ParseJSON extends JSON {
  def main(args: Array[String]) {
    val reader = new FileReader(args(0))
    // parseAll is overloaded: takes sequence or input reader as a second argument
    println(parseAll(value, reader))
  }
}

// example JSON in a file:
{
  "address book": {
    "name": "John Smith",
    "address": {
      "street": "10 Market Street",
      "city"  : "San Francisco, CA",
      "zip"   : 94111
    },
    "phone numbers": [
      "408 338-4238",
      "408 111-6892"
    ]
  }
}

// output:
// [14.2] parsed: (({~List((("address book"~:)~(({~List((("name"~:)~"John Smith"),
// ((("address"~:)~(({~List((("street"~:)~"10 Market Street"), (("city"~:)~
// "San Francisco, CA"), (("zip"~:)~94111)))~})), (("phone numbers"~:)~(([~List(
// "408 338-4238", "408 111-6892"))~])))~})))~})
```

766 - Parser output

- in order to customize the parser output, we first need to know what the individual parsers and combinator frameworks return as a result:
- each parser written as a string returns the parsed string itself
- regex parsers also return parsed string
- a sequential composition P~Q returns the results of both P and Q in an instance of a case class that is also written as ~. So if P returns "true" and Q returns "?", then P~Q returns `~("true", "?")`, which prints as `(true~?)`
- an alternative composition P | Q returns the result of either P or Q, whichever one succeeds
- the repetition `rep(P)` or `repsep(P, separator)` returns a list of the results of all runs of P
- an option `opt(P)` returns an instance of `Option` type, thus returning `Some(R)` if P succeeds with result R or `None` if P fails

- a better output of the JSON parser would be to map a JSON object into an internal Scala representation:
- JSON object is represented as a map of type `Map[String, Any]`
- JSON array is represented as a list of type `List[Any]`
- JSON string is represented as Scala `String`
- JSON numeric literal is represented as a `Double`
- values `true`, `false` and `null` are represented as corresponding Scala values
- to produce this representation, you need to make use of one more combination form for parsers: `^^` operator, which transforms the result of a parser
- in `P ^^ f`, where `P` is a parser and `f` is a function, whenever `P` returns with some result `R`, the result of `P ^^ f` is `f(R)`:

```
// parser that parses floating point number and converts it to Double:
floatingPointNumber ^^ (_.toDouble)

// parser that parses the string "true" and returns Scala's boolean 'true' value:
"true" ^^ (x => true)

// a new version of a JSON parser that returns a map:
def obj: Parser[Map[String, Any]] =   // may be improved
  "{"~repsep(member, ","}~"}" ^^ { case "{"~ms~"}" => Map() ++ ms }

// as we mentioned, the '~' operator produces an instance of a case class with that name
// here's the definition of that class, which is an inner class of trait 'Parsers':
case class ~[+A, +B](x: A, y: B) {
  override def toString = "(" + x + "~" + y + ")"
}

// since the name of the class is the same as the name of the sequence combinator method
// we can match parser results with patterns that follow the same structure as the
// parsers themselves
// e.g. the pattern "{"~ms~"}" matches a result string "{" followed by a result variable
// 'ms', which is followed by a result string "}"
// the purpose of the "{"~ms~"}" pattern is to strip off the braces so that you can get
// at the list of members resulting from the 'repsep(member, ",")' parser
```

- in cases like the previous code, there is also an alternative that avoids producing unnecessary parser results that are immediately discarded by the pattern match
- the alternative makes use of `~>` and `<~` parser combinators, which express sequential composition like `~`, but `~>` keeps only the result of its right operand, whereas `<~` keeps only the result of its left operand
- using these combinators, the JSON parser can be expressed more succinctly:

```
def obj: Parser[Map[String, Any]] =
  "{" ~> repsep(member, ",") <~ "}" ^^ (Map() ++ _)

// final version of JSON parser:
class JSON1 extends JavaTokenParsers {
  def obj: Parser[Map[String, Any]] =
    "{" ~> repsep(member, ",") <~ "}" ^^ (Map() ++ _)
  def arr: Parser[List[Any]] =
    "[" ~> repsep(value, ",") <~ "]"
```

```
  def member: Parser[(String, Any)] =
    stringLiteral~":"~value ^^ { case name~":"~value => (name, value) }
  def value: Parser[Any] = (
    obj
    | arr
    | stringLiteral
    | floatingPointNumber ^^ (_.toDouble)
    | "null"  ^^ (x => null)
    | "true"  ^^ (x => true)
    | "false" ^^ (x => false)
  )
}

/* output:
[14.2] parsed:  Map(
  "address book" -> Map(
    "name" -> "John Smith",
    "address" -> Map(
      "street" -> "10 Market Street",
      "city" -> "San Francisco, CA",
      "zip" -> 94111.0),
    "phone numbers" -> List("408 338-4238", "408 111-6892")
  )
)*/
```

Symbolic versus alphanumeric names

```
/* Summary of parser combinators */
"..."              // literal
"...".r            // regular expression
P~Q                // sequential composition
P<~Q, P~>Q         // sequential composition; keep left/right only
P | Q              // alternative
opt(P)             // option
rep(P)             // repetition
repsep(P, Q)       // interleaved repetition
P ^^ f             // result conversion
```

- many of the parser combinators in the above table use symbolic names, which has its advantages (symbolic names are short and can be chosen to have the right precedences and associativities) and disadvantages (symbolic names take time to learn)

Turning off semicolon inference

- note that the body of the `value` parser in the example above is enclosed in parentheses. This is a little trick to disable semicolon inference in parser expressions
- semicolons are inserted between any two lines that can be separate statements syntactically, unless the first line ends in an infix operator or the two lines are enclosed in parentheses or square brackets

```
// we could have written 'value' like this:
def value: Parser[Any] =
  obj |
  arr |
  stringLiteral |
  // ...
```

```
// and then, semicolon insertion would have worked correctly
// but most developers prefer to see the | operator at the beginning of the second
// alternative, rather than the end of the first, which would lead to incorrect
// semicolon insertion:
   obj;        // semicolon implicitly inserted
   | arr;
```

772 - Implementing combinator parsers

Guidelines for choosing between symbolic and alphabetic names

- use symbolic names in cases where they already have a universally established meaning. For example, nobody would recommend writing `add` instead of `+` for addition
- otherwise, give preference to alphabetic names if you want your code to be understandable to casual readers
- you can still choose symbolic names for DSL libraries, if this gives clear advantage in legibility and you don't expect, anyway, that a casual reader, without a firm grounding in the domain, would be able to understand the code immediately

- the core of Scala's combinator parsing framework is contained in the trait `scala.util.parsing.combinator.Parsers`. This trait defines the `Parser` type, as well as all fundamental combinators
- a `Parser` is in essence just a function from some input type to a parse result
- as a first approximation, the type could be written like this:

```
type Parser[T] = Input => ParseResult[T]
```

Parser input

- sometimes, a parser reads a stream of tokens instead of a raw sequence of characters. Then, a separate lexical analyzer is used to convert a stream of raw characters into a stream of tokens
- the type of parser inputs is defined:

```
type Input = Reader[Elem]
```

- the class `Reader` comes from the package `scala.util.parsing.input` and is similar to a `Stream`, but also keeps track of the position of all the elements it reads
- the type `Elem` represents individual input elements. It's an abstract type member of the `Parsers`trait:

```
type Elem
```

- this means that subclasses and subtraits of `Parsers` need to instantiate class `Elem` to the type of input elements that are being parsed. E.g. `RegexParsers` and `JavaTokenParsers` fix `Elem` to be equal to `Char`. It would also be possible to set `Elem` to some other type, such as the type of tokens returned from a separate lexer

Parser results

- a parser might either succeed or fail on some given input. Consequently, class `ParseResult` has two subclasses for representing success and failure:

```
sealed abstract class ParseResult[+T]
case class Success[T](result: T, in: Input) extends ParseResult[T]
case class Failure[T](msg: String, in: Input) extends ParseResult[Nothing]
```

- `Success` carries the result returned from the parser in its `result` parameter
- the type of parser results is arbitrary. That is why `ParseResult`, `Success` and `Parser` are all parameterized with a type parameter `T`, which represents the kinds of results returned by a given parser
- `Success` also takes a second parameter, `in`, which refers to the input immediately following the part that the parser consumed. This field is needed for chaining parsers, so that one parser can operate on result of another
- note that this is purely functional approach to parsing. Input is not read as a side effect, but is kept in a stream. A parser analyzes some part of the input stream and then returns the remaining part in its result
- the other subclass of `ParseResult` is `Failure`, which takes as a parameter a message that describes why the parser failed
- `Failure` also takes a second parameter, but it's not needed for chaining (parser doesn't continue), but to position the error message at the correct place in the input stream
- parse results are defined to be covariant in the type parameter `T`, i.e. a parser returning `String` as result is compatible with a parser returning `AnyRef`

The `Parser` class

- `Parser` is in reality a class that inherits from the function type `Input => ParseResult[T]` and additionally defines methods:

```
abstract class Parser[+T] extends (Input => ParseResult[T]) {
  p =>
    // unspecified method that defines the behavior of this parser
    def apply(in: Input): ParseResult[T]
    def ~ // ...
    def | // ...
    // ...
}
```

- since parsers inherit from functions, they need to define `apply` method
- the abstract `apply` method in class `Parser` needs to be implemented in the individual parsers that inherit from `Parser`

Aliasing `this`

- the body of the `Parser` class starts with:

```
abstract class Parser[+T] extends ... { p =>   // alias
```

- a clause such as `id =>` immediately after the opening brace of a class template defines the identifier `id` as an **alias** for `this` inside the class. It's as if we had written `val id = this` in the class body, except that the compiler knows that `id` is an alias for `this`
- so `id.m` is completely equivalent to `this.m` , except that the first expression would not compile if `id` was just defined as a `val`, because then, the compiler would treat `id` as a normal identifier
- ***aliasing*** can also be a good abbreviation when you need to access `this` of an outer class:

```
class Outer { outer =>
  class Inner {
    println(Outer.this eq outer)   // true
  }
}
// in Java, you'd use 'Outer.this'
// in Scala, we use alias 'outer'
```

Single-token parsers

- trait `Parsers` defines a generic parser `elem` that can be used to parse any single token:

```
def elem(kind: String, p: Elem => Boolean) =
  new Parser[Elem] {
    def apply(in: Input) =
      if (p(in.first)) Success(in.first, in.rest)
      else Failure(kind + " expected", in)
  }

// 'kind' describes what kind of token should be parsed
// 'p'    a predicate on 'Elems', which indicates whether an element fits the class
//        of tokens to be parsed
// When applying the parser 'elem(kind, p)' to some input 'in', the first element of the
// input stream is tested with predicate. If 'p' returns 'true', the parser succeeds
// Its result is the element itself, and its remaining input is the input stream starting
// just after the element that was parsed
// If 'p' returns 'false', the parser fails with an error message that indicates what
// kind of token was expected
```

Sequential composition

- the `elem` parser only consumes a single element. To parse more interesting phrases, you can string parsers together with the sequential composition operator `~`
- `P~Q` applies first the parser `P` to a given input string, and then, if `P` succeeds, the `Q` parser is applied to the input that's left after `P` did its job
- `~` is implemented as a method in class `Parser`:

```
// method ~
abstract class Parser[+T] /* ... */ { p =>
  // ...
  def ~ [U](q: => Parser[U]) = new Parser[T~U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
        // if 'p' succeeds, 'q' is run on remainder of input 'in1'
        q(in1) match {
          // if 'q' also succeeds, the whole parser succeeds
```

```
            case Success(y, in2) => Success(new ~(x, y), in2)
            case failure => failure
          }
        case failure => failure
      }
    }
```

- inside `Parser` class, `p` is specified by `p =>` part as an alias of `this`, so `p` designates the left operand (or ***receiver***) of `~`
- its right operand is represented by parameter `q`
- the parser's result is a `~` object containing both the result of `p` (i.e. `x`) and the result of `q` (i.e. `y`)
- if either `p` or `q` fails, the result of `p~q` is the `Failure` object returned by `p` or `q`
- the result type of `~` is a parser that returns an instance of the case class `~` with elements of types `T` and `U`
- the type expression `T~U` is just a more legible shorthand for the parameterized type `~[T, U]`
- generally, Scala always interprets a binary type operation such as `A op B` as the parameterized type `op[A, B]`, which is analogous to the situation for patterns, where a binary pattern `P op Q` is also interpreted as an application, i.e. `op(P, Q)`
- the other two sequential composition operators, `<~` and `~>`, could be defined just like `~`, only with some small adjustment in how the result is computed
- a more legible technique is to define them in terms of `~`:

```
def <~ [U](q: => Parser[U]): Parser[T] =
  (p~q) ^^ { case x~y => x }
def ~> [U](q: => Parser[U]): Parser[U] =
  (p~q) ^^ { case x~y => y }
```

Alternative composition

- alternative composition `P | Q` applies either `P` or `Q` to a given input
- it first tries `P`, and if it succeeds, the whole parser succeeds with the result of `P`
- if `P` fails, then `Q` is tried **on the same input** as `P`. The result of `Q` is then the result of the whole parser

```
// definition of | method of class 'Parser':
def | (q: => Parser[T]) = new Parser[T] {
  def apply(in: Input) = p(in) match {
    case s1 @ Success(_, _) => s1
    case failure => q(in)
  }
}
```

- if `P` and `Q` both fails, the failure message is determined by `Q`

Dealing with recursion

- `q` parameter in methods `~` and `|` is by-name (its type is preceded by `=>`)
- this means that the actual parser argument will be evaluated only when `q` is needed, which should only be the case after `p` has run. This makes possible to write recursive parsers:

```
// a recursive parser that parses a number enclosed in arbitrarily many parentheses:
def parens = floatingPointNumber | "(" ~ parens ~ ")"
```

- if `|` and `~` took ***by-value parameters*** this definition would immediately cause a stack overflow without reading anything, because the value of `parens` occurs in the middle of its right-hand side

Result conversion

- the last method of class `Parser`, `^^` converts a parser's result
- the parser `P ^^ f` succeeds exactly when `P` succeeds. In that case it returns `P`'s result converted using the function `f`:

```
// implementation of the method ^^
def ^^ [U](f: T => U): Parser[U] = new Parser[U] {
  def apply(in: Input) = p(in) match {
    case Success(x, in1) => Success(f(x), in1)
    case failure => failure
  }
}
```

Parsers that don't read any input

- there are also two parsers that do not consume any input: `success` and `failure`
- parser `success(result)` always succeeds with the given `result`
- parser `failure(msg)` always fails with error message `msg`
- both are implemented as methods in trait `Parsers`, the outer trait that also contains class `Parser`:

```
def success[T](v: T) = new Parser[T] {
  def apply(in: Input) = Success(v, in)
}
def failure(msg: String) = new Parser[Nothing] {
  def apply(in: Input) = Failure(msg, in)
}
```

Option and repetition

- option and repetition combinators `opt`, `rep` and `repsep` are also defined in trait `Parsers`. They are all implemented in terms of sequential composition, alternative and result conversion:

```
def opt[T](p: => Parser[T]): Parser[Option[T]] = (
  p ^^ Some(_)
  | success(None)
)

def rep[T](p: => Parser[T]): Parser[List[T]] = (
  p~rep(p) ^^ { case x~xs => x :: xs }
  | success(List())
)

def repsep[T](p: => Parser[T], q: => Parser[Any]): Parser[List[T]] = (
  p~rep(q ~> p) ^^ { case r~rs => r :: rs}
  | success(List())
```

```
)
```

781 - String literals and regular expressions

- the parsers we used so far made use of string literals and regular expressions to parse single words. The support for these comes from `RegexParsers`, a subtrait of `Parsers`:

```
trait RegexParsers extends Parsers {    // ...
```

- the `RegexParsers` trait is more specialized than trait `Parsers` in that it only works for inputs that are sequences of characters:

```
type Elem = Char
```

- it defines two methods, `literal` and `regex`:

```
implicit def literal(s: String): Parser[String] =    // ...
implicit def regex(r: Regex): Parser[String] =    // ...
```

- both methods have an `implicit` modifier, so they are automatically applied whenever a `String`or `Regex` is given in a place where a `Parser` is expected
- this is why we can write string literals and regexes directly in a grammar, without having to wrap them with one of these methods
- e.g. the parser `"("~expr~")"` will be automatically expanded to `literal("(")~expr~literal(")")`
- the `RegexParsers` trait also takes care of handling white space between symbols, by calling a method named `handleWhiteSpace` before running a `literal` or `regex` parser
- `handleWhiteSpace` method skips the longest input sequence that conforms to the `whiteSpace`regular expression, which is defined like this:

```
protected val whiteSpace = """\s+""".r
// you can override it if you want to treat whitespace differently
// e.g. if you want white space not to be skipped at all:
object MyParsers extends RegexParsers {
   override val whiteSpace = "".r
   // ...
}
```

782 - Lexing and parsing

- syntax analysis is often split in two phases, the **lexer** phase recognizes individual words in the input and classifies them into some `token` classes. This phase is also called **lexical analysis**
- it is followed by a **syntactical analysis** phase that analyzes sequences of tokens
- the `Parsers` trait can be used for either phase, because its input elements are of the abstract type `Elem`
- for lexical analysis, `Elem` would be instantiated to `Char` (individual characters that make up a word are being parsed)

- syntactical analyzer would instantiate `Elem` to the type of token returned by the lexer
- Scala's parsing combinators provide several utility classes for lexical and syntactical analysis. These are contained in two sub-packages:

```
scala.util.parsing.combinator.lexical
scala.util.parsing.combinator.syntactical
```

- if you want to split your parser into a separate lexer and syntactical analyzer, you should consult the ***Scaladoc*** for these packages. For simple pairs, the regex expression based approach shown previously is usually sufficient

782 - Error reporting

- how does the parser issue an error message? One problem is that, when a parser rejects some input, it generally has encountered many different failures. Each alternative parse must have failed (possibly recursively, at each choice point). How to decide which of these should be emitted to the user?
- Scala's parsing library implements a simple heuristic: among all failures, the one that occurred at the latest position in the input is chosen. I.e. the parser picks the longest prefix that is still valid and issues an error message that describes why parsing the prefix could not be continued further. If there are several failure points at that latest position, the one that was visited last is chosen

```
// consider running the JSON parser on a faulty address book which starts with:
{ "name": John,

// the longest legal prefix of this phrase is ' { "name": ', so the JSON parser will flag
// the word 'John' as an error
// The JSON parser expects a value at this point, but 'John' is an identifier

// [1.13] failure: "false" expected but identifier John found
//     { "name": John,
//               ^

// the message stated that "false" was expected. This comes from the fact that "false"
// is the last alternative of the production for value in the JSON grammar

// a better error message can be produced by adding a "catch-all" failure point:
def value: Parser[Any] =
   obj | arr | stringLit | floatingPointNumber | "null"
   | "true" | "false" | failure("illegal start of value")

// this addition does not change the set of valid inputs, it only improves error msg
```

- the implementation of the ***last possible*** scheme of error reporting uses a field named `lastFailure` in `Parsers` trait to mark the failure that occurred at the latest position in the input:

```
var lastFailure: Option[Failure] = None

// the field is initialized to 'None'. It's updated in the constructor of class 'Failure'
case class Failure(msg: String, in: Input) extends ParseResult[Nothing] {
   if (lastFailure.isDefined && lastFailure.get.in.pos <= in.pos)
      lastFailure = Some(this)
```

```
}

// the field is read by the 'phrase' method (implemented in the 'Parsers' trait),
// which emits the final error message if the parser fails:
def phrase[T](p: Parser[T]) = new Parser[T] {
  lastFailure = None
  def apply(in: Input) = p(in) match {
    case s @ Success(out, in1) =>
      if (in1.atEnd) s
      else Failure("end of input expected", in1)
    case f: Failure =>
      lastFailure
  }
}
// 'lastFailure' is updated as a side-effect of the constructor of 'Failure' and by the
// 'phrase' method itself
```

- the `phrase` method runs its argument parser `p` and if `p` succeeds with a completely consumed input, the success result of `p` is returned
- if `p` succeeds but the input is not read completely, a failure with message "end of input expected" is returned
- if `p` fails, the failure or error stored in `lastFailure` is returned

784 - Backtracking versus LL(1)

- parser combinators employ **backtracking** to choose between different parsers
- in expression `P | Q`, if `P` fails, then `Q` is run on the same input as `P`. This happens even if `P`has parsed some tokens before failing. In this case, the same tokens will be parsed again by `Q`
- ***backtracking*** imposes only a few restrictions on how to formulate a grammar so that it can be parsed. You just need to avoid left-recursive productions, such as:

```
expr ::= expr "+" term | term
// this always fails, because 'expr' immediately calls itself and thus never progresses
// any further
```

- ***backtracking*** is potentially costly, because the same input can be parsed several times. Consider for instance:

```
expr ::= term "+" expr | term
// what happens if the 'expr' parser is applied to an input such as:
(1 + 2) * 3
// the first alternative would be tried, and would fail when matching the + sign
// then the second alternative would be tried on the same term, and it would succeed
// the point is that the term ended up being parsed twice

// it's often possible to modify the grammar to avoid backtracking, e.g. in the case of
// arithmetic expressions, either one of the following productions would work:
expr ::= term ["+" expr]
expr ::= term {"+" term}
```

- many languages admit so-called **LL(1) grammars**

- when a combinator parser is formed from such a grammar, it will never backtrack, i.e. the input position will never be reset to the earlier value
- the combinator parsing framework allows you to express the expectation that a grammar is ***LL(1)*** explicitly, using a new operator `~!`
- this operator is like sequential composition `~` but it will never backtrack to input elements that have already been parsed
- using this operator, the productions in the arithmetic expression parser could alternatively be written as:

```
def expr: Parser[Any] =
  term ~! rep("+" ~! term | "-" ~! term)
def term: Parser[Any] =
  factor ~! rep("*" ~! factor | "/" ~! factor)
def factor: Parser[Any] =
  "(" ~! expr ~! ")" | floatingPointNumber
```