Functional Programming Principles in Scala

Martin Odersky

September 12, 2012

Programming Paradigms

Paradigm: In science, a *paradigm* describes distinct concepts or thought patterns in some scientific discipline.

Main programming paradigms:

- imperative programming
- functional programming
- logic programming

Orthogonal to it:

- object-oriented programming

Review: Imperative programming

Imperative programming is about

- modifying mutable variables,
- using assignments
- and control structures such as if-then-else, loops, break, continue, return.

The most common informal way to understand imperative programs is as instruction sequences for a Von Neumann computer.

Imperative Programs and Computers

There's a strong correspondence between

Mutable variables	$\approx$	memory cells
Variable dereferences	$\approx$	load instructions
Variable assignments	$\approx$	store instructions
Control structures	$\approx$	jumps

Problem: Scaling up. How can we avoid conceptualizing programs word by word?

Reference: John Backus, Can Programming Be Liberated from the von. Neumann Style?, Turing Award Lecture 1978.

Scaling Up

In the end, pure imperative programming is limited by the "Von Neumann" bottleneck:

> *One tends to conceptualize data structures word-by-word.*

We need other techniques for defining high-level abstractions such as collections, polynomials, geometric shapes, strings, documents.

Ideally: Develop *theories* of collections, shapes, strings, ...

What is a Theory?

A theory consists of

- one or more data types
- operations on these types
- laws that describe the relationships between values and operations

Normally, a theory does not describe mutations!

Theories without Mutation

For instance the theory of polynomials defines the sum of two polynomials by laws such as:

```
(a*x + b) + (c*x + d) = (x+c)*x + (b+d)
```

But it does not define an operator to change a coefficient while keeping the polynomial the same!

Theories without Mutation

For instance the theory of polynomials defines the sum of two polynomials by laws such as:

```
(a*x + b) + (c*x + d) = (x+c)*x + (b+d)
```

But it does not define an operator to change a coefficient while keeping the polynomial the same!

Other example:

The theory of strings defines a concatenation operator $++$ which is associative:

```
(a ++ b) ++ c  =  a ++ (b ++ c)
```

But it does not define an operator to change a sequence element while keeping the sequence the same!

Consequences for Programming

Let's

- concentrate on defining theories for operators,
- minimize state changes,
- treat operators as functions, often composed of simpler functions.

Functional Programming

- In a *restricted* sense, functional programming (FP) means programming without mutable variables, assignments, loops, and other imperative control structures.
- In a *wider* sense, functional programming means focusing on the functions.
- In particular, functions can be values that are produced, consumed, and composed.
- All this becomes easier in a functional language.

Functional Programming Languages

- In a *restricted* sense, a functional programming language is one which does not have mutable variables, assignments, or imperative control structures.
- In a *wider* sense, a functional programming language enables the construction of elegant programs that focus on functions.
- In particular, functions in a FP language are first-class citizens. This means
  - they can be defined anywhere, including inside other functions
  - like any other value, they can be passed as parameters to functions and returned as results
  - as for other values, there exists a set operators to compose functions

Some functional programming languages

In the restricted sense:

- Pure Lisp, XSLT, XPath, XQuery, FP
- Haskell (without I/O Monad or UnsafePerformIO)

In the wider sense:

- Lisp, Scheme, Racket, Clojure
- SML, Ocaml, F#
- Haskell (full language)
- Scala
- Smalltalk, Ruby (!)

History of FP languages

1959	Lisp
1975-77	ML, FP, Scheme
1978	Smalltalk
1986	Standard ML
1990	Haskell, Erlang
1999	XSLT
2000	OCaml
2003	Scala, XQuery
2005	F#
2007	Clojure

Elements of Programming

September 11, 2012

Elements of Programming

Every non-trivial programming language provides:

- primitive expressions representing the simplest elements
- ways to *combine* expressions
- ways to *abstract* expressions, which introduce a name for an expression by which it can then be referred to.

The Read-Eval-Print Loop

Functional programming is a bit like using a calculator

An interactive shell (or REPL, for Read-Eval-Print-Loop) lets one write expressions and responds with their value.

The Scala REPL can be started by simply typing

```
> scala
```

Expressions

Here are some simple interactions with the REPL

```
scala> 87 + 145
232
```

Functional programming languages are more than simple calcululators because they let one define values and functions:

```
scala> def size = 2
size: => Int

scala> 5 * size
10
```

Evaluation

A non-primitive expression is evaluated as follows.

1. Take the leftmost operator
2. Evaluate its operands (left before right)
3. Apply the operator to the operands

A name is evaluated by replacing it with the right hand side of its definition

The evaluation process stops once it results in a value

A value is a number (for the moment)

Later on we will consider also other kinds of values

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

```
(2 * 3.14159) * radius
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

```
(2 * 3.14159) * radius
```

```
6.28318 * radius
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

```
(2 * 3.14159) * radius
```

```
6.28318 * radius
```

```
6.28318 * 10
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius

(2 * 3.14159) * radius

6.28318 * radius

6.28318 * 10

62.8318
```

Parameters

Definitions can have parameters. For instance:

```
scala> def square(x: Double) = x * x
square: (Double)Double

scala> square(2)
4.0

scala> square(5 + 4)
81.0

scala> square(square(4))
256.0

def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double
```

Parameter and Return Types

Function parameters come with their type, which is given after a colon

```
def power(x: Double, y: Int): Double = ...
```

If a return type is given, it follows the parameter list.

Primitive types are as in Java, but are written capitalized, e.g:

`Int`	32-bit integers
`Double`	64-bit floating point numbers
`Boolean`	boolean values `true` and `false`

Evaluation of Function Applications

Applications of parameterized functions are evaluated in a similar way as operators:

1. Evaluate all function arguments, from left to right
2. Replace the function application by the function's right-hand side, and, at the same time
3. Replace the formal parameters of the function by the actual arguments.

Example

```
sumOfSquares(3, 2+2)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
25
```

The substitution model

This scheme of expression evaluation is called the *substitution model*.

The idea underlying this model is that all evaluation does is *reduce an expression to a value*.

It can be applied to all expressions, as long as they have no side effects.

The substitution model is formalized in the *λ-calculus*, which gives a foundation for functional programming.

Termination

- *Does every expression reduce to a value (in a finite number of steps)?*

Termination

- *Does every expression reduce to a value (in a finite number of steps)?*
- *No. Here is a counter-example*

```
def loop: Int = loop
```

loop $\longrightarrow$ loop $\longrightarrow$ . ...

loop

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
9 + 4 * 4
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
9 + 4 * 4
25
```

Call-by-name and call-by-value

The first evaluation strategy is known as *call-by-value*, the second is is known as *call-by-name*.

Both strategies reduce to the same final values as long as

- the reduced expression consists of pure functions, and
- both evaluations terminate.

Call-by-value has the advantage that it evaluates every function argument only once.

Call-by-name has the advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.

Call-by-name vs call-by-value

Question: Say you are given the following function definition:

```
def test(x: Int, y: Int) = x * x
```

For each of the following function applications, indicate which evaluation strategy is fastest (has the fewest reduction steps)

```
  CBV      CBN       same
fastest  fastest   #steps

   O        O         O           test(2, 3)
   O        O         O           test(3+4, 8)
   O        O         O           test(7, 2*4)
   O        O         O           test(3+4, 2*4)
```

Call-by-name vs call-by-value

```scala
def test(x: Int, y: Int) = x * x
```

```scala
test(2, 3)
test(3+4, 8)
test(7, 2*4)
test(3+4, 2*4)
```

test(2,3)
↓
2 * 2
↓
4

Same

test(3+4, 8)

test(7,8)
↓
7*7
↓
49

(3+4)*(3+4)
↓
7 * (3+4)
↓
7*7
↓
49

CBV

test(7, 2*4)

test(7,8)
↓
7*7
↓
49

7 * 7
↓
49

CBN

Evaluation Strategies and Termination

August 31, 2012

Call-by-name, Call-by-value and termination

You know from the last module that the call-by-name and call-by-value evaluation strategies reduce an expression to the same value, as long as both evaluations terminate.

But what if termination is not guaranteed?

We have:

- If CBV evaluation of an expression e terminates, then CBN evaluation of e terminates, too.
- The other direction is not true

Non-termination example

Question: Find an expression that terminates under CBN but not under CBV.

Non-termination example

Let's define

```
def first(x: Int, y: Int) = x
```

and consider the expression `first(1, loop)`.

Under CBN:

```
first(1, loop)
```

Under CBV:

```
first(1, loop)
```

Scala's evaluation strategy

Scala normally uses call-by-value.

But if the type of a function parameter starts with => it uses call-by-name.

Example:

```scala
def constOne(x: Int, y: => Int) = 1
```

Let's trace the evaluations of

```scala
constOne(1+2, loop)
```

and

```scala
constOne(loop, 1+2)
```

Trace of `constOne(1 + 2, loop)`

```
constOne(1 + 2, loop)
```

Trace of `constOne(loop, 1 + 2)`

```
constOne(loop, 1 + 2)
```

Conditionals and Value Definitions

August 31, 2012

Conditional Expressions

To express choosing between two alternatives, Scala has a conditional expression `if-else`.

It looks like a `if-else` in Java, but is used for expressions, not statements.

Example:

```
def abs(x: Int) = if (x >= 0) x else -x
```

`x >= 0` is a *predicate*, of type `Boolean`.

Boolean Expressions

Boolean expressions `b` can be composed of

```
true  false      // Constants
!b               // Negation
b && b           // Conjunction
b || b           // Disjunction
```

and of the usual comparison operations:

```
e <= e, e >= e, e < e, e > e, e == e, e != e
```

Rewrite rules for Booleans

Here are reduction rules for Boolean expressions (`e` is an arbitrary expression):

```
!true       -->   false
!false      -->   true
true && e   -->   e
false && e  -->   false
true || e   -->   true
false || e  -->   e
```

Note that `&&` and `||` do not always need their right operand to be evaluated.

We say, these expressions use "short-circuit evaluation".

Exercise: Formulate rewrite rules for if-else

$$\text{if } (b)\ e_1 \text{ then } e_2$$

$$\text{if } (\text{true})\ e_1 \text{ else } e_2 \longrightarrow e_1$$

$$\text{if } (\text{false})\ e_1 \text{ else } e_2 \longrightarrow e_2$$

Value Definitions

We have seen that function parameters can be passed by value or be passed by name.

The same distinction applies to definitions.

The `def` form is "by-name", its right hand side is evaluated on each use.

There is also a `val` for, which is "by-value". Example:

def z = 3+4

z

```
val x = 2
val y = square(x)
```

The right-hand side of a `val` definition is evaluated at the point of the definition itself.

Afterwards, the name refers to the value.

For instance, `y` above refers to `4`, not `square(2)`.

Value Definitions and Termination

The difference between `val` and `def` becomes apparent when the right hand side does not terminate. Given

```
def loop: Boolean = loop
```

A definition

```
def x = loop
```

is OK, but a definition

```
val x = loop
```

will lead to an infinite loop.

Exercise

Write functions `and` and `or` such that for all argument expressions `x` and `y`:

```
and(x, y)    ==    x && y
or(x, y)     ==    x || y
```

(do not use `||` and `&&` in your implementation)

What are good operands to test that the equalities hold?

Example: Square roots with Newton's method

August 31, 2012

Task

We will define in this session a function

```
/** Calculates the square root of parameter x */
def sqrt(x: Double): Double = ...
```

The classical way to achieve this is by successive approximations using Newton's method.

Method

To compute `sqrt(x)`:

- Start with an initial *estimate* `y` (let's pick `y = 1`).
- Repeatedly improve the estimate by taking the mean of `y` and `x/y`.

Example: $x = 2$

Estimation	Quotient	Mean
1	2 / 1 = 2	1.5
1.5	2 / 1.5 = 1.333	1.4167
1.4167	2 / 1.4167 = 1.4118	1.4142
1.4142	...	...

Implementation in Scala (1)

First, define a function which computes one iteration step

```
def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)
```

Note that `sqrtIter` is *recursive*, its right-hand side calls itself.

Recursive functions need an explicit return type in Scala.

For non-recursive functions, the return type is optional

Implementation in Scala (2)

Second, define a function `improve` to improve an estimate and a test to check for terminatation:

```
def improve(guess: Double, x: Double) =
  (guess + x / guess) / 2

def isGoodEnough(guess: Double, x: Double) =
  abs(guess * guess - x) < 0.001
```

Implementation in Scala (3)

Third, define the `sqrt` function:

```
def sqrt(x: Double) = srqtIter(1.0, x)
```

Exercise

1. The `isGoodEnough` test is not very precise for small numbers and can lead to non-termination for very large numbers. Explain why.
2. Design a different version of `isGoodEnough` that does not have these problems.
3. Test your version with some very very small and large numbers, e.g.

```
0.001
0.1e-20
1.0e20
1.0e50
```

Blocks and Lexical Scope

August 31, 2012

Nested functions

It's good functional programming style to split up a task into many small functions.

But the names of functions like `sqrtIter`, `improve`, and `isGoodEnough` matter only for the *implementation* of `sqrt`, not for its *usage*.

Normally we would not like users to access these functions directly.

We can achieve this and at the same time avoid "name-space pollution" by putting the auxciliary functions inside `sqrt`.

The sqrt Function, Take 2

```scala
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0, x)
}
```

Blocks in Scala

- A block is delimited by braces `{ ... }`.

```
{ val x = f(3)
  x * x
}
```

- It contains a sequence of definitions or expressions.
- The last element of a block is an expression that defines its value.
- This return expression can be preceded by auxiliary definitions.
- Blocks are themselves expressions; a block may appear everywhere an expression can.

Blocks and Visibility

```scala
val x = 0
def f(y: Int) = y + 1
val result = {
  val x = f(3)
  x * x
}
```

- The definitions inside a block are only visible from within the block.
- The definitions inside a block *shadow* definitions of the same names outside the block.

Exercise: Scope Rules

Question: What is the value of `result` in the following program?

```
val x = 0
def f(y: Int) = y + 1
val result = {
  val x = f(3)
  x * x
} + x
```

x = 4
16

Possible answers:

O	0
●	16
O	32
O	reduction does not terminate

Lexical Scoping

Definitions of outer blocks are visible inside a block unless they are shadowed.

Therefore, we can simplify `sqrt` by eliminating redundant occurrences of the `x` parameter, which means everywhere the same thing:

The sqrt Function, Take 3

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))

  def improve(guess: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0)
}
```

Semicolons

In Scala, semicolons at the end of lines are in most cases optional

You could write

```
val x = 1;
```

but most people would omit the semicolon.

On the other hand, if there are more than one statements on a line, they need to be separated by semicolons:

```
val y = x + 1; y * y
```

Semicolons and infix operators

One issue with Scala's semicolon convention is how to write expressions that span several lines. For instance

```
someLongExpression
+ someOtherExpression
```

would be interpreted as *two* expressions:

```
someLongExpression;
+ someOtherExpression
```

Semicolons and infix operators

There are two ways to overcome this problem.

You could write the multi-line expression in parentheses, because semicolons are never inserted inside `(...)`:

```
(someLongExpression
 + someOtherExpression)
```

Or you could write the operator on the first line, because this tells the Scala compiler that the expression is not yet finished:

```
someLongExpression +
someOtherExpression
```

Summary

You have seen simple elements of functional programing in Scala.

- arithmetic and boolean expressions
- conditional expressions if-else
- functions with recursion
- nesting and lexical scope

You have learned the difference between the call-by-name and call-by-value evaluation strategies.

You have learned a way to reason about program execution: reduce expressions using the substitution model.

This model will be an important tool for the coming sessions.

Tail Recursion

Review: Evaluating a Function Application

One simple rule : One evaluates a function application $f(e_1, ..., e_n)$

- by evaluating the expressions $e_1, \ldots, e_n$ resulting in the values $v_1, ..., v_n$, then
- by replacing the application with the body of the function `f`, in which
- the actual parameters $v_1, ..., v_n$ replace the formal parameters of `f`.

Application Rewriting Rule

This can be formalized as a *rewriting of the program itself*:

$$
\begin{array}{ll}
 & \mathtt{def}\ \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{B};\ \ldots\ \mathtt{f}(\mathtt{v}_1, \ldots, \mathtt{v}_n) \\
\rightarrow & \\
 & \mathtt{def}\ \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{B};\ \ldots\ [\mathtt{v}_1/\mathtt{x}_1, \ldots, \mathtt{v}_n/\mathtt{x}_n]\,\mathtt{B}
\end{array}
$$

Here, $[\mathtt{v}_1/\mathtt{x}_1, \ldots, \mathtt{v}_n/\mathtt{x}_n]\,\mathtt{B}$ means:

The expression $\mathtt{B}$ in which all occurrences of $\mathtt{x_i}$ have been replaced by $\mathtt{v_i}$.

$[\mathtt{v}_1/\mathtt{x}_1, \ldots, \mathtt{v}_n/\mathtt{x}_n]$ is called a *substitution*.

Rewriting example:

Consider `gcd`, the function that computes the greatest common divisor of two numbers.

Here's an implementation of `gcd` using Euclid's algorithm.

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

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

$\rightarrow$ `if (21 == 0) 14 else gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
→ if (0 == 0) 7 else gcd(0, 7 % 0)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

$\rightarrow$ `gcd(14, 21)`

$\rightarrow$ `if (21 == 0) 14 else gcd(21, 14 % 21)`

$\rightarrow$ `if (false) 14 else gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14)`

$\rightarrow$ `if (14 == 0) 21 else gcd(14, 21 % 14)`

$\twoheadrightarrow$ `gcd(14, 7)`

$\twoheadrightarrow$ `gcd(7, 0)`

$\rightarrow$ `if (0 == 0) 7 else gcd(0, 7 % 0)`

$\rightarrow$ `7`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

```
factorial(4)
```

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

```
factorial(4)
```

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

Another rewriting example:

Consider `factorial`:

```scala
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0))))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

Another rewriting example:

Consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0))))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

$\twoheadrightarrow$ `120`

What are the differences between the two sequences?

Tail Recursion

Implementation Consideration: If a function calls itself as its last action, the function's stack frame can be reused. This is called *tail recursion*.

$\Rightarrow$ Tail recursive functions are iterative processes.

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called *tail-calls*.

Tail Recursion in Scala

In Scala, only directly recursive calls to the current function are optimized.

One can require that a function is tail-recursive using a `@tailrec` annotation:

```scala
@tailrec
def gcd(a: Int, b: Int): Int = ...
```

If the annotation is given, and the implementation of `gcd` were not tail recursive, an error would be issued.

Exercise: Tail recursion

Design a tail recursive version of `factorial`.

Higher-Order Functions

Higher-Order Functions

Functional languages treat functions as *first-class values*.

This means that, like any other value, a function can be passed as a parameter and returned as a result.

This provides a flexible way to compose programs.

Functions that take other functions as parameters or that return functions as results are called *higher order functions*.

Example:

Take the sum of the integers between a and b:

```scala
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0 else a + sumInts(a + 1, b)
```

Take the sum of the cubes of all the integers between a and b :

```scala
def cube(x: Int): Int = x * x * x

def sumCubes(a: Int, b: Int): Int =
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Example (ctd)

Take the sum of the factorials of all the integers between `a` and `b` :

```
def sumFactorials(a: Int, b: Int): Int =
  if (a > b) 0 else fact(a) + sumFactorials(a + 1, b)
```

These are special cases of

$$\sum_{n=a}^{b} f(n)$$

for different values of f.

Can we factor out the common pattern?

Summing with Higher-Order Functions

Let's define:

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0
  else f(a) + sum(f, a + 1, b)
```

We can then write:

```
def sumInts(a: Int, b: Int)       = sum(id, a, b)
def sumCubes(a: Int, b: Int)      = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

where

```
def id(x: Int): Int   = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if (x == 0) 1 else fact(x - 1)
```

Function Types

The type `A => B` is the type of a *function* that takes an argument of type `A` and returns a result of type `B`.

So, `Int => Int` is the type of functions that map integers to integers.

Anonymous Functions

Passing functions as parameters leads to the creation of many small functions.

- Sometimes it is tedious to have to define (and name) these functions using `def`.

Compare to strings: We do not need to define a string using `def`. Instead of

```
def str = "abc"; println(str)
```

We can directly write

```
println("abc")
```

because strings exist as *literals*. Analogously we would like function literals, which let us write a function without giving it a name.

These are called *anonymous functions*.

Anonymous Function Syntax

Example: A function that raises its argument to a cube:

```
(x: Int) => x * x * x
```

Here, `(x: Int)` is the *parameter* of the function, and `x * x * x` is it's *body*.

- The type of the parameter can be omitted if it can be inferred by the compiler from the context.

If there are several parameters, they are separated by commas:

```
(x: Int, y: Int) => x + y
```

Anonymous Functions are Syntactic Sugar

An anonymous function $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$ can always be expressed using `def` as follows:

$$\{ \text{def } f(x_1 : T_1, ..., x_n : T_n) = E; f \}$$

where f is an arbitrary, fresh name (that's not yet used in the program).

- One can therefore say that anonymous functions are *syntactic sugar*.

Summation with Anonymous Functions

Using anonymous functions, we can write sums in a shorter way:

```
def sumInts(a: Int, b: Int)  = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
```

Exercise

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.
2. Write `factorial` in terms of `product`.
3. Can you write a more general function, which generalizes both `sum` and `product`?

Currying

Principles of Functional Programming

Motivation

Look again at the summation functions:

```scala
def sumInts(a: Int, b: Int)       = sum(x => x, a, b)
def sumCubes(a: Int, b: Int)      = sum(x => x * x * x, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

Question

Note that `a` and `b` get passed unchanged from `sumInts` and `sumCubes` into `sum`.

Can we be even shorter by getting rid of these parameters?

Functions Returning Functions

Let's rewrite `sum` as follows.

```
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0
    else f(a) + sumF(a + 1, b)
  sumF
}
```

`sum` is now a function that returns another function.

The returned function `sumF` applies the given function parameter `f` and sums the results.

Stepwise Applications

We can then define:

```
def sumInts       = sum(x => x)
def sumCubes      = sum(x => x * x * x)
def sumFactorials = sum(fact)
```

These functions can in turn be applied like any other function:

```
sumCubes(1, 10) + sumFactorials(10, 20)
```

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, ... middlemen?

Of course:

```
sum (cube) (1, 10)
```

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, ... middlemen?

Of course:

```
sum (cube) (1, 10)
```

- `sum(cube)` applies `sum` to `cube` and returns the *sum of cubes* function.
- `sum(cube)` is therefore equivalent to `sumCubes`.
- This function is next applied to the arguments `(1, 10)`.

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, … middlemen?

Of course:

```
sum (cube) (1, 10)
```

- `sum(cube)` applies `sum` to `cube` and returns the *sum of cubes* function.
- `sum(cube)` is therefore equivalent to `sumCubes`.
- This function is next applied to the arguments `(1, 10)`.

Generally, function application associates to the left:

```
sum(cube)(1, 10)   ==   (sum (cube)) (1, 10)
```

Multiple Parameter Lists

The definition of functions that return functions is so useful in functional programming that there is a special syntax for it in Scala.

For example, the following definition of `sum` is equivalent to the one with the nested `sumF` function, but shorter:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

Expansion of Multiple Parameter Lists

In general, a definition of a function with multiple parameter lists

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_n) = \texttt{E}$$

where $n > 1$, is equivalent to

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_{n-1}) = \{\texttt{def g}(\texttt{args}_n) = \texttt{E}; \texttt{g}\}$$

where g is a fresh identifier. Or for short:

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_{n-1}) = (\texttt{args}_n \Rightarrow \texttt{E})$$

Expansion of Multiple Parameter Lists (2)

By repeating the process *n* times

$$\texttt{def f}(\texttt{args}_1)...(\texttt{args}_{n-1})(\texttt{args}_n) = \texttt{E}$$

is shown to be equivalent to

$$\texttt{def f} = (\texttt{args}_1 \Rightarrow (\texttt{args}_2 \Rightarrow ...(\texttt{args}_n \Rightarrow \texttt{E})...))$$

This style of definition and function application is called *currying*, named for its instigator, Haskell Brooks Curry (1900-1982), a twentieth century logician.

In fact, the idea goes back even further to Schönfinkel and Frege, but the term "currying" has stuck.

More Function Types

Question: Given,

```
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum` ?

More Function Types

Question: Given,

```scala
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum` ?

Answer:

```scala
(Int => Int) => (Int, Int) => Int
```

Note that functional types associate to the right. That is to say that

```scala
Int => Int => Int
```

is equivalent to

```scala
Int => (Int => Int)
```

Exercise

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.
2. Write `factorial` in terms of `product`.
3. Can you write a more general function, which generalizes both `sum` and `product`?

Example: Finding Fixed Points

Finding a fixed point of a function

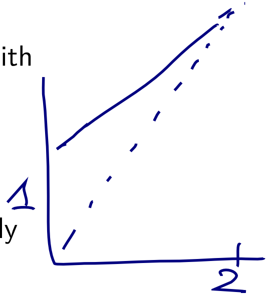
A number `x` is called a *fixed point* of a function `f` if

```
f(x) = x
```

For some functions `f` we can locate the fixed points by starting with an initial estimate and then by applying `f` in a repetitive way.

```
x, f(x), f(f(x)), f(f(f(x))), ...
```

until the value does not vary anymore (or the change is sufficiently small).



$$x \Rightarrow 1 + \frac{x}{2}$$

Programmatic Solution

This leads to the following function for finding a fixed point:

```scala
val tolerance = 0.0001
def isCloseEnough(x: Double, y: Double) =
  abs((x - y) / x) / x < tolerance
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

Return to Square Roots

Here is a *specification* of the `sqrt` function:

> `sqrt(x)` = the number `y` such that `y * y = x`.

Or, by dividing both sides of the equation with `y`:

> `sqrt(x)` = the number `y` such that `y = x / y`.

Consequently, `sqrt(x)` is a fixed point of the function `(y => x / y)`.

First Attempt

This suggests to calculate `sqrt(x)` by iteration towards a fixed point:

```
def sqrt(x: Double) =
  fixedPoint(y => x / y)(1.0)
```

Unfortunately, this does not converge.

Let's add a `println` instruction to the function `fixedPoint` so we can follow the current value of `guess`:

First Attempt (2)

```
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    println(next)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

`sqrt(2)` then produces:

```
2.0
1.0
2.0
1.0
```

Average Damping

One way to control such oscillations is to prevent the estimation from varying too much. This is done by *averaging* successive values of the original sequence:

```
def sqrt(x: Double) = fixedPoint(y => (y + x / y) / 2)(1.0)
```

This produces

```
1.5
1.4166666666666665
1.4142156862745097
1.414213562373099
1.414213562373099
```

In fact, if we expand the fixed point function `fixedPoint` we find a similar square root function to what we developed last week.

Functions as Return Values

The previous examples have shown that the expressive power of a language is greatly increased if we can pass function arguments.

The following example shows that functions that return functions can also be very useful.

Consider again iteration towards a fixed point.

We begin by observing that $\sqrt{x}$ is a fixed point of the function `y => x / y`.

Then, the iteration converges by averaging successive values.

This technique of *stabilizing by averaging* is general enough to merit being abstracted into its own function.

```
def averageDamp(f: Double => Double)(x: Double) = (x + f(x)) / 2
```

Exercise:

Write a square root function using `fixedPoint` and `averageDamp`.

Final Formulation of Square Root

```
def sqrt(x: Double) = fixedPoint(averageDamp(y => x/y))(1.0)
```

This expresses the elements of the algorithm as clearly as possible.

Summary

We saw last week that the functions are essential abstractions because they allow us to introduce general methods to perform computations as explicit and named elements in our programming language.

This week, we've seen that these abstractions can be combined with higher-order functions to create new abstractions.

As a programmer, one must look for opportunities to abstract and reuse.

The highest level of abstraction is not always the best, but it is important to know the techniques of abstraction, so as to use them when appropriate.

Scala Syntax Summary

Language Elements Seen So Far:

We have seen language elements to express types, expressions and definitions.

Below, we give their context-free syntax in Extended Backus-Naur form (EBNF), where

`|` denotes an alternative,

`[...]` an option (0 or 1),

`{...}` a repetition (0 or more).

Types

```
Type          = SimpleType | FunctionType
FunctionType  = SimpleType ‘=>’ Type
              | ‘(’ [Types] ‘)’ ‘=>’ Type
SimpleType    = Ident
Types         = Type {‘,’ Type}
```

A *type* can be:

- A *numeric type*: `Int`, `Double` (and `Byte`, `Short`, `Char`, `Long`, `Float`),
- The `Boolean` type with the values `true` and `false`,
- The `String` type,
- A *function type*, like `Int => Int`, `(Int, Int) => Int`.

Later we will see more forms of types.

Expressions

```
Expr         = InfixExpr | FunctionExpr
             | if ‘(’ Expr ‘)’ Expr else Expr
InfixExpr    = PrefixExpr | InfixExpr Operator InfixExpr
Operator     = ident
PrefixExpr   = [‘+’ | ‘-’ | ‘!’ | ‘~’ ] SimpleExpr
SimpleExpr   = ident | literal | SimpleExpr ‘.’ ident
             | Block
FunctionExpr = Bindings ‘=>‘ Expr
Bindings     = ident [‘:’ SimpleType]
             | ‘(’ [Binding {‘,’ Binding}] ‘)’
Binding      = ident [‘:’ Type]
Block        = ‘{’ {Def ‘;’} Expr ‘}’
```

Expressions (2)

An *expression* can be:

- An *identifier* such as `x`, `isGoodEnough`,
- A *literal*, like `0`, `1.0`, "abc",
- A *function application*, like `sqrt(x)`,
- An *operator application*, like `-x`, `y + x`,
- A *selection*, like `math.abs`,
- A *conditional expression*, like `if (x < 0) -x else x`,
- A *block*, like `{ val x = math.abs(y) ; x * 2 }`
- An *anonymous function*, like `x => x + 1`.

Definitions

```
Def          = FunDef   |   ValDef
FunDef       = def ident {‘(’ [Parameters] ‘)’}
               [‘:’ Type] ‘=’ Expr
ValDef       = val ident [‘:’ Type] ‘=’ Expr
Parameter    = ident ‘:’ [ ‘=>’ ] Type
Parameters   = Parameter {‘,’ Parameter}
```

A *definition* can be:

- A *function definition*, like `def square(x: Int) = x * x`
- A *value definition*, like `val y = square(2)`

A *parameter* can be:

- A *call-by-value parameter*, like `(x: Int)`,
- A *call-by-name parameter*, like `(y: => Double)`.

Functions and Data

Functions and Data

In this section, we'll learn how functions create and encapsulate data structures.

Example

Rational Numbers

We want to design a package for doing rational arithmetic.

A rational number $\frac{x}{y}$ is represented by two integers:

- its *numerator* x, and
- its *denominator* y.

Rational Addition

Suppose we want to implement the addition of two rational numbers.

```
def addRationalNumerator(n1: Int, d1: Int, n2: Int, d2: Int): Int
def addRationalDenominator(n1: Int, d1: Int, n2: Int, d2: Int): Int
```

but it would be difficult to manage all these numerators and denominators.

A better choice is to combine the numerator and denominator of a rational number in a data structure.

Classes

In Scala, we do this by defining a *class*:

```
class Rational(x: Int, y: Int) {
  def numer = x
  def denom = y
}
```

This definition introduces two entities:

- A new *type*, named `Rational`.
- A *constructor* `Rational` to create elements of this type.

Scala keeps the names of types and values in *different namespaces*.
So there's no conflict between the two defintions of `Rational`.

Objects

We call the elements of a class type *objects*.

We create an object by prefixing an application of the constructor of the class with the operator `new`.

Example

```
new Rational(1, 2)
```

Members of an Object

Objects of the class `Rational` have two *members*, `numer` and `denom`.

We select the members of an object with the infix operator '.' (like in Java).

Example

```
val x = new Rational(1, 2)    > x: Rational = Rational@2abe0e27
x.numer                       > 1
x.denom                       > 2
```

Rational Arithmetic

We can now define the arithmetic functions that implement the standard rules.

$$
\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2} \\
\frac{n_1}{d_1} \Big/ \frac{n_2}{d_2} &= \frac{n_1 d_2}{d_1 n_2} \\
\frac{n_1}{d_1} = \frac{n_2}{d_2} &\quad \text{iff} \quad n_1 d_2 = d_1 n_2
\end{aligned}
$$

Implementing Rational Arithmetic

```
def addRational(r: Rational, s: Rational): Rational =
  new Rational(
    r.numer * s.denom + s.numer * r.denom,
    r.denom * s.denom)

def makeString(r: Rational) =
  r.numer + "/" + r.denom
```

```
makeString(addRational(new Rational(1, 2), new Rational(2, 3)))   > 7/6
```

Methods

One can go further and also package functions operating on a data abstraction in the data abstraction itself.

Such functions are called *methods*.

Example

Rational numbers now would have, in addition to the functions `numer` and `denom`, the functions `add`, `sub`, `mul`, `div`, `equal`, `toString`.

Methods for Rationals

Here's a possible implementation:

```
class Rational(x: Int, y: Int) {
  def numer = x
  def denom = y
  def add(r: Rational) =
    new Rational(numer * r.denom + r.numer * denom,
                 denom * r.denom)
  def mul(r: Rational) = ...
  ...
  override def toString = numer + "/" + denom
}
```

Remark: the modifier `override` declares that `toString` redefines a method that already exists (in the class `java.lang.Object`).

Calling Methods

Here is how one might use the new `Rational` abstraction:

```
val x = new Rational(1, 3)
val y = new Rational(5, 7)
val z = new Rational(3, 2)
x.add(y).mul(z)
```

Exercise

1. In your worksheet, add a method `neg` to class Rational that is used like this:

```
x.neg          // evaluates to -x
```

2. Add a method `sub` to subtract two rational numbers.
3. With the values of `x`, `y`, `z` as given in the previous slide, what is the result of

```
x - y - z
```

?

More Fun with Rationals

Data Abstraction

The previous example has shown that rational numbers aren't always represented in their simplest form. (Why?)

One would expect the rational numbers to be *simplified*:

- reduce them to their smallest numerator and denominator by dividing both with a divisor.

We could implement this in each rational operation, but it would be easy to forget this division in an operation.

A better alternative consists of simplifying the representation in the class when the objects are constructed:

Rationals with Data Abstraction

```scala
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
  def denom = y / g
  ...
}
```

`gcd` and `g` are *private* members; we can only access them from inside the `Rational` class.

In this example, we calculate `gcd` immediately, so that its value can be re-used in the calculations of `numer` and `denom`.

Rationals with Data Abstraction (2)

It is also possible to call `gcd` in the code of `numer` and `denom`:

```
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  def numer = x / gcd(x, y)
  def denom = y / gcd(x, y)
}
```

This can be advantageous if it is expected that the functions `numer` and `denom` are called infrequently.

Rationals with Data Abstraction (3)

It is equally possible to turn `numer` and `denom` into `val`s, so that they are computed only once:

```
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  val numer = x / gcd(x, y)
  val denom = y / gcd(x, y)
}
```

This can be advantageous if the functions `numer` and `denom` are called often.

The Client's View

Clients observe exactly the same behavior in each case.

This ability to choose different implementations of the data without affecting clients is called *data abstraction*.

It is a cornerstone of software engineering.

Self Reference

On the inside of a class, the name `this` represents the object on which the current method is executed.

Example

Add the functions `less` and `max` to the class `Rational`.

```
class Rational(x: Int, y: Int) {
  ...
  def less(that: Rational) =
    numer * that.denom < that.numer * denom

  def max(that: Rational) =
    if (this.less(that)) that else this
}
```

Self Reference (2)

Note that a simple name `x`, which refers to another member of the class, is an abbreviation of `this.x`. Thus, an equivalent way to formulate `less` is as follows.

```
def less(that: Rational) =
  this.numer * that.denom < that.numer * this.denom
```

Preconditions

Let's say our `Rational` class requires that the denominator is positive.

We can enforce this by calling the `require` function.

```
class Rational(x: Int, y: Int) {
  require(y > 0, "denominator must be positive")
  ...
}
```

`require` is a predefined function.

It takes a condition and an optional message string.

If the condition passed to `require` is `false`, an `IllegalArgumentException` is thrown with the given message string.

Assertions

Besides `require`, there is also `assert`.

Assert also takes a condition and an optional message string as parameters. E.g.

```
val x = sqrt(y)
assert(x >= 0)
```

Like `require`, a failing `assert` will also throw an exception, but it's a different one: `AssertionError` for `assert`, `IllegalArgumentException` for `require`.

This reflects a difference in intent

- `require` is used to enforce a precondition on the caller of a function.
- `assert` is used as to check the code of the function itself.

Constructors

In Scala, a class implicitly introduces a constructor. This one is called the *primary constructor* of the class.

The primary constructor

- takes the parameters of the class
- and executes all statements in the class body (such as the `require` a couple of slides back).

Auxiliary Constructors

Scala also allows the declaration of *auxiliary constructors*.

These are methods named `this`

ExampleAdding an auxiliary constructor to the class `Rational`.

```
class Rational(x: Int, y: Int) {
  def this(x: Int) = this(x, 1)
  ...
}
```

new Rational(2) > *2/1*

Exercise

Modify the `Rational` class so that rational numbers are kept unsimplified internally, but the simplification is applied when numbers are converted to strings.

Do clients observe the same behavior when interacting with the rational class?

- O yes
- O no
- O yes for small sizes of denominators and nominators and small numbers of operations.

Evaluation and Operators

Classes and Substitutions

We previously defined the meaning of a function application using a computation model based on substitution. Now we extend this model to classes and objects.

Question: How is an instantiation of the class `new C(e1, ..., em)` evaluated?

Answer: The expression arguments $e_1, \ldots, e_m$ are evaluated like the arguments of a normal function. That's it.

The resulting expresion, say, `new C(v1, ..., vm)`, is already a value.

Classes and Substitutions

Now suppose that we have a class definition,

$$\texttt{class C}(\texttt{x}_1, \ldots, \texttt{x}_m)\{ \ \ldots \ \texttt{def f}(\texttt{y}_1, \ldots, \texttt{y}_n) = \texttt{b} \ \ldots \ \}$$

where

- The formal parameters of the class are $\texttt{x}_1, \ldots, \texttt{x}_m$.
- The class defines a method $\texttt{f}$ with formal parameters $\texttt{y}_1, \ldots, \texttt{y}_n$.

(The list of function parameters can be absent. For simplicity, we have omitted the parameter types.)

Question: How is the following expression evaluated?

$$\texttt{new C}(\texttt{v}_1, \ldots, \texttt{v}_m).\texttt{f}(\texttt{w}_1, \ldots, \texttt{w}_n)$$

Classes and Substitutions (2)

Answer: The expression $\texttt{new C}(\texttt{v}_1, \ldots, \texttt{v}_m).\texttt{f}(\texttt{w}_1, \ldots, \texttt{w}_n)$ is rewritten to:

$$[\texttt{w}_1/\texttt{y}_1, \ldots, \texttt{w}_n/\texttt{y}_n][\texttt{v}_1/\texttt{x}_1, \ldots, \texttt{v}_m/\texttt{x}_m][\texttt{new C}(\texttt{v}_1, \ldots, \texttt{v}_m)/\texttt{this}]\,\texttt{b}$$

There are three substitutions at work here:

- the substitution of the formal parameters $\texttt{y}_1, \ldots, \texttt{y}_n$ of the function $\texttt{f}$ by the arguments $\texttt{w}_1, \ldots, \texttt{w}_n$,
- the substitution of the formal parameters $\texttt{x}_1, \ldots, \texttt{x}_m$ of the class $\texttt{C}$ by the class arguments $\texttt{v}_1, \ldots, \texttt{v}_m$,
- the substitution of the self reference *this* by the value of the object $\texttt{new C}(\texttt{v}_1, \ldots, \texttt{v}_n)$.

class C (x₁,...,xm) {

def f (y₁,...,yn) = b...this...

}

Object Rewriting Examples

```
new Rational(1, 2).numer
```

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\rightarrow [1/\mathtt{x}, 2/\mathtt{y}]\ []\ [\mathtt{new\ Rational}(1, 2)/\mathtt{this}]\ \mathtt{x}$

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\rightarrow [1/\mathtt{x}, 2/\mathtt{y}]\ []\ [\mathtt{new\ Rational}(1, 2)/\mathtt{this}]\ \mathtt{x}$

$= 1$

Object Rewriting Examples

`new Rational(1, 2).numer`

$\rightarrow [1/x, 2/y]\ []\ [\texttt{new Rational}(1, 2)/\texttt{this}]\ \texttt{x}$

$= 1$

`new Rational(1, 2).less(new Rational(2, 3))`

Object Rewriting Examples

```
new Rational(1, 2).numer
```

$\rightarrow$ $[1/\mathtt{x}, 2/\mathtt{y}]$ $[]$ $[\mathtt{new\ Rational}(1, 2)/\mathtt{this}]$ `x`

$=$ `1`

```
new Rational(1, 2).less(new Rational(2, 3))
```

$\rightarrow$ $[1/\mathtt{x}, 2/\mathtt{y}]$ $[\mathtt{newRational}(2, 3)/\mathtt{that}]$ $[\mathtt{new\ Rational}(1, 2)/\mathtt{this}]$

```
this.numer * that.denom < that.numer * this.denom
```

Object Rewriting Examples

`new Rational(1, 2).numer`

$\rightarrow [1/x, 2/y]\ []\ [\texttt{new Rational}(1, 2)/\texttt{this}]$ `x`

$=$ `1`

`new Rational(1, 2).less(new Rational(2, 3))`

$\rightarrow [1/x, 2/y]\ [\texttt{newRational}(2, 3)/\texttt{that}]\ [\texttt{new Rational}(1, 2)/\texttt{this}]$

`this.numer * that.denom < that.numer * this.denom`

$=$ `new Rational(1, 2).numer * new Rational(2, 3).denom <`

`new Rational(2, 3).numer * new Rational(1, 2).denom`

Object Rewriting Examples

`new Rational(1, 2).numer`

$\rightarrow$ $[1/x, 2/y]$ $[]$ $[\texttt{new Rational}(1, 2)/\texttt{this}]$ `x`

$=$ `1`

`new Rational(1, 2).less(new Rational(2, 3))`

$\rightarrow$ $[1/x, 2/y]$ $[\texttt{newRational}(2, 3)/\texttt{that}]$ $[\texttt{new Rational}(1, 2)/\texttt{this}]$

`this.numer * that.denom < that.numer * this.denom`

$=$ `new Rational(1, 2).numer * new Rational(2, 3).denom <`

`new Rational(2, 3).numer * new Rational(1, 2).denom`

$\twoheadrightarrow$ `1 * 3 < 2 * 2`

$\twoheadrightarrow$ `true`

Operators

In principle, the rational numbers defined by `Rational` are as natural as integers.

But for the user of these abstractions, there is a noticeable difference:

- We write `x + y`, if `x` and `y` are integers, but
- We write `r.add(s)` if `r` and `s` are rational numbers.

In Scala, we can eliminate this difference. We procede in two steps.

Step 1: Infix Notation

Any method with a parameter can be used like an infix operator.

It is therefore possible to write

```
r add s                                  r.add(s)
r less s        /* in place of */        r.less(s)
r max s                                  r.max(s)
```

Step 2: Relaxed Identifiers

Operators can be used as identifiers.

Thus, an identifier can be:

- *Alphanumeric*: starting with a letter, followed by a sequence of letters or numbers
- *Symbolic*: starting with an operator symbol, followed by other operator symbols.
- The underscore character '_' counts as a letter.
- Alphanumeric identifiers can also end in an underscore, followed by some operator symbols.

Examples of identifiers:

```
x1      *      +?%&      vector_++      counter_=
```

Operators for Rationals

A more natural definition of class `Rational`:

```
class Rational(x: Int, y: Int) {
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
  def denom = y / g
  def + (r: Rational) =
    new Rational(
      numer * r.denom + r.numer * denom,
      denom * r.denom)
  def - (r: Rational) = ...
  def * (r: Rational) = ...
  ...
}
```

Operators for Rationals

... and rational numbers can be used like `Int` or `Double`:

```
val x = new Rational(1, 2)
val y = new Rational(1, 3)
(x * x) + (y * y)
```

Precedence Rules

The *precedence* of an operator is determined by its first character.

The following table lists the characters in increasing order of priority precedence:

```
(all letters)
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)
```

Exercise

Provide a fully parenthized version of

```
((a + b) ^? (c ?^ d)) less ((a ==> b) | c)
```

Every binary operation needs to be put into parentheses, but the structure of the expression should not change.

Class Hierarchies

Abstract Classes

Consider the task of writing a class for sets of integers with the following operations.

```scala
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

`IntSet` is an *abstract class*.

Abstract classes can contain members which are missing an implementation (in our case, `incl` and `contains`).

Consequently, no instances of an abstract class can be created with the operator `new`.

Class Extensions

Let's consider implementing sets as binary trees.

There are two types of possible trees: a tree for the empty set, and a tree consisting of an integer and two sub-trees.

Here are their implementations:

```
class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}
```

Class Extensions (2)

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {

  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true

  def incl(x: Int): IntSet =
    if (x < elem) new NonEmpty(elem, left incl x, right)
    else if (x > elem) new NonEmpty(elem, left, right incl x)
    else this
}
```

Terminology

`Empty` and `NonEmpty` both *extend* the class `IntSet`.

This implies that the types `Empty` and `NonEmpty` *conform* to the type `IntSet`

- an object of type `Empty` or `NonEmpty` can be used wherever an object of type `IntSet` is required.

Base Classes and Subclasses

`IntSet` is called the *superclass* of `Empty` and `NonEmpty`.

`Empty` and `NonEmpty` are *subclasses* of `IntSet`.

In Scala, any user-defined class extends another class.

If no superclass is given, the standard class `Object` in the Java package `java.lang` is assumed.

The direct or indirect superclasses of a class `C` are called *base classes* of `C`.

So, the base classes of `NonEmpty` are `IntSet` and `Object`.

Implementation and Overriding

The definitions of `contains` and `incl` in the classes `Empty` and `NonEmpty` *implement* the abstract functions in the base trait `IntSet`.

It is also possible to *redefine* an existing, non-abstract definition in a subclass by using `override`.

Example

```
abstract class Base {
  def foo = 1
  def bar: Int
}
```

```
class Sub extends Base {
  override def foo = 2
  def bar = 3
}
```

Object Definitions

In the `IntSet` example, one could argue that there is really only a single empty `IntSet`.

So it seems overkill to have the user create many instances of it.

We can express this case better with an *object definition*:

```scala
object Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, Empty, Empty)
}
```

This defines a *singleton object* named `Empty`.

No other `Empty` instances can be (or need to be) created.

Singleton objects are values, so `Empty` evaluates to itself.

Programs

So far we have executed all Scala code from the REPL or the worksheet.

But it is also possible to create standalone applications in Scala.

Each such application contains an object with a `main` method.

For instance, here is the "Hello World!" program in Scala.

```
object Hello {
  def main(args: Array[String]) = println("hello world!")
}
```

Once this program is compiled, you can start it from the command line with

```
> scala Hello
```

Exercise

Write a method `union` for forming the union of two sets. You should implement the following abstract class.

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
  def union(other: IntSet): IntSet
}
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty contains 1
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

`Empty contains 1`

$\rightarrow$ $[1/\mathtt{x}]$ $[\mathtt{Empty}/\mathtt{this}]$ `false`

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

`Empty contains 1`

$\rightarrow$ [1/x] [Empty/this] `false`

$=$ `false`

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ [7/elem] [7/x] [new NonEmpty(7, Empty, Empty)/this]

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

→ [7/elem] [7/x] [new NonEmpty(7, Empty, Empty)/this]

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

```
= if (7 < 7) new NonEmpty(7, Empty, Empty).left contains 7
    else if (7 > 7) new NonEmpty(7, Empty, Empty).right
         contains 7 else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(new NonEmpty(7, Empty, Empty)) contains 7
```

$\rightarrow$ [7/elem] [7/x] [new NonEmpty(7, Empty, Empty)/this]

```
    if (x < elem) this.left contains x
      else if (x > elem) this.right contains x else true
```

```
= if (7 < 7) new NonEmpty(7, Empty, Empty).left contains 7
    else if (7 > 7) new NonEmpty(7, Empty, Empty).right
         contains 7 else true
```

$\rightarrow$ `true`

Something to Ponder

Dynamic dispatch of methods is analogous to calls to higher-order functions.

Question:

Can we implement one concept in terms of the other?

- Objects in terms of higher-order functions?
- Higher-order functions in terms of objects?

How Classes are Organized

Packages

Classes and objects are organized in packages.

To place a class or object inside a package, use a package clause at the top of your source file.

```
package progfun.examples

object Hello { ... }
```

This would place `Hello` in the package `progfun.examples`.

You can then refer to `Hello` by its *fully qualified name* `progfun.examples.Hello`. For instance, to run the `Hello` program:

```
> scala progfun.examples.Hello
```

Imports

Say we have a class `Rational` in package `week3`.

You can use the class using its fully qualified name:

```
val r = new week3.Rational(1, 2)
```

Alternatively, you can use an import:

```
import week3.Rational
val r = new Rational(1, 2)
```

Forms of Imports

Imports come in several forms:

```
import week3.Rational           // imports just Rational
import week3.{Rational, Hello}  // imports both Rational and Hello
import week3._                  // imports everything in package week3
```

The first two forms are called *named imports*.

The last form is called a *wildcard import*.

You can import from either a package or an object.

Automatic Imports

Some entities are automatically imported in any Scala program.

These are:

- All members of package `scala`
- All members of package `java.lang`
- All members of the singleton object `scala.Predef`.

Here are the fully qualified names of some types and functions which you have seen so far:

`Int`	`scala.Int`
`Boolean`	`scala.Boolean`
`Object`	`java.lang.Object`
`require`	`scala.Predef.require`
`assert`	`scala.Predef.assert`

Scaladoc

You can explore the standard Scala library using the scaladoc web pages.

You can start at

www.scala-lang.org/api/current

Traits

In Java, as well as in Scala, a class can only have one superclass.

But what if a class has several natural supertypes to which it conforms or from which it wants to inherit code?

Here, you could use `traits`.

A trait is declared like an abstract class, just with `trait` instead of `abstract class`.

```
trait Planar {
  def height: Int
  def width: Int
  def surface = height * width
}
```

single inheritance

Traits (2)

Classes, objects and traits can inherit from at most one class but arbitrary many traits.

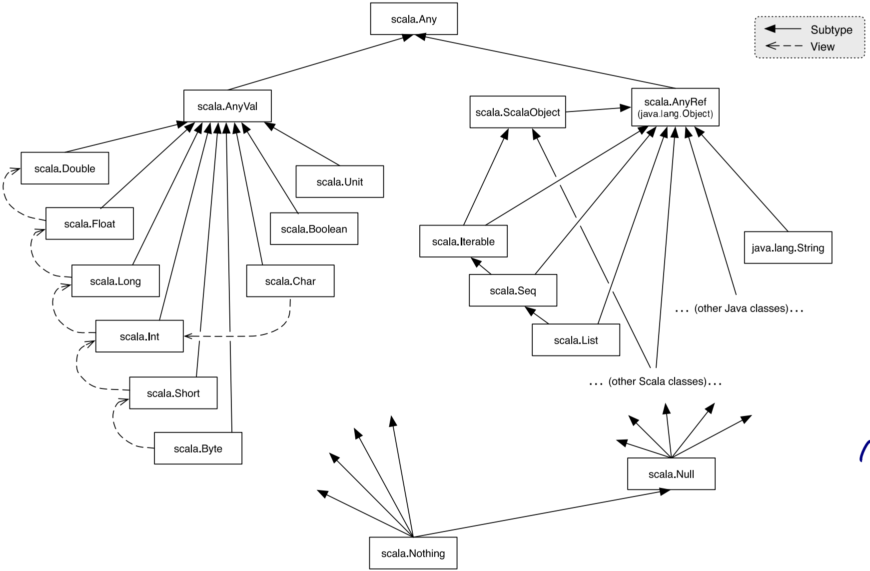
Example:

```
class Square extends Shape with Planar with Movable ...
```

Traits resemble interfaces in Java, but are more powerful because they can contains fields and concrete methods.

On the other hand, traits cannot have (value) parameters, only classes can.

Scala's Class Hierarchy



null

Top Types

At the top of the type hierarchy we find:

`Any`	the base type of all types Methods: ‘==‘, ‘!=‘, ‘equals‘, ‘hashCode, ‘toString‘
`AnyRef`	The base type of all reference types; Alias of ‘java.lang.Object‘
`AnyVal`	The base type of all primitive types.

The Nothing Type

`Nothing` is at the bottom of Scala's type hierarchy. It is a subtype of every other type.

There is no value of type `Nothing`.

Why is that useful?

- To signal abnormal termination
- As an element type of empty collections (see next session)

Set [Nothing]

Exceptions

Scala's exception handling is similar to Java's.

The expression

```
throw Exc
```

aborts evaluation with the exception `Exc`.

The type of this expression is `Nothing`.

The Null Type

Every reference class type also has `null` as a value.

The type of `null` is `Null`.

`Null` is a subtype of every class that inherits from `Object`; it is incompatible with subtypes of `AnyVal`.

```
val x = null          // x: Null
val y: String = null // y: String
val z: Int = null    // error: type mismatch
```

Exercise

What is the type of

```
if (true) 1 else false
```

```
O         Int
O         Boolean
O         AnyVal
O         Object
O         Any
```

Polymorphism

Cons-Lists

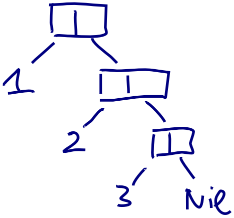
A fundamental data structure in many functional languages is the immutable linked list.

It is constructed from two building blocks:

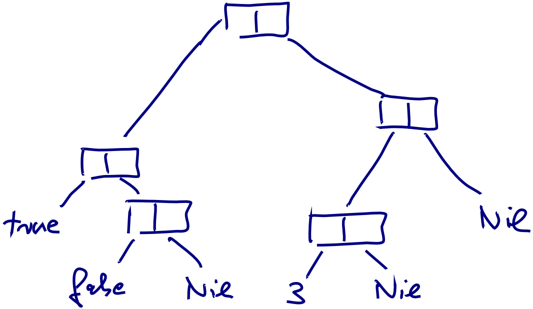
`Nil`	the empty list
`Cons`	a cell containing an element and the remainder of the list.

Examples for Cons-Lists

```
List(1, 2, 3)
```




```
List(List(true, false), List(3))
```



Cons-Lists in Scala

Here's an outline of a class hierarchy that represents lists of integers in this fashion:

```
package week4

trait IntList ...
class Cons(val head: Int, val tail: IntList) extends IntList ...
class Nil extends IntList ...
```

A list is either

- an empty list `new Nil`, or
- a list `new Cons(x, xs)` consisting of a `head` element `x` and a `tail` list `xs`.

Value Parameters

Note the abbreviation `(val head: Int, val tail: IntList)` in the definition of `Cons`.

This defines at the same time parameters and fields of a class.

It is equivalent to:

```
class Cons(_head: Int, _tail: IntList) extends IntList {
  val head = _head
  val tail = _tail
}
```

where `_head` and `_tail` are otherwise unused names.

Type Parameters

It seems too narrow to define only lists with `Int` elements.

We'd need another class hierarchy for `Double` lists, and so on, one for each possible element type.

We can generalize the definition using a type parameter:

```
package week4

trait List[T]
class Cons[T](val head: T, val tail: List[T]) extends List[T]
class Nil[T] extends List[T]
```

Type parameters are written in square brackets, e.g. `[T]`.

Complete Definition of List

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  def isEmpty = false
}

class Nil[T] extends List[T] {
  def isEmpty = true
  def head = throw new NoSuchElementException(”Nil.head”)
  def tail = throw new NoSuchElementException(”Nil.tail”)
}
```

Generic Functions

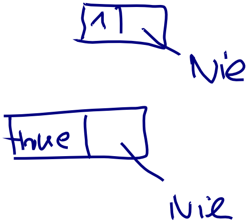
Like classes, functions can have type parameters.

For instance, here is a function that creates a list consisting of a single element.

```scala
def singleton[T](elem: T) = new Cons[T](elem, new Nil[T])
```

We can then write:

```scala
singleton[Int](1)
singleton[Boolean](true)
```



Type Inference

In fact, the Scala compiler can usually deduce the correct type parameters from the value arguments of a function call.

So, in most cases, type parameters can be left out. You could also write:

[Int

```
singleton(1)
singleton(true)
```

Types and Evaluation

Type parameters do not affect evaluation in Scala.

We can assume that all type parameters and type arguments are removed before evaluating the program.

This is also called *type erasure*.

Languages that use type erasure include Java, Scala, Haskell, ML, OCaml.

Some other languages keep the type parameters around at run time, these include C++, C#, F#.

Polymorphism

Polymorphism means that a function type comes "in many forms".

In programming it means that

- the function can be applied to arguments of many types, or
- the type can have instances of many types.

We have seen two principal forms of polymorphism:

- subtyping: instances of a subclass can be passed to a base class
- generics: instances of a function or class are created by type parameterization.

List

Nil Cons

Exercise

Write a function `nth` that takes an integer `n` and a list and selects the `n`'th element of the list.

Elements are numbered from 0.

If index is outside the range from `0` up the the length of the list minus one, a `IndexOutOfBoundsException` should be thrown.

Objects Everywhere

Pure Object Orientation

A pure object-oriented language is one in which every value is an object.

If the language is based on classes, this means that the type of each value is a class.

Is Scala a pure object-oriented language?

At first glance, there seem to be some exceptions: primitive types, functions.

But, let's look closer:

Standard Classes

Conceptually, types such as `Int` or `Boolean` do not receive special treatment in Scala. They are like the other classes, defined in the package `scala`.

For reasons of efficiency, the Scala compiler represents the values of type `scala.Int` by 32-bit integers, and the values of type `scala.Boolean` by Java's Booleans, etc.

Pure Booleans

The `Boolean` type maps to the JVM's primitive type `boolean`.

But one *could* define it as a class from first principles:

```scala
package idealized.scala
abstract class Boolean {
  def ifThenElse[T](t: => T, e: => T): T

  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def unary_!: Boolean             = ifThenElse(false, true)

  def == (x: Boolean): Boolean     = ifThenElse(x, x.unary_!)
  def != (x: Boolean): Boolean     = ifThenElse(x.unary_!, x)
  ...
}
```

if (cond) te else ee

cond.ifThenElse(te, ee)

Boolean Constants

Here are constants `true` and `false` that go with `Boolean` in the `idealized.scala`:

```
package idealized.scala

object true extends Boolean {
  def ifThenElse[T](t: => T, e: => T) = t
}

object false extends Boolean {
  def ifThenElse[T](t: => T, e: => T) = e
}
```

if (true) te else ee

= te

Exercise

Provide an implementation of the comparison operator < in class `idealized.scala.Boolean`.

Assume for this that `false` $<$ `true`.

Exercise

Provide an implementation of the comparison operator < in class `idealized.scala.Boolean`.

Assume for this that `false` $<$ `true`.

```
class Boolean {
  def < (x: Boolean) =
      ifThenElse (false, x)
}
```

The class Int

Here is a partial specification of the class `scala.Int`.

```
class Int {
  def + (that: Double): Double
  def + (that: Float): Float
  def + (that: Long): Long
  def + (that: Int): Int         // same for -, *, /, %

  def << (cnt: Int): Int         // same for >>, >>>  */

  def & (that: Long): Long
  def & (that: Int): Int         // same for |, ^ */
```

1 + 2.0

The class Int (2)

```
    def == (that: Double): Boolean
    def == (that: Float): Boolean
    def == (that: Long): Boolean   // same for !=, <, >, <=, >=
    ...
  }
```

Can it be represented as a class from first principles (i.e. not using primitive `int`s?

Exercise

Provide an implementation of the abstract class `Nat` that represents non-negative integers.

```
abstract class Nat {
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def + (that: Nat): Nat
  def - (that: Nat): Nat
}
```

Exercise (2)

Do not use standard numerical classes in this implementation.

Rather, implement a sub-object and a sub-class:

```
object Zero extends Nat
class Succ(n: Nat) extends Nat
```

One for the number zero, the other for strictly positive numbers.

(this one is a bit more involved than previous quizzes).

Functions as Objects

Functions as Objects

We have seen that Scala's numeric types and the `Boolean` type can be implemented like normal classes.

But what about functions?

Functions as Objects

We have seen that Scala's numeric types and the `Boolean` type can be implemented like normal classes.

But what about functions?

In fact function values *are* treated as objects in Scala.

The function type `A => B` is just an abbreviation for the class `scala.Function1[A, B]`, which is defined as follows.

```
package scala
trait Function1[A, B] {
  def apply(x: A): B
}
```

So functions are objects with `apply` methods.

There are also traits `Function2`, `Function3`, ... for functions which take more parameters (currently up to 22).

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

```
{ class AnonFun extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFun
}
```

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

```
{ class AnonFun extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFun
}
```

or, shorter, using *anonymous class syntax*:

```
new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
```

Expansion of Function Calls

A function call, such as `f(a, b)`, where `f` is a value of some class type, is expanded to

```
f.apply(a, b)
```

So the OO-translation of

```
val f = (x: Int) => x * x
f(7)
```

would be

```
val f = new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
f.apply(7)
```

Functions and Methods

Note that a method such as

```
def f(x: Int): Boolean = ...
```

is not itself a function value.

But if `f` is used in a place where a Function type is expected, it is converted automatically to the function value

```
(x: Int) => f(x)
```

or, expanded:

```
new Function1[Int, Boolean] {
  def apply(x: Int) = f(x)
}
```

Exercise

In package `week4`, define an

```
object List {
  ...
}
```

with 3 functions in it so that users can create lists of lengths 0-2 using syntax

```
List()         // the empty list
List(1)        // the list with single element 1
List(2, 3)     // the list with elements 2 and 3.
```

Subtyping and Generics

Polymorphism

Two principal forms of polymorphism:

- subtyping
- generics

In this session we will look at their interactions.

Two main areas:

- bounds
- variance

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

```
def assertAllPos(s: IntSet): IntSet
```

In most situations this is fine, but can one be more precise?

assertAllPos(Empty) = Empty

-"- (NonEmpty(...)) = $\begin{cases} \text{NonEmpty(...)} \\ \text{throw Error} \end{cases}$

Type Bounds

One might want to express that `assertAllPos` takes `Empty` sets to `Empty` sets and `NonEmpty` sets to `NonEmpty` sets.

A way to express this is:

```
def assertAllPos[S <: IntSet](r: S): S = ...
```

Here, "`<: IntSet`" is an *upper bound* of the type parameter `S`:

It means that `S` can be instantiated only to types that conform to `IntSet`.

Generally, the notation

- `S <: T` means: `S` *is a subtype of* `T`, and
- `S >: T` means: `S` *is a supertype of* `T`, or `T` *is a subtype of* `S`.

$$\begin{array}{c} T \\ \uparrow \\ S \end{array}$$

Lower Bounds

You can also use a lower bound for a type variable.

Example

```
[S >: NonEmpty]
```

introduces a type parameter `S` that can range only over *supertypes* of `NonEmpty`.

So `S` could be one of `NonEmpty`, `IntSet`, `AnyRef`, or `Any`.

We will see later on in this session where lower bounds are useful.

Mixed Bounds

Finally, it is also possible to mix a lower bound with an upper bound.

For instance,

```
[S >: NonEmpty <: IntSet]
```

would restrict `S` any type on the interval between `NonEmpty` and `IntSet`.

IntSet

NonEmpty

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

NonEmpty

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

Intuitively, this makes sense: A list of non-empty sets is a special case of a list of arbitrary sets.

We call types for which this relationship holds *covariant* because their subtyping relationship varies with the type parameter.

Does covariance make sense for all types, not just for `List`?

Arrays

For perspective, let's look at arrays in Java (and C#).

Reminder:

- An array of `T` elements is written `T[]` in Java.
- In Scala we use parameterized type syntax `Array[T]` to refer to the same type.

Arrays in Java are covariant, so one would have:

```
NonEmpty[] <: IntSet[]
```

Array Typing Problem

But covariant array typing causes problems.

To see why, consider the Java code below.

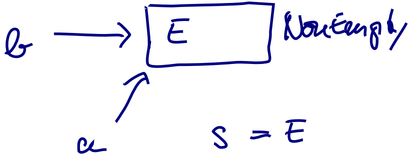
```
NonEmpty[] a = new NonEmpty[]{new NonEmpty(1, Empty, Empty)}
IntSet[] b = a
b[0] = Empty
NonEmpty s = a[0]
```

ArrayStoreException

It looks like we assigned in the last line an `Empty` set to a variable of type `NonEmpty`!

What went wrong?

sort(Object[] a)



The Liskov Substitution Principle

The following principle, stated by Barbara Liskov, tells us when a type can be a subtype of another.

> *If* `A <: B`*, then everything one can to do with a value of type* `B` *one should also be able to do with a value of type* `A`*.*

[The actual definition Liskov used is a bit more formal. It says:

> *Let* `q(x)` *be a property provable about objects* `x` *of type* `B`*. Then* `q(y)` *should be provable for objects* `y` *of type* `A` *where* `A <: B`*.*

]

$$\begin{array}{c} B \\ \uparrow \\ A \end{array}$$

Exercise

The problematic array example would be written as follows in Scala:

```scala
val a: Array[NonEmpty] = Array(new NonEmpty(1, Empty, Empty))
val b: Array[IntSet] = a
b(0) = Empty
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

O	A type error in line 1
O	A type error in line 2
O	A type error in line 3
O	A type error in line 4
O	A program that compiles and throws an exception at run-time
O	A program that compiles and runs without exception

Exercise

The problematic array example would be written as follows in Scala:

```scala
val a: Array[NonEmpty] = Array(new NonEmpty(1, Empty, Empty))
val b: Array[IntSet] = a
b(0) = Empty
val s: NonEmpty = a(0)
```

Array [IntSet]

$\not\uparrow$

Array [NonEmpty]

When you try out this example, what do you observe?

o A type error in line 1
● A type error in line 2
o A type error in line 3
o A type error in line 4
o A program that compiles and throws an exception at run-time
o A program that compiles and runs without exception

Variance

September 29, 2012

Variance

You have seen the the previous session that some types should be covariant whereas others should not.

Roughly speaking, a type that accepts mutations of its elements should not be covariant.

But immutable types can be covariant, if some conditions on methods are met.

List ✓

Array ✗

Definition of Variance

Say `C[T]` is a parameterized type and `A`, `B` are types such that `A <: B`.

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`C[A] <: C[B]`	`C` is *covariant*
`C[A] >: C[B]`	`C` is *contravariant*
neither `C[A]` nor `C[B]` is a subtype of the other	`C` is *nonvariant*

$$\begin{array}{ccc} B & & C[B] \\ \uparrow & & \\ A & & C[A] \end{array}$$

Definition of Variance

Say `C[T]` is a parameterized type and `A`, `B` are types such that `A <: B`.

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`C[A] <: C[B]`	`C` is *covariant*
`C[A] >: C[B]`	`C` is *contravariant*
neither `C[A]` nor `C[B]` is a subtype of the other	`C` is *nonvariant*

Scala lets you declare the variance of a type by annotating the type parameter:

`class C[+A] { ... }`	`C` is *covariant*
`class C[-A] { ... }`	`C` is *contravariant*
`class C[A] { ... }`	`C` is *nonvariant*

Exercise

Say you have two function types:

```
type A = IntSet => NonEmpty
type B = NonEmpty => IntSet
```

According to the Liskov Substitution Principle, which of the following should be true?

```
O          A <: B
O          B <: A
O          A and B are unrelated.
```

Exercise

Say you have two function types:

```
type A = IntSet => NonEmpty
type B = NonEmpty => IntSet
```

According to the Liskov Substitution Principle, which of the following should be true?

- ● `A <: B`
- O `B <: A`
- O `A and B are unrelated.`

Typing Rules for Functions

Generally, we have the following rule for subtyping between function types:

If `A2 <: A1` and `B1 <: B2`, then

```
A1 => B1   <:   A2 => B2
```

A2 => B2

A1 => B1

Function Trait Declaration

So functions are *contravariant* in their argument type(s) and *covariant* in their result type.

This leads to the following revised definition of the `Function1` trait:

```
package scala
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Variance Checks

We have seen in the array example that the combination of covariance with certain operations is unsound.

In this case the problematic operation was the update operation on an array.

If we turn `Array` into a class, and `update` into a method, it would look like this:

```
class Array[+T] {
  def update(x: T) ...
}
```

The problematic combination is

- the covariant type parameter `T`
- which appears in parameter position of the method `update`.

Variance Checks (2)

The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations.

Roughly,

- *covariant* type parameters can only appear in method results.
- *contravariant* type parameters can only appear in method parameters.
- *invariant* type parameters can appear anywhere.

The precise rules are a bit more involved, fortunately the Scala compiler performs them for us.

Variance-Checking the Function Trait

Let's have a look again at Function1:

```
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Here,

- T is contravariant and appears only as a method parameter type
- U is covariant and appears only as a method result type

So the method is checks out OK.

Variance and Lists

Let's get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Variance and Lists

Let's get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Here are the essential modifications:

```
trait List[+T] { ... }
object Empty extends List[Nothing] { ... }
```

Making Classes Covariant

Sometimes, we have to put in a bit of work to make a class covariant.

Consider adding a `prepend` method to `List` which prepends a given element, yielding a new list.

A first implementation of `prepend` could look like this:

```
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

But that does not work!

Exercise

Why does the following code not type-check?

```
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

Possible answers:

O `prepend` turns `List` into a mutable class.

O `prepend` fails variance checking.

O `prepend`'s right-hand side contains a type error.

Exercise

Why does the following code not type-check?

```
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

Possible answers:

0 `prepend` turns `List` into a mutable class.

0 `prepend` fails variance checking.

0 `prepend`'s right-hand side contains a type error.

Prepend Violates LSP

Indeed, the compiler is right to throw out `List` with `prepend`, because it violates the Liskov Substitution Principle:

Here's something one can do with a list `xs` of type `List[IntSet]`:

```
xs.prepend(Empty)
```

But the same operation on a list `ys` of type `List[NonEmpty]` would lead to a type error:

```
ys.prepend(Empty)
           ^ type mismatch
           required: NonEmpty
           found: Empty
```

So, `List[NonEmpty]` cannot be a subtype of `List[IntSet]`.

Lower Bounds

But `prepend` is a natural method to have on immutable lists!

Question: How can we make it variance-correct?

Lower Bounds

But `prepend` is a natural method to have on immutable lists!

Question: How can we make it variance-correct?

We can use a *lower bound*:

```
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

This passes variance checks, because:

- covariant type parameters may appear in lower bounds of method type parameters
- contravariant type parameters may appear in upper bounds of method

Exercise

Implement `prepend` as shown in trait `List`.

```
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

What is the result type of this function:

```
def f(xs: List[NonEmpty], x: Empty) = xs prepend x    ?
```

Possible answers:

O does not type check
O `List[NonEmpty]`
O `List[Empty]`
O `List[IntSet]`
O `List[Any]`

Exercise

Implement `prepend` as shown in trait `List`.

```
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

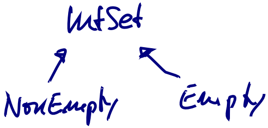
"NonEmpty "Empty

U = IntSet

What is the result type of this function:

```
def f(xs: List[NonEmpty], x: Empty) = xs prepend x    ?
```

: List[IntSet]



Possible answers:

O	does not type check
O	`List[NonEmpty]`
O	`List[Empty]`
O	`List[IntSet]`
O	`List[Any]`

Decomposition

Decomposition

Suppose you want to write a small interpreter for arithmetic expressions.

To keep it simple, let's restrict ourselves to numbers and additions.

Expressions can be represented as a class hierarchy, with a base trait `Expr` and two subclasses, `Number` and `Sum`.

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us to the following implementation.

Expressions

isVar
isProd

name

```
trait Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
```

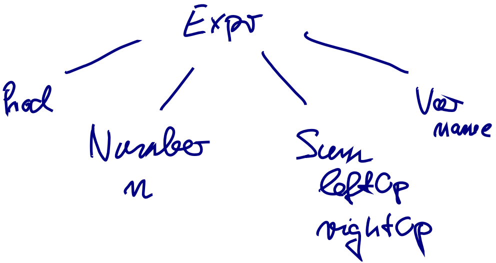
Classification

Accessor

8

```
class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = throw new Error("Number.leftOp")
  def rightOp: Expr = throw new Error("Number.rightOp")
}
```

8



Expressions (2)

```
class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = throw new Error(”Sum.numValue”)
  def leftOp: Expr = e1
  def rightOp: Expr = e2
      8
}
```

new Sum(e_1, e_2) $\approx e_1 + e_2$

Evaluation of Expressions

You can now write an evaluation function as follows.

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else throw new Error(”Unknown expression ” + e)
}
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

eval (Sum (Num (1), Num (2)) = 3

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

```
class Prod(e1: Expr, e2: Expr) extends Expr   // e1 * e2
class Var(x: String) extends Expr             // Variable ‘x’
```

You need to add methods for classification and access to all classes defined above.

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

- ○ 9
- ○ 10
- ○ 19
- ● 25
- ○ 35
- ○ 40

quadratic increase of methods

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

- O 9
- O 10
- O 19
- O 25
- O 35
- O 40

Non-Solution: Type Tests and Type Casts

A “hacky” solution could use type tests and type casts.

Scala let’s you do these using methods defined in class `Any`:

```
def isInstanceOf[T]: Boolean  // checks whether this object’s type conforms to ‘T‘
def asInstanceOf[T]: T        // treats this object as an instance of type ‘T‘
                              // throws ‘ClassCastException‘ if it isn’t.
```

These correspond to Java’s type tests and casts

```
Scala                 Java

x.isInstanceOf[T]     x instanceof T
x.asInstanceOf[T]     (T) x
```

But their use in Scala is discouraged, because there are better alternatives.

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if (e.isInstanceOf[Number])
    e.asInstanceOf[Number].numValue
  else if (e.isInstanceOf[Sum])
    eval(e.asInstanceOf[Sum].leftOp) +
    eval(e.asInstanceOf[Sum].rightOp)
  else throw new Error(”Unknown expression ” + e)
```

Assessment of this solution:

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if (e.isInstanceOf[Number])
    e.asInstanceOf[Number].numValue
  else if (e.isInstanceOf[Sum])
    eval(e.asInstanceOf[Sum].leftOp) +
    eval(e.asInstanceOf[Sum].rightOp)
  else throw new Error("Unknown expression " + e)
```

Expr

Num
numValue

Sum
leftOp
rightOp

Assessment of this solution:

+ no need for classification methods, access methods only for classes where the value is defined.
– low-level and potentially unsafe.

Solution 1: Object-Oriented Decomposition

For example, suppose that all you want to do is *evaluate* expressions.

You could then define:

```
trait Expr {
  def eval: Int ;  def show : String
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

But what happens if you'd like to display expressions now?

You have to define new methods in all the subclasses.

Limitations of OO Decomposition

And what if you want to simplify the expressions, say using the rule:

```
a * b + a * c   ->   a * (b + c)
```

Problem: This is a non-local simplification. It cannot be encapsulated in the method of a single object.

You are back to square one; you need test and access methods for all the different subclasses.

Pattern Matching

Reminder: Decomposition

The task we are trying to solve is find a general and convenient way to access objects in a extensible class hierarchy.

Expr

Number Sum Prod Var

eval
show
simplify

Attempts seen previously:

- *Classification and access methods*: quadratic explosion 40
- *Type tests and casts*: unsafe, low-level
- *Object-oriented decomposition*: does not always work, need to touch all classes to add a new method.

Solution 2: Functional Decomposition with Pattern Matching

Observation: the sole purpose of test and accessor functions is to *reverse* the construction process:

- Which subclass was used?
- What were the arguments of the constructor?

This situation is so common that many functional languages, Scala included, automate it.

new Sum(e_1, e_2)

Case Classes

A *case class* definition is similar to a normal class definition, except that it is preceded by the modifier `case`. For example:

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Like before, this defines a trait `Expr`, and two concrete subclasses `Number` and `Sum`.

Case Classes (2)

It also implicitly defines companion objects with `apply` methods.

```
object Number {
  def apply(n: Int) = new Number(n)
}
object Sum {
  def apply(e1: Expr, e2: Expr) = new Sum(e1, e2)
}
```

Number (2)

$\downarrow$

Number.apply (

so you can write `Number(1)` instead of `new Number(1)`.

However, these classes are now empty. So how can we access the members?

Pattern Matching

Pattern matching is a generalization of `switch` from C/Java to class hierarchies.

It's expressed in Scala using the keyword `match`.

Example

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

Match Syntax

Rules:

- `match` is followed by a sequence of *cases*, `pat => expr`.
- Each case associates an *expression* `expr` with a *pattern* `pat`.
- A `MatchError` exception is thrown if no pattern matches the value of the selector.

```
e match {
  case pat1 => expr1
      ⋮
  case patn => exprn
}
```

Forms of Patterns

Patterns are constructed from:

- *constructors*, e.g. `Number`, `Sum`,
- *variables*, e.g. `n`, `e1`, `e2`,
- *wildcard patterns* `_`,
- *constants*, e.g. `1`, `true`.

Number (n)

Number (_)

1, true, "abc", N

val N = 2

Sum (Number (1), Var (x)) =>

Variables always begin with a lowercase letter.

Variable n

Constant N

The same variable name can only appear once in a pattern. So, `Sum(x, x)` is not a legal pattern.

Sum (x, y)

Names of constants begin with a capital letter, with the exception of the reserved words `null`, `true`, `false`.

Evaluating Match Expressions

An expression of the form

$$\texttt{e match } \{ \texttt{ case } \texttt{p}_1 => \texttt{e}_1 \ \ldots \ \texttt{case } \texttt{p}_n => \texttt{e}_n \ \}$$

matches the value of the selector `e` with the patterns $p_1, \ldots, p_n$ in the order in which they are written.

The whole match expression is rewritten to the right-hand side of the first case where the pattern matches the selector *e*.

References to pattern variables are replaced by the corresponding parts in the selector.

What Do Patterns Match?

- A constructor pattern $C(p_1, ..., p_n)$ matches all the values of type C (or a subtype) that have been constructed with arguments matching the patterns $p_1, ..., p_n$.
- A variable pattern x matches any value, and *binds* the name of the variable to this value.
- A constant pattern c matches values that are equal to c (in the sense of ==)

Example

Example

```
eval(Sum(Number(1), Number(2)))
```

$\rightarrow$

```
Sum(Number(1), Number(2)) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

$\rightarrow$

```
eval(Number(1)) + eval(Number(2))
```

Example (2)

$\rightarrow$

```scala
Number(1) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
} + eval(Number(2))
```

$\rightarrow$

```scala
1 + eval(Number(2))
```

$\twoheadrightarrow$

```scala
3
```

Pattern Matching and Methods

Of course, it's also possible to define the evaluation function as a method of the base trait.

Example

```
trait Expr {
  def eval: Int = this match {
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
  }
}
```

eval(e)

Expression Problem"

Expr

def eval

Sum

def eval = ...

Number

def eval = ..

Exercise

Write a function `show` that uses pattern matching to return the representation of a given expressions as a string.

```
def show(e: Expr): String = ???
```

Exercise (Optional, Harder)

Add case classes `Var` for variables `x` and `Prod` for products `x * y` as discussed previously.

Change your `show` function so that it also deals with products.

Pay attention you get operator precedence right but to use as few parentheses as possible.

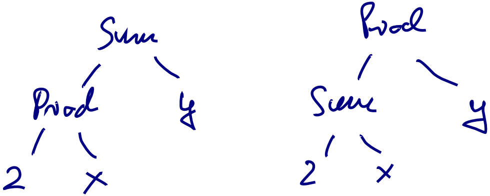
Example

```
Sum(Prod(2, Var("x")), Var("y"))
```

should print as "`2 * x + y`". But

```
Prod(Sum(2, Var("x")), Var("y"))
```

should print as "`(2 + x) * y`".



2 +(x * y)

(2 + x) * y

Lists

Lists

The list is a fundamental data structure in functional programming.

A list having $x_1, \ldots, x_n$ as elements is written List($x_1, \ldots, x_n$)

Example

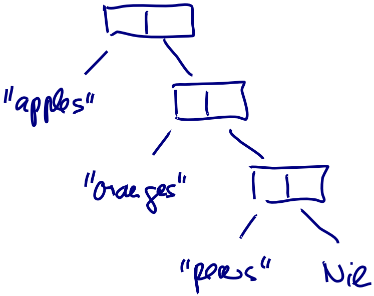
```
val fruit  = List("apples", "oranges", "pears")
val nums   = List(1, 2, 3, 4)
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty  = List()
```

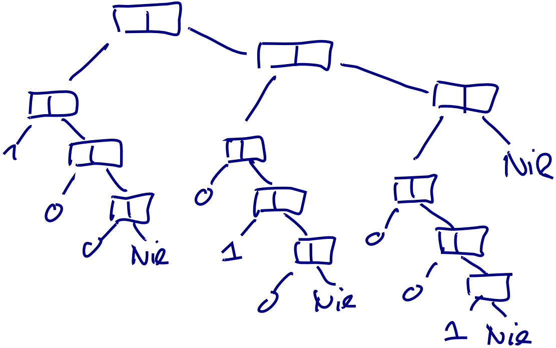
There are two important differences between lists and arrays.

- Lists are immutable — the elements of a list cannot be changed.
- Lists are recursive, while arrays are flat.

Lists

```
val fruit  = List("apples", "oranges", "pears")
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
```





The List Type

Like arrays, lists are homogeneous: the elements of a list must all have the same type.

The type of a list with elements of type `T` is written `scala.List[T]` or shorter just `List[T]`

Example

```
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]       = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Nothing]   = List()
```

Constructors of Lists

All lists are constructed from:

- the empty list `Nil`, and
- the construction operation `::` (pronounced *cons*):
  `x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`.

For example:

```
fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
empty = Nil
```

new Cons(x, xs)

x :: xs

Right Associativity

Convention: Operators ending in ":" associate to the right.

`A :: B :: C` is interpreted as `A :: (B :: C)`.

We can thus omit the parentheses in the definition above.

Example

```
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
```

Operators ending in ":" are also different in the they are seen as method calls of the *right-hand* operand.

So the expression above is equivalent to

:: ≈ prepend

```
Nil.::(4).::(3).::(2).::(1)
```

Operations on Lists

All operations on lists can be expressed in terms of the following three operations:

`head`	the first element of the list
`tail`	the list composed of all the elements except the first.
`isEmpty`	‘true‘ if the list is empty, ‘false‘ otherwise.

These operations are defined as methods of objects of type list. For example:

```
fruit.head      == "apples"
fruit.tail.head == "oranges"
diag3.head      == List(1, 0, 0)
empty.head      == throw new NoSuchElementException("head of empty list")
```

List Patterns

It is also possible to decompose lists with pattern matching.

`Nil`	The `Nil` constant
`p :: ps`	A pattern that matches a list with a `head` matching `p` and a `tail` matching `ps`.
`List(p1, ..., pn)`	same as `p1 :: ... :: pn :: Nil`

Example

`1 :: 2 :: xs`	Lists of that start with `1` and then `2`
`x :: Nil`	Lists of length 1
`List(x)`	Same as `x :: Nil`
`List()`	The empty list, same as `Nil`
`List(2 :: xs)`	A list that contains as only element another list that starts with `2`.

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

O	`L == 3`
O	`L == 4`
O	`L == 5`
O	`L >= 3`
O	`L >= 4`
O	`L >= 5`

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

- O `L == 3`
- O `L == 4`
- O `L == 5`
- ● `L >= 3`
- O `L >= 4`
- O `L >= 5`

Sorting Lists

Suppose we want to sort a list of numbers in ascending order:

- One way to sort the list `List(7, 3, 9, 2)` is to sort the tail `List(3, 9, 2)` to obtain `List(2, 3, 9)`.
- The next step is to insert the head `7` in the right place to obtain the result `List(2, 3, 7, 9)`.

This idea describes *Insertion Sort* :

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

Exercise

Complete the definition insertion sort by filling in the `???`s in the definition below:

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

What is the worst-case complexity of insertion sort relative to the length of the input list `N`?

```
O       the sort takes constant time
O       proportional to N
O       proportional to N log(N)
O       proportional to N * N
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

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

What is the worst-case complexity of insertion sort relative to the length of the input list N?

```
O      the sort takes constant time
O      proportional to N
O      proportional to N log(N)
O      proportional to N * N
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
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)
}
```

What is the worst-case complexity of insertion sort relative to the length of the input list N?

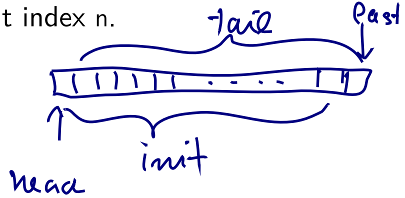
```
O      the sort takes constant time
O      proportional to N
O      proportional to N * log(N)
O      proportional to N * N
```

More Functions on Lists

List Methods (1)

Sublists and element access:

`xs.length`	The number of elements of `xs`.
`xs.last`	The list's last element, exception if `xs` is empty.
`xs.init`	A list consisting of all elements of `xs` except the last one, exception if `xs` is empty.
`xs take n`	A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
`xs drop n`	The rest of the collection after taking `n` elements.
`xs(n)`	(or, written out, `xs apply n`). The element of `xs` at index `n`.



List Methods (2)

Creating new lists:

`xs ++ ys`	The list consisting of all elements of `xs` followed by all elements of `ys`.
`xs.reverse`	The list containing the elements of `xs` in reversed order.
`xs updated (n, x)`	The list containing the same elements as `xs`, except at index `n` where it contains `x`.

Finding elements:

`xs indexOf x`	The index of the first element in `xs` equal to `x`, or `-1` if `x` does not appear in `xs`.
`xs contains x`	same as `xs indexOf x >= 0`

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error(”last of empty list”)
  case List(x) =>
  case y :: ys =>
}
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```scala
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error("last of empty list")
  case List(x) => x
  case y :: ys =>
}
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error(”last of empty list”)
  case List(x) => x
  case y :: ys => last(ys)
}
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
}
```

So, `last` takes steps proportional to the length of the list `xs`.

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match {
  case List() => throw new Error("init of empty list")
  case List(x) => ???
  case y :: ys => ???
}
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match {
  case List() => throw new Error("init of empty list")
  case List(x) =>
  case y :: ys =>
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) =
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() =>
  case z :: zs =>
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs =>
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs => z :: concat(zs, ys)
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs => z :: concat(zs, ys)
}
```

What is the complexity of concat? $|xs|$

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing a stand-alone function:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() =>  xs
  case y :: ys =>  reverse(ys) ++ List(y)
}
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing a stand-alone function:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() => List()
  case y :: ys => reverse(ys) ++ List(y)
}
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing a stand-alone function:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() => List()
  case y :: ys => reverse(ys) ++ List(y)
}
```

$n * n$

What is the complexity of `reverse`?

Can we do better? (to be solved later).

Exercise

Remove the `n`'th element of a list `xs`. If `n` is out of bounds, return `xs` itself.

```
def removeAt[T](xs: List[T], n: Int) = ???
```

Usage example:

```
removeAt(1, List('a', 'b', 'c', 'd'))
```

> *List(a, c, d)*

Exercise (Harder, Optional)

Flatten a list structure:

```
def flatten(xs: List[Any]): List[Any] = ???

flatten(List(List(1, 1), 2, List(3, List(5, 8))))
            >   res0: List[Any] = List(1, 1, 2, 3, 5, 8)
```

Pairs and Tuples

Sorting Lists Faster

As a non-trivial example, let's design a function to sort lists that is more efficient than insertion sort.

A good algorithm for this is *merge sort*. The idea is as follows:

If the list consists of zero or one elements, it is already sorted.

Otherwise,

- Separate the list into two sub-lists, each containing around half of the elements of the original list.
- Sort the two sub-lists.
- Merge the two sorted sub-lists into a single sorted list.

First MergeSort Implementation

Here is the implementation of that algorithm in Scala:

```scala
def msort(xs: List[Int]): List[Int] = {
  val n = xs.length/2
  if (n == 0) xs
  else {
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs splitAt n
    merge(msort(fst), msort(snd))
  }
}
```

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) =
  xs match {
    case Nil =>
      ys
    case x :: xs1 =>
      ys match {
        case Nil =>
          xs
        case y :: ys1 =>
          if (x < y) x :: merge(xs1, ys)
          else y :: merge(xs, ys1)
      }
    }
```

The SplitAt Function

The `splitAt` function on lists returns two sublists

- the elements up the the given index
- the elements from that index

The lists are returned in a *pair*.

Detour: Pair and Tuples

The pair consisting of `x` and `y` is written `(x, y)` in Scala.

Example

```
val pair = ("answer", 42)
```
> *pair : (String, Int) = (answer,42)*

The type of `pair` above is `(String, Int)`.

Pairs can also be used as patterns:

```
val (label, value) = pair
```
> *label : String = answer*
| *value : Int = 42*

This works analogously for tuples with more than two elements.

Translation of Tuples

A tuple type $(\mathsf{T}_1, ..., \mathsf{T}_n)$ is an abbreviation of the parameterized type

$$\texttt{scala.Tuple}n[\mathsf{T}_1, ..., \mathsf{T}_n]$$

A tuple expression $(\mathsf{e}_1, ..., \mathsf{e}_n)$ is equivalent to the function application

$$\texttt{scala.Tuple}n(\mathsf{e}_1, ..., \mathsf{e}_n)$$

A tuple pattern $(\mathsf{p}_1, ..., \mathsf{p}_n)$ is equivalent to the constructor pattern

$$\texttt{scala.Tuple}n(\mathsf{p}_1, ..., \mathsf{p}_n)$$

The Tuple class

Here, all `Tuple`*n* classes are modeled after the following pattern:

```
case class Tuple2[T1, T2](_1: +T1, _2: +T2) {
  override def toString = "(" + _1 + "," + _2 +")"
}
```

The fields of a tuple can be accessed with names _1, _2, ...

So instead of the pattern binding

```
val (label, value) = pair
```

one could also have written:

```
val label = pair._1
val value = pair._2
```

But the pattern matching form is generally preferred.

Exercise

The `merge` function as given uses a nested pattern match.

This does not reflect the inherent symmetry of the merge algorithm.

Rewrite `merge` using a pattern matching over pairs.

```
def merge(xs: List[Int], ys: List[Int]): List[Int] =
  (xs, ys) match {
    ???
  }
```

Implicit Parameters

Making Sort more General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ...
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) = {
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
}
```

Merge then needs to be adapted as follows:

```
def merge(xs: List[T], ys: List[T]) = (xs, ys) match {
  ...
  case (x :: xs1, y :: ys1) =>
    if (lt(x, y)) ...
    else ...
}
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruit = List("apple", "pear", "orange", "pineapple")

merge(xs)((x: Int, y: Int) => x < y)
merge(fruit)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `merge(xs)`:

```
merge(xs)((x, y) => x < y)
```

Parametrization with Ordered

There is already a class in the standard library that represents orderings.

```
scala.math.Ordering[T]
```

provides ways to compare elements of type `T`. So instead of parameterizing with the `lt` operation directly, we could parameterize with `Ordering` instead:

```
def msort[T](xs: List[T])(ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst)(ord), msort(snd)(ord)) ...
```

Ordered Instances:

Calling the new `msort` can be done like this:

```
import math.Ordering

msort(nums)(Ordering.Int)
msort(fruits)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

Aside: Implicit Parameters

Problem: Passing around `lt` or `ord` values is cumbersome.

We can avoid this by making `ord` an implicit parameter.

```
def msort[T](xs: List[T])(implicit ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst), msort(snd)) ...
```

Then calls to `msort` can avoid the ordering parameters:

```
msort(nums)
msort(fruits)
```

The compiler will figure out the right implicit to pass based on the demanded type.

Rules for Implicit Parameters

Say, a function takes an implicit parameter of type `T`.

The compiler will search an implicit definition that

- is marked `implicit`
- has a type compatible with `T`
- is visible at the point of the function call, or is defined in a companion object associated with `T`.

If there is a single (most specific) definition, it will be taken as actual argument for the implicit parameter.

Otherwise it's an error.

Exercise: Implicit Parameters

Consider the following line of the definition of `msort`:

```
... merge(msort(fst), msort(snd)) ...
```

Which implicit argument is inserted?

O `Ordering.Int`

O `Ordering.String`

O `the "ord" parameter of "msort"`

Higher-order List Functions

Recurring Patterns for Computations on Lists

The examples have shown that functions on lists often have similar structures.

We can identify several recurring patterns, like,

- transforming each element in a list in a certain way,
- retrieving a list of all elements satisfying a criterion,
- combining the elements of a list using an operator.

Functional languages allow programmers to write generic functions that implement patterns such as these using higher-order functions.

Applying a Function to Elements of a List

A common operation is to transform each element of a list and then return the list of results.

For example, to multiply each element of a list by the same factor, you could write:

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match {
  case Nil     => xs
  case y :: ys => y * factor :: scaleList(ys, factor)
}
```

Map

This scheme can be generalized to the method `map` of the `List` class.
A simple way to define `map` is as follows:

```
abstract class List[T] { ...
  def map[U](f: T => U): List[U] = this match {
    case Nil      => this
    case x :: xs => f(x) :: xs.map(f)
  }
```

(in fact, the actual definition of `map` is a bit more complicated, because it is tail-recursive, and also because it works for arbitrary collections, not just lists).

Using `map`, `scaleList` can be written more concisely.

```
def scaleList(xs: List[Double], factor: Double) =
  xs map (x => x * factor)
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of `squareList`.

```
def squareList(xs: List[Int]): List[Int] = xs match {
  case Nil     => ???
  case y :: ys => ???
}

def squareList(xs: List[Int]): List[Int] =
  xs map ???
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of `squareList`.

```
def squareList(xs: List[Int]): List[Int] = xs match {
  case Nil     =>  Nil
  case y :: ys =>  y*y :: squareList(ys)
}
```

```
def squareList(xs: List[Int]): List[Int] =
  xs map (x => x*x)
```

Filtering

Another common operation on lists is the selection of all elements satisfying a given condition. For example:

```
def posElems(xs: List[Int]): List[Int] = xs match {
  case Nil      => xs
  case y :: ys => if (y > 0) y :: posElems(ys) else posElems(ys)
}
```

Filter

This pattern is generalized by the method `filter` of the `List` class:

```
abstract class List[T] {
  ...
  def filter(p: T => Boolean): List[T] = this match {
    case Nil     => this
    case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
  }
}
```

Using `filter`, `posElems` can be written more concisely.

```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x => x > 0)
```

Variations of Filter

Besides filter, there are also the following methods that extract sublists based on a predicate:

`xs filterNot p`	Same as `xs filter (x => !p(x))`; The list consisting of those elements of `xs` that do not satisfy the predicate `p`.
`xs partition p`	Same as `(xs filter p, xs filterNot p)`, but computed in a single traversal of the list `xs`.
`xs takeWhile p`	The longest prefix of list `xs` consisting of elements that all satisfy the predicate `p`.
`xs dropWhile p`	The remainder of the list `xs` after any leading elements satisfying `p` have been removed.
`xs span p`	Same as `(xs takeWhile p, xs dropWhile p)` but computed in a single traversal of the list `xs`.

Exercise

Write a function pack that packs consecutive duplicates of list elements into sublists. For instance,

```
pack(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(List("a", "a", "a"), List("b"), List("c", "c"), List("a")).
```

You can use the following template:

```
def pack[T](xs: List[T]): List[List[T]] = xs match {
  case Nil      => Nil
  case x :: xs1 => ???
}
```

Exercise

Using pack, write a function encode that produces the run-length encoding of a list.

The idea is to encode n consecutive duplicates of an element x as a pair (x, n). For instance,

```
encode(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(("a", 3), ("b", 1), ("c", 2), ("a", 1)).
```

Reduction of Lists

Reduction of Lists

Another common operation on lists is to combine the elements of a list using a given operator.

For example:

```
sum(List(x1, ..., xn))          =  0 + x1 + ... + xn
product(List(x1, ..., xn))      =  1 * x1 * ... * xn
```

We can implement this with the usual recursive schema:

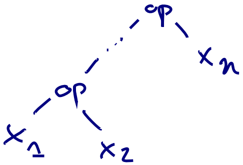
```
def sum(xs: List[Int]): Int = xs match {
  case Nil      => 0
  case y :: ys => y + sum(ys)
}
```

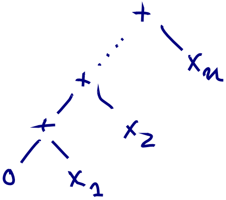
ReduceLeft

This pattern can be abstracted out using the generic method `reduceLeft`:

`reduceLeft` inserts a given binary operator between adjacent elements of a list:

```
List(x1, ..., xn) reduceLeft op   =  (...(x1 op x2) op ... ) op xn
```





Using `reduceLeft`, we can simplify:

```scala
def sum(xs: List[Int])     = (0 :: xs) reduceLeft ((x, y) => x + y)
def product(xs: List[Int]) = (1 :: xs) reduceLeft ((x, y) => x * y)
```

A Shorter Way to Write Functions

Instead of `((x, y) => x * y))`, one can also write shorter:

```
(_ * _)
```

((x,y) => (x * y))

Every _ represents a new parameter, going from left to right.

The parameters are defined at the next outer pair of parentheses (or the whole expression if there are no enclosing parentheses).

So, `sum` and `product` can also be expressed like this:

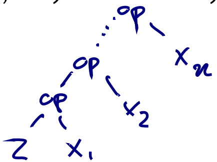
```
def sum(xs: List[Int])     = (0 :: xs) reduceLeft (_ + _)
def product(xs: List[Int]) = (1 :: xs) reduceLeft (_ * _)
```

FoldLeft

The function `reduceLeft` is defined in terms of a more general function, `foldLeft`.

`foldLeft` is like `reduceLeft` but takes an *accumulator*, `z`, as an additional parameter, which is returned when `foldLeft` is called on an empty list.

```
(List(x1, ..., xn) foldLeft z)(op)   =  (...(z op x1) op ... ) op xn
```



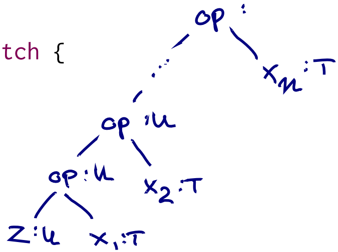
So, `sum` and `product` can also be defined as follows:

```
def sum(xs: List[Int])      = (xs foldLeft 0) (_ + _)
def product(xs: List[Int])  = (xs foldLeft 1) (_ * _)
```

Implementations of ReduceLeft and FoldLeft

`foldLeft` and `reduceLeft` can be implemented in class `List` as follows.

```scala
abstract class List[T] { ...
  def reduceLeft(op: (T, T) => T): T = this match {
    case Nil     => throw new Error("Nil.reduceLeft")
    case x :: xs => (xs foldLeft x)(op)
  }
  def foldLeft[U](z: U)(op: (U, T) => U): U = this match {
    case Nil     => z
    case x :: xs => (xs foldLeft op(z, x))(op)
  }
}
```

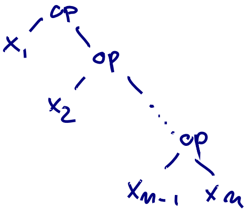


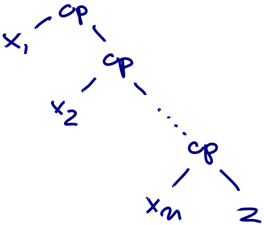
FoldRight and ReduceRight

Applications of `foldLeft` and `reduceLeft` unfold on trees that lean to the left.

They have two dual functions, `foldRight` and `reduceRight`, which produce trees which lean to the right, i.e.,

```
List(x1, ..., x{n-1}, xn) reduceRight op = x1 op ( ... (x{n-1} op xn) ... )
(List(x1, ..., xn) foldRight acc)(op)    = x1 op ( ... (xn op acc) ... )
```

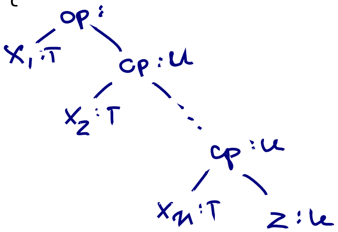




Implementation of FoldRight and ReduceRight

They are defined as follows

```
def reduceRight(op: (T, T) => T): T = this match {
  case Nil => throw new Error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))
}
def foldRight[U](z: U)(op: (T, U) => U): U = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}
```



Difference between FoldLeft and FoldRight

For operators that are associative and commutative, `foldLeft` and `foldRight` are equivalent (even though there may be a difference in efficiency).

But sometimes, only one of the two operators is appropriate.

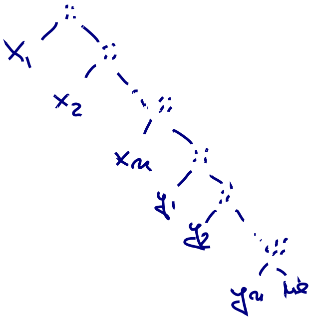
Exercise

Here is another formulation of concat:

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  (xs foldRight ys) (_ :: _)
```

Here, it isn't possible to replace foldRight by foldLeft. Why?

```
O     The types would not work out
O     The resulting function would not terminate
O     The result would be reversed
```



Back to Reversing Lists

We now develop a function for reversing lists which has a linear cost.

The idea is to use the operation `foldLeft`:

```
def reverse[T](xs: List[T]): List[T] = (xs foldLeft z?)(op?)
```

All that remains is to replace the parts `z?` and `op?`.

Let's try to *compute* them from examples.

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil

=   reverse(Nil)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil
=   reverse(Nil)
=   (Nil foldLeft z?)(op)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil
=   reverse(Nil)
=   (Nil foldLeft z?)(op)
=   z?
```

Consequently, `z? = List()`

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))

=   (List(x) foldLeft Nil)(op?)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)
=   reverse(List(x))
=   (List(x) foldLeft Nil)(op?)
=   op?(Nil, x)
```

Consequently, `op?(Nil, x) = List(x) = x :: List()`.

This suggests to take for `op?` the operator `::` but with its operands swapped.

Deduction of Reverse(3)

We thus arrive at the following implementation of `reverse`.

```
def reverse[a](xs: List[T]): List[T] =
  (xs foldLeft List[T]())((xs, x) => x :: xs)
```

Remark: the type parameter in `List[T]()` is necessary for type inference.

Question: What is the complexity of this implementation of `reverse` ?

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  (xs foldRight List[U]())( ??? )

def lengthFun[T](xs: List[T]): Int =
  (xs foldRight 0)( ??? )
```

Reasoning About Lists

Laws of Concat

Recall the concatenation operation ++ on lists. concat

We would like to verify that concatenation is associative, and that it admits the empty list `Nil` as neutral element to the left and to the right:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
       xs ++ Nil  =  xs
       Nil ++ xs  =  xs
```

Q: How can we prove properties like these?

Laws of Concat

Recall the concatenation operation `++` on lists.

We would like to verify that concatenation is associative, and that it admits the empty list `Nil` as neutral element to the left and to the right:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
       xs ++ Nil  =  xs
       Nil ++ xs  =  xs
```

Q: How can we prove properties like these?

A: By *structural induction* on lists.

Reminder: Natural Induction

Recall the principle of proof by *natural induction*:

To show a property $\mathsf{P}(\mathsf{n})$ for all the integers $\mathsf{n} \geq \mathsf{b}$,

- Show that we have $\mathsf{P}(\mathsf{b})$ (*base case*),
- for all integers $\mathsf{n} \geq \mathsf{b}$ show the *induction step*:

  if one has $\mathsf{P}(\mathsf{n})$*, then one also has* $\mathsf{P}(\mathsf{n}+1)$*.*

Example

Given:

```
def factorial(n: Int): Int =
  if (n == 0) 1              // 1st clause
  else n * factorial(n-1)  // 2nd clause
```

Show that, for all `n >= 4`

```
factorial(n) >= power(2, n)
```

2^n

Base Case

Base case: 4

This case is established by simple calculations:

```
factorial(4) = 24 >= 16 = power(2, 4)
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)
```

$\text{factorial}(n) \geq 2^n$

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)    // by 2nd clause in factorial
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)          // by calculating
```

Induction Step

Induction step: n+1

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)          // by calculating

>=  2 * power(2, n)           // by induction hypothesis
```

Induction Step

Induction step: n+1

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)          // by calculating

>=  2 * power(2, n)           // by induction hypothesis

=   power(2, n + 1)           // by definition of power
```

$2 * 2^n = 2^{n+1}$

Referential Transparency

Note that a proof can freely apply reduction steps as equalities to some part of a term.

That works because pure functional programs don't have side effects; so that a term is equivalent to the term to which it reduces.

This principle is called *referential transparency*.

Structural Induction

The principle of structural induction is analogous to natural induction:

To prove a property `P(xs)` for all lists `xs`,

- show that `P(Nil)` holds (*base case*),
- for a list `xs` and some element `x`, show the *induction step*:

  if `P(xs)` *holds, then* `P(x :: xs)` *also holds.*

Example

Let's show that, for lists `xs`, `ys`, `zs`:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
```

To do this, use structural induction on `xs`. From the previous implementation of `concat`,

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case x :: xs1 => x :: concat(xs1, ys)
}
```

distill two *defining clauses* of `++`:

```
       Nil ++ ys  =  ys                    // 1st clause
(x :: xs1) ++ ys  =  x :: (xs1 ++ ys)      // 2nd clause
```

Base Case

Base case: `Nil`

For the left-hand side we have:

```
(Nil ++ ys) ++ zs
```

Base Case

$\boxed{\textbf{Base case: } \texttt{Nil}}$

For the left-hand side we have:

```
(Nil ++ ys) ++ zs

=   ys ++ zs        // by 1st clause of ++
```

Base Case

$\boxed{\textbf{Base case: } \texttt{Nil}}$

For the left-hand side we have:

```
  (Nil ++ ys) ++ zs

=    ys ++ zs          // by 1st clause of ++
```

For the right-hand side, we have:

```
  Nil ++ (ys ++ zs)
```

Base Case

Base case: `Nil`

For the left-hand side we have:

```
  (Nil ++ ys) ++ zs

=   ys ++ zs         // by 1st clause of ++
```

For the right-hand side, we have:

```
  Nil ++ (ys ++ zs)

=   ys ++ zs         // by 1st clause of ++
```

This case is therefore established.

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs
```

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs

=   (x :: (xs ++ ys)) ++ zs     // by 2nd clause of ++
```

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
  ((x :: xs) ++ ys) ++ zs

=   (x :: (xs ++ ys)) ++ zs     // by 2nd clause of ++

=   x :: ((xs ++ ys) ++ zs)     // by 2nd clause of ++
```

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
  ((x :: xs) ++ ys) ++ zs

=   (x :: (xs ++ ys)) ++ zs    // by 2nd clause of ++

=   x :: ((xs ++ ys) ++ zs)    // by 2nd clause of ++

=   x :: (xs ++ (ys ++ zs))    // by induction hypothesis
```

Induction Step: RHS

For the right hand side we have:

```
(x :: xs) ++ (ys ++ zs)
```

Induction Step: RHS

For the right hand side we have:

```
(x :: xs) ++ (ys ++ zs)

=   x :: (xs ++ (ys ++ zs))    // by 2nd clause of ++
```

So this case (and with it, the property) is established.

Exercise

Show by induction on `xs` that `xs ++ Nil = xs`.

How many equations do you need for the inductive step?

●	2
0	3
0	4

Base case: xs = Nil

Nil ++ Nil

= Nil // by 1st clause

Induction step: x :: xs

(x :: xs) ++ Nil = x :: xs ?

= x :: (xs ++ Nil) // 2nd clause

= x :: xs // by i.h.

= ✓

A Larger Equational Proof on Lists

A Law of Reverse

For a more difficult example, let's consider the `reverse` function.

We pick its inefficient definition, because its more amenable to equational proofs:

```
      Nil.reverse  =  Nil                         // 1st clause
(x :: xs).reverse  =  xs.reverse ++ List(x)       // 2nd clause
```

We'd like to prove the following proposition

```
xs.reverse.reverse  =  xs
```

Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse            // by 1st clause of reverse
=   Nil                    // by 1st clause of reverse
```

Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse              // by 1st clause of reverse
=   Nil                      // by 1st clause of reverse
```

For the induction step, let's try:

```
(x :: xs).reverse.reverse
=   (xs.reverse ++ List(x)).reverse // by 2nd clause of reverse
```

Proof

By induction on xs. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse              // by 1st clause of reverse
=   Nil                      // by 1st clause of reverse
```

For the induction step, let's try:

```
(x :: xs).reverse.reverse
=   (xs.reverse ++ List(x)).reverse // by 2nd clause of reverse
```

We can't do anything more with this expression, therefore we turn to the right-hand side:

```
x :: xs
=   x :: xs.reverse.reverse           // by induction hypothesis
```

Both sides are simplified in different expressions.

To Do

We still need to show:

ys ys

```
(xs.reverse ++ List(x)).reverse  =  x :: xs.reverse.reverse
```

Trying to prove it directly by induction doesn't work.

We must instead try to *generalize* the equation. For *any* list `ys`,

```
(ys ++ List(x)).reverse  =  x :: ys.reverse
```

This equation can be proved by a second induction argument on `ys`.

Auxiliary Equation, Base Case

ys = Nil

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse       // to show: =  x :: Nil.reverse

=   List(x).reverse            // by 1st clause of ++
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++

=   (x :: Nil).reverse          // by definition of List
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++

=   (x :: Nil).reverse          // by definition of List
=   Nil.reverse ++ List(x)
=   Nil ++ (x :: Nil)           // by 2nd clause of reverse
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse       // to show: =  x :: Nil.reverse

=   List(x).reverse            // by 1st clause of ++

=   (x :: Nil).reverse         // by definition of List

=   Nil ++ (x :: Nil)          // by 2nd clause of reverse

=   x :: Nil                   // by 1st clause of ++
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse       // to show: =  x :: Nil.reverse

=   List(x).reverse            // by 1st clause of ++

=   (x :: Nil).reverse         // by definition of List

=   Nil ++ (x :: Nil)          // by 2nd clause of reverse

=   x :: Nil                   // by 1st clause of ++

=   x :: Nil.reverse           // by 1st clause of reverse
```

≈

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse            // to show: =  x :: (y :: ys).reverse
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse            // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse        // by 2nd clause of ++
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse              // to show: =  x :: (y :: ys).reverse

=    (y :: (ys ++ List(x))).reverse         // by 2nd clause of ++

=    (ys ++ List(x)).reverse ++ List(y)     // by 2nd clause of reverse
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse            // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse        // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)    // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)          // by the induction hypothesis
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse             // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse         // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)     // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)           // by the induction hypothesis

=   x :: (ys.reverse ++ List(y))           // by 1st clause of ++
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse           // to show: =  x :: (y :: ys).reverse
=   (y :: (ys ++ List(x))).reverse       // by 2nd clause of ++
=   (ys ++ List(x)).reverse ++ List(y)   // by 2nd clause of reverse
=   (x :: ys.reverse) ++ List(y)         // by the induction hypothesis
=   x :: (ys.reverse ++ List(y))         // by 1st clause of ++
=   x :: (y :: ys).reverse               // by 2nd clause of reverse
```

unfold

fold

=

This establishes the auxiliary equation, and with it the main proposition.

fold / unfold method

Exercise (Open-Ended, Harder)

Prove the following distribution law for `map` over concatenation.

For any lists `xs`, `ys`, function `f`:

```
(xs ++ ys) map f  =  (xs map f) ++ (ys map f)
```

You will need the clauses of $++$ as well as the following clauses for `map`:

```
      Nil map f  =  Nil
(x :: xs) map f  =  f(x) :: (xs map f)
```

Other Collections

Other Sequences

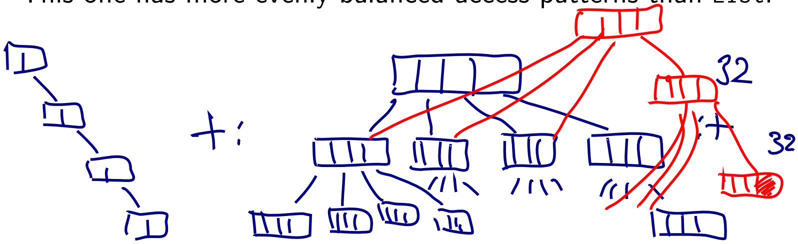
We have seen that lists are *linear*: Access to the first element is much faster than access to the middle or end of a list.

The Scala library also defines an alternative sequence implementation, `Vector`.

This one has more evenly balanced access patterns than `List`.

$\log_{32}(N)$

$\log_{32}(N)$



$32 * 32 = 2^{10}$

2^{15}

2^{20}

2^{25}

2^{30}

Operations on Vectors

Vectors are created analogously to lists:

```
val nums = Vector(1, 2, 3, -88)
val people = Vector("Bob", "James", "Peter")
```

They support the same operations as lists, with the exception of `::`

Instead of `x :: xs`, there is

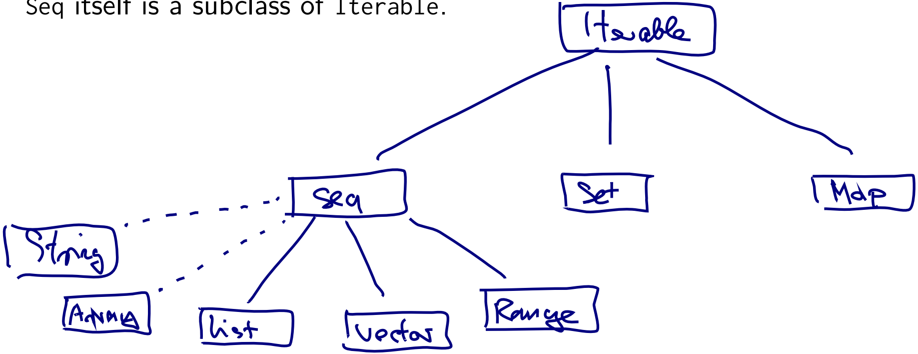
`x +: xs`	Create a new vector with leading element `x`, followed by all elements of `xs`.
`xs :+ x`	Create a new vector with trailing element `x`, preceded by all elements of `xs`.

(Note that the `:` always points to the sequence.)

Collection Hierarchy

A common base class of `List` and `Vector` is `Seq`, the class of all *sequences*.

`Seq` itself is a subclass of `Iterable`.



Arrays and Strings

Arrays and Strings support the same operations as `Seq` and can implicitly be converted to sequences where needed.

(They cannot be subclasses of `Seq` because they come from Java)

```
val xs: Array[Int] = Array(1, 2, 3)
xs map (x => 2 * x)

val ys: String = "Hello world!"
ys filter (_.isUpper)
```

Ranges

Another simple kind of sequence is the *range*.

It represents a sequence of evenly spaced integers.

Three operators:

`to` (inclusive), `until` (exclusive), `by` (to determine step value):

```
val r: Range = 1 until 5    // 1, 2, 3, 4
val s: Range = 1 to 5       // 1, 2, 3, 4, 5
1 to 10 by 3                // 1, 4, 7, 10
6 to 1 by -2                // 6, 4, 2
```

Ranges a represented as single objects with three fields: lower bound, upper bound, step value.

Some more Sequence Operations:

`xs exists p`	`true` if there is an element `x` of `xs` such that `p(x)` holds, `false` otherwise.
`xs forall p`	`true` if `p(x)` holds for all elements `x` of `xs`, `false` otherwise.
`xs zip ys`	A sequence of pairs drawn from corresponding elements of sequences `xs` and `ys`.
`xs.unzip`	Splits a sequence of pairs `xs` into two sequences consisting of the first, respectively second halves of all pairs.
`xs.flatMap f`	Applies collection-valued function `f` to all elements of `xs` and concatenates the results
`xs.sum`	The sum of all elements of this numeric collection.
`xs.product`	The product of all elements of this numeric collection
`xs.max`	The maximum of all elements of this collection (an `Ordering` must exist)
`xs.min`	The minimum of all elements of this collection

Example: Combinations

To list all combinations of numbers x and y where x is drawn from 1..M and y is drawn from 1..N:

```
(1 to M) flatMap (x => (1..N) map (y => (x,y)))
```

Example: Combinations

To list all combinations of numbers `x` and `y` where `x` is drawn from `1..M` and `y` is drawn from `1..N`:

```
(1 to M) flatMap (x => (1 to N) map (y => (x, y)))
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map(xy => xy._1 * xy._2).sum
```

$$\sum_{i=1}^{n} x_i * y_i$$

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map(xy => xy._1 * xy._2).sum
```

An alternative way to write this is with a *pattern matching function value*.

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map{ case (x, y) => x * y }.sum
```

Generally, the function value

```
{ case p1 => e1 ... case pn => en }
```

is equivalent to

```
x => x match { case p1 => e1 ... case pn => en }
```

Exercise:

A number `n` is *prime* if the only divisors of `n` are `1` and `n` itself.

What is a high-level way to write a test for primality of numbers?
For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean = ???
```

Exercise:

A number n is *prime* if the only divisors of n are 1 and n itself.

What is a high-level way to write a test for primality of numbers?
For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean = (2 until n) forall (d => n % d != 0)
```

Combinatorial Search and For-Expressions

Handling Nested Sequences

We can extend the usage of higher order functions on sequences to many calculations which are usually expressed using nested loops.

Example: Given a positive integer `n`, find all pairs of positive integers `i` and `j`, with `1 <= j < i < n` such that `i + j` is prime.

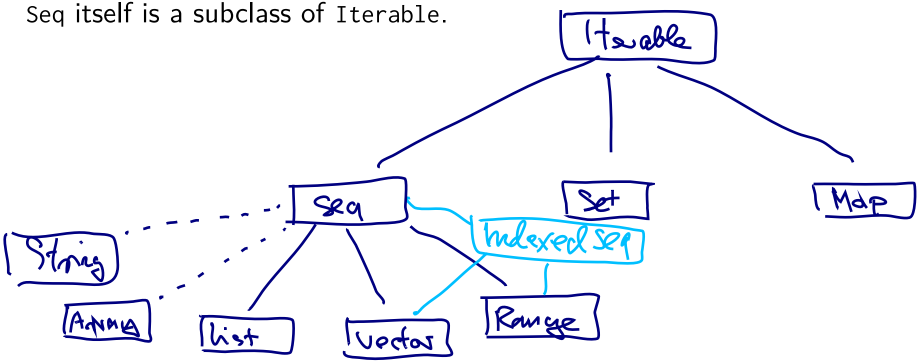
For example, if `n = 7`, the sought pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

Collection Hierarchy

A common base class of `List` and `Vector` is `Seq`, the class of all *sequences*.

`Seq` itself is a subclass of `Iterable`.



Algorithm

A natural way to do this is to:

- Generate the sequence of all pairs of integers `(i, j)` such that `1 <= j < i < n`.
- Filter the pairs for which `i + j` is prime.

One natural way to generate the sequence of pairs is to:

- Generate all the integers `i` between `1` and `n` (excluded).
- For each integer `i`, generate the list of pairs `(i, 1), ..., (i, i-1)`.

This can be achieved by combining `until` and `map`:

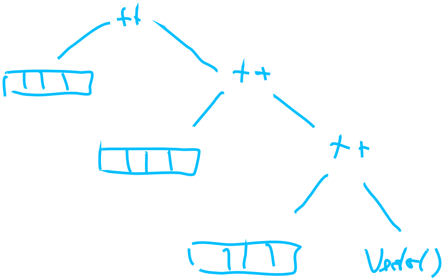
```
(1 until n) map (i =>
  (1 until i) map (j => (i, j)))
```

Generate Pairs

The previous step gave a sequence of sequences, let's call it `xss`.

We can combine all the sub-sequences using `foldRight` with `++`:

```
(xss foldRight Seq[Int]())(_ ++ _)
```



Or, equivalently, we use the built-in method `flatten`

```
xss.flatten
```

This gives:

```
((1 until n) map (i =>
  (1 until i) map (j => (i, j)))).flatten
```

Generate Pairs (2)

Here's a useful law:

```
xs flatMap f  =  (xs map f).flatten
```

Hence, the above expression can be simplified to

```
(1 until n) flatMap (i =>
    (1 until i) map (j => (i, j)))
```

Assembling the pieces

By reassembling the pieces, we obtain the following expression:

```
(1 until n) flatMap (i =>
   (1 until i) map (j => (i, j))) filter ( pair =>
      isPrime(pair._1 + pair._2))
```

This works, but makes most people's head hurt.

Is there a simpler way?

For-Expressions

Higher-order functions such as `map`, `flatMap` or `filter` provide powerful constructs for manipulating lists.

But sometimes the level of abstraction required by these function make the program difficult to understand.

In this case, Scala's `for` expression notation can help.

For-Expression Example

Let `persons` be a list of elements of class `Person`, with fields `name` and `age`.

```
case class Person(name: String, age: Int)
```

To obtain the names of persons over 20 years old, you can write:

```
for ( p <- persons if p.age > 20 ) yield p.name
```

which is equivalent to:

```
persons filter (p => p.age > 20) map (p => p.name)
```

The for-expression is similar to loops in imperative languages, except that it builds a list of the results of all iterations.

Syntax of For

A for-expression is of the form

```
for ( s ) yield e
```

where `s` is a sequence of *generators* and *filters*, and `e` is an expression whose value is returned by an iteration.

- A *generator* is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection.
- A *filter* is of the form `if f` where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators in the sequence, the last generators vary faster than the first.

Instead of `( s )`, braces `{ s }` can also be used, and then the sequence of generators and filters can be written on multiple lines without requiring semicolons.

Use of For

Here are two examples which were previously solved with higher-order functions:

Given a positive integer `n`, find all the pairs of positive integers `(i, j)` such that `1 <= j < i < n`, and `i + j` is prime.

```
for {
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
} yield (i, j)
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =
      (for ((x,y) <- xs zip ys) yield x*y).sum
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =

  (for ((x, y) <- xs zip ys) yield x * y).sum
```

Combinatorial Search Example

Sets

Sets are another basic abstraction in the Scala collections.

A set is written analogously to a sequence:

```
val fruit = Set("apple", "banana", "pear")
val s = (1 to 6).toSet
```

Most operations on sequences are also available on sets:

```
s map (_ + 2)
fruit filter (_.startsWith == "app")
s.nonEmpty
```

(see `Iterables` Scaladoc for a list of all supported operations)

Sets vs Sequences

The principal differences between sets and sequences are:

1. Sets are unordered; the elements of a set do not have a predefined order in which they appear in the set

2. sets do not have duplicate elements:

```
s map (_ / 2)       // Set(2, 0, 3, 1)
```

3. The fundamental operation on sets is `contains`:

```
s contains 5        // true
```

Example: N-Queens

The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.

- In other words, there can't be two queens in the same row, column, or diagonal.

We now develop a solution for a chessboard of any size, not just 8.

One way to solve the problem is to place a queen on each row.

Once we have placed `k - 1` queens, one must place the `k`th queen in a column where it's not "in check" with any other queen on the board.

Algorithm

We can solve this problem with a recursive algorithm:

- Suppose that we have already generated all the solutions consisting of placing `k-1` queens on a board of size `n`.
- Each solution is represented by a list (of length `k-1`) containing the numbers of columns (between `0` and `n-1`).
- The column number of the queen in the `k-1`th row comes first in the list, followed by the column number of the queen in row `k-2`, etc.
- The solution set is thus represented as a set of lists, with one element for each solution.
- Now, to place the `k`th queen, we generate all possible extensions of each solution preceded by a new queen:

Implementation

```scala
def queens(n: Int) = {
  def placeQueens(k: Int): Set[List[Int]] = {
    if (k == 0) Set(List())
    else
      for {
        queens <- placeQueens(k - 1)
        col <- 0 until n
        if isSafe(col, queens)
      } yield col :: queens
  }
  placeQueens(n)
}
```

Exercise

Write a function

```
def isSafe(col: Int, queens: List[Int]): Boolean
```

which tests if a queen placed in an indicated column `col` is secure amongst the other placed queens.

It is assumed that the new queen is placed in the next availabale row after the other placed queens (in other words: in row `queens.length`).

List(0, 3, 1) --> List((2, 0), (1, 3), (0, 1))

Maps

Map

Another fundamental collection type is the *map*.

A map of type `Map[Key, Value]` is a data structure that associates keys of type `Key` with values of type `Value`.

Examples:

```
val romanNumerals = Map(”I” -> 1, ”V” -> 5, ”X” -> 10)
val capitalOfCountry = Map(”US” -> ”Washington”, ”Switzerland” -> ”Bern”)
```

Maps are Iterables

Class `Map[Key, Value]` extends the collection type `Iterable[(Key, Value)]`.

Therefore, maps support the same collection operations as other iterables do. Example:

```
val countryOfCapital = capitalOfCountry map {
  case(x, y) => (y, x)
}                     // Map(”Washington” -> ”US”, ”Bern” -> ”Switzerland”)
```

Note that maps extend iterables of key/value *pairs*.

In fact, the syntax `key -> value` is just an alternative way to write the pair `(key, value)`.

Maps are Functions

Class `Map[Key, Value]` also extends the function type `Key => Value`, so maps can be used everywhere functions can.

In particular, maps can be applied to key arguments:

```
capitalOfCountry(”US”)         // ”Washington”
```

Querying Map

Applying a map to a non-existing key gives an error:

```
capitalOfCountry("Andorra")
  // java.util.NoSuchElementException: key not found: Andorra
```

To query a map without knowing beforehand whether it contains a given key, you can use the `get` operation:

```
capitalOfCountry get "US"      // Some("Washington")
capitalOfCountry get "Andorra" // None
```

The result of a `get` operation is an `Option` value.

The `Option` Type

The `Option` type is defined as:

```scala
trait Option[+A]
case class Some[+A](value: A) extends Option[A]
object None extends Option[Nothing]
```

The expression `map get key` returns

- `None` if `map` does not contain the given `key`,
- `Some(x)` if `map` associates the given `key` with the value `x`.

Decomposing Option

Since options are defined as case classes, they can be decomposed using pattern matching:

```
def showCapital(country: String) = capitalOfCountry.get(country) match {
  case Some(capital) => capital
  case None => "missing data"
}

showCapital("US")       // "Washington"
showCapital("Andorra") // "missing data"
```

Options also support quite a few operations of the other collections.

I invite you to try them out!

Sorted and GroupBy

Two useful operation of SQL queries in addition to for-expressions are `groupBy` and `orderBy`.

`orderBy` on a collection can be expressed by `sortWith` and `sorted`.

```
val fruit = List("apple", "pear", "orange", "pineapple")
fruit sortWith (_.length < _.length)  // List("pear", "apple", "orange", "pineapple")
fruit.sorted      // List("apple", "orange", "pear", "pineapple")
```

`groupBy` is available on Scala collections. It partitions a collection into a map of collections according to a *discriminator function* `f`.

Example:

```
fruit groupBy (_.head)  //> Map(p -> List(pear, pineapple),
                        //|     a -> List(apple),
                        //|     o -> List(orange))
```

Map Example

A polynomial can be seen as a map from exponents to coefficients.

For instance, $x^3 - 2x + 5$ can be represented with the map.

```
Map(0 -> 5, 1 -> -2, 3 -> 1)
```

Based on this observation, let's design a class `Polynom` that represents polynomials as maps.

Default Values

So far, maps were *partial functions*: Applying a map to a key value in `map(key)` could lead to an exception, if the key was not stored in the map.

There is an operation `withDefaultValue` that turns a map into a total function:

```
val cap1 = capitalOfCountry withDefaultValue "<unknown>"
cap1("Andorra")              // "<unknown>"
```

Variable Length Argument Lists

It's quite inconvenient to have to write

```
Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))
```

Can one do without the `Map(...)`?

Problem: The number of `key -> value` pairs passed to `Map` can vary.

Variable Length Argument Lists

It's quite inconvenient to have to write

```
Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))
```

Can one do without the `Map(...)`?

Problem: The number of `key -> value` pairs passed to `Map` can vary.

We can accommodate this pattern using a *repeated parameter*:

```
def Polynom(bindings: (Int, Double)*) =
  new Polynom(bindings.toMap withDefaultValue 0)

Polynom(1 -> 2.0, 3 -> 4.0, 5 -> 6.2)
```

Inside the `Polynom` function, `bindings` is seen as a `Seq[(Int, Double)]`.

Final Implementation of Polynom

```
class Poly(terms0: Map[Int, Double]) {
  def this(bindings: (Int, Double)*) = this(bindings.toMap)
  val terms = terms0 withDefaultValue 0.0
  def + (other: Poly) = new Poly(terms ++ (other.terms map adjust))
  def adjust(term: (Int, Double)): (Int, Double) = {
    val (exp, coeff) = term
    exp -> (coeff + terms(exp))
  }

override def toString =
  (for ((exp, coeff) <- terms.toList.sorted.reverse)
    yield coeff+"x^"+exp) mkString " + "
}
```

Exercise

The + operation on `Poly` used map concatenation with `++`. Design another version of + in terms of `foldLeft`:

```
def + (other: Poly) =
  new Poly((other.terms foldLeft ???)(addTerm)

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  ???
```

Which of the two versions do you believe is more efficient?

```
  O        The version using ++
  O        The version using foldLeft
```

Exercise

The + operation on `Poly` used map concatenation with `++`. Design another version of + in terms of `foldLeft`:

```
def + (other: Poly) =
  new Poly((other.terms foldLeft ???)(addTerm)

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  ???
```

Which of the two versions do you believe is more efficient?

```
  O        The version using ++
  O        The version using foldLeft
```

Putting the Pieces Together

Task

Phone keys have mnemonics assigned to them.

```
val mnemonics = Map(
      '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
      '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")
```

Assume you are given a dictionary `words` as a list of words.

Design a method `translate` such that

```
translate(phoneNumber)
```

produces all phrases of words that can serve as mnemonics for the phone number.

Example: The phone number "7225247386" should have the mnemonic `Scala is fun` as one element of the set of solution phrases.

Background

This example was taken from:

> *Lutz Prechelt: An Empirical Comparison of Seven Programming Languages. IEEE Computer 33(10): 23-29 (2000)*

Tested with Tcl, Python, Perl, Rexx, Java, C++, C.

Code size medians:

- 100 loc for scripting languages
- 200-300 loc for the others

The Future?

Scala's immutable collections are:

- *easy to use*: few steps to do the job.
- *concise*: one word replaces a whole loop.
- *safe*: type checker is really good at catching errors.
- *fast*: collection ops are tuned, can be parallelized.
- *universal*: one vocabulary to work on all kinds of collections.

This makes them a very attractive tool for software development