v1.1
20140402

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO

a.a 2013/2014

Lecturer: Alessandro Ricci

[module 1.5]

INTEGRATING OOP AND FP

SUMMARY

- Functional Programming and OOP
  - Java 8 - project Lambda
  - C#
  - JavaScript
  - Scala

A MULTI-PARADIGM APPROACH

- A possible way to deal with these problems is to embrace multi-paradigm programming
  - don't try to capture every aspect with a single paradigm
  - think more about how to *integrate* different programming paradigms
    - within the same language(s)
  - keeping each paradigm as pure as possible

OOP + FUNCTIONAL PROGRAMMING

- How to do it: three main possibilities
  - conceive a multi-paradigm language from scratch featuring elements of both the paradigm
    - example: Oz language
  - start from a FP language and extend it with OOP features
    - examples: Common Lisp, OCAML, OOHaskell...
  - start from an OOP language and embed some key features of Functional Programming
    - Scala, Java 8, JavaScript, C#

CASE STUDIES

- Java 8 - Project Lambda
- C# - delegates
- JavaScript functions
- Scala

JAVA 8 - PROJECT “LAMBDA”

- Bringing some features of functional programming inside Java
  - JSR 335 “Project Lambda”
  - going to be included in Java 8 - JSR 337
    - 2013/5 Public Review
    - 2013/6 Proposed Final Draft
    - 2013/8 Final Release

MOTIVATIONS

- Some objects encode nothing more than a function
  - typical case: *callback interfaces*

```
public interface ActionListener {
  void actionPerformed(ActionEvent ev);
 }
```

- Typical solution: anonymous classes

```
button.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent ev){
    ui.dazzle(ev.getModifiers);
  };})
```

- Many libraries rely on this pattern

THE PROBLEM

- Increasing relevance of callbacks and other functional-style idioms => it is important that *modeling code as data in Java be as lightweight as possible*
- Problem of anonymous inner classes:
  - bulky syntax
  - confusion surrounding the meaning of names and this
  - inflexible class loading and instance-creation semantics
  - inability to capture non-final local var
  - inability to abstract over control flow
- Project Lambda
  - focussed in particular on the two first points
  - *find a solution without altering Java type system, reusing as much as possible existing code*

FUNCTIONAL INTERFACES

- Idea
  - functions are represented by *functional interfaces*
    - interfaces with only one method
  - e.g. Runnable, Comparable

```
interface Runnable { void run();}
interface MyFunc<T1,T2> { T2 apply(T1 arg); }
```

- Nothing special needs to be done to declare an interface as functional
  - detected by compiler, *given its structure*

LAMBDA EXPRESSIONS

- *Lambda expressions* are anonymous methods, declared like lambda abstractions
  - representing either function literals or closures
  - whose type is defined by functional interfaces
  - converted into instances of functional interfaces
- Syntax: `(argument list) -> body`
  - body could be a single expression or a statement block
- Examples

```
(int x, int y) -> x + y
() -> 42
(String s) -> { System.out.println(s); }
```

LAMBDA EXPRESSIONS

- Lambda expressions appearing in statements:

```
FileFilter javaFiles =
                (File f) -> f.getName().endsWith(".java");

String user =
    doPrivileged(() -> System.getProperty("user.name"));

new Thread(() -> {
      connectToService();
      sendNotification();
}).start()

List<String> list =
                   Arrays.asList("looong","short","tiny");
Collections.sort(list,
 (String s1, String s2) -> s1.length() - s2.length());
```

TARGET TYPING

- The type of the lambda expression (i.e. the functional interface) is inferred by the compiler given the context where the expression is used.
  - e.g.

```
ActionListener l =
(ActionEvent ev) -> ui.dazzle(ev.getModifiers());
```

- So the same lambda expression can have different types in different context:
  - `Callable<String> c = () -> "done";`
    - here the expression is an instance of `Callable`
  - `PrivilegedAction<String> a = () -> "done";`
    - here the same expression is an instance of `PrivilegedAction`

TYPE INFERENCE

- Since a functional interface target type *already knows* what types the lambda expression's formal parameters should have, it is ofter unnecessary to repeat them, since they are *inferred* by the compiler
- Example:

```
Comparator<String> c =
    (s1, s2) -> s1.compareToIgnoreCase(s2);
```

METHOD AND CONSTRUCTOR REFERENCES

- Existing methods and constructors can be used as lambda expressions.
- Example:

```
class Person {
  private final String name;
  private final int age;
  public static int compareByAge(Person a, Person b){...}
  public static int compareByName(Person a, Person b){...}
}

Person[] people = ...
Arrays.sort(people, Person::compareByAge);
```

STATIC METHODS REFERENCE

- Referring static methods:

```
interface Block<T> { void run(T arg); }
...
Block<Integer> b1 = System::exit();
Block<String[]> b2 = Arrays::sort;
Block<String> b3 = MyProgram::main;
Runnable r = MyProgram::main
```

INSTANCE METHODS/CONSTRUCTORS

- Besides static methods, one can use also:
  - an instance method of a particular object

```
class ComparisonProvider {
  public int compareByAge(Person a, Person b){...}
  public int compareByName(Person a, Person b){...}
}
...
Arrays::sort(people,comparisonProvider::compareByName);
```

  - an instance method of an arbitrary object of a particular type

```
Arrays.sort(names, String::compareToIgnoreCase);
```

  - constructors (using the new keyword):

```
SocketImplFactory factory = MySocketImpl::new;
```

LAMBDA EXPRESSIONS AS CLOSURES

- Lambda expressions can be closures, accessing variables outside its immediate lexical scope:

```
...
String msg = "hello";
JButton button = ...
...
button.addActionListener(
  (ActionEvent ev) -> { button.setEnable(false); });
```

HIGH-ORDER FUNCTIONS

- Passing lambda expressions to methods

```
public interface Command<T> { void apply(T t); }
...
public void doWithContact(String fileName, Command<Contact> block){
 try {
  String contactStr = FileUtils.readFileToString(new File(fileName));
  Contact contact = AContactParser.parse(contactStr);
  block.apply(contact);
 } catch (IOException ex){ ...
 } catch (ParseException ex){ ...}
}
```

```
// usage
Command<Contact> saveCmd = c -> ContactDao.save(c));
Command<Contact> printCmd = c -> System.out.println(c.toString()));
....
doWithContact("customerXYZ.vcf", printCmd);
doWithContact("customerXYZ.vcf", saveCmd);
```

COLLECTIONS

- Collections lib has been extended to allow for high-order functions and function chaining
  - http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html
  - http://www.oracle.com/technetwork/articles/java/architect-lambdas-part1-2080972.html
  - http://www.oracle.com/technetwork/articles/java/architect-lambdas-part2-2081439.html
- Main elements:
  - `Stream<T>` interface
  - method `stream()` in Collection class

STREAM INTERFACE

```
public interface Stream<T> {
    void forEach(Block<? super T> block);
    Stream<T> filter(Predicate<? super T> predicate);
    <R> Stream<R> map(Mapper<? extends R, ? super T> mapper);
    T reduce(T base, BinaryOperator<T> op);
    <A extends Destination<? super T>> A into(A target);
    Stream<T> sorted(Comparator<? super T> comparator);
    <U> U fold(Factory<U> baseFactory,
               Combiner<U, U, T> reducer,
               BinaryOperator<U> combiner);
    boolean anyMatch(Predicate<? super T> predicate);
    boolean allMatch(Predicate<? super T> predicate);
    boolean noneMatch(Predicate<? super T> predicate);
    Optional<T> findFirst();
    Optional<T> findAny();
    ....
}
```

COMMON FUNCTIONAL INTERFACES

- The package `java.util.function` contains a "starter set" of functional interfaces:
  - Predicate
    - a property of the object passed as argument
  - Block
    - an action to be performed with the object passed as argument
  - Function
    - transform a T to a U
  - Supplier
    - provide an instance of a T (such as a factory)
  - UnaryOperator
    - a unary operator from T -> T
  - BinaryOperator
    - a binary operator from (T, T) -> T

EXAMPLES

- forEach

```
List<> list = Arrays.asList("hello","how","are","you?");
list.stream().forEach(s -> { System.out.println(s); });
```

- filter

```
list.stream()
    .filter(s -> s.length() < 4)
    .forEach(s -> { System.out.println(s); });
```

- map

```
list.stream()
    .filter(s -> s.length() < 4)
    .map(s -> s.length())
    .forEach(v -> { System.out.println(v); })
```

EXAMPLES

- reduce

```
List<> list = Arrays.asList(new Integer[]{1,2,3,4});
int sum = list.stream().reduce((x,y) -> x + y);
```

- into

```
List<Integer> newList = list.stream()
                        .filter(s -> s.length() < 4)
                        .map(s -> s.length())
                        .into(new ArrayList<>());
```

C#

- Statically and strongly typed object-oriented (class-based) extension of C-like languages
  - based on the CLR .NET virtual machine
  - one of the programming languages designed for the Common Language Infrastructure (CLI)
- Developed by Microsoft within its .NET initiative and later approved as a standard by Ecma and ISO
  - developer team lead by A. Hejlsberg
    - Turbo Pascal, Delphi, TypeScript
- Multi-paradigm
  - imperative, declarative, functional, generic, object-oriented, concurrent
- The most recent version is C# 5.0 (2012)
- Strong similarities with Java

C# TASTE

```
using System;
namespace HelloCounter
{
    class Counter
    {
        private int Count {get; };
        public Counter(int c) { Count = c; }
        public void Inc() { Count++; }
    }
    class Hello
    {
        static void Main()
        {
            Counter counter = new Counter(10);
            counter.Inc()
            Console.WriteLine(counter.get());
        }
    }
}
```

FP WITH DELAGATES

- Lambda expressions can be implemented as *delegates*
  - a delegate is an object that represents a method and, optionally, the *this* associated to the object
    - when the delegate is invoked, the corresponding method is invoked
    - a delegate can be passed to methods, references can be stored in structures or classes
  - used to implement handlers, callbacks, event management, etc

DELEGATE EXAMPLE

```
delegate void MyDelegate(int i);

class Program
{
   public static void Main()
   {
      TakesADelegate(new MyDelegate(DelegFunction));
   }
   public static void TakesADelegate(MyDelegate SomeFunction)
   {
      SomeFunction(21);
   }
   public static void DelegFunction(int i)
   {
      System.Console.WriteLine(
                        "Called by delegate with number: {0}.", i);
   }
}
```

LAMBDA EXPRESSIONS

- A lambda expression is an anonymous function that can be used to create delegates
- Example:

```
delegate int del(int i);

static void Main(string[] args)
{
  del myDelegate = x => x * x;
  int j = myDelegate(5); // j = 25
}
```

EXPRESSION LAMBDAS

- The syntax of a expression lambda is analogous to lambda abstractions:

```
(input parameters ) => expression
```

- Example:

```
(x, y) => x == y
```

- Parentheses are optional if only one parameter
- Zero parameters case: `() => SomeMethod()`
- Type inference supported - types mandatory when type inference can be not enough:

```
(int x, string s) => s.Length > x
```

STATEMENT LAMBDAS

- Like expression lambda, but with a block with statement(s) as body:

```
(input parameters ) => { statement; ...}
```

```
delegate void TestDelegate(string s);
...
TestDelegate myDel = n => {
 string s = n + " World";
 Console.WriteLine(s);
};

myDel("hello");
```

IMPLEMENTING CLOSURES

- Statement/expression lambda can refer to outer variables, implementing closures:

```
delegate bool D();
delegate bool D2(int i);

class Test
{
  D del;
  D2 del2;
  public void TestMethod(int input)
  {
    int j = 0;
    del = () =>
            { j = 10;  return j > input; };
    del2 = (x) => {return x == j; };
    Console.WriteLine("j = {0}", j);
    bool boolResult = del();
    // Output: j = 10 b = True
     Console.WriteLine("j = {0}. b = {1}", j, boolResult);
  }
  static void Main(){...}
}
```

```
static void Main()
{
  Test test = new Test();
  test.TestMethod(5);

  bool result = test.del2(10);
  // Output: True
  Console.WriteLine(result);
  Console.ReadKey();
}
```

USING HIGH-ORDER FUNCTIONS WITH LINQ

- Language-Integrated Query (LINQ)
  - set of features that provide *query capabilities* to the language syntax of C# and Visual Basic.
  - introduces standard patterns for querying and updating data
  - the technology can be extended to support potentially any kind of data store
- Strong synergy with lambda expressions

EXAMPLE OF A SIMPLE QUERY

```
class SimpleLambda
{
    static void Main()
    {

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80",
                          highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

EXAMPLE OF COMPLEX QUERY

- Example of query *without* LINQ & lambda expressions:

```
IEnumerable<string> GetExpensiveProducts() {
   // Create new list for storing the results
   List<string> filteredInfo = new List<string>();
   // Iterate over all products from the data source
   foreach(Product product in Products) {
      if (product.UnitPrice > 75.0M) {
         // Add expensive product to a list of results
         filteredInfo.Add(String.Format("{0} - ${1}",
            product.ProductName, product.UnitPrice));
      }
   }
   return filteredInfo;
}
```

EXAMPLE OF QUERY

- Example of data retrieval with LINQ & lambda expressions:

```
IEnumerable<string> GetExpensiveProducts() {
  return Products
          .Where(product => product.UnitPrice > 75.0M)
          .Select(product => String.Format("{0} - ${1}",
                                product.ProductName,
                                product.UnitPrice))
          .ToList();
}
```

JAVASCRIPT

- Dynamically weakly-typed object-based scripting languages, supporting also functions as first-class entities
  - syntax influenced by the language C
  - key design principles taken from the Self and Scheme
- Formalized in the ECMAScript language standard
  - part of a web browser (client-side JavaScript)

JAVASCRIPT & APPS

- Originally implemented as scripting language part of web browsers, nowadays JavaScript is the reference language for developing web apps (client + servers), integrated with HTML5
  - scripts could interact with the user, control the browser, communicate asynchronously, and alter the document content that was displayed.
- Fast JavaScript VMs available (e.g. V8)
- Client-side and server-side libraries, frameworks getting more and more popular
  - e.g. jQuery (client side), Node.js (server side)

OBJECT-BASED + FUNCTIONS

- object based
  - no classes, only objects with support for prototypes for implementing delegation (instead of inheritance)
  - objects are associative arrays of properties (fields + methods)
    - properties and their values can be added, changed, or deleted at run-time
- Functional
  - functions are first-class, they are objects themselves
    - as such, they have properties and methods, such as .call() and .bind()
  - they can be assigned to variables, passed as arguments, returned by other functions, and manipulated like any other object
  - any reference to a function allows it to be invoked using the () operator
    - nested functions and closures

JAVASCRIPT TASTE (WITHIN A PAGE)

```
<!DOCTYPE html>
<html>
<head>
<script>
  var myCounter = {
    count: 1,
    inc: function() {
           this.count++;
         }
  }
  function callback()
  {
    myCounter.inc()
    alert("Hello World! " + myCounter.count);
  }
</script>
</head>
<body>
<button onclick="callback()">Try it</button>
</body>
</html>
```

FUNCTION AS FIRST-CLASS OBJECTS

```
function forEach(array, action){
  for (var i = 0; i < array.length; i++){
    action(array[i])
  }
}

forEach(["Hello","how","are","you?"], print);
```

ANONYMOUS FUNCTIONS

- Lambda expressions/abstraction/closures can be defined as anonymous functions with the `function` keyword:

```
function sum(numbers){
  var total = 0;
  forEach( numbers, function (num){
    total +=  num;
  });
  return total;
}
...
show(sum([4,13,20]);
```

- Note that
  - function (num){ ... } is a closure (access to total..)
  - function (num){ ... } is not a pure function

HIGH-ORDER FUNCTIONS: MAP

- A non-pure version of the map functions in JavaScript:

```
function map(func, array) {
  var result = [];
  forEach(array, function (element) {
    result.push(func(element));
  });
  return result;
}
...
show(map(Math.round, [0.01,2, 3.03, Math.PI]));
```

HIGH-ORDER FUNCTIONS: MAP

- A non-pure version of the reduce functions in JavaScript:

```
function reduce(combine, base, array){
  forEach(array, function (element) {
    base = combine(base, element);
  });
  return base;
}

function add(a, b) { return a + b; }

/* usage */
function sum(numbers) {
  return reduce(add, 0, numbers);
}
```

INTEGRATING OOP & FP IN SCALA

SCALA

- “Scalable Language”
  - statically strongly typed object-oriented
  - designed to integrate functional programming along with Java's OOP model
    - functional programming features inspired by Scheme, Standard ML and Haskell
    - interoperability with Java and running on top of the Java virtual machine (JVM)
- History
  - design started in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) by Martin Odersky
  - released early 2004, second version released in March 2006
  - In 2011/01 the Scala team won a 5 year research grant of over €2.3 million from the European Research Council.
  - In 2011/05 Odersky and collaborators launched Typesafe Inc
    - a company to provide commercial support, training, and services for Scala

A TASTE OF SCALA

```
trait Incrementable {
  def inc()
  def inc(delta: Int){
    for (i <- 1 until delta){
      inc()
    }
  }
}
```

```
class Counter extends Incrementable {
  private var count = 0

  def inc() {
    count+=1
  }
  def getValue(): Int = count
}
```

```
object Test {
  def main(args: Array[String]) {
    val c = new Counter()
    c inc()
    println(">> "+(c.getValue))
    c inc(delta = 10)
    println(">> "+(c.getValue))
  }
}
```

DISTINGUISHED FEATURES COMPARED TO JAVA - SYNTAX

- Omitting semicolons
- Omitting () when no params
- Using {} instead of () when 1 param
- Method invocation without "."
- Value types are capitalized: Int, Double, Boolean instead of int, double, boolean (but they are objects...)
- Parameter and return types follow, rather than precede
- Array elem access with parenthesis ()
- Unmodified variables preceded by val
- Short-forms for class definition / constructors
- Return can be omitted in function
- instead of Java's import foo.*;, Scala uses import foo._.
- ...

DISTINGUISHED FEATURES COMPARED TO JAVA - CONCEPTS

- On the OO side
  - uniform type system - everything is an object - also integers (Int), booleans (Bool), etc
  - traits
  - operator overloading
  - optional and named parameters
  - singleton objects
  - no static elements
  - object constructions
  - ...

DISTINGUISHED FEATURES COMPARED TO JAVA - CONCEPTS

- In general
  - functions as first-class entities, mapped onto objects
  - flexible control abstractions
  - type inference
  - pattern matching
  - support for XML
  - actors as reference concurrent model
- ...

DEFINING FUNCTIONS

- As methods of objects

```
import scala.io.Source;
object LongLines {
  def processFile(filename: String, width: Int){
    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(filename, width, line)
  }
  private def processLine(filename: String, width: Int, line: String){
    if (line.length > width){
      println(filename +": "+line.trim)
    }
  }
}
object FindLongLines {
  def main(args: Array[String]){
    val width = args(0).toInt
    for (arg <- args.drop(1))
      LongLines.processFile(arg, width)
  }
}
```

LOCAL FUNCTIONS

- Functions defined in the scope of methods

```
import scala.io.Source;
object LongLines {
  def processFile(filename: String, width: Int){
    def processLine(filename: String, width: Int, line: String){
      if (line.length > width){
        println(filename +": "+line.trim)
      }
    }
    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(filename, width, line)
  }
}
```

FIRST-CLASS FUNCTIONS

- Functions as unnamed literals, to be passed as values
  - a **function literal** is compiled into a class that when is instantiated at runtime is a ***functional value***
    - i.e. instance of some class that extends one of the traits FunctionN available in package `scala`
      - `Function0` is for no params, `Function1` with one param, etc
      - these traits have an `apply` method
    - so function literals exist in source code (like classes), function values are objects at runtime..
- Syntax: `(ParamList) => Body`
  `(x: Int) => x + 1`

FIRST-CLASS FUNCTIONS

- Function values are objects - so it can be assigned as objects:

```
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function1>
scala> increase(10)
res: Int = 11
```

- The body can be a block of statements:

```
increase = (x: Int) => {
  println("how")
  println("are")
  println("you")
  x + 1
}
```

SHORT FORMS OF FUNCTION LITERALS

- Exploiting type inference to avoid explicit type specification:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumber.filter((x) => x > 0)
res: List[Int] = List(5,10)
```

- So the type of function literals depend on the context (like in Java 8 case)
  - *target typing*

SHORT FORMS OF FUNCTION LITERALS

- Avoid using parenthesis when not needed (e.g. 1 param only)

```
scala> someNumber.filter( x => x > 0)
```

- Underscores can be used as placeholders of parameters

```
scala> someNumber.filter( _ > 0)
```

PARTIALLY APPLIED FUNCTIONS

- *Underscores* can be used to avoid specifying all the parameters when invoking a function
  - as a result, a function is returned (=> curried functions)

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int

scala> sum(1,2,3)
res: Int = 6

scala> val b = sum(1,_:Int,3)
b: (Int) => Int = <function1>

scala> b(5)
res: Int = 9
```

CLOSURES

- Function literals with *free variables*:

```
scala> var more = 1
more: Int = 1

scala> var addMore = (x: Int) => x +
more
addMore: (Int) => <function1>

scala> addMore(10)
res: Int = 11
```

CLOSURES

- Closures in Scala *capture variables themselves*, not the values to which variables refer
  - this happens instead to Java 8, for instance
- So in the example if we change the value of the var more, also the closure sees it

```
scala> more = 2
scala> addMore(10)
res: Int = 12
```

- This is true also in the opposite direction, i.e. changes made by a closure to a free variable are visible also outside the closure

```
scala> var = changeMore(x: Int) => { more+=x }
scala> changeMore(100)
scala> print(more)
more: Int = 102
```

TAIL RECURSION

- Tail recursion makes it possible to have recursive functions that are as efficient (in time and space) as the iterative/imperative version
- Example:

```
def approximate(guess: Double): Double =
   if (isGoodEnough(guess)) guess
   else approximate(improve(guess)
```

- Imperative version:

```
def approximate(initialGuess: Double): Double = {
  var guess = initialGuess
  while (!isGoodEnough(guess))
    guess = improve(guess)
  guess
}
```

TAIL RECURSION

- The compiler recognizes that `approximate` is tail recursive and replace the recursive function call simply with a *jump* back to the beginning of the function, after updating the function parameters with new values

TRACING TAIL-RECURSIVE FUNC

- A simple test to show that a tail-recursive function doesn't build a new stack frame (activation record) for each call
- Consider this non-tail recursive function

```
def boom(x: Int): Int =
  if (x == 0) throw new Exception("boom!")
  else boom(x - 1) + 1
```

- Running it:

```
scala> boom(3)

java.lang.Exception: boom!
    at .boom(<console>:5)
    at .boom(<console>:6)
    at .boom(<console>:6)
    at .boom(<console>:6)
```

The stack trace shows the the presence of 4 stack frames for boom

TRACING TAIL-RECURSIVE FUNC

- Consider then the following *tail-recursive* version:

```
def bang(x: Int): Int =
  if (x == 0) throw new Exception("bang!")
  else bang(x - 1)
```

- Running it:

```
scala> bang(3)
java.lang.Exception: bang!
   at .bang(<console>:5)
   at .<init>(<console>:6)
```

The stack trace shows the the presence of only 1 stack frame for the func call

HIGH-ORDER FUNCTIONS

- High-Order functions are an important conceptual tool for factorizing and reusing code
- Example
  - starting class

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles
    def filesEnding(query: String) =
    for (file <- filesHere; if (file.getName.endsWith(query)))
      yield file
}
```

  - adding incrementally features:

```
def filesContaining(query: String) =
  for (file <- filesHere; if (file.getName.contains(query)))
    yield file

def filesRegex(query: String) =
  for (file <- filesHere; if (file.getName.matches(query)))
    yield file
```

HIGH-ORDER FUNCTIONS

- Factorizing the code with HO functions:

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles

  def filesMatching(query: String, matcher: (String, String) => Boolean) =
        for (file <- filesHere; if (matcher(file.getName,query))
          yield file

  def filesEnding(query: String) = filesMatching(_.endsWith(query))
  def filesContaining(query: String) = filesMatching(_.contains(query))
  def filesRegex(query: String) = filesMatching(_.matches(query))
}
```

  - where function values: `_.endsWith(query)` is a short form for

```
(name: String, query: String) =>  name.endsWith(query)
```

ANOTHER EXAMPLE

- Executing some function/procedure, measuring its duration

```
def measure[T](func => T): T = {
  val start = System.nanoTime()
  val result = func
  val elapsed = System.nanoTime() - start
  println("duration: %s ns".format(elapsed))
  result
}

def myCallback = {
  Thread.sleep(1000)
  "done"
}

val result = measure(myCallback);
> duration: 1002500000 ns
```

COMPOSING CALLBACKS

```
def doWithContact(fileName: String, handle: Contact => Unit): Unit = {
  try {
    val contactStr = io.Source.fromFile(fileName).mkString
    val contact = AContactParser.parse(contactStr)
    handle(contact)
  } catch { ... }}

val storeCallback = (c:Contact) => ContactDao.save(c)
val sendCallback = (c:Contact) => {
  val msgBody = AConverter.convert(c)
  RESTservice.send(msgBody)
}
val combineCallback = (c:Contact) => {
  storeCallback(c)
  sendCallback(c)
}
..
doWithContact(“customerX.vcf”, storeCallback)
doWithContact(“customerY.vcf”, sendCallback)
doWithContact(“customerZ.vcf”, combinedCallback)
```

CURRYING

- Scala supports the definition and invocation of curried functions
  - by separating the list of arguments
- Example - curried function:

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum (x: Int)(y: Int)Int
scala> curriedSum(1)(2)
res: Int = 3
```

- Non curried version:

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (x: Int, y: Int)Int
scala> plainOldSum(1,2)
res: Int = 3
```

CURRYING

- Applying curried functions:

```
scala> val add2 = curriedSum(2)(_)
add2: Int => Int = <function1>

scala> add2(5)
res: Int = 7
```

  - alternative

```
scala> def add2 = curriedSum(2)(_)
add2: Int => Int

scala> add2(5)
res: Int = 7
```

WRITING NEW CONTROL STRUCTURES

- High-order functions are often used in Scala to make new control structures
  - even though the syntax of the language is fixed
- Example
  - twice control structure, which repeats an operation two times and return a result:

```
scala> def twice(op: Double => Double, x: Double) = op(op(x))
scala> twice(_ + 1, 5)
res: Double = 7
```

IMPLEMENTING CONTROL PATTERNS

- Anytime that one finds a control pattern repeated in multiple parts of the code, the implementation of a new control structure could be useful to simplify and factorize the code
- Example of a coding pattern: *loan pattern*
  - "*open a resource, operate on it and the CLOSE the resource"*

```
def withPrintWriter(file: File, op: PrintWriter => Unit)  {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close(file)
  }
}
```

IMPLEMENTING CONTROL PATTERNS

- To make client code look a bit more like a built-in control structure, curly braces can be used instead of parenthesis to surround the argument list
  - this is possible in Scala for any method in which we are passing exactly one argument

```
val file = new File("date.txt")

withPrintWriter(file) {
  writer => writer.println(new java.util.Date)
}
```

COLLECTIONS & HIGH-ORDER FUNCTIONS

- Collections have directly methods functioning as high-order functions.
- Example: List
  - mapping over lists: `map, flatMap, foreach`
  - filtering lists: `filter, partition, find, takeWhile, dropWhile, span`
  - predicates over lists: `forall, exits`
  - folding lists: `/:, \:`
  - sorting lists: `sortWith`
  - ...

MAP EXAMPLES

```
scala> List(1,2,3) map (_ + 1)
res: List[Int] = List(2,3,4)

scala> val words = List("the","quick","brown","fox")
words: List[String] = List(the, quick, brown, fox)

scala> words map (_.length)
res: List[Int] = List(3, 5, 5, 3)

scala> words map (_.toList.reverse.mkString)
res: List[String] = List(“eht", “kciuq", “nworb”,
                         “xof")
```

FOREACH & FILTER EXAMPLES

```
scala> var sum = 0
scala> List(1,2,3,4,5,6) foreach (sum += _)
scala> sum
res: Int = 15
```

```
scala> List(1,2,3,4,5) filter (_ % 2 == 0)
res: List[Int] = List(2,4)
```

FORALL AND EXISTS EXAMPLES

- `forall` returns true if all the elements of a list satisfy some predicate p

```
scala> List(1,2,3,4,5) forall(_ > 0)
res: Boolean = True
```

- `exists` returns true if one element of a list satisfies the predicate

```
scala> List(1,2,3,4,5) exists(_ < 0)
res: Boolean = False
```

FOLDING EXAMPLE

- folding left operator /:
  - shortest-form
    - `(z /: List(a,b,c))(op)`
      is equal to `op(op(op(z,a),b),c)`
    - example:

```
scala> (0 /: List(1,2,3,4,5)) (_ + _)
res: Int = 15
```

  - equivalent non-short form:

```
List(1,2,3,4,5)./:(0)((x: Int, y: Int) => x + y)
```

    that is: /: is a method of lists

- Another example (reverse)

```
(List[Int]() /: List(1,2,3,4,5)) ((x,y) => (y :: x))
```

FOLDING EXAMPLE

- folding right operator :\
  - shortest-form:

    `(List(a,b,c) :\ z)(op)`

    which is equal to `op(a,op(b,op(c,z)))`

  - example:

```
scala> (List(1,2,3,4,5) :\ List[Int]())
                        ((x,y) => (y ::: List(x))
res: List[Int] = List(5, 4, 3, 2, 1)
```

SORTING

```
scala> List(1,-3,4,2) sortWith (_ < _)
res: List[Int] = List(-3,1,2,4)
```

FUNCTION CHAINING - EXAMPLE

```
class Photo(name:String, sizeKb:Int, ratings:List[Int])
val p1 = Photo(“photo1.jpg”,344,List(9,6,7,3))
val p2 = ...
val photos = List(p1,p2,p3,...)
..
val names = photos.map(p => p.name)
val bigPhotos = photos.filter(p => p.sizeKb > 20000).map(p => p.name)
..

// returns the names of all photos whose average rating is higher
// than 6, sorted by the total amount of ratings given
val avg = (l:List[Int]) => l.sum / l.size
val minAvgRating = 6
vat result = photos.filter(p => avg(p.ratings) > minAvgRating)
                   .sortBy(p => p.ratings.size)
                   .map(p => p.name)
```

LIST COMPREHENSION: FOR

- Every `for` expression in Scala can be translated into an expression using `map`, `flatMap` and `withFilter`
- For expressions are in the form:

  `for ( seq ) yield (expr)`

  seq is a sequence of generators, definitions and filters, separated by semicolons
  - generator: `pat <- expr`
    - pattern pat is matched one-by-one by against all elements of the collection expr
  - definition: `pat = expr`
    - binding pat to the value of expr
  - filters: `if expr`
- Analogous to *list comprehension* in FP languages

EXAMPLE

```
class Person(name: String, isMale: Boolean; children: Person*)
val lara = Person("Lara",false)
val bob = Person("Bob",true)
val julie = Person("Julie",false, lara,bob)
val persons = List(lara, bob, julie)

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

scala> for (x <- List(1,2); y <- List("one","two")) yield (x,y)
red: List[(Int,String)] = List((1,one),(1,two),(2,one),(2,two))
```

TRANSLATIONS

- Every for expression translation:
  - `for (x <- expr1) yield expr2`

    can be translated in:

    `expr1.map(x => expr2)`
  - `for (x <- expr1 if expr2) yield expr3`

    can be translated in:

    `expr1 withFilter(x => expr2) map (x => expr3)`
  - `for (x <- expr1) body`

    can be translated in:

    `expr1 foreach (x => body)`

SUMMARY

- Functional Programming and OOP
  - Java 8 - project Lambda
  - C#
  - JavaScript
  - Scala

BIBLIOGRAPHY

- JSR 335 (Lambda Expressions for the Java Programming Language)
  - http://openjdk.java.net/projects/lambda/
  - “State of the Lambda” doc overview
    - http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html
- Java 8 vs. Scala: A Feature Comparison. Urs Peter and Sander van der Berg. InfoQ. http://www.infoq.com/articles/java-8-vs-scala
- Programming in Scala - Odersky, Spoon, Venners
- “Java 8 vs Scala: a Feature Comparison” by Urs Peter and Sender van der Berg, 2012/06, InfoQ