ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Functional programming: an appetizer

Imperative programming

Java and C# are fundamentally based on the imperative paradigm of programming. Fundamental elements:

- State (variables)
- State modification (assignment)
- Iteration (sequencing)

An imperative program is a sequence of state modifications on variables.

```
int power(int x, int n) { // x to the nth
    int result = 1;
    for (int i = n; i > 0; i--) result *= x;
    return result;
}
```

Functional programming

Functional programming is a different paradigm, based on the notion of mathematical function. Fundamental elements:

- Data (values)
- Functions on data (maps)
- Functional forms (function composition, higher-order functions)

A functional program is the application of a function on values, which returns values without side effects.

```
int power(int x, int n) { // x to the nth
    if (n == 0) return 1;
    else return x*power(x, n-1);
}
```

Functional programming languages

Theoretical foundations:

- Lambda calculus (Church, 1932)

Some functional programming languages:

- LISP (McCarthy, 1958)
  - its many dialects: Clojure, Scheme, Racket, ...
- ML (Milner & al., 1973)
- Erlang (Ericsson, 1986)
- Haskell (Peyton Jones & al., 1990)
- F# (Microsoft .NET, ~2000s)

Functional + object-oriented

A few programming languages have tried to combine functional and object-oriented programming models.

- Functional: data-oriented computations, no side-effects, higher-order functions.
- Object-oriented: encapsulation, inheritance, polymorphism.

Two noticeable examples:

- Scala is a Java dialect combining object-oriented and functional.
- C# has incorporated elements of functional programming since version 3 (mostly for the LINQ framework).

Functional elements

We now make a short presentation of some functional features available in Scala and C#.

- Data-oriented computations
- Anonymous functions
- Type inference
- List comprehensions
- Pattern matching
- Immutable collections

This is not meant to be a comprehensive (or even self-contained) presentation.

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Scala: a short overview

Scala

From www.scala-lang.org:

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive.

Scala

Mainly developed by Martin Odersky's group at EPFL.

- First version: 2003
- Latest version: 2.10

- Scala compiles to Java bytecode, and hence it runs on the JVM.
- It is interoperable with Java: Java library can be used in Scala.
- Elements of functional programming with object-oriented encapsulation mechanisms
- Gets rid of some of Java's "legacy" features (e.g., primitive types are objects too in Scala)
- Main selling points: powerful collection library and parallelization

Data-oriented computations

In Scala, there's no distinction between statements and expressions: every expression has a value. The last expression evaluated in a function definition is the one returned.

```
     // function definition
def square(x: Int): Int = {
     x*x;
}
     // attribute definition
var sq: Int = square(2);
     // constant definition
val sq: Int = square(2);
```

Anonymous functions

In Scala, functions are objects. A convenient syntax exists to declare functional expressions which can be passed around as arguments.

```
// anonymous function definition
(x: Int) => x*x*x
// passed as argument
doAll(myList, (x: Int) => x*x*x);
// defined and immediately evaluated
val res = ((x: Int) => x*x*x)(5);
println(res); // print 125=5*5*5
```

Type inference

An extensive type inference system makes type declarations optional in many cases.

```
    // Return type inferred
def square(x: Int) = { x*x; }
    // Variable type inferred
var sq = square(2);
    // Expression type inferred
    (x: Int) => x > 10
```

List comprehensions

List comprehensions define complex expressions over lists (and other collections). They can translate complex function applications concisely and efficiently.

```
// List of integers from 1 to 10
for (x <- 1 to 10) yield x
// List of squares of odd numbers
   between 1 and 100
for (x <- 1 to 100 if x % 2 == 1) yield x*x
// Nested expressions
for (x <- 1 to 10; y <- 1 to x; if x % y == 0)
   yield x+y;
```

Pattern matching

Pattern matching is a flexible construct to define complex conditional expressions. It is typically used on lists or collections.

```
// Sum all elements in list
def Sum(list: List[Int]): Int = list match {
   // if list is empty, the sum is zero
  case Nil => 0;
   // first element (head) followed by tail
  case head :: tail => head + Sum(tail);
}
```

Immutable collections

Scala distinguishes between mutable and immutable objects. All Scala collection classes are immutable by default (mutable variants are still available). Methods working on immutable objects behave as mathematical functions: they return fresh objects with the result, but do not modify the argument objects.

ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Functional programming with C#: a short overview

LINQ

The Language Integrated Query (LINQ) framework extends .NET with native-language support for SQL-like queries and other declarative/functional features.

- First version: 2007 with .NET 3.5

LINQ greatly increases the expressiveness of C#, and makes it possible to introduce significant elements of functional programming in generic C# applications.

Anonymous functions

LINQ offers lambda expressions as a convenient syntax to declare functional expressions which can be passed around as arguments. Lambda expressions can also be assigned to variables of delegate types.

```
// anonymous function definition
(int x) => x*x*x
// passed as argument
doAll(myList, (int x) => x*x*x);
// defined and then evaluated
delegate int Int2Int(int i);
Int2Int fun = (int x) => x*x*x;
      // print 125=5*5*5
Console.WriteLine(fun(5));
```

Type inference

We have already seen C#'s type inference mechanisms in action with variables declared as `var` and with generic actual parameters.

```
public static T copy <T> (T o)
        where T: ICloneable {
                return (T) o.Clone();
}
        // Variable type inferred
        var sq = square(2);
        // Generic type inferred
        var c = copy(new int[] {1, 4, 5});
        // Expression type inferred
        (int x) => x > 10
```

List comprehensions

List comprehensions define complex expressions over lists (and other collections). They are available in LINQ through SQL-like syntax.

```
// IEnumerable<int> of integers from 1 to 10
Enumerable.Range(1, 10)
// Squares of odd numbers between 1 and 100
from x in Enumerable.Range(1, 100)
    where x % 2 == 1 select x*x
// Nested expressions
from x in Enumerable.Range(1, 10)
from y in Enumerable.Range(1, x)
where x % y == 0 select x+y;
```

Pattern matching

C# and LINQ do not offer general support for pattern matching, but head-tail recursion is often implementable through other library features.

```
// Sum all elements in list
def List<int> Sum(List<int> list) {
   // if list is empty, the sum is zero
  if (list.Count == 0)
   return 0;
   // first element (head) followed by tail
  else return list[0] +
   Sum(Enumerable.Skip(list, 1).ToList());
}
```

Immutable collections

C# generic collections are mutable. However, most of the extension methods provided as part of the LINQ framework behave as mathematical functions: they return fresh objects with the result, but do not modify the argument objects.

See the example of `Skip` in the previous slide.

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Example: Quicksort in functional style

Quicksort

To sort an array `a`, the quicksort algorithm work as follows.

1. Pick an element in `a` – called the "pivot".
   (The correctness of quicksort doesn't depend on the pivot choice, but its running time does. We'll just pick the middle element as pivot in our examples).
2. Partition `a` in three segments:
   - Lower part: elements less than the pivot
   - Mid part: elements equal to the pivot
   - Upper part: elements greater than the pivot
3. Recursively sort the lower and upper parts.

Quicksort in Scala

```
  def sort(a: List[Int]): List[Int] = {
   if (a.length <= 1) {
     a;
   } else {
     val pivot = a(a.length / 2);
     val low = a.filter(x => x < pivot);
     val mid = a.filter(x => x == pivot);
     val up = a.filter(x => x > pivot);
     sort(low) ++ mid ++ sort(up);
   }
 }
```

Quicksort in C# with LINQ

```
IEnumerable<int> sort(IEnumerable<int> a) {
 if (a.Count() <= 1) {
   return a;
 } else {
   var pivot = a.ElementAt(a.Count() / 2);
   var low = from x in a where x < pivot select x;
      // Alternative syntax
   var mid = a.Where(x => x == pivot);
   var up = from x in a where x > pivot select x;
   return sort(low).Concat(mid.Concat(sort(up)));
  }
}
```

Complexity

What's the complexity of Quicksort implemented functionally?

- Asymptotically it's the same as in the imperative implementations: partitioning has linear cost, and the number of recursive calls is the same.
- There is, however, less control on how the compiler handles recursion and memory allocation.
  - List comprehensions are normally highly optimized
  - But they cannot always beat custom-made optimizations or in-place implementations

Observation

Standard imperative implementations of Quicksort partition in two parts (less than or equal to, and greater than) rather than three. What happens if you set up the recursion as is now but with this two-fold partitioning?

- Hint: consider the list [3, 2, 9, 1, 7]