Functional programming 1
Where are we today

Peter Sestoft

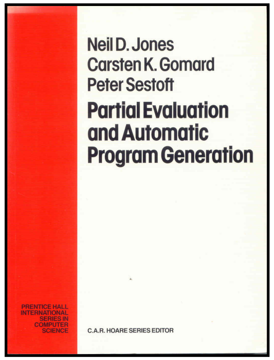
IT University of Copenhagen

Ingeniørforeningen, IDA-IT

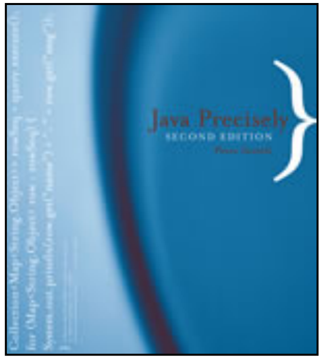
Wednesday 2014-09-24

The speaker

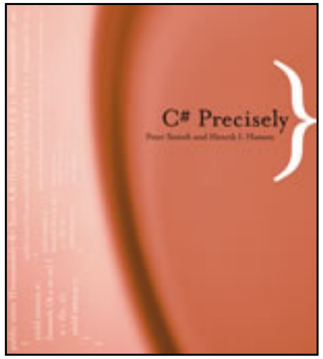
- MSc 1988 computer science and mathematics and PhD 1991, DIKU, Copenhagen University
- Programming languages, compilers, software development, ...
- Open source software:
  - Moscow ML, a functional language, since 1994
  - C5 Generic Collection Library for C#/.NET, since 2006
- Author of some books:



1993

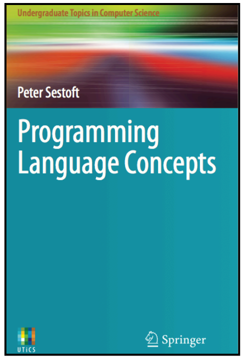


2002, 2005, 2015

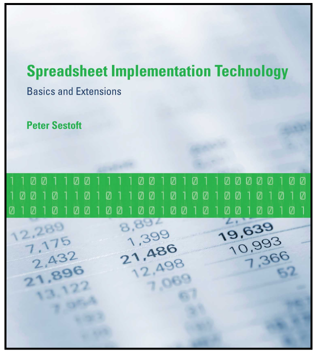


2004, 2012

2007

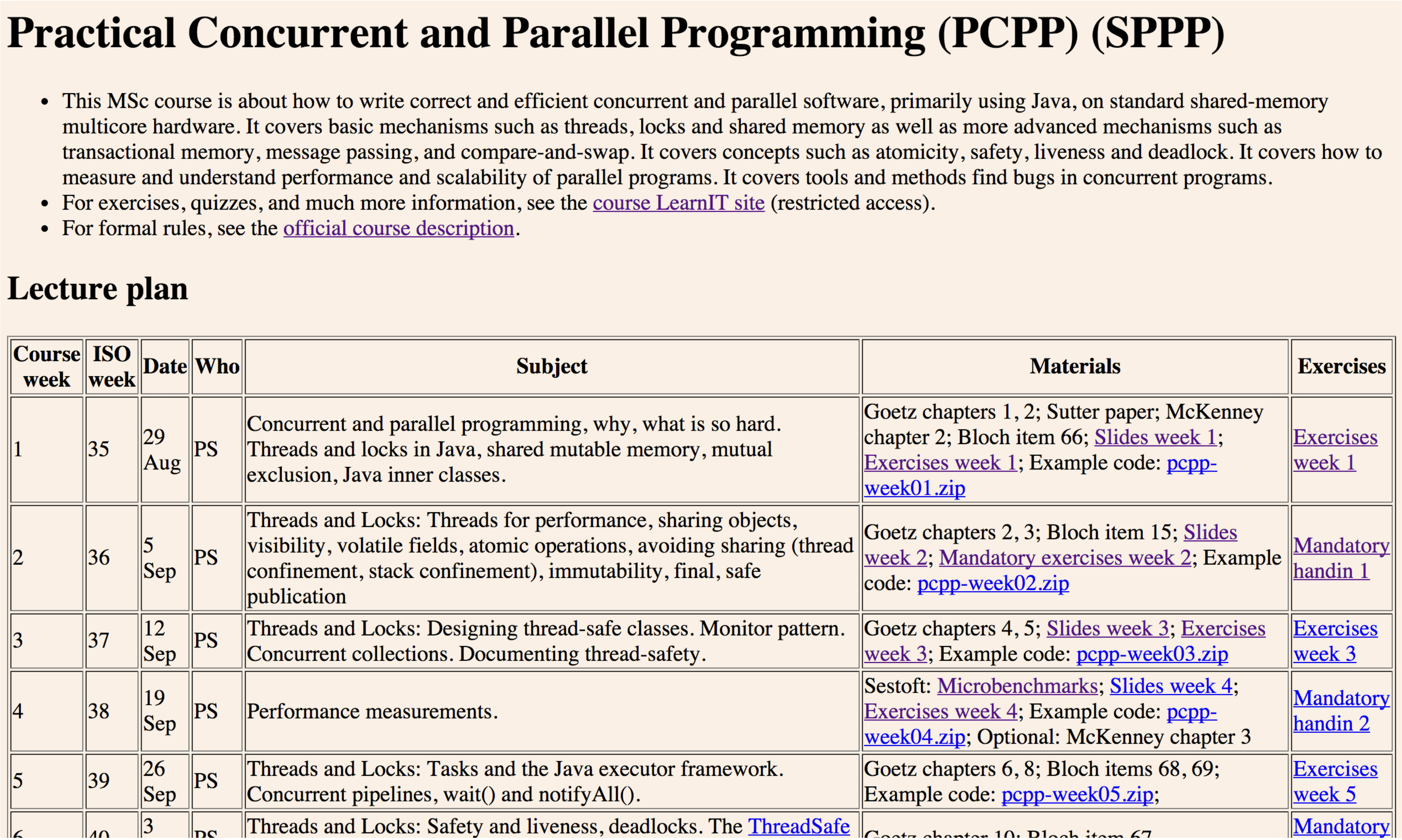


2012



2014

My current obsession: new ITU course



Practical Concurrent and Parallel Programming (PCPP) (SPPP)

- This MSc course is about how to write correct and efficient concurrent and parallel software, primarily using Java, on standard shared-memory multicore hardware. It covers basic mechanisms such as threads, locks and shared memory as well as more advanced mechanisms such as transactional memory, message passing, and compare-and-swap. It covers concepts such as atomicity, safety, liveness and deadlock. It covers how to measure and understand performance and scalability of parallel programs. It covers tools and methods find bugs in concurrent programs.
- For exercises, quizzes, and much more information, see the course LearnIT site (restricted access).
- For formal rules, see the official course description.

Lecture plan

Course week	ISO week	Date	Who	Subject	Materials	Exercises
1	35	29 Aug	PS	Concurrent and parallel programming, why, what is so hard. Threads and locks in Java, shared mutable memory, mutual exclusion, Java inner classes.	Goetz chapters 1, 2; Sutter paper; McKenney chapter 2; Bloch item 66; Slides week 1; Exercises week 1; Example code: pcpp-week01.zip	Exercises week 1
2	36	5 Sep	PS	Threads and Locks: Threads for performance, sharing objects, visibility, volatile fields, atomic operations, avoiding sharing (thread confinement, stack confinement), immutability, final, safe publication	Goetz chapters 2, 3; Bloch item 15; Slides week 2; Mandatory exercises week 2; Example code: pcpp-week02.zip	Mandatory handin 1
3	37	12 Sep	PS	Threads and Locks: Designing thread-safe classes. Monitor pattern. Concurrent collections. Documenting thread-safety.	Goetz chapters 4, 5; Slides week 3; Exercises week 3; Example code: pcpp-week03.zip	Exercises week 3
4	38	19 Sep	PS	Performance measurements.	Sestoft: Microbenchmarks; Slides week 4; Exercises week 4; Example code: pcpp-week04.zip; Optional: McKenney chapter 3	Mandatory handin 2
5	39	26 Sep	PS	Threads and Locks: Tasks and the Java executor framework. Concurrent pipelines, wait() and notifyAll().	Goetz chapters 6, 8; Bloch items 68, 69; Example code: pcpp-week05.zip;	Exercises week 5
6	40	3	PS	Threads and Locks: Safety and liveness, deadlocks. The ThreadSafe	Goetz chapter 10; Bloch item 67	Mandatory

Plan for today

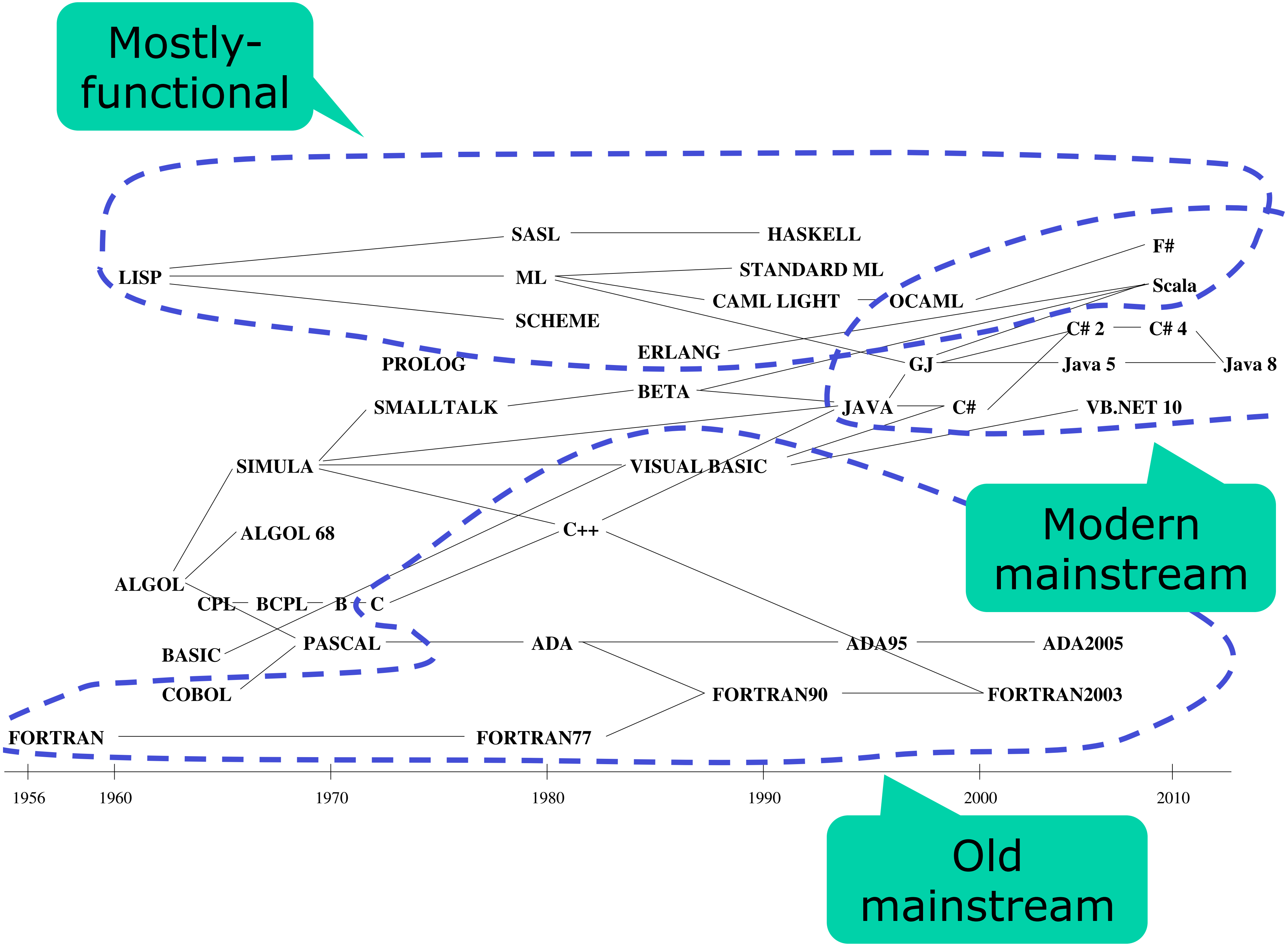
- Programming language genealogy
- Why functional programming, why now
- F#, an ML dialect
- Algebraic datatypes
- Pattern matching
- Higher-order functions
- Polymorphic type inference
- Sequences
- Functional programming in the mainstream
  - C# 5
  - Java 8
  - Scala

What is it? In a nutshell

- Compute with values, not locations
  - Data values are immutable
  - Functions have no side effects
- Build results as new data
  - Do not destructively update existing data
  - Example: **`add(set,x)`** produces a new set instead of updating the existing collection **`set`**
  - Cheap: immutable data structures can be shared
- Higher-order functions
- Static type, polymorphic types, and more

Why functional programming?

- Powerful modularization facilities:
  - abstraction: higher-order functions
  - statically checkable documentation: types
- Easier to reason about
- Types without tears due to type inference
- Easier to parallelize, exploit multicore
  - Shared mutable data is the root of all evil
  - Avoid *mutable*, and many problems go away



Mostly-functional

SASL
HASKELL
F#
LISP
ML
STANDARD ML
Scala
CAML LIGHT
OCAML
SCHEME
C# 2
C# 4
ERLANG
PROLOG
GJ
Java 5
Java 8
BETA
SMALLTALK
JAVA
C#
VB.NET 10
SIMULA
VISUAL BASIC
C++
ALGOL 68
ALGOL
CPL
BCPL
B
C
PASCAL
ADA
ADA95
ADA2005
BASIC
COBOL
FORTRAN90
FORTRAN2003
FORTRAN
FORTRAN77

Modern mainstream

Old mainstream

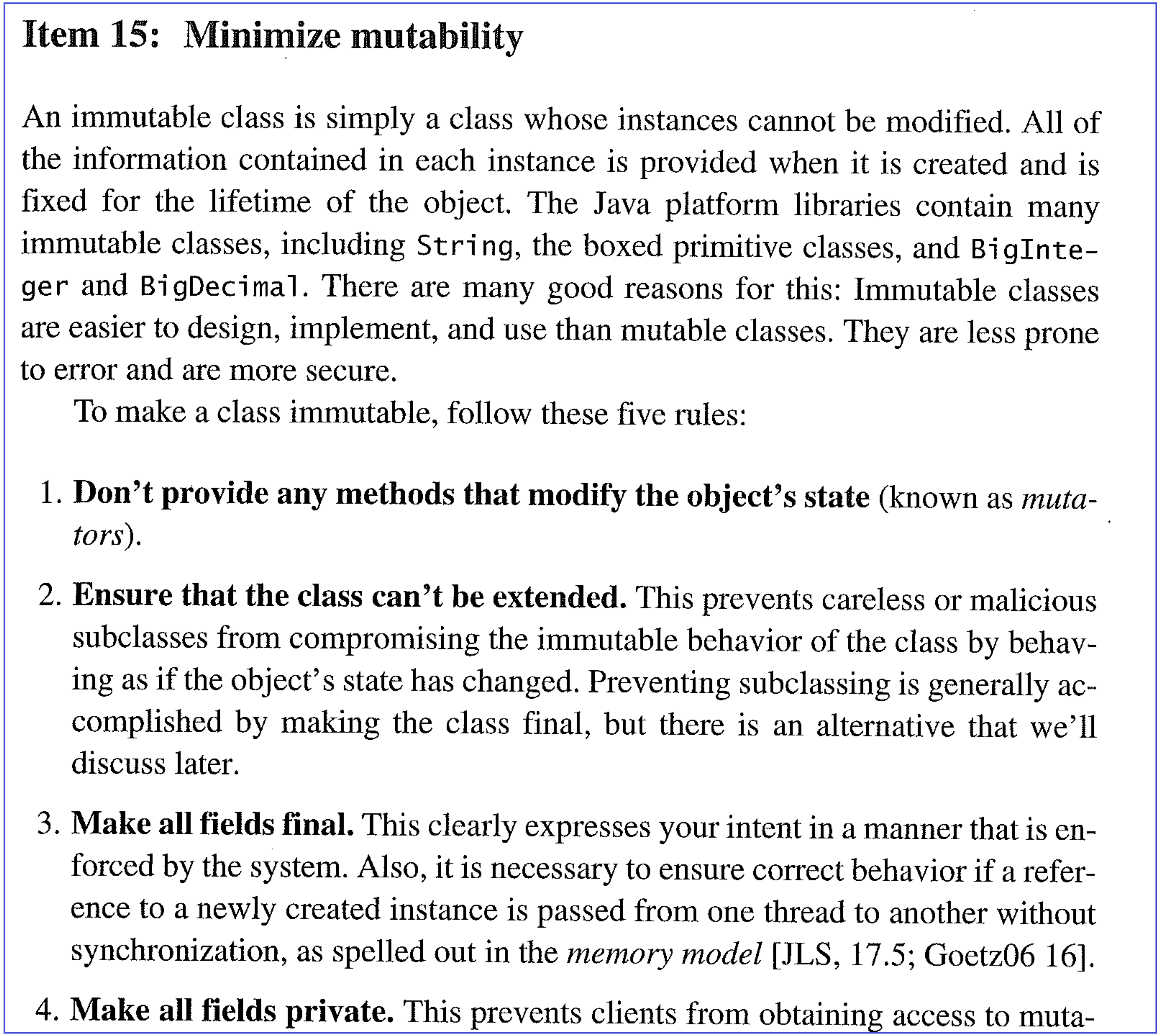
Why now? It has been here for ages

- Functional programming languages are old
  - Lisp 1960, Scheme 1978, dynamic types
  - ML 1978, polymorphic (generic) types
  - SASL 1976, Miranda, Lazy ML, Haskell 1989, lazy
- Also many classic books
  - Burge: *Recursive programming techniques*, 1975
  - Henderson: *Functional programming*, 1980
  - Peyton-Jones: Implementation func prog lang, 1987
  - Bird & Wadler: *Intro functional programming*, 1988
- Many old applications
  - Program analysis and transformation, artificial intelligence, computer-aided design, ...

Affordable, acceptable, necessary

- Technological advances: *affordable now*
  - Hardware has become bigger and faster
  - Garbage collection technology has matured
- Psychological advances: *acceptable now*
  - Java Virtual Machine (1994) and .NET (2000) led to accept of “managed platforms”, garbage collection
- Harder problems: *better tools needed now*
  - Generic types for modelling and specification
  - Higher abstractions are useful and effective
    - Eg. bulk data processing with C# LINQ and Java streams
  - Most functional computations are easy to parallelize
    - Eg. Parallel LINQ and Java 8 parallel streams

General trend towards functional



Item 15: Minimize mutability

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object. The Java platform libraries contain many immutable classes, including `String`, the boxed primitive classes, and `BigInteger` and `BigDecimal`. There are many good reasons for this: Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.

To make a class immutable, follow these five rules:

1. **Don't provide any methods that modify the object's state** (known as *mutators*).
2. **Ensure that the class can't be extended.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally accomplished by making the class final, but there is an alternative that we'll discuss later.
3. **Make all fields final.** This clearly expresses your intent in a manner that is enforced by the system. Also, it is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model* [JLS, 17.5; Goetz06 16].
4. **Make all fields private.** This prevents clients from obtaining access to muta-

Bloch: *Effective* Java, 2008, p. 73

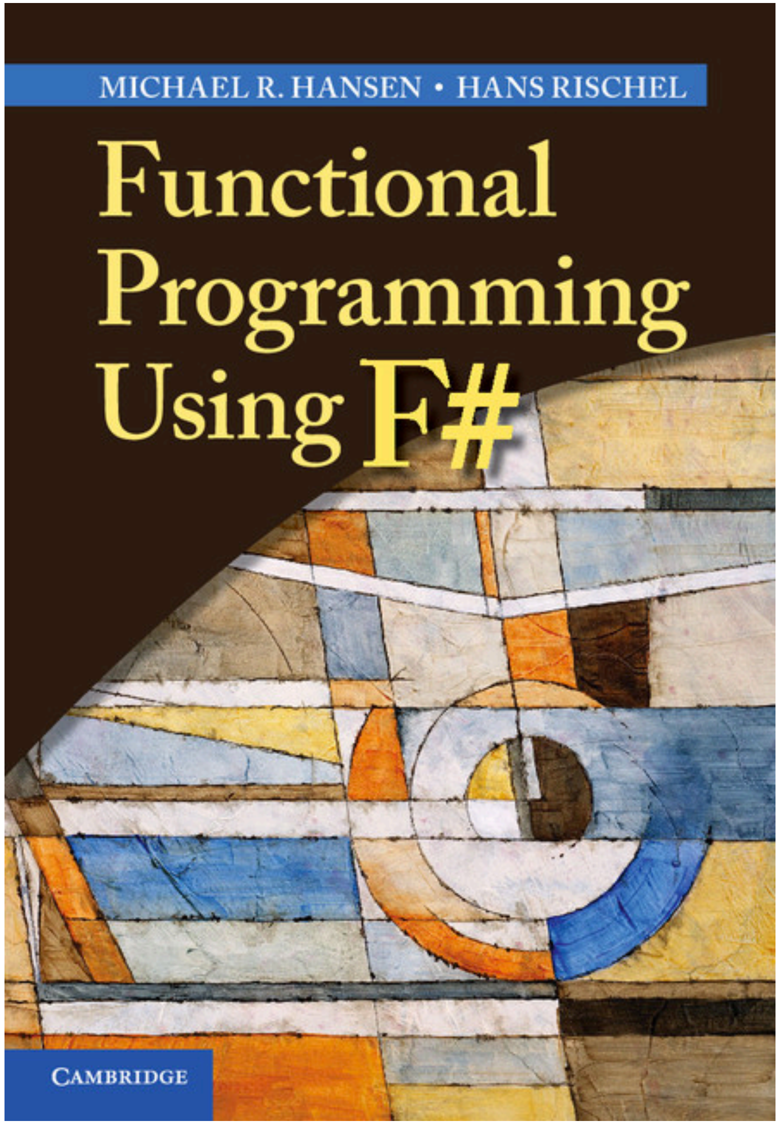
Josh Bloch designed the Java collection classes

A serious Java (or C#) developer should own and use this book

The F# functional language

- Runs on Microsoft .NET and Mono platforms
  - Can use standard .NET libraries, interface C#
  - Excellent performance
- Descends from OCaml and ML
- Many innovations:
  - Asynchronous computations
  - Units of measure type system
  - Type providers
- Used in finance and data analysis
- Don Syme, Microsoft Research UK

Recommended F# textbook



Written at DTU

Used at DTU and ITU

Hansen and Rischel: *Functional Programming with F#*, Cambridge University Press 2013

F# values, declarations and types

```
F# Interactive for F# 3.1 (Open Source Edition)
> let res = 3+4;;
val res : int = 7
```

Inferred type

Computed value

```
> let y = sqrt 2.0;;
val y : float = 1.414213562

> let large = 10 < res;;
val large : bool = false
```

- Bindings to immutable variables, **not** assignment
- Types inferred automatically

F# function definitions

```
> let circleArea r = System.Math.PI * r * r;;
val circleArea : r:float -> float

> let mul2 x = 2.0 * x;;
val mul2 : x:float -> float
```

Function type

- Calling a function:

```
> circleArea 10.0;;
val it : float = 314.1592654

> circleArea(10.0);;
val it : float = 314.1592654
```

F# recursion, pattern matching

- Defining factorial

```
> let rec fac n =
-       if n=0 then 1
-       else n * fac(n-1);;

val fac : n:int -> int
```

- Same, using pattern matching:

```
> let rec fac n =
-       match n with
-           | 0 -> 1
-           | _ -> n * fac(n-1);;

val fac : n:int -> int
```

F# pairs and tuples

```
> let p = (2, 3);;
val p : int * int = (2, 3)
```

Pair type

```
> let w = (2, true, 3.4, "blah");;
val w : int * bool * float * string
      = (2, true, 3.4, "blah")

> let add (x, y) = x + y;;
val add : x:int * y:int -> int
```

Function from pair to int

- A “two-argument” function is really a function from a single pair of arguments

F# lists

```
> let x1 = [7; 9; 13];;
val x1 : int list = [7; 9; 13]

> let x2 = 7 :: 9 :: 13 :: [];;
val x2 : int list = [7; 9; 13]

> x1 = x2;;
val it : bool = true
```

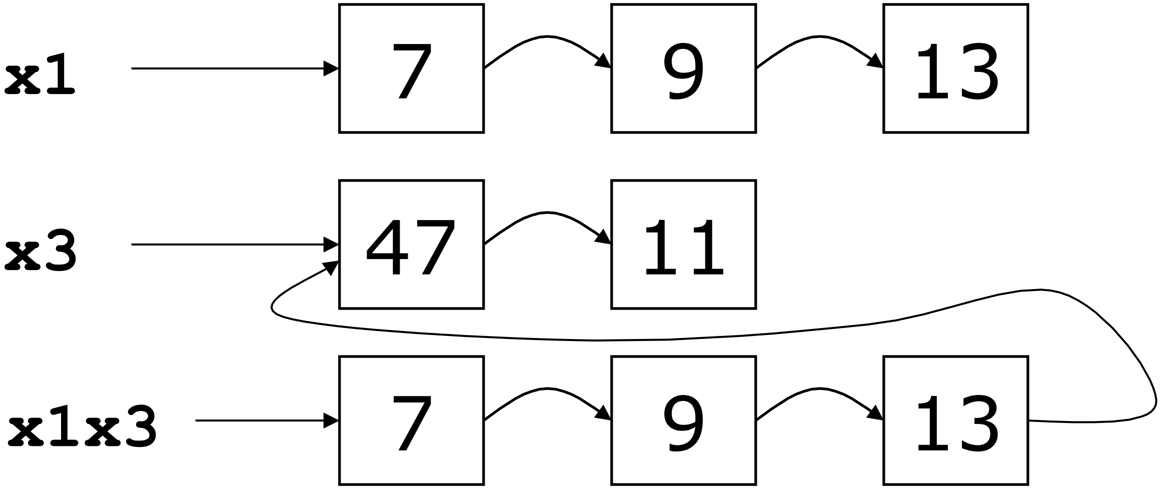
- Data structures compose to any depth
  - Eg a list of pairs of name and age

```
> let friends = [("Hans", 52); ("Hanne", 49)];;
val friends : (string * int) list = [("Hans", 52); ("Hanne", 49)]
```

List of pairs of string and int

List append (@)

```
> let x1 = [7; 9; 13];;
> let x3 = [47; 11];;
> let x1x3 = x1 @ x3;;
val x1x3 : int list = [7; 9; 13; 47; 11]
```



- F# data (lists, pairs, ...) are *immutable*
- This makes list tail sharing *unobservable*
- Admits economy impossible in C, Java, C#, ...

F# defining functions on lists

```
> let rec sum xs =
-     match xs with
-       | []    -> 0
-       | x::xr -> x + sum xr;;
val sum : xs:int list -> int

> sum x1;;
val it : int = 29
```

F# algebraic datatypes

- A person is either a teacher or a student:

```
type person =
     | Student of string
     | Teacher of string * int;;
```

Defines a type and two constructors

```
> let people = [Student "Niels"; Teacher("Peter", 5083)];;
val people : person list = [Student "Niels"; Teac ...]

> let getphone person =
-     match person with
-       | Teacher(name, phone) -> phone
-       | Student name         -> failwith "no phone";;
val getphone : person:person -> int
```

Matching on constructors

- Checks exhaustiveness and irredundancy
- OO would use abstract class Person with subclasses Teacher and Student

F# polymorphic functions

```
- let rec len xs =
-   match xs with
-     | []    -> 0
-     | x::xr -> 1 + len xr;;

val len : xs:'a list -> int
```

The function doesn’t look at the list elements

The function type is polymorphic

```
len [7; 9; 13]
len [true; true; false; true]
len ["foo"; "bar"]
len [("Peter", 50)]
```

It works on any type of list

- Same as a generic **method** in Java or C#

```
static int Count<T>(IEnumerable<T> xs) { ... }
```

F# polymorphic types: generic tree

```
type 'a tree =
   | Lf
   | Br of 'a * 'a tree * 'a tree;;
```

Defines a polymorphic type ’a tree and two constructors

```
> Br(42, Lf, Lf);;
val it : int tree = Br (42,Lf,Lf)

> Br("quoi?", Lf, Lf);;
val it : string tree = Br ("quoi?",Lf,Lf)

> Br(("Peter", 50), Lf, Lf);;
val it : (string * int) tree = Br (("Peter", 50),Lf,Lf)
```

- Same as a generic **type** in Java or C#
- But in F#, types are inferred automatically

F# sequence expressions

- Like “set comprehensions” in mathematics

```
- seq { 1..200 };;
val it : seq<int>

> seq { for x in 1..200 do yield 3*x };;
val it : seq<int> = seq [3; 6; 9; 12; ...]
```

{ 3*x | x in 1..200 }

```
Seq.sum(seq { for x in 1..200 do
              if x%5<>0 && x%7<>0
              then yield 1.0/float x })
```

Σ{ 1/x | x in 1..200 ∧ 5 and 7 do not divide x }

Pattern matching example: Symbolic differentiation

- Represent expression by algebraic datatype:

```
type expr =
     | Cst of int
     | Var of string
     | Add of expr * expr
     | Sub of expr * expr
     | Mul of expr * expr;;
```

- Examples:

```
> Mul(Cst 42, Var "x");;
val it : expr = Mul (Cst 42,Var "x")
```

42 * x

```
> Mul(Var "x", Mul(Var "x", Var "x"))
```

x * (x * x)

Pattern matching example: Symbolic differentiation wrt x

diff(k) = 0

diff(x) = 1

diff(y) = 0

diff(a + b) = diff(a) + diff(b)

diff(a * b) = diff(a) * b + a * diff(b)

diff(a – b) = diff(a) – diff(b)

```
let rec diffX (e : expr) =
    match e with
      | Cst i -> Cst 0
      | Var y when y="x" -> Cst 1
      | Var y             -> Cst 0
      | Add(e1, e2) -> Add(diffX e1, diffX e2)
      | Mul(e1, e2) -> Add(Mul(diffX e1, e2), Mul(e1, diffX e2))
      | Sub(e1, e2) -> Sub(diffX e1, diffX e2)
```

Differentiation works but results could be simplified

```
> diffX(Mul(Cst 42, Var "x"));;
val it : expr = Add (Mul (Cst 0,Var "x"),Mul (Cst 42,Cst 1))
```

Should be:
42

```
> diffX(Mul(Var "x", Var "x"));;
val it : expr = Add (Mul (Cst 1,Var "x"),Mul (Var "x",Cst 1))
```

Should be:
2 * x

```
> diffX (Mul(Var "x", Mul(Var "x", Var "x")));;
val it : expr =
  Add
    (Mul (Cst 1,Mul (Var "x",Var "x")),
     Mul (Var "x",Add (Mul (Cst 1,Var "x"),Mul (Var "x",Cst 1))))
```

Should be:
3 * x * x

Expression simplification

```
let rec simp e =
    match e with
     | Add(Cst 0,  e2)          -> simp e2
     | Add(e1,     Cst n)       -> Add(Cst n, simp e1)
     | Sub(e1,     Cst 0)       -> simp e1
     | Mul(Cst 0,  e2)          -> Cst 0
     | Mul(Cst 1,  e2)          -> simp e2
     | Mul(e1,     Cst n)       -> Mul(Cst n, simp e1)
     | Add(Cst i1, Cst i2)      -> Cst (i1+i2)
     | Mul(Cst i1, Cst i2)      -> Cst (i1*i2)
     | Sub(Cst i1, Cst i2)      -> Cst (i1-i2)
     | Add(e1, e2) when e1=e2 -> Mul(Cst 2, simp e1)
     | Add(e1, e2)              -> Add(simp e1, simp e2)
     | Mul(e1, e2)              -> Mul(simp e1, simp e2)
     | Sub(e1, e2) when e1=e2 -> Cst 0
     | Sub(e1, e2)              -> Sub(simp e1, simp e2)
     | _ -> e;;
```

$0+e = e$

$e+n = n+e$

$e-0 = e$

$0*e = 0$

$1*e = e$

$e*n = n*e$

$e+e = 2*e$

$e-e = 0$

```
let rec simplify e =
    let simpler = simp e
    in if e=simpler then e else simplify simpler;;
```

The simplifier works

```
> simplify(diffX(Mul(Cst 42, Var "x")));;
val it : expr = Cst 42
```

OK

```
> simplify(diffX(Mul(Var "x", Var "x")));;
val it : expr = Mul (Cst 2,Var "x")
```

OK

```
> simplify(diffX(Mul(Var "x", Mul(Var "x", Var "x"))));;
val it : expr = Add (Mul (Var "x",Var "x"),
                     Mul (Var "x",Mul (Cst 2,Var "x")))
```

x * x + x * (2 * x)

- Need more rules:

```
e1*(n*e2) = n*(e1*e2)
n*e + m*e = (n+m)*e
```

- Easy to add thanks to pattern matching

C# adopts functional concepts

- 1.0: Object-oriented, 2001
  - simple types, delegates

Kennedy and Syme

- 2.0: Generic types and methods, 2005
  - iterator blocks as stream generators

Proebsting

- 3.0: Functional programming and LINQ, 2007
  - lambda expressions, in-core LINQ is just functions

Meijer

- 4.0: Task Parallel Library, 2010
  - uses functions everywhere
- 5.0: Asynchronous methods, 2012
- 6.0: More functional programming, 2015?
  - pattern matching, immutable collections

C# anonymous functions (lambdas)

- Anonymous method (delegate) syntax C# 3:

```
delegate (int x) { return x%2==0; }
```

```
(int x) => x%2==0
```

Same meaning

```
x => x%2==0
```

Type inferred

C# generic delegate types

```
Action
Action<A1>
Action<A1,A2>
...
Func<R>
Func<A1,R>
Func<A1,A2,R>
...
```

.NET 3.5
(2007)

```
unit   -> unit
A1     -> unit
A1*A2  -> unit
...
unit   -> R
A1     -> R
A1*A2  -> R
...
```

F# or
Standard ML
(1978)

C# functional programming

- A method to compose a function with itself

```
public static Func<T,T> Twice<T>(Func<T,T> f) {
  return x => f(f(x));
}
```

- Some lambdas and computed functions

```
var fun1 = Twice<int>(x => 3*x);
Func<int,int> triple = x => 3*x;
var fun2 = Twice(triple);
Func<Func<int,int>, Func<int,int>> twice
                     = f => x => f(f(x));
var fun3 = twice(triple);
var res = fun1(4) + fun2(5) + fun3(6);
```

Linq, language integrated query

- Linq in C#:

```
from x in primes where x*x < 100 select 3*x
```

- Set comprehensions, ZF notation:

$$\{ 3x \mid x \in \text{primes},\ x^2 < 100 \}$$

transformer “select”

generator “from”

filter “where”

- Miranda (1985) list comprehensions, Haskell
- F# sequence expressions

From queries to method calls

- A query such as

```
from x in primes where x*x < 100 select 3*x
```

is transformed to an ordinary C# expression:

```
primes.Where(x => x*x < 100)
      .Select(x => 3 * x)
```

Functions as arguments

- There Where and Select methods are higher-order functions
- LINQ is disguised functional programming

Basic extension methods for Linq

```
IEnumerable<T> Where<T>(this IEnumerable<T> xs,
                        Func<T,bool> p)
```

- As list comprehension:
  [x | x <- xs, p(x)]

Filter

```
IEnumerable<U> Select<T,U>(this IEnumerable<T> xs,
                           Func<T,U> f)
```

- As list comprehension:
  [f(x) | x <- xs]

Map

Extension methods on IEnumerable

- Most support Linq for collections
- But an enumerable is nearly a lazy list, so they also support functional programming
- The F# sequence expression in C#:

```
double sum = Enumerable.Range(1, 200)
              .Where(x => x%5!=0 && x%7!=0)
              .Select(x => 1.0/x)
              .Sum();
```

Same

```
double sum =
   (from x in Enumerable.Range(1, 200)
    where x%5!=0 && x%7!=0
    select 1.0/x).Sum();
```

Java 8 function interfaces, 2014

- Java 1.1-7 have anonymous inner classes:

```
Thread t = new Thread(
   new Runnable() { public void run() { ... } }
);
```

An anonymous inner class, and an instance of it

- Java 8 *function interface:* exactly one method

```
interface Runnable { void run(); }
```

- Java 8 *anonymous function, "lambda"*

```
Thread t = new Thread(() -> ...);
```

Anonymous `void` function, compatible with Runnable

Java 8 streams, 2014

- Like .NET Enumerables & extension methods
  - In package java.util.stream
- The F# and C# example, in Java 8:

```
double sum =
  IntStream.range(1, 200)
  .filter(x -> x%5!=0 && x%7!=0)
  .mapToDouble(x -> 1.0/x)
  .sum();
```

- No LINQ-style syntactic sugar (so far)
- Java streams are easily parallelizable

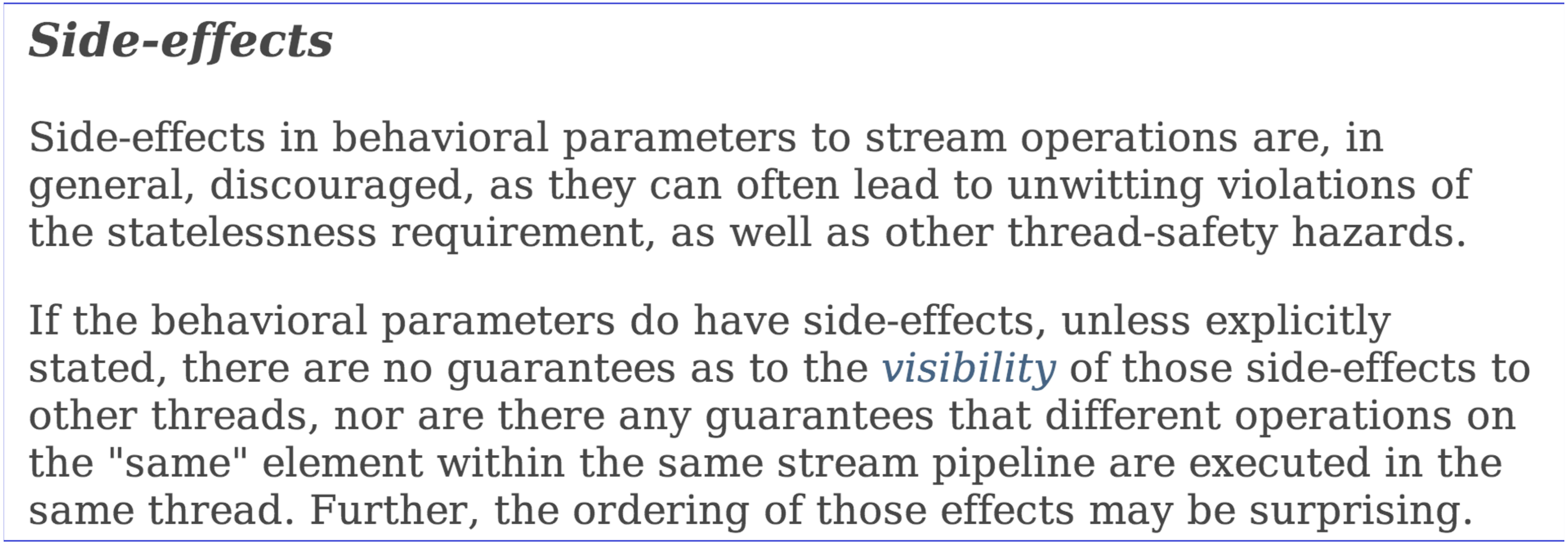
Java 8 streams are parallelizable

```
double sum =
  IntStream.range(1, 200).parallel()
  .filter(x -> x%5!=0 && x%7!=0)
  .mapToDouble(x -> 1.0/x)
  .sum();
```

Parallel!

But not faster: too little work

- Safe only if you program functionally:



Side-effects

Side-effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards.

If the behavioral parameters do have side-effects, unless explicitly stated, there are no guarantees as to the *visibility* of those side-effects to other threads, nor are there any guarantees that different operations on the "same" element within the same stream pipeline are executed in the same thread. Further, the ordering of those effects may be surprising.

Java 8 class library documentation

The Scala programming language

- Compiles to the Java platform
  - can work with Java class libraries and Java
  - is quite easy to pick up if you know Java
  - is much more concise and powerful
- Scala has classes, like Java and C#
  - Neat combination of functional and object-oriented
  - No interfaces, but traits = partial classes
- Many innovations
  - Very general libraries
  - Thanks to complex type system
  - Many ideas get adopted by C# and Java now
- Martin Odersky and others, EPFL, CH

Java versus Scala

Java

```
class PrintOptions {
  public static void main(String[] args) {
    for (String arg : args)
      if (arg.startsWith("-"))
        System.out.println(arg.substring(1));
  }
}
```

Singleton class; no statics

Declaration syntax

Array[T] is generic type

Sca

for expression

```
object PrintOptions {
  def main(args: Array[String]) = {
    for (arg <- args; if arg startsWith "-")
      println(arg substring 1)
  }
}
```

Can use Java class libraries

Interactive Scala

- Scala also has an interactive top-level
  - Like F#, Scheme, most functional languages

```
sestoft@mac ~/scala $ scala
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit...).

scala> def fac(n: Int): Int = if (n==0) 1 else n*fac(n-1)
fac: (n: Int)Int

scala> fac(10)
res0: Int = 3628800
```

java.util.BigInteger

```
scala> def fac(n: Int): BigInt = if (n==0) 1 else n*fac(n-1)
fac: (n: Int)BigInt

scala> fac(100)
res1: BigInt = 9332621544394415268169923885626670049071596
8264381621468592963895217599993229915608941463976156518286
253697920827223758251185210916864000000000000000000000000
```

Commercial uses of functional programming

- Financial sector
  - Functional is big in London and New York
  - Eg Jane Street Capital, Standard Chartered Bank
  - Denmark: Simcorp, financial back office systems
- Web services
  - Twitter, LinkedIn use Scala
- Security and high-integrity systems
  - Galois Inc
- Chip design and FPGA generation
  - Xilinx
- Stochastic testing
  - Qvik, QuickCheck for Erlang etc.