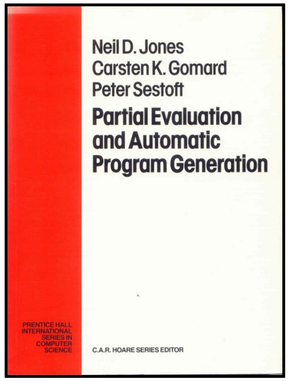
Parallel Functional Programming in Java 8

Peter Sestoft
IT University of Copenhagen

Parallel Functional Programming
Chalmers, Thursday 2023-05-04

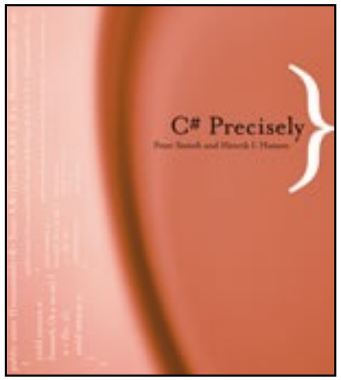
The speaker

- MSc 1988 computer science and mathematics and PhD 1991, DIKU, Copenhagen University
- KU, DTU, KVL and ITU; and Glasgow U, AT&T Bell Labs, Microsoft Research UK, Harvard University
- Functional, object-oriented, and parallel software

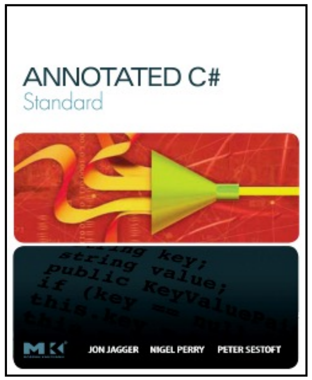


1993

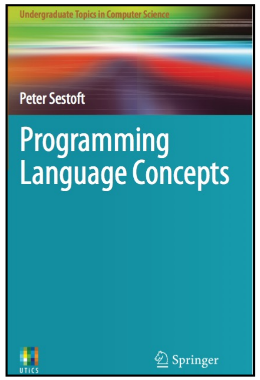
2002, 2005, 2016



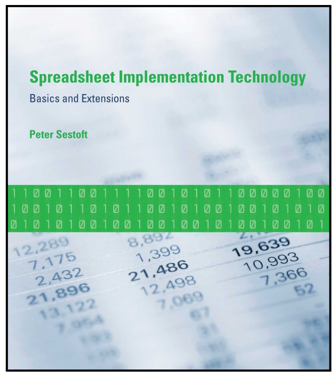
2004, 2012



2007



2012, 2017



2014

Plan

- Java 8 functional programming
  - Package java.util.function
  - Lambda expressions, method reference expressions
  - Functional interfaces, targeted function type
- Java 8 streams for bulk data
  - Package java.util.stream
- High-level parallel programming
  - Streams: primes, top-down parsing, n-queens, ...
  - Array parallel prefix operations
    - Class java.util.Arrays static methods

Materials

- Sestoft: *Java Precisely* 3rd ed., MIT Press 2016
  - § 11.13: Lambda expressions
  - § 11.14: Method reference expressions
  - § 23: Functional interfaces
  - § 24: Streams for bulk data
  - § 25: Class Optional<T>

- Book examples are called Example154.java etc
  - Get them from the book homepage http://www.itu.dk/people/sestoft/javaprecisely/

Functional and stream programming in Java 8 (2014)

- Lambda expressions

```
(String s) -> s.length
```

- Method reference expressions

```
String::length
```

- Functional interfaces

```
Function<String,Integer>
```

- Streams for bulk data processing

```
Stream<String> ss = ...
Stream<Integer> is = ss.map(String::length)
```

- Parallel stream processing

```
is = ss.parallel().map(String::length)
```

- Parallel array operations

```
Arrays.parallelSetAll(arr, i -> sin(i/PI/100.0))
Arrays.parallelPrefix(arr, (x, y) -> x+y)
```

Functional programming in Java

- *Immutable data* instead of objects with state
- *Recursion* instead of loops
- *Higher-order functions* that may
  - take functions as argument
  - return functions as result

Immutable list of T

Example154.java

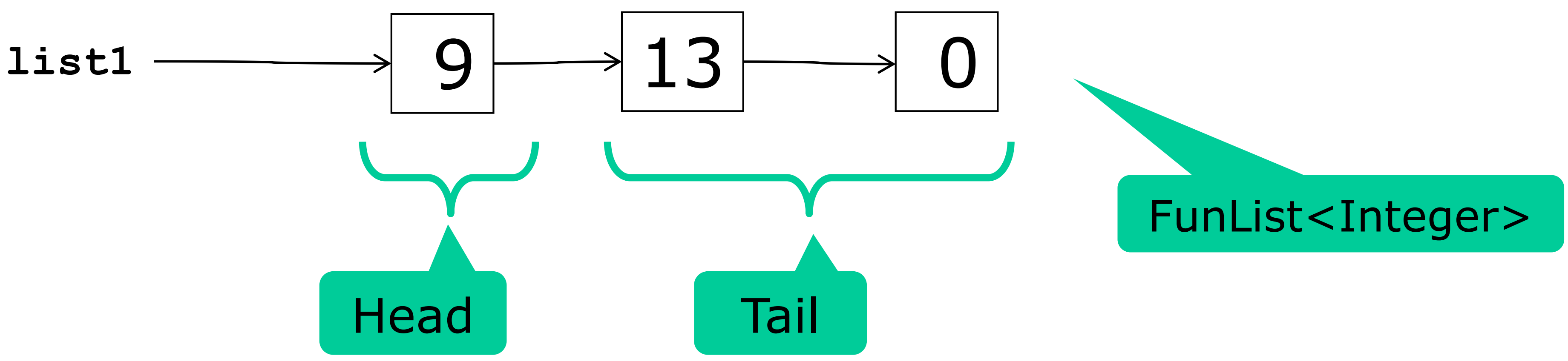
```
class FunList<T> {
  final Node<T> first;
  protected static class Node<U> {
    public final U item;
    public final Node<U> next;
    public Node(U item, Node<U> next) { ... }
  }
  ...
}
```

Immutable data

- FunList<T>, linked lists of nodes

```
class FunList<T> {
  final Node<T> first;
  protected static class Node<U> {
    public final U item;
    public final Node<U> next;
    public Node(U item, Node<U> next) { ... }
  }
  static <T> FunList<T> cons(T item, FunList<T> list) {
    return new FunList<T>(new Node<T>(item, list.first));
  }
```

Example154.java

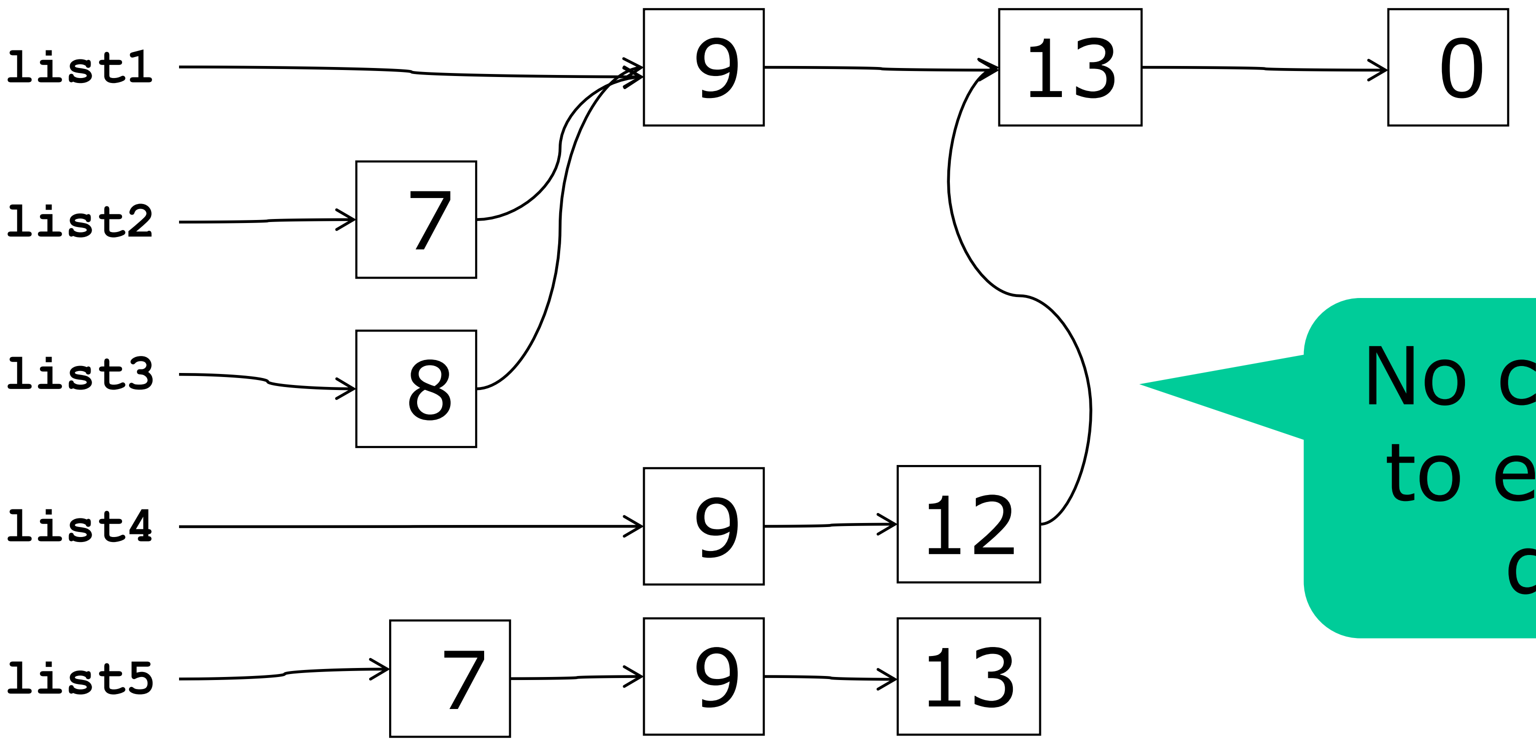


Immutability: No changes to existing data

```java
FunList<Integer> empty = new FunList<>(null),
  list1 = cons(9, cons(13, cons(0, empty))),
  list2 = cons(7, list1),
  list3 = cons(8, list1),
  list4 = list1.insert(1, 12),
  list5 = list2.removeAt(3);
```

Insert 12 before element number 1

Example154.java



No changes to existing data

Recursion in insert

Example154.java

```
public FunList<T> insert(int i, T item) {
  return new FunList<T>(insert(i, item, this.first));
}

static <T> Node<T> insert(int i, T item, Node<T> xs) {
  return i == 0 ? new Node<T>(item, xs)
         : new Node<T>(xs.item, insert(i-1, item, xs.next));
}
```

- "If **`i`** is zero, put **`item`** in a new node, and let its tail be the old list **`xs`**"
- "Otherwise, put the first element of **`xs`** in a new node, and let its tail be the result of inserting **`item`** in position **`i-1`** of the tail of **`xs`**"

Functional programming and immutable data: Pros and cons

- Immutability leads to more data allocation
  - Takes time and space
  - But modern allocators, garbage collectors are fast
- Immutable data can be safely shared
  - May actually reduce amount of allocation
- Immutable data are automatically threadsafe
  - No (other) thread can destructively update it; very good for parallel programming
  - And also due to visibility effects of `final` modifier

Subtle Java point; Java Precisely § 20.5.2

Mutable and/or shared memory

- Mutable: Data can be updated, reassigned
- Shared: Data can be accessed from two threads

	Shared	Unshared
Mutable	Imperative: C, Java, …; threads, locks, semaphore	Message passing: Erlang, Scala Akka
Immutable	Functional: Haskell, F#, Java parallel func. streams	

Languages, techniques

	Shared	Unshared
Mutable	Difficult, race conditions	Easy
Immutable	Easy	Easy

Conceptual difficulty

	Shared	Unshared
Mutable	Slow (cache states M ↔ I)	Fast (cache state M)
Immutable	Fast (cache state S)	Fast (cache state E)

MESI cache hardware

Java lambda expressions 1

Example64.java

- One argument lambda expressions:

```
Function<String,Integer>
  fsi1 = s -> Integer.parseInt(s);

... fsi1.apply("004711") ...
```

Function that takes a string s and parses it as an integer

Calling the function

Same, written in other ways

```
Function<String,Integer>
  fsi2 = s -> { return Integer.parseInt(s); },
  fsi3 = (String s) -> Integer.parseInt(s);
```

- Two-argument lambda expressions:

```
BiFunction<String,Integer,String>
  fsis1 = (s, i) -> s.substring(i, Math.min(i+3, s.length()));
```

Java method reference expressions

```
BiFunction<String,Integer,Character> charat
  = String::charAt;

System.out.println(charat.apply("ABCDEF", 1));
```

Same as (s,i) -> s.charAt(i)

Example67.java

```
Function<String,Integer> parseint = Integer::parseInt;
```

Same as fsi1, fsi2 and fsi3

```
Function<Integer,Character> hex1
  = "0123456789ABCDEF"::charAt;
```

Conversion to hex digit

Class and array constructors

```
Function<Integer,C> makeC = C::new;
Function<Integer,Double[]> make1DArray = Double[]::new;
```

Targeted function type (TFT)

- A lambda expression or method reference expression *does not have a type in itself*
- Therefore must have a *targeted function type*
- Lambda or method reference must appear as
  - Assignment right hand side:
    - ```
      Function<String,Integer> f = Integer::parseInt;
      ```
      TFT
  - Argument to call:
    - ```
      stringList.map(Integer::parseInt)
      ```
      map's argument type is TFT
  - In a cast:
    - ```
      (Function<String,Integer>)Integer::parseInt
      ```
      TFT
  - Argument to **return** statement:
    - ```
      return Integer::parseInt;
      ```
      Enclosing method's return type is TFT

Functions as arguments: map

```
public <U> FunList<U> map(Function<T,U> f) {
  return new FunList<U>(map(f, first));
}
static <T,U> Node<U> map(Function<T,U> f, Node<T> xs) {
  return xs == null ? null
       : new Node<U>(f.apply(xs.item), map(f, xs.next));
}
```

Example154.java

- Function **`map`** encodes general behavior
  - Transform each list element to make a new list
  - Argument **`f`** expresses the specific transformation
- Just as in Haskell, Scala, F#, Scheme, …
- Same effect as OO “template method pattern”

Calling map

```
7 9 13
```

```
FunList<Double> list8 = list5.map(i -> 2.5 * i);
```

```
17.5 22.5 32.5
```

```
FunList<Boolean> list9 = list5.map(i -> i < 10);
```

```
true true false
```

Java 8 functional interfaces

- A *functional interface* has exactly one abstract method

```
interface Function<T,R> {
  R apply(T x);
}
```

Type of functions from T to R

C#: Func<T,R>

F#: T -> R

```
interface Consumer<T> {
  void accept(T x);
}
```

Type of functions from T to void

C#: Action<T>

F#: T -> unit

(Too) many Java functional interfaces

Interface	Sec.	Function Type	Single Abstract Method Signature
		One-Argument Functions and Predicates	
Function<T,R>	23.5	`T -> R`	`R apply(T)`
UnaryOperator<T>	23.6	`T -> T`	`T apply(T)`
Predicate<T>	23.7	`T -> boolean`	`boolean test(T)`
Consumer<T>	23.8	`T -> void`	`void accept(T)`
Supplier<T>	23.9	`void -> T`	`T get()`
Runnable		`void -> void`	`void run()`
		Two-Argument Functions and Predicates	
BiFunction<T,U,R>	23.10	`T * U -> R`	`R apply(T, U)`
BinaryOperator<T>	23.11	`T * T -> T`	`T apply(T, T)`
BiPredicate<T,U>	23.7	`T * U -> boolean`	`boolean test(T, U)`
BiConsumer<T,U>	23.8	`T * U -> void`	`void accept(T, U)`
		Primitive-Type Specialized Versions of the Generic Functional Interfaces	
DoubleToIntFunction	23.5	`double -> int`	`int applyAsInt(double)`
DoubleToLongFunction	23.5	`double -> long`	`long applyAsLong(double)`
IntToDoubleFunction	23.5	`int -> double`	`double applyAsDouble(int)`
IntToLongFunction	23.5	`int -> long`	`long applyAsLong(int)`
LongToDoubleFunction	23.5	`long -> double`	`double applyAsDouble(long)`
LongToIntFunction	23.5	`long -> int`	`int applyAsInt(long)`
DoubleFunction<R>	23.5	`double -> R`	`R apply(double)`
IntFunction<R>	23.5	`int -> R`	`R apply(int)`
LongFunction<R>	23.5	`long -> R`	`R apply(long)`
ToDoubleFunction<T>	23.5	`T -> double`	`double applyAsDouble(T)`
ToIntFunction<T>	23.5	`T -> int`	`int applyAsInt(T)`
ToLongFunction<T>	23.5	`T -> long`	`long applyAsLong(T)`
ToDoubleBiFunction<T,U>	23.10	`T * U -> double`	`double applyAsDouble(T, U)`
ToIntBiFunction<T,U>	23.10	`T * U -> int`	`int applyAsInt(T, U)`
ToLongBiFunction<T,U>	23.10	`T * U -> long`	`long applyAsLong(T, U)`
DoubleUnaryOperator	23.6	`double -> double`	`double applyAsDouble(double)`
IntUnaryOperator	23.6	`int -> int`	`int applyAsInt(int)`
LongUnaryOperator	23.6	`long -> long`	`long applyAsLong(long)`
DoubleBinaryOperator	23.11	`double * double -> double`	`double applyAsDouble(double, double)`
IntBinaryOperator	23.11	`int * int -> int`	`int applyAsInt(int, int)`
LongBinaryOperator	23.11	`long * long -> long`	`long applyAsLong(long, long)`
DoublePredicate	23.7	`double -> boolean`	`boolean test(double)`
IntPredicate	23.7	`int -> boolean`	`boolean test(int)`
LongPredicate	23.7	`long -> boolean`	`boolean test(long)`
DoubleConsumer	23.8	`double -> void`	`void accept(double)`
IntConsumer	23.8	`int -> void`	`void accept(int)`
LongConsumer	23.8	`long -> void`	`void accept(long)`
ObjDoubleConsumer<T>	23.8	`T * double -> void`	`void accept(T, double)`
ObjIntConsumer<T>	23.8	`T * int -> void`	`void accept(T, int)`
ObjLongConsumer<T>	23.8	`T * long -> void`	`void accept(T, long)`
BooleanSupplier	23.9	`void -> boolean`	`boolean getAsBoolean()`
DoubleSupplier	23.9	`void -> double`	`double getAsDouble()`
IntSupplier	23.9	`void -> int`	`int getAsInt()`
LongSupplier	23.9	`void -> long`	`long getAsLong()`

```
interface IntFunction<R> {
  R apply(int x);
}
```

Use instead of Function<Integer,R> to avoid (un)boxing

Primitive-type specialized interfaces

Java Precisely page 125

Primitive-type specialized interfaces for int, double, and long

```
interface Function<T,R> {
  R apply(T x);
}
```

```
interface IntFunction<R> {
  R apply(int x);
}
```

Why both?

What difference?

```
Function<Integer,String> f1 = i -> "#" + i;
IntFunction<String> f2 = i -> "#" + i;
```

- Calling **`f1.apply(42)`** will *box* **`42`** as Integer
  - Allocating object in heap, takes time and memory
- Calling **`f2.apply(42)`** avoids boxing, is faster
- Purely for performance

Java streams for bulk data

- Stream<T> is a finite or infinite sequence of T
  - Possibly lazily generated
  - Possibly parallel
- Stream methods
  - **`map`**, **`flatMap`**, **`reduce`**, **`filter`**, ...
  - These take functions as arguments
  - Can be combined into pipelines
  - Java optimizes (and parallelizes) the pipelines well
- Similar to
  - Java Iterators, but very different implementation
  - The Enumerable extension methods underlying C#/.NET Language Integrated Query (Linq)

Some stream operations

- `Stream<Integer> s = Stream.of(2, 3, 5)`
- `s.filter(p)` = those `x` where `p.test(x)` holds

  `s.filter(x -> x%2==0)` gives 2
- `s.map(f)` = results of `f.apply(x)` for `x` in `s`

  `s.map(x -> 3*x)` gives 6, 9, 15
- `s.flatMap(f)` = a flattening of the streams created by `f.apply(x)` for `x` in `s`

  `s.flatMap(x -> Stream.of(x,x+1))` gives 2,3,3,4,5,6
- `s.findAny()` = some element of `s`, if any, or else the absent Option<T> value

  `s.findAny()` gives 2 or 3 or 5
- `s.reduce(x0, op)` = `x0❖s0❖...❖sn` if we write `op.apply(x,y)` as `x❖y`

  `s.reduce(1, (x,y)->x*y)` gives 1*2*3*5 = 30

Similar functions are everywhere

- Java stream **`map`** is called
  - **`map`** in Haskell, Scala, F#, Clojure
  - **`Select`** in C# Linq
- Java stream **`flatMap`** is called
  - **`concatMap`** in Haskell
  - **`flatMap`** in Scala
  - **`collect`** in F#
  - **`SelectMany`** in C# Linq
  - **`mapcat`** in Clojure
- Java **`reduce`** is a special (assoc. op.) case of
  - **`foldl`** in Haskell
  - **`foldLeft`** in Scala
  - **`fold`** in F#
  - **`Aggregate`** in C# Linq
  - **`reduce`** in Clojure

Counting primes on Java 8 streams

- A standard Java for loop:

```
int count = 0;
for (int i=0; i<range; i++)
  if (isPrime(i))
    count++;
```

Classical efficient imperative loop

- Sequential Java 8 stream:

```
IntStream.range(0, range)
.filter(i -> isPrime(i))
.count()
```

Pure functional programming ...

- Parallel Java 8 stream:

```
IntStream.range(0, range)
.parallel()
.filter(i -> isPrime(i))
.count()
```

... and thus parallelizable and thread-safe

Performance results (!!)

- Counting the primes in 0 ...99,999

Method	Intel i7 (ms)	AMD Opteron (ms)
Sequential for-loop	9.9	40.5
Sequential stream	9.9	40.8
Parallel stream	2.8	1.7
Best thread-parallel	3.0	4.9
Best task-parallel	2.6	1.9

- Functional streams give the simplest solution
- Nearly as fast as tasks and threads, or faster:
  - Intel i7 (4 cores) speed-up: 3.6 x
  - AMD Opteron (32 cores) speed-up: 24.2 x
  - ARM Cortex-A7 (RP 2B) (4 cores) speed-up: 3.5 x
- The future is parallel – and functional ☺

Side-effect freedom: functional!

- From the java.util.stream package docs:

> ***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.

function-type

Should say
”catastrophic”

- Java compiler and type system cannot enforce side-effect freedom – unlike Haskell
- Java runtime cannot detect violations

Creating Java streams 1

- Explicitly or from array, collection or map:

```
IntStream is = IntStream.of(2, 3, 5, 7, 11, 13);

String[] a = { "Hoover", "Roosevelt", ...};
Stream<String> presidents = Arrays.stream(a);

Collection<String> coll = ...;
Stream<String> countries = coll.stream();

Map<String,Integer> phoneNumbers = ...;
Stream<Map.Entry<String,Integer>> phones
  = phoneNumbers.entrySet().stream();
```

Example164.java

- Finite, ordered, sequential, lazily generated

Creating Java streams 2

- Useful special-case streams:
  - `IntStream.range(0, 100_000)`
  - `random.ints(5_000)`
  - `bufferedReader.lines()`
  - `regex.matcher(text).results()`
  - `bitset.stream()`
- Functional iterators for infinite streams
- Imperative generators for infinite streams
- StreamBuilder<T>: eager, only finite streams

Example164.java

Creating Java streams 3: generators

- Generating 0, 1, 2, 3, ...

Functional

```
IntStream nats1 = IntStream.iterate(0, x -> x+1);
```

Most efficient (!!), and parallelizable

Object-oriented imperative

```
IntStream nats2 = IntStream.generate(new IntSupplier() {
  private int next = 0;
  public int getAsInt() { return next++; }
});
```

Imperative, using final array for mutable state

```
final int[] next = { 0 };
IntStream nats3 = IntStream.generate(() -> next[0]++);
```

Example165.java

Streams & floating-point sum

- Eg. compute series sum: $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}$ for N=999,999,999
- Could make a DoubleStream, and use **`.sum()`**

```
IntStream.range(1, N)
   .mapToDouble(i -> 1.0/i)
   .sum();
```

21.300481501347942

- Or *parallel* DoubleStream and **`.sum()`**

```
IntStream.range(1, N)
   .parallel()
   .mapToDouble(i -> 1.0/i)
   .sum();
```

21.300481501347942

Precise (not exact) is 21.300481501347944

TestStreamSums.java

- What about a good old-fashioned for loop?

Summation with good old for-loops

- Compute series sum: for N=999,999,999 $$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{N}$$
- For-loop, forwards summation

```
double sum = 0.0;
for (int i=1; i<N; i++)
  sum += 1.0/i;
```

21.300481501348550

Different!

- For-loop, backwards summation

```
double sum = 0.0;
for (int i=1; i<N; i++)
  sum += 1.0/(N-i);
```

21.300481501346148

TestStreamSums.java

- Floating-point is sensitive to summation order
- Java DoubleStream **`.sum()`** uses *Kahan summation* to avoid these problems

Streams for top-down parsing 1

- Three forms of grammar Rule:

a	Literal: parse a
r1 r2 … rn	Sequence: parse r1, then r2, …
r1 | r2	Alternative: parse r1 or r2

```
static abstract class Rule {
  final Rule first;

  public abstract IntStream match(String s, int p);

  public static boolean match(String s) {
    return first.match(s, 0).anyMatch(pos -> pos == s.length());
  }
}
static class Lit extends Rule { ... }
static class Seq extends Rule { ... }
static class Alt extends Rule { ... }
```

Stream of integers q such that the rule can match s[p..q-1]

Problem from Advent of Code 19 Dec 2020

Streams for top-down parsing 2

```
static class Lit extends Rule {
  final String val;
  public IntStream match(String s, int pos) {
    return s.startsWith(val, pos) ? IntStream.of(pos+val.length())
                                  : IntStream.empty();
  }
}

static class Seq extends Rule {
  final Rule[] rules;
  public IntStream match(String s, int pos0) {
    IntStream result = IntStream.of(pos0);
    for (int rule : rules)
      result = result.flatMap(pos -> rule.match(s, pos));
    return result;
} }

static class Alt extends Rule {
  final Seq alt1, alt2;
  public IntStream match(String s, int pos0) {
    return IntStream.concat(alt1.match(s, pos0), alt2.match(s, pos0));
} }
```

Streams for backtracking

- Eg. generate all n-permutations of 0, 1, ..., n-1
  n=3: [2,1,0], [1,2,0], [2,0,1], [0,2,1], [0,1,2], [1,0,2]

Set of numbers not yet used in **tail**

An incomplete permutation

```
public static Stream<IntList> perms(BitSet todo, IntList tail) {
  if (todo.isEmpty())
    return Stream.of(tail);
  else
    return todo.stream().boxed()
     .flatMap(r -> perms(minus(todo, r), new IntList(r, tail)));
}
```

Example175.java

```
public static Stream<IntList> perms(int n) {
  BitSet todo = new BitSet(n); todo.flip(0, n);
  return perms(todo, null);
}
```

{ 0, ..., n-1 }

Empty permutation [ ]

A closer look at generation for n=3

todo tail

({0,1,2}, [])

r=0 ({1,2}, [0])

({2}, [1,0])

({}, [2,1,0]) To result stream

({1}, [2,0])

({}, [1,2,0]) To result stream

r=1 ({0,2}, [1])

({2}, [0,1])

({}, [2,0,1]) To result stream

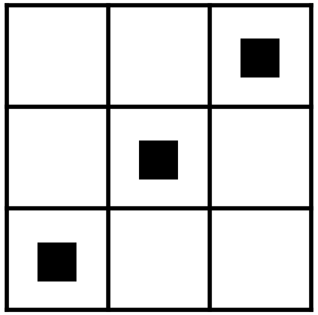
({0}, [2,1])

({}, [0,2,1]) To result stream

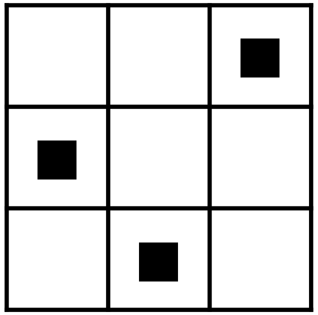
r=2 ({0,1}, [2])

...

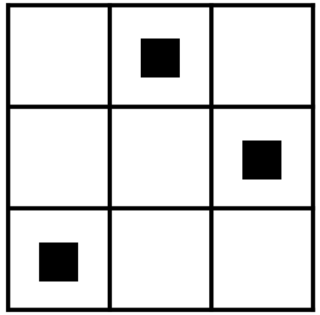
A permutation is a safe rook (tårn) placement on a chessboard



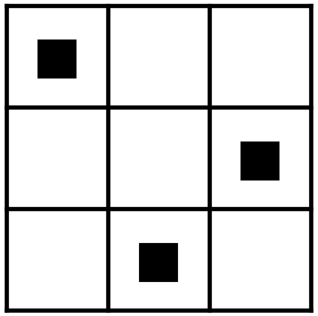
[2, 1, 0]



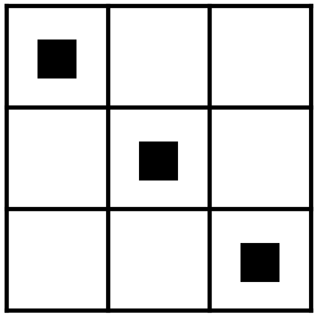
[1, 2, 0]



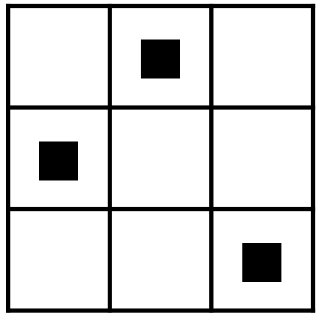
[2, 0, 1]



[0, 2, 1]



[0, 1, 2]



[1, 0, 2]

Solutions to the n-queens problem

- For queens, just take diagonals into account:
  - consider only r that are safe for the partial solution

```
public static Stream<IntList> queens(BitSet todo, IntList tail) {
  if (todo.isEmpty())
    return Stream.of(tail);
  else
    return todo.stream()
     .filter(r -> safe(r, tail)).boxed()
     .flatMap(r -> queens(minus(todo, r), new IntList(r, tail)));
}
```

Diagonal check

.parallel()

Example176.java

```
public static boolean safe(int mid, IntList tail) {
  return safe(mid+1, mid-1, tail);
}
public static boolean safe(int d1, int d2, IntList tail) {
  return tail==null || d1!=tail.item && d2!=tail.item && safe(d1+1, d2-1, tail.next);
}
```

- Simple, and parallelizable for free, 3.5 x faster
- Solve or generate sudokus: much the same

Versatility of streams

- Many uses of a stream of solutions
  - Print the number of solutions

```
System.out.println(queens(8).count());
```

  - Print all solutions

```
queens(8).forEach(System.out::println);
```

  - Print an arbitrary solution (if there is one)

```
System.out.println(queens(8).findAny());
```

  - Print the 20 first solutions

```
queens(8).limit(20).forEach(System.out::println);
```

Example174.java

- Much harder in an imperative version
- Separation of concerns (Dijkstra): *production* of solutions versus *consumption* of solutions

Streams for quasi-infinite sequences

- van der Corput numbers
  - 1/2, 1/4, 3/4, 1/8, 5/8, 3/8, 7/8, 1/16, ...
  - Dense and uniform in interval [0, 1]
  - For simulation and finance, Black-Scholes options
- Trick: vd Corput numbers as base-2 fractions

  0.1, 0.01, 0.11, 0.001, 0.101, 0.011, 0.111, ...

  are *bit-reversals* of 1, 2, 3, 4, 5, 6, 7, ... in binary

```
public static DoubleStream vanDerCorput() {
  return IntStream.range(1, 31).asDoubleStream()
        .flatMap(b -> bitReversedRange((int)b));
}

private static DoubleStream bitReversedRange(int b) {
  final long bp = Math.round(Math.pow(2, b));
  return LongStream.range(bp/2, bp)
        .mapToDouble(i -> (double)(bitReverse((int)i) >>> (32-b)) / bp);
}
```

Example183.java

Java 8 stream properties

- Some stream dimensions
  - Finite versus infinite
  - Lazily generated (by **`iterate`**, **`generate`**, ...) versus eagerly generated (stream builders)
  - Ordered (**`map`**, **`filter`**, **`limit`** ... preserve element order) versus unordered
  - Sequential (all elements processed on one thread) versus parallel
- Java streams
  - can be lazily generated, like Haskell lists
  - but are *use-once*, unlike Haskell lists
    - reduces risk of space leaks
    - limits expressiveness, harder to compute average …

How are Java streams implemented?

- Spliterators

```
interface Spliterator<T> {
  long estimateSize();
  void forEachRemaining(Consumer<T> action);
  boolean tryAdvance(Consumer<T> action);
  void Spliterator<T> trySplit();
}
```

  - Many method calls, well inlined/fused by the JIT
- Parallelization
  - Divide stream into chunks using **`trySplit`**
  - Process each chunk in a Java task (Haskell “spark”)
  - Run on thread pool using work-stealing queues
  - ... thus similar to Haskell parBuffer/parListChunk

Parallel (functional) array operations

- Simulating random motion on a line
  - Take n random steps of length at most [-1, +1]:

```
double[] a = new Random().doubles(n, -1.0, +1.0)
                         .toArray();
```

Example25.java

  - Compute the positions at end of each step:

    a[0], a[0]+a[1], a[0]+a[1]+a[2], ...

```
Arrays.parallelPrefix(a, (x,y) -> x+y);
```

NB: Updates array **`a`**

  - Find the maximal absolute distance from start:

```
double maxDist = Arrays.stream(a).map(Math::abs)
                             .max().getAsDouble();
```

- A lot done, fast, without loops or assignments
  - Just arrays and streams and functions

Array and streams and parallel ...

- Associative array aggregation

```
Arrays.parallelPrefix(a, (x,y) -> x+y);
```

- Such operations can be parallelized well
  - So-called *prefix scans* (Blelloch 1990)

- Streams and arrays complement each other:
- Streams: lazy, possibly infinite, non-materialized, use-once, parallel pipelines
- Arrays: eager, always finite, materialized, use-many-times, parallel prefix scans

Some problems with Java streams

- Streams are use-once & have other restrictions
  - Probably to permit easy parallelization
- Hard to create lazy finite streams
  - Probably to allow simple high-performance implementation
- Difficult to control resource consumption
- A single side-effect may mess all up completely
- Sometimes **`.parallel()`** hurts performance a lot
  - See exercise
  - And strange behavior, in parallel + limit in Sudoku generator

- Laziness in Java easily goes wrong, try-with-resource:

```
static Stream<String> getPageAsStream(String url) throws IOException {
  try (BufferedReader in
         = new BufferedReader(new InputStreamReader(
                                          new URL(url).openStream()))) {
        return in.lines();
     }
  }
```

Example216.java

Closes the Reader eagerly, so any use of the Stream<String> causes IOException: Stream closed

Useless

A multicore performance mystery

2P

- K-means clustering 2P: Assign – Update – Assign – Update … till convergence

Pseudocode

```
while (!converged) {
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      myCluster[pi] = closest(points[pi], clusters);
  }
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      myCluster[pi].addToMean(points[pi]);
  }
  ...
}
```

Assign

Update

TestKMeansSolution.java

Imperative

- Assign: writes a point to **`myCluster[pi]`**
- Update: calls **`addToMean`** on **`myCluster[pi]`**

A multicore performance mystery

2Q

- ”Improved” version 2Q:
  - call **addToMean** directly on point
  - instead of first writing it to **myCluster** array

```
while (!converged) {
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      closest(points[pi], clusters).addToMean(points[pi]);
  }
  ...
}
```

Performance of k-means clustering

- Sequential: as you would expect, 5% speedup
- Parallel: surprisingly bad!

"Improved"

	2P	2Q	Ratio 2Q/2P
Sequential	4.240	4.019	0.95
4-core parallel i7	1.310	2.234	1.70
24-core parallel Xeon	0.852	6.587	7.70

Time in seconds for 200,000 points, 81 clusters, 1/8/48 tasks, 108 iterations

Bad

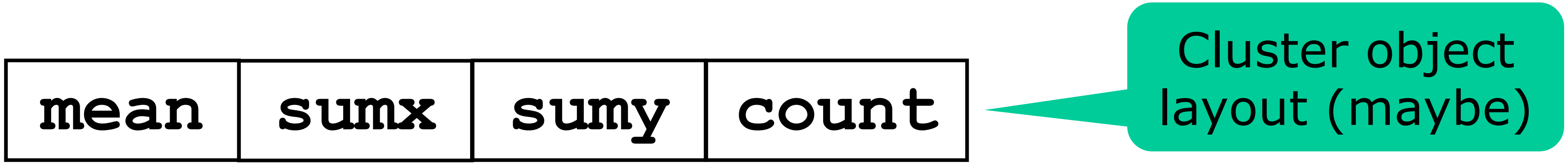
Very bad

- Q: WHY is the "improved" code slower?
- A: Cache invalidation and false sharing

The Point and Cluster classes

```
class Point {
  public final double x, y;
}
```

```
static class Cluster extends ClusterBase {
  private volatile Point mean;
  private double sumx, sumy;
  private int count;
  public synchronized void addToMean(Point p) {
    sumx += p.x;
    sumy += p.y;
    count++;
  }
  ...
}
```



Parallel streams to the rescue, 3P

3P

```
while (!converged) {
  final Cluster[] clustersLocal = clusters;
  Map<Cluster, List<Point>> groups =
    Arrays.stream(points).parallel()
          .collect(Collectors.groupingBy(p -> closest(p,clustersLocal)));
  clusters = groups.entrySet().stream().parallel()
    .map(kv -> new Cluster(kv.getKey().getMean(), kv.getValue()))
    .toArray(Cluster[]::new);
  Cluster[] newClusters =
    Arrays.stream(clusters).parallel()
          .map(Cluster::computeMean).toArray(Cluster[]::new);
  converged = Arrays.equals(clusters, newClusters);
  clusters = newClusters;
}
```

Assign

Update

Functional

	2P	2Q	3P stream
Sequential	4.240	4.019	5.353
4-core parallel i7	1.310	2.234	1.350
24-core parallel Xeon	0.852	6.587	0.553

!!!

Time in seconds for 200,000 points, 81 clusters, 1/8/48 tasks, 108 iterations

Mutable and/or shared memory

- Mutable: Data can be updated, reassigned
- Shared: Data can be accessed from two threads

	Shared	Unshared
Mutable	Imperative: C, Java, …; threads, locks, semaphore	Message passing: Erlang, Scala Akka
Immutable	Functional: Haskell, F#, Java parallel func. streams	

Languages, techniques

	Shared	Unshared
Mutable	Difficult, race conditions	Easy
Immutable	Easy	Easy

Conceptual difficulty

	Shared	Unshared
Mutable	Slow (cache states M ↔ I)	Fast (cache state M)
Immutable	Fast (cache state S)	Fast (cache state E)

MESI cache hardware

Materials

- Reading
  - Java Precisely 3rd ed. § 11.13, 11.14, 23, 24, 25
  - Optional:
    - http://www.itu.dk/people/sestoft/papers/benchmarking.pdf
    - http://www.itu.dk/people/sestoft/papers/cpucache-20170319.pdf

"A multicore performance mystery"

- Exercises
  - Extend immutable list class with functional programming; use parallel array operations; use streams of words and streams of numbers
  - Alternatively: Make a faster and more scalable k-means clustering implementation, if possible, in any language