

Practical Concurrent and Parallel Programming 3

Peter Sestoft
IT University of Copenhagen

Friday 2015-09-11*

Comments on exercises

- True: If program p fails when tested, then it is not thread-safe
- **False:** If program p does not fail when tested, then it is thread-safe
- Rules for exercise hand-in have been posted
- Questions or observations about exercises?

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, queens
 - Array parallel prefix operations
 - Class `java.util.Arrays` static methods

Materials for today

- *Java Precisely* 3rd edition (draft 27 July 2015)
 - §11.13: Lambda expressions
 - §11.14: Method reference expressions
 - §23: Functional interfaces
 - §24: Streams for bulk data
 - §25: Class `Optional<T>`
 - Get the PDF on LearnIT
- Book examples are called `Example154.java` etc
 - Get them from the course homepage

New in Java 8

- Lambda expressions
`(String s) -> s.length`
- Method reference expressions
`String::length`
- Functional interfaces
`Function<String, Integer>`
- Streams for bulk data
`Stream<Integer> is = ss.map(String::length)`
- Parallel streams
`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

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

```
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
list of T

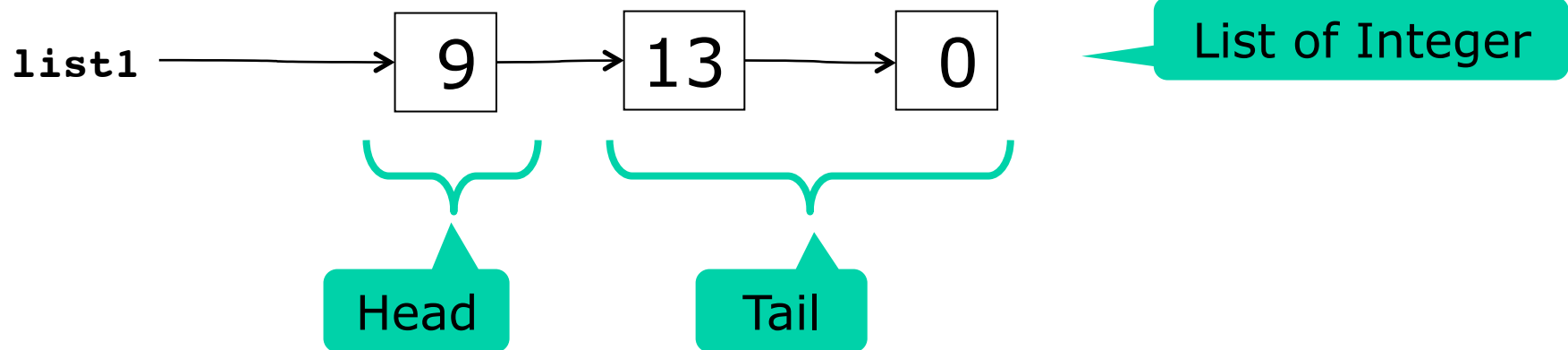
Example154.java

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) { ... }  
    }  
}
```

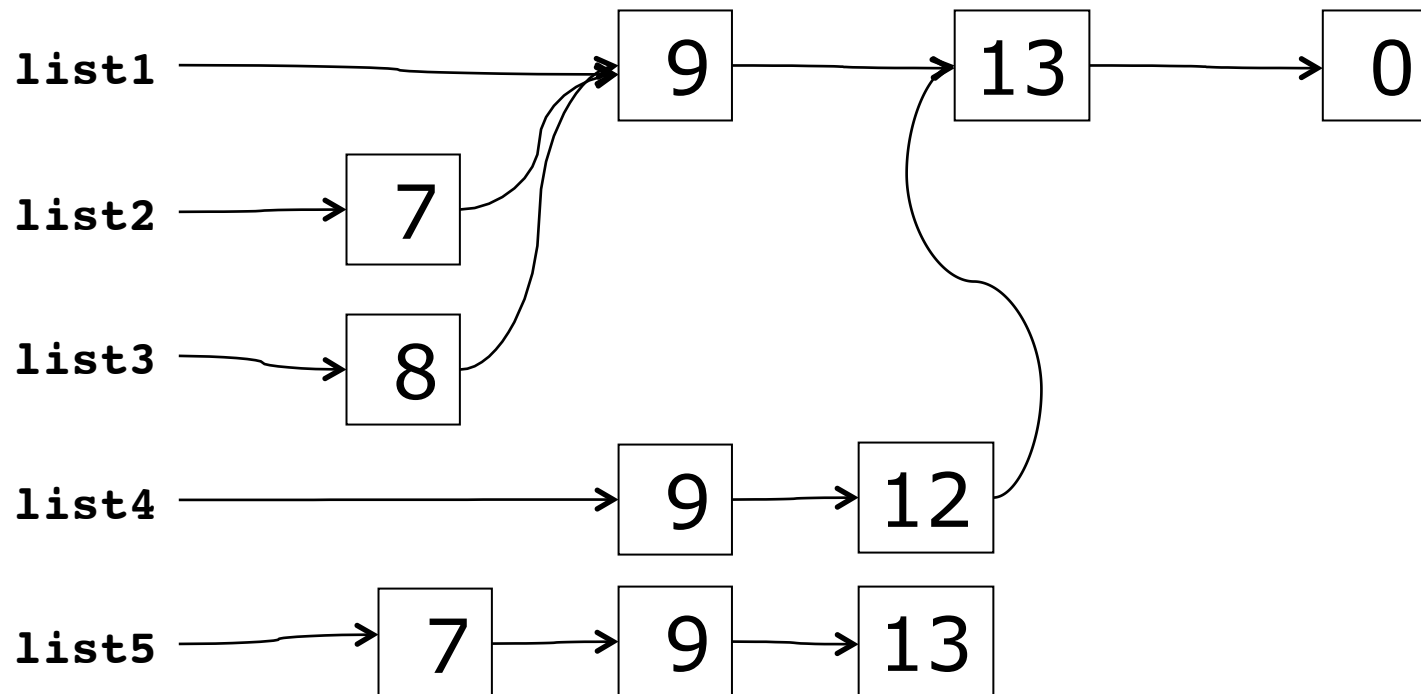
Example154.java



Existing data do not change

```
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);
```

Example154.java



Recursion in insert

```
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));  
}
```

Example154.java

- “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**”

Immutable data: Bad and good

- Immutability leads to more allocation
 - Takes time and space
 - But modern 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 mess with it, either
 - (And also due to visibility effects of **final** modifier)

Lambda expressions 1

Example64.java

- One argument lambda expressions:

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

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

Calling the function

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

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()));
```

Lambda expressions 2

Example64.java

- Zero-argument lambda expression:

```
Supplier<String>  
now = () -> new java.util.Date().toString();
```

- One-argument result-less lambda ("void"):

```
Consumer<String>  
show1 = s -> System.out.println(">>>" + s + "<<<");
```

```
Consumer<String>  
show2 = s -> { System.out.println(">>>" + s + "<<<"); };
```

Method reference expressions

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

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

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

Example67.java

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

Same as fs1, fs2 and fs3

```
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;`

– Argument to call:

- `stringList.map(Integer::parseInt)`

– In a cast:

- `(Function<String, Integer>) Integer::parseInt`

– Argument to `return` statement:

- `return Integer::parseInt;`

TFT

map's argument type is TFT

TFT

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
- Similar to template method pattern in OO

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

Functions as arguments: reduce

```
static <T,U> U reduce(U x0, BiFunction<U,T,U> op, Node<T> xs) {  
    return xs == null ? x0  
        : reduce(op.apply(x0, xs.item), op, xs.next);  
}
```

- **`list.reduce(x0, op)`**
 = $x_0 \diamond x_1 \diamond \dots \diamond x_n$
 if we write `op.apply(x, y)` as $x \diamond y$
- Example: **`list.reduce(0, (x, y) -> x+y)`**
 = $0 + x_1 + \dots + x_n$

Example154.java

Calling reduce

17.5 22.5 32.5

```
double sum = list8.reduce(0.0, (res, item) -> res + item);
```

72.5

Example154.java

```
double product = list8.reduce(1.0, (res, item) -> res * item);
```

12796.875

```
boolean allBig  
    = list8.reduce(true, (res, item) -> res && item > 10);
```

true

Tail recursion and loops

```
static <T,U> U reduce(U x0, BiFunction<U,T,U> op, Node<T> xs) {  
    return xs == null ? x0  
        : reduce(op.apply(x0, xs.item), op, xs.next);  
}
```

Tail call

- A call that is the func's last action is a tail call
- A tail-recursive func can be replaced by a loop

```
static <T,U> U reduce(U x0, BiFunction<U,T,U> op, Node<T> xs) {  
    while (xs != null) {  
        x0 = op.apply(x0, xs.item);  
        xs = xs.next;  
    }  
    return x0;  
}
```

Loop version
of reduce

Example154.java

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 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

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(i)** involves *boxing* of **i** as Integer
 - Allocating object in heap, takes time and memory
- Calling **f2(i)** avoids boxing, is faster
- Purely a matter of performance

Functions that return functions

- Conversion of n to English numeral, cases

$n < 20$: one, two, ..., nineteen

$n < 100$: twenty three, ...

Same pattern

$n \geq 100$: two hundred forty three, ...

$n \geq 1000$: three thousand two hundred forty three...

$n \geq 1$ million: ...

$n \geq 1$ billion: ...

```
static String less100(long n) {  
    return n < 20 ? ones[(int)n  
        : tens[(int)n/10-2] + after(ones[(int)n%10]);  
}  
static LongFunction<String> less(long limit, String unit,  
    LongFunction<String> conv) {  
    return n -> n < limit ? conv.apply(n)  
        : conv.apply(n/limit) + " " + unit  
        + after(conv.apply(n%limit));  
}
```

Convert $n < 100$

Example158.java

Functions that return functions

- Using the general higher-order function

```
static final LongFunction<String>
    less1K = less(          100, "hundred",  Example158::less100),
    less1M = less(       1_000, "thousand", less1K),
    less1B = less(    1_000_000, "million",  less1M),
    less1G = less(1_000_000_000, "billion",  less1B);
```

Example158.java

- Converting to English numerals:

```
public static String toEnglish(long n) {
    return n==0 ? "zero" : n<0 ? "minus " + less1G.apply(-n)
                               : less1G.apply(n);
}
```

toEnglish(2147483647)

**two billion one hundred forty seven million four
hundred eighty three thousand six hundred forty seven**

Streams for bulk data

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

ExampleXXX.java

Some stream operations

- **Stream<Integer> s = Stream.of(2, 3, 5)**
- **s.filter(p)** = the **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** \diamond **s0** \diamond ... \diamond **sn** if we write **op.apply(x, y)** as **x** \diamond **y**
`s.reduce(1, (x, y) -> x*y)` gives $1*2*3*5 = 30$

Counting primes on Java 8 streams

- Our old 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, or faster:
 - Intel i7 (4 cores) speed-up: 3.6 x
 - AMD Opteron (32 cores) speed-up: 24.2 x
- The future is parallel – and functional 😊

Creating 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 streams 2

- Useful special-case streams:
- `IntStream.range(0, 10_000)`
- `random.ints(5_000)`
- `bufferedReader.lines()`
- `bitset.stream()`
- Functional iterators for infinite streams
- Imperative generators for infinite streams
- `StreamBuilder<T>`: eager, only finite streams

Example164.java

Creating streams 3: generators

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

Functional

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

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

Creating streams 4: StreamBuilder

- Convert own linked IntList to an IntStream

```
class IntList {
    public final int item;
    public final IntList next;
    ...
    public static IntStream stream(IntList xs) {
        IntStream.Builder sb = IntStream.builder();
        while (xs != null) {
            sb.accept(xs.item);
            xs = xs.next;
        }
        return sb.build();
    }
}
```

Example182.java

- Eager: no stream element output until end
- Finite: does not work on cyclic lists

Streams for backtracking

- Generate all n-permutations of 0, 1, ..., n-1
 - Eg [2,1,0], [1,2,0], [2,0,1], [0,2,1], [0,1,2], ...

Set of numbers
not yet used

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 $n=3$

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

({1,2}, [0])

({2}, [1,0])

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

Output to stream

({1}, [2,0])

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

Output to stream

({0,2}, [1])

({2}, [0,1])

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

Output to stream

({0}, [2,1])

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

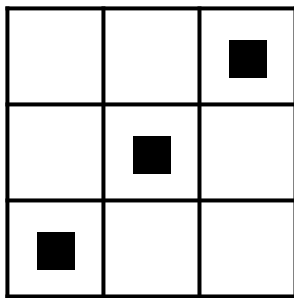
Output to stream

({0,1}, [2])

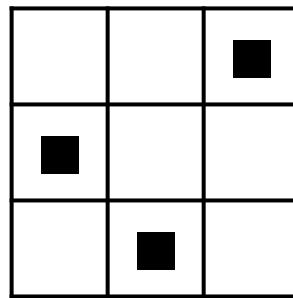
...

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

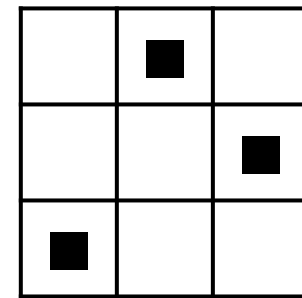
- Uses each column (position) exactly once
- Uses each row (number) exactly once



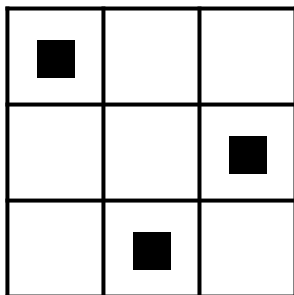
[2, 1, 0]



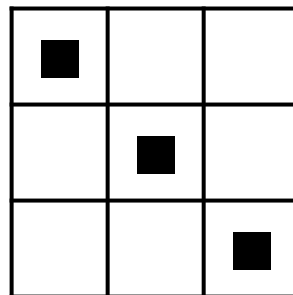
[1, 2, 0]



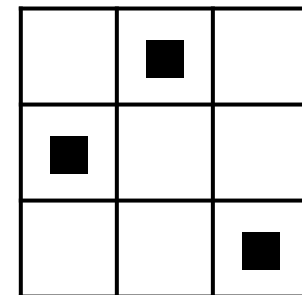
[2, 0, 1]



[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)));  
}
```

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, d2-1, tail.next);  
}
```

Diagonal
check

.parallel()

- Simple, and parallelizable for free
- Solve and 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);
```

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

Example174.java

Streams for quasi-infinite sequences

- van der Corput numbers
 - $1/2, 1/4, 3/4, 1/8, 5/8, 3/8, 7/8, \dots$
 - Dense and uniform in interval $[0, 1]$
 - For simulation and finance, Black-Scholes options
- Trick: v d Corput numbers as binary fractions
 $0.1, 0.01, 0.11, 0.001, 0.101, 0.011, 0.111 \dots$
are bit-reversals of $1, 2, 3, 4, 5, 6, 7, \dots$ 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

Collectors: aggregation of streams

- To format an IntList as string "[2, 3, 5, 7]"
 - Convert the list to an IntStream
 - Convert each element to get Stream<String>
 - Use a predefined Collector to build final result

```
public String toString() {  
    return stream(this).mapToObj(String::valueOf)  
        .collect(Collectors.joining(", ", "[", "]"));  
}
```

Example182.java

```
public static String toString(IntList xs) {  
    StringBuilder sb = new StringBuilder();  
    sb.append("[");  
    boolean first = true;  
    while (xs != null) {  
        if (!first)  
            sb.append(", ");  
        first = false;  
        sb.append(xs.item);  
        xs = xs.next;  
    }  
    return sb.append("]").toString();  
}
```

The alternative "direct" solution requires care and cleverness

Parallel array operations

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

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

- 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);
```

- 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 ...

- Side-effect free 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
- Array: eager, finite, materialized, use-many-times, parallel prefix scans

Some problems with streams

- Streams are use-once & have other restrictions
 - Probably to permit easy parallelization
- Hard to create lazy finite streams
 - Probably to allow 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 bug, or very strange behavior, in `parallel + limit`
- Laziness is subtle, easily goes wrong:

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

Closes the reader too early, so any use of the `Stream<String>` causes `IOException: Stream closed`

Example216.java

Useless

This week

- Reading
 - Java Precisely 3rd ed. §11.13, 11.14, 23, 24, 25
- Exercises
 - Extend immutable list class with functional programming; use parallel array operations; use streams of words and streams of numbers
- Read before next week's lecture
 - Sestoft: Microbenchmarks in Java and C#
 - Optional: McKenney chapter 3