Advanced Programming: Java 8 Stream Pipelines

Peter Sestoft
IT University of Copenhagen

Thursday 1 October 2015

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 book homepage (3rd edition): http://www.itu.dk/people/sestoft/javaprecisely/

Plan for today

- **New features in Java 8**
- Stream teaser
- Java 8 functional programming
- Streams, pipelines and parallel pipelines
- Parallel array operations
- Streams for backtracking and search
- Solving and generating sudoku puzzles

New in Java 8

- Lambda expressions

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

- Method reference expressions

```
String::length
```

- Functional interfaces = function types

```
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)
```

Blelloch-style parallel scan

Java 8 streams for bulk data

- Stream<T> is a finite or infinite sequence of T
  - Possibly lazily generated, else eager
  - Possibly parallel, else sequential
  - Possibly ordered, else unordered
  - Possibly infinite, else finite
- 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
  - Java iterators, but very different implementation
  - The extension methods underlying .NET (P)Linq

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❖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

Counting primes on Java 8 streams

- Classic Java/C/C++/C# 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 ☺

Streams for floating-point sums

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

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

- For-loop, backwards summation

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

Different results?

- A stream pipeline, and a parallel one

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

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

Different results?

TestStreamSums.java

Results: execution time (ms)

Method	Intel i7 (4 c)	AMD Opteron (32 c)
Forwards for-loop	5408	6707
Backwards for-loop	7140	6489
Stream	8570	16887
Parallel stream	2044	759

- Stream is slower than for-loop
- Parallel stream is faster than loop 2.6x – 8.8x
- But stream sum is *more precise than loop*!!??
  - The summation in sum() reduces impact of parallel execution, and guarantees precision in general
  - See exercises

Plan for today

- New features in Java 8
- Stream teaser
- **Java 8 functional programming**
- Streams, pipelines and parallel pipelines
- Streams for backtracking and search
- Parallel array operations
- Solving and generating sudoku puzzles

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()));
```

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;

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, 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;`

      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

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()`

```java
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(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

Plan for today

- New features in Java 8
- Stream teaser
- **Java 8 functional programming**
- Streams, pipelines and parallel pipelines
- Streams for backtracking and search
- Parallel array operations
- Solving and generating sudoku puzzles

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);
```

Example165.java

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]++);
```

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 quasi-infinite sequences

- van der Corput numbers
  - 1/2, 1/4, 3/4, 1/8, 5/8, 3/8, 7/8, ...
  - 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 ...

  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

Collectors: aggregation of streams

- To format an IntList as string "`[2, 3, 5, 7]`"
  - Convert the list to an IntStream
  - Convert each int element, to get a 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 must handle first element specially

Java 8 streams vs Scala lazy lists

Java 8 stream	Scala (C&B §5) lazy list
Values generated on demand	Values generated on demand
Values consumable just once	Values consumable many times
May be unordered	Always ordered
Easily parallelizable	Inherently sequential
Little risk of space leaks	High risk of space leaks
Hard to merge sorted streams	Easy to merge sorted lists
Hard to lazily create finite stream	Easy to lazily create finite list
Side effects are bad	Side effects are bad

Plan for today

- New features in Java 8
- Stream teaser
- Java 8 functional programming
- Streams, pipelines and parallel pipelines
- **Streams for backtracking and search**
- Parallel array operations
- Solving and generating sudoku puzzles

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], [1,0,2]

Set of numbers not yet used

A partial 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-set

partial permutation

({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

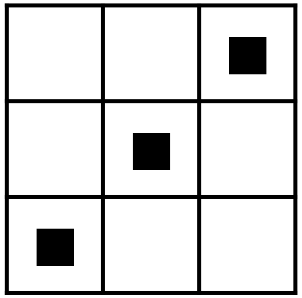
({0}, [2,1])

({}, [0,2,1]) Output to stream

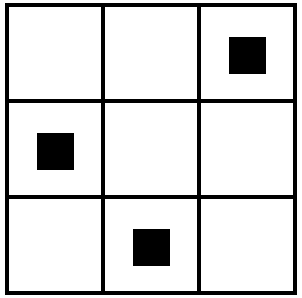
({0,1}, [2]) ...

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

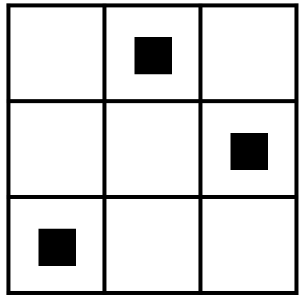
- Uses each column (position) exactly once
- Uses each row (number) exactly once



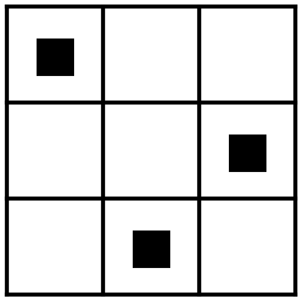
[2, 1, 0]



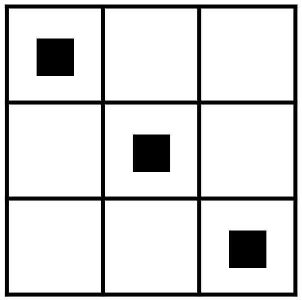
[1, 2, 0]



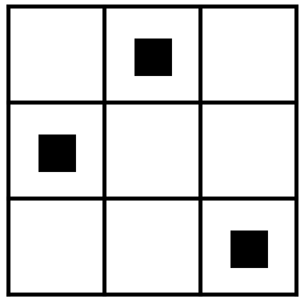
[2, 0, 1]



[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)));
}
```

Diagonal check

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);
}
```

`.parallel()`

- Simple, and parallelizable for free!

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

Plan for today

- New features in Java 8
- Stream teaser
- Java 8 functional programming
- Streams, pipelines and parallel pipelines
- Streams for backtracking and search
- **Parallel array operations**
- Solving and generating sudoku puzzles

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

Example25.java

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
- Hard (impossible?) to lazily merge two lazy streams
- Unclear how to control resource consumption
- A single side-effect may mess up everything completely
- Sometimes **`.parallel()`** hurts performance a lot
  - And bad interaction of generate+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

Monads in Java …

- Interface Stream<T>
  - with unit(x) = **`Stream.of(x)`**
  - with bind(f, xs) = **`xs.flatMap(f)`**

  flatMap(xs)(f)

  where Stream.of(x).flatMap(f).equals(f.apply(x))
  and xs.flatMap(Stream::of).equals(xs)

- Class Optional<T>
  - with unit(x) = **`Optional.of(x)`**
  - with bind(f, opt) = **`opt.flatMap(f)`**

  where Optional.of(x).flatMap(f).equals(f.apply(x))
  and opt.flatMap(Optional::of).equals(opt)

Plan for today

- New features in Java 8
- Stream teaser
- Java 8 functional programming
- Streams, pipelines and parallel pipelines
- Streams for backtracking and search
- Parallel array operations
- **Solving and generating sudoku puzzles**

Sudoku: rules and example

- Every row must contain 1, 2, ..., 9
- Every column must contain 1, 2, ..., 9
- Every block must contain 1, 2, ..., 9

1								3
		8	3	2	5	7		
	2		7		6		8	
	6		1		3		9	
	9	5	6	4	8	3	7	
	7		5		2		4	
	1		2		4		3	
		2	9	3	7	1		
7								4

Block 2

012
345
678

s-example

Representation of a puzzle

1								3
		8	3	2	5	7		
	2		7		6		8	
	6		1		3		9	
	9	5	6	4	8	3	7	
	7		5		2		4	
	1		2		4		3	
		2	9	3	7	1		
7								4

```
class State {
  private final int[][] board = new int[9][9];
  public int get(int r, int c) {
    return board[r][c];
  }
  ...
}
```

Row r=0…8,
column c=0…8

0 means blank,
else 1…9

AdproSudoku.java

A simple “greedy” sudoku solver

- For each blank cell, find set of candidates
  - If empty, puzzle has no solution; fail or backtrack
  - If exactly one number, fill it in; can’t be wrong
- Repeat
  - until no more blanks: a solution has been found
  - or until all blanks have >= 2 candidates: clone board, choose a candidate, fill it in and continue; *and be prepared to get back and try another one*

Representing subsets of { 1, 2, … 9 }

- Bit pattern, least significant bit represents {1}
  - eg …000001100 = { 3, 4 }
  - eg **0x1FF** = 111111111 = { 1, …, 9 } = the full set
- Set operations:
  - **s1 & s2** = set intersection
  - **s1 | s2** = set union
  - **~S & 0x1FF** = set complement
- The singleton set { n } is:

```
private static int singleton(int n) {
  return n == 0 ? 0 : 1 << (n - 1);
}
```

Numbers that *could* be in cell (r,c)

- The set of numbers not yet used in row r:

```
public int R(int r) {
  int used = ~0x1FF;
  for (int c=0; c<9; c++)
    used |= singleton(get(r, c));
  return ~used;
}
```

- C(c) = not yet used in column c
- B(b) = not yet used in block b
- First suggestion for candidates for cell (r, c):

```
public int A(int r, int c) {
  return R(r) & C(c) & B(b(r, c));
}
```

- Example: cell (8,3) can only contain 8

AdproSudoku.java

Exclusion: what *must* be in ...

- Numbers that must be in column c of block b = those that cannot be used in c outside b

```
public int CB(int c, int b) {
  int res = C(c);
  for (int r=0; r<9; r++)
    if (get(r, c) == 0 && b(r, c) != b)
      res &= ~R(r);
  return res;
}
```

- Example: col 4 block 1 must contain 3
- So cell (2,4) must have X = 3

```
-------------------------
|   5   | 4     |   3 2 |
|   3 4 |   2   | 7   6 |
|   9 2 | 6 X   | 5 8 4 |
-------------------------
|   8   |   5 4 |   7   |
| 9 7 5 | 8 6 1 | 2 4 3 |
|   4   |   9   | 8 6 5 |
-------------------------
| 5 6 7 |   4   | 3   8 |
| 4 1 9 |   8   | 6 2 7 |
| 3 2 8 |     6 | 4 5   |
-------------------------
```

s1split

Exclusion: what *must* be in cell (r, c)

- Encoding of some sudoku laws:

```
public int E(int r, int c) {
  int r0 = r/3*3, r1 = r0+(r+1)%3, r2 = r0+(r+2)%3;
  int c0 = c/3*3, c1 = c0+(c+1)%3, c2 = c0+(c+2)%3;
  int b = b(r, c), block = B(b);
  int rex = (~R(r1) & ~R(r2) | RB(r, b)) & block,
      cex = (~C(c1) & ~C(c2) | CB(c, b)) & block;
  int Erc = rex & cex;
  Erc |= get(r, c2)!=0 ? rex & ~C(c1) : 0;
  Erc |= get(r, c1)!=0 ? rex & ~C(c2) : 0;
  Erc |= get(r2, c)!=0 ? ~R(r1) & cex : 0;
  Erc |= get(r1, c)!=0 ? ~R(r2) & cex : 0;
  Erc |= get(r, c1)!=0 && get(r, c2)!=0 ? rex : 0;
  Erc |= get(r1, c)!=0 && get(r2, c)!=0 ? cex : 0;
  return Erc;
}
```

If empty we know nothing; if non-empty, one of the numbers must be in (r,c)

- Example: cell (6,8) must contain one of { 7 }

Finding a cell (r,c) to fill in

```
protected BranchPoint deterministic() {
  BranchPoint bp = null;
  for (int r=0; r<9; r++)
    for (int c=0; c<9; c++)
      if (get(r, c) == 0) {
        int Arc = A(r, c), Erc = E(r, c),
            cand = Erc == 0 ? Arc : Arc & Erc;
        ArrayList<Integer> possible = members(cand);
        if (possible.size() == 1) {
          setDestructive(r, c, possible.get(0));
          return null;
        } else if (bp == null || possible.size() < bp.possible.size())
            bp = new BranchPoint(r, c, permutation(cand));
      }
  return bp;
}
```

Candidates for (r,c)

Null if no blanks, else a branchpoint

- Repeat until all uniquely determined cells filled

```
BranchPoint bp = null;
while (bp == null && state.blankCount() > 0)
  bp = state.deterministic();
```

Using lazy streams for backtracking

```
public Stream<State> solutions() {
  State state = new State(this);
  BranchPoint bp = null;
  while (bp == null && state.blankCount() > 0)
    bp = state.deterministic();
  if (state.blankCount() == 0)
    ... to do ...
  else {
    int r = bp.r, c = bp.c;
    return ... to do ...
  }
}
```

Fill all uniquely determined cells

If no blanks left, we have a solution

Else try all cell (r, c)'s candidates

Creating sudoku puzzles

- Requirements:
  - A puzzle must have exactly one solution
  - To be solvable by humans, puzzles must not require too much backtracking (guessing)
- General idea:
  - Start with a complete filled-in solution
  - Randomly erase numbers so long as there is a single solution, and (eg.) no backtracking needed
    - Use the solver to determine the latter properties

Creating complete solutions

- Start with a random "skeleton", eg
  - Random filled-in block 0
  - Random filled-in (rest of) row 0 and column 0
  - Random filled-in block 3 and 8 and 4
- Use solver to find complete solution, if any

```
-------------------------------
| 8  7  6 | 1  9  2 | 5  3  4 |
| 9  1  3 |         |         |
| 4  2  5 |         |         |
-------------------------------
| 2       | 7       |    1    |
| 5       | 6       |    8    |
| 3       | 4       |    9    |
-------------------------------
| 1       |         | 9       |
| 7       |         | 8       |
| 6       |         | 2       |
-------------------------------
```

```
-------------------------------
| 8  7  6 | 1  9  2 | 5  3  4 |
| 9  1  3 | 5  4  7 | 6  2  8 |
| 4  2  5 | 3  8  6 | 1  7  9 |
-------------------------------
| 2  4  8 | 7  5  9 | 3  1  6 |
| 5  9  7 | 6  3  1 | 4  8  2 |
| 3  6  1 | 4  2  8 | 7  9  5 |
-------------------------------
| 1  8  4 | 2  7  5 | 9  6  3 |
| 7  3  2 | 9  6  4 | 8  5  1 |
| 6  5  9 | 8  1  3 | 2  4  7 |
-------------------------------
```

- At least $9! * 6!^2 * 3!^2$ = 6,772,211,712,000 different skeletons

Creating puzzles with Java streams

- Make lazy infinite streams of skeletons
- Try to complete each skeleton, returning at most one completion per skeleton
- Result: an "infinite" stream of solved sudokus

```
Stream<State> skeleta
   = Stream.generate(() -> makeSkeleton());
Stream<State> sudokus
   = skeleta.flatMap(s -> s.solutions().limit(1));
```

- Erase from each solution to get enough blanks

```
Stream<State> puzzles
  = sudokus.flatMap(s -> new State(s.board)
                        .eraseEnough(blanks).limit(1));
```

At most one, else many puzzles have same solutions, boring

How to erase enough

- If the puzzle has enough blanks, return it
- Else
  - try several ways to erase a random cell, and
  - recursively erase enough from each of those,
  - until we have one puzzle with enough blanks

```
protected Stream<State> eraseEnough(int blanks) {
  if (blankCount() >= blanks)
    return ... to do ...
  else
    return eraseOne()
          .flatMap(... to do ...);
}
```

- Result is an infinite stream of puzzles

How to erase one more cell

- To erase one more cell
  - choose a few (3) non-blank cells at random
  - try to erase each of them
  - keep those for which there is still just 1 solution

```
protected Stream<State> eraseOne() {
  ArrayList<Cell> nonBlanks = nonBlanks();
  ... to do ...
}
```

Using the infinite stream of puzzles

- Disregard puzzles that require backtracking

```
Stream<State> puzzles = State.make(blanks)
    .filter(s -> s.solutions().findAny().get().getGuesses()==0);
```

- Print puzzles (& solution) during generation

```
puzzles.forEach(s ->
         { s.print(); s.solutions().findAny().get().print(); });
```

- Collect **n** puzzles in a **State[]** array, then generate Latex code for puzzles and solutions

```
latexDoc(puzzles.limit(n).toArray(State[]::new));
```

- Java 8 streams are great for backtracking, and help separating concerns

This week

- Reading
  - Java Precisely 3rd ed. §11.13, 11.14, 23, 24, 25
- Exercises
  - Use streams of words and streams of numbers
- Mini-project
  - Solve sudoku puzzles, and generate sudoku puzzles, using streams
  - more ...