Java (8)

Brian Vermeer (@BrianVerm)

Java

Java 5 (Tiger) - Sept 2004

Java 6 (Mustang) - Dec 2006

Java 7 (Dolphin) - July 2011

Java 8 (Spider) - March 2014

Java 9 - Sept 2017

```
int million = 1_000_000;
```

Java

Java 5 (Tiger) - Sept 2004

Java 6 (Mustang) - Dec 2006

Java 7 (Dolphin) - July 2011

Java 8 (Spider) - March 2014

Java 9 - Sept 2017

Java 8

March 2014

Working from a more functional perspective

Some “life changing” code constructions

In retrospect:
What is GOOD, BAD and UGLY

Brian Vermeer

Software Engineer

blue4IT

Disclaimer

Best Practices are based on opinions

Different conclusions are possible

Think, and make your own judgement

Why Java 8

Doing things in parallel

Less code

Minimise Errors

New (functional) solutions

Lambda

Anonymous Inner Function.
(Nameless function without boilerplate code)

Satisfying a Functional Interface
(Interface with basically 1 abstract method)

<parameters> **->** <function body>

Functional Interfaces

```
public interface Predicate<T> {
    boolean test(T t);
}

public interface Function<T,R> {
    R apply(T t);
}

public interface Consumer<T> {
    void accept(T t);
}

public interface Supplier<T> {
    T get();
}
```

Functional Interfaces

```
public Predicate<Integer> pred =  i -> i % 2 == 0;

public Consumer<Integer> cons = i -> System.out.println(i);

public Supplier<Double> sup = () -> Math.random();

public Function<Integer, String> func = i -> "Input is " + i;
```

Functional Interfaces

```
public Predicate<Integer> pred =  i -> i % 2 == 0;

public Consumer<Integer> cons = i -> System.out.println(i);

public Supplier<Double> sup = () -> Math.random();

public Function<Integer, String> func = i -> "Input is " + i;


public BiFunction<String, String, String> bifunc1 =
 (str1, str2) -> str1 + "-" + str2;
```

Functional Interfaces

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}

public interface Callable<V> {
    V call() throws Exception;
}

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

Java 7 Comparator

```
List<Beer> beers = new ArrayList<>();
beers.add(new Beer("La Chouffe", 8.0));
beers.add(new Beer("Duvel", 8.5));
beers.add(new Beer("Jupiler", 5.2));


Collections.sort(beers, new Comparator<Beer>() {
    @Override
    public int compare(Beer b1, Beer b2) {
        return b1.getName().compareTo(b2.getName());
    }
});
```

Java 7 Comparator

```
List<Beer> beers = new ArrayList<>();
beers.add(new Beer("La Chouffe", 8.0));
beers.add(new Beer("Duvel", 8.5));
beers.add(new Beer("Jupiler", 5.2));


Collections.sort(beers, new Comparator<Beer>() {
    @Override
    public int compare(Beer b1, Beer b2) {
        return b1.getName().compareTo(b2.getName());
    }
});
```

Java 8 Comparator

```
List<Beer> beers = new ArrayList<>();
beers.add(new Beer("La Chouffe", 8.0));
beers.add(new Beer("Duvel", 8.5));
beers.add(new Beer("Jupiler", 5.2));)


Collections.sort(beers, (b1,b2) -> b1.getName().compareTo(b2.getName()));

beers.sort((b1,b2) -> b1.getName().compareTo(b2.getName()));
```

Abstraction

```
private void logUpper(String str) {
    //super interesting code
    System.out.println(str.toUpperCase());
    //even more interesting code
}

private void logLower(String str) {
    //super interesting code
    System.out.println(str.toLowerCase());
    //even more interesting code
}
```

Abstraction

```
private void logUpper(String string) {
    doFoo(string, s -> s.toUpperCase());
}

private void logLower(String string) {
    doFoo(string, s -> s.toLowerCase());
}

private void doFoo(String str, Function<String, String> func) {
    //super interesting code
    System.out.println(func.apply(str));
    //even more interesting code
}
```

Writing a Lamda

```
//Single line Lambda

int1 -> (int1 / 2) * 3;
```

```
//Multi line Lambda (block lambda)

int1 -> { //do Some code
    // do Some more code
    // return something
    }
```

Writing a lambda

```
Consumer<String> con1  = str -> System.out.println(str);

Consumer<String> con1 = System.out::println;
```

Writing a lambda

```
Consumer<String> con1  = str -> System.out.println(str);

Consumer<String> con1 = System.out::println;
```

```
Consumer<String> con1  = str -> System.out.println("- " + str);

~~Consumer<String> con1 = System.out::println;~~
```

Puzzler

```
List<Beer> beers = new ArrayList<>();
beers.add(new Beer("La Chouffe", 8.0));
beers.add(new Beer("Duvel", 8.5));
beers.add(new Beer("Jupiler", 5.2));)

beers.sort((b1,b2) -> b1.getAlcohol().compareTo(b2.getAlcohol()));  //1
beers.sort((b1,b2) -> { b1.getAlcohol().compareTo(b2.getAlcohol());}); //2
```

A) Line 1 compiles , Line 2 does not.
B) Line 2 compiles , Line 1 does not.
C) Both lines will compile.
D) I don’t care, just give me one of these beers.

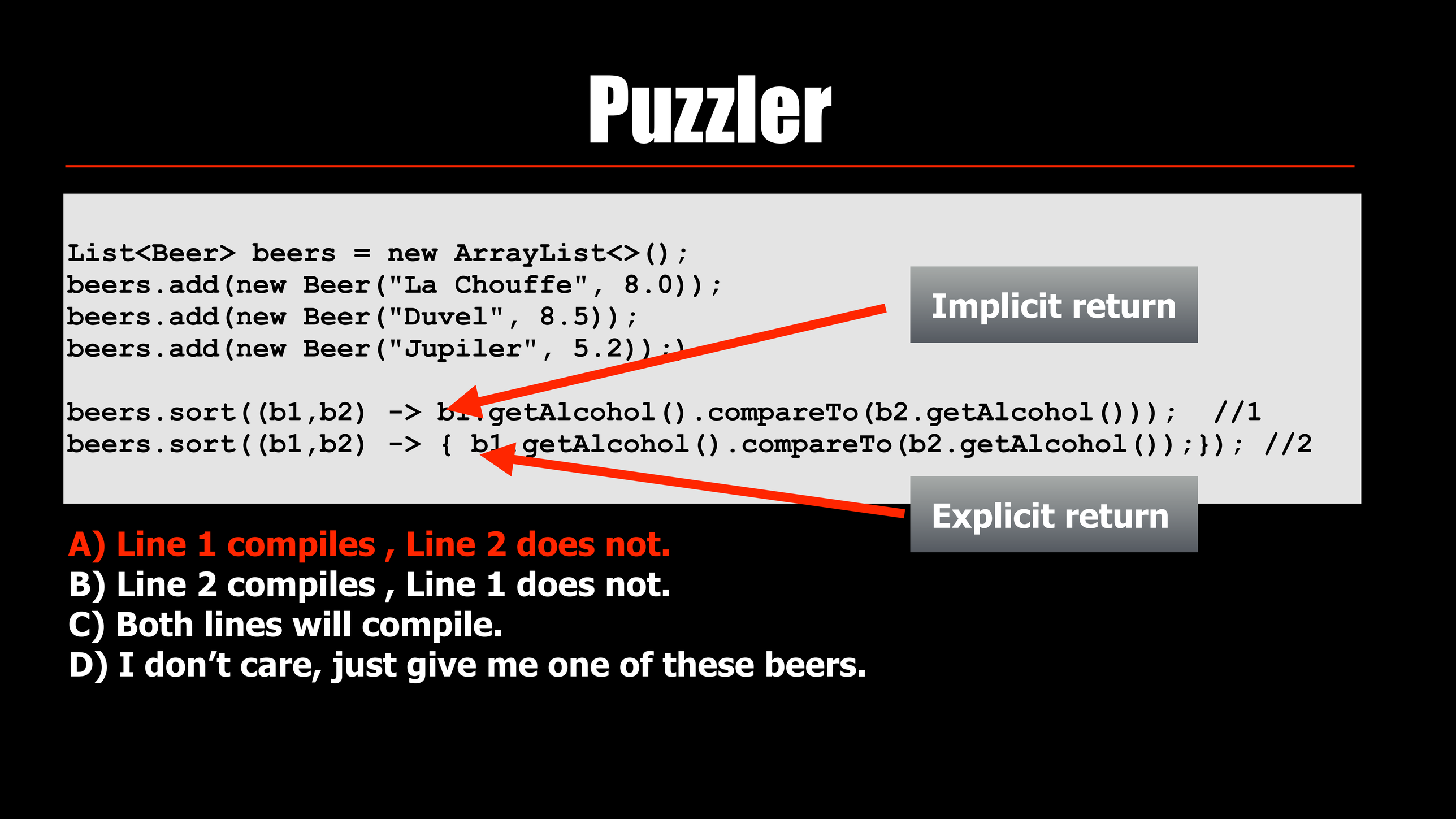
Puzzler

```
List<Beer> beers = new ArrayList<>();
beers.add(new Beer("La Chouffe", 8.0));
beers.add(new Beer("Duvel", 8.5));
beers.add(new Beer("Jupiler", 5.2));)

beers.sort((b1,b2) -> b1.getAlcohol().compareTo(b2.getAlcohol()));  //1
beers.sort((b1,b2) -> { b1.getAlcohol().compareTo(b2.getAlcohol());}); //2
```

A) Line 1 compiles , Line 2 does not.
B) Line 2 compiles , Line 1 does not.
C) Both lines will compile.
D) I don’t care, just give me one of these beers.

Puzzler

```
List<Beer> beers = new ArrayList<>();
beers.add(new Beer("La Chouffe", 8.0));
beers.add(new Beer("Duvel", 8.5));
beers.add(new Beer("Jupiler", 5.2));

beers.sort((b1,b2) -> b1.getAlcohol().compareTo(b2.getAlcohol()));  //1
beers.sort((b1,b2) -> { b1.getAlcohol().compareTo(b2.getAlcohol());}); //2
```



A) Line 1 compiles , Line 2 does not.
B) Line 2 compiles , Line 1 does not.
C) Both lines will compile.
D) I don’t care, just give me one of these beers.

Block Lambda

```
str1 -> {
    System.out.println("Hello world");
    return str1 + System.currentTimeMillis();
};
```

Don’t do Block Lambda

```
str1 -> {
    System.out.println("Hello world");
    return str1 + System.currentTimeMillis();
};
```

```
str1 -> myMethod(str1);

private String myMethod(final String str1) {
    System.out.println("Hello world");
    return str1 + System.currentTimeMillis();
}
```

Don’t do Block Lambda

```
beer -> {
    String name;
    if (beer.getName().contains(" ")) {
        name = beer.getName().replace(" ", "");
    } else {
        name = beer.getName();
    }

    try {
        name += Integer.parseInt(beer.getAlcoholPrecentage().toString());
    } catch (NumberFormatException nfe) {
        name += beer.getAlcoholPrecentage();
    }

    return name;
}
```

Type inference

Make use of automatic type inferencing.

Only specify types if compiler does not understand

```
(Beer b1,Beer b2) -> b1.getAlcohol().compareTo(b2.getAlcohol())


(b1,b2) -> b1.getAlcohol().compareTo(b2.getAlcohol())
```

Parentheses

Dont’s use parentheses if not needed

(6-3)+2 is the same as 6 - 3 + 2

```
(int1) -> int1 * 2 - 1


int1 -> int1 * 2 - 1
```


Exceptions

Exception handling

How can I throw CHECKED exceptions from inside Java 8 streams/ lambdas?

Exception handling

How can I throw CHECKED exceptions from inside Java 8 streams/ lambdas?

The simple answer to your question is:
You can't, at least not directly.

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { …}

Function <Beer,Beer> fBeer = beer -> doSomething(beer)
```

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { …}

Function <Beer,Beer> fBeer = beer -> doSomething(beer)
```

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { …}


beer -> {
        try{
            return doSomething(beer);
        } catch (IsEmptyException e) {
            throw new RuntimeException(e);
        }
};

//not very pretty
```

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { …}

private Beer wrappedDoSomeThing(Beer beer) {
    try{
        return doSomething(beer);
    } catch (IsEmptyException e) {
        throw new RuntimeException(e);
    }
}

beer -> wrappedDoSomeThing(beer)
```

Exception Utility

```
@FunctionalInterface
public interface CheckedFunction<T, R> {
    public R apply(T t) throws Throwable;
}


public static <T, R> Function<T, R> wrap(CheckedFunction<T, R> function) {
    return t -> {
        try {
            return function.apply(t);
        } catch (Throwable ex) {
            throw new RuntimeException(ex);
        }
    };
};

wrap(beer -> doSomething(beer))
```

Optional

Optional

New type added in Java 8

Wrapper class around a value that might be absent

Designed to tackle the NULL reference problem

Different opinions (awesome or ugly)

Optional

Optional<String>
- empty
- literal String (“Hello World”);

```
public String giveMeAString() { return “Hello World” }
public String giveMeAString() { return null }

public Optional<String> giveMeAString() { return Optional.of(“Hello World”);}
public Optional<String> giveMeAString() { return Optional.empty(); }
```

Optional

```
Optional<String> a = Optional.of(“Hello World”);

Optional<String> b = Optional.empty();

Optional<String> c = null //please avoid this
```

Be careful

Optional can be null

Always avoid this in your own code!

Using Optional

Don’t use unnecessary optionals

```
//avoid
if (Optional.ofNullable(str1).isPresent()) { … }

Optional.ofNullable(str1).ifPresent(x -> doSomething(x));

//instead
if (str1 == null) { … }
```

Using Optional

Use Optional in a more functional way

```
public Optional<Beer> findBeer(String name) { … }


Beer b = findBeer(name).orElse(Beer.DEFAULT);
Beer b1 = findBeer(name).orElseThrow(NotFoundException::new);
```

Optional

Optional can have 3 options, be aware ….

Never assign to null in your own code.

Choose when to use Optional en stick to that
- everywhere
- only what is publicly available
- don’t use it at all

Optional is designed as return type, not as input type

Streams

What is Stream (in Java8)

Not a data structure, not storage

Flow of data derived from a Collection

Intermediate result

Lazy evaluated by nature

Can transform data, cannot mutate data

Can create a pipeline of function that can be evaluated

JAVA 8 Streams

```
stringLists.stream()
        .map(str -> str.toUpperCase())
        .collect(Collectors.toList());
```

Intermediate

filter
distinct
map
flatMap
limit
skip
sorted
peek

Terminal

reduce
collect
toArray
count
max
min
findAny
findFirst
forEach
allMatch
anyMatch

Stream Example

```
List<Beer> beers = getBeers();
List<String> goodBeers = new ArrayList<>();
for (Beer beer : beers) {
    if (beer.getAlcohol() > 7.5) {
        goodBeers.add(beer.getName());
    }
}
```

Stream Example

```
List<Beer> beers = getBeers();
List<String> goodBeers = new ArrayList<>();
for (Beer beer : beers) {
    if (beer.getAlcohol() > 7.5) {
        goodBeers.add(beer.getName());
    }
}
```

```
List<Beer> beers = getBeers();
List<String> goodBeers = beers.stream()
        .filter(beer -> beer.getAlcohol() > 7.5)
        .map(beer -> beer.getName())
        .collect(Collectors.toList());
```

forEach ?

```
List<Beer> beers = getBeers();
List<String> goodBeers = new ArrayList<>();
for (Beer beer : beers) {
    if (beer.getAlcohol() > 7.5) {
        goodBeers.add(beer.getName());
    }
}
```

```
List<Beer> beers = getBeers();
List<String> goodBeers = new ArrayList<>();
beers.forEach(beer -> {
    if (beer.getAlcoholPrecentage() > 7.5) { goodBeers.add(beer); }
});

//bad idea
```

forEach ??

```
List<Beer> beers = getBeers();

//enrich with ratings
beers.forEach(beer -> beer.setRating(findRating(beer.getName())));

//enrich with reviews
beers.forEach(beer -> beer.setReviews(findReviews(beer.getName())));
```

```
List<Beer> beers = getBeers();
beers.stream()
     .map(beer -> beer.newBeerWithRating(findRating(beer.getName()))
     .map(beer -> beer.newBeerWithReviews(findReviews(beer.getName()))
     .collect(Collectors.toList())
```

Puzzler

```
List<String> teams = new ArrayList<>();
teams.add("alpha");
teams.add("bravo");
teams.add("charlie");
teams.add("delta");
Stream<String> teamsInMission = teams.stream();
teams.remove("bravo");
teams.add("echo");
teamsInMission.forEach(team -> System.out.println(team));
```

A) alpha, bravo, charlie, delta
B) alpha, bravo, charlie, delta, ConcurrentModificationException
C) alpha, charlie, delta, echo
D) ConcurrentModificationException

Puzzler

```
List<String> teams = new ArrayList<>();
teams.add("alpha");
teams.add("bravo");
teams.add("charlie");
teams.add("delta");
Stream<String> teamsInMission = teams.stream();
teams.remove("bravo");
teams.add("echo");
teamsInMission.forEach(team -> System.out.println(team));
```

A) alpha, bravo, charlie, delta

B) alpha, bravo, charlie, delta, ConcurrentModificationException

C) alpha, charlie, delta, echo

D) ConcurrentModificationException

Only use a Stream once

```
List<Beer> beers = getBeers();
Stream<Beer> beerStream = beers.stream();

beerStream.forEach(b ->System.out.println(b.getName())); //1

beerStream.forEach(b ->System.out.println(b.getAlcohol())); //2
```

Line 2 will give:

java.lang.IllegalStateException: stream has already been operated upon or closed

Consume the stream

```
beers.stream()
        .limit(10)
        .map(i -> i.getAlcohol())
        .peek(i -> {
            if (i > 7.0)
                throw new RuntimeException();
        });
```

Don’t forget to consume the stream!

Multiple filters

```
beers.stream()
        .filter(beer -> beer.getAlcohol() > 7.5)
        .filter(beer -> beer.getRating() > 4)
        .map(beer -> beer.getName())
        .map(beer -> beer.toUpperCase())
        .collect(Collectors.toList());
```

Infinite stream

```
IntStream.iterate(0, i -> i + 1)
         .forEach(i -> System.out.println(i));
```

Infinite stream

```
IntStream.iterate(0, i -> i + 1)
         .forEach(i -> System.out.println(i));
```

```
IntStream.iterate(0, i -> i + 1)
         .limit(10)
         .forEach(i -> System.out.println(i));
```

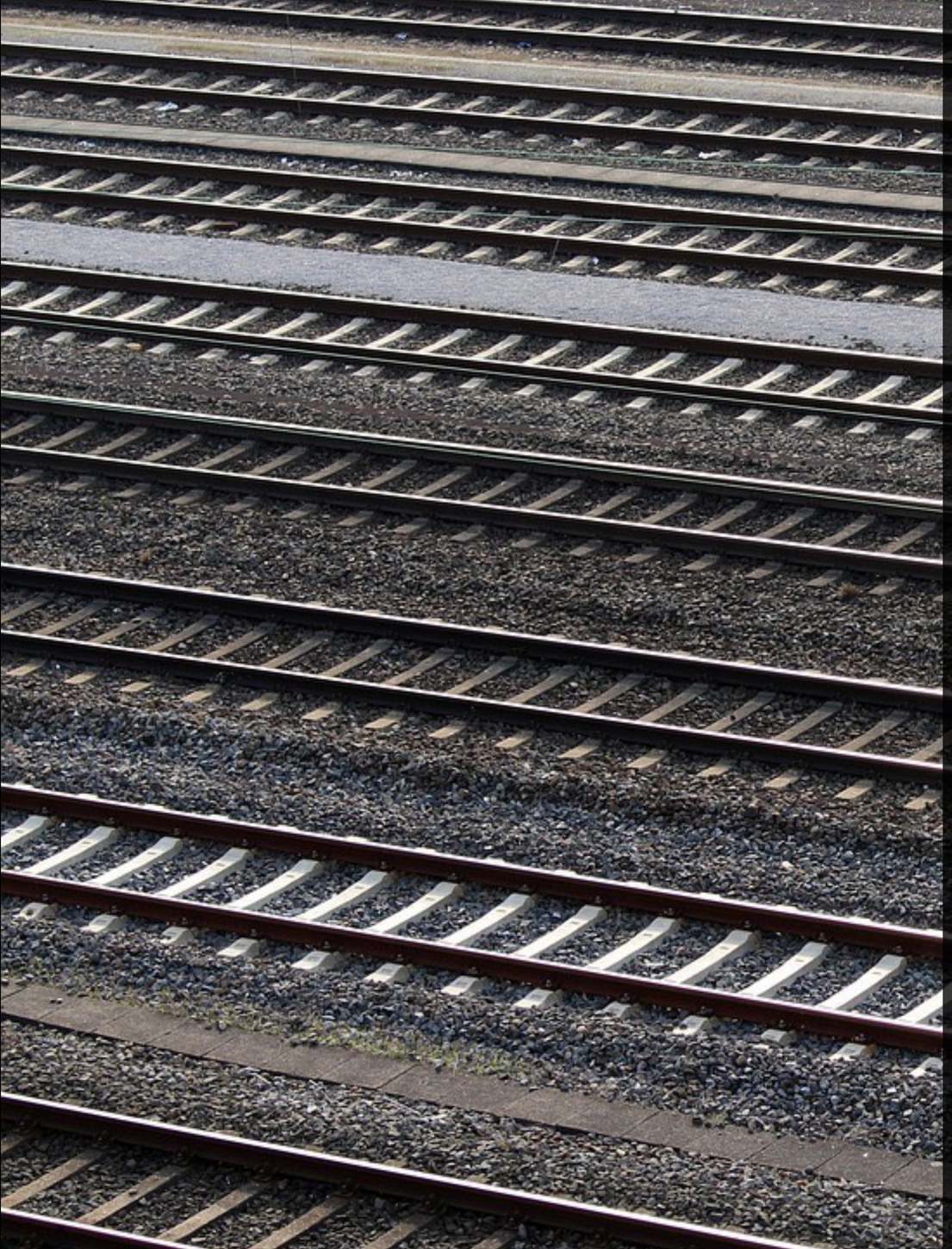
Infinite stream

```
//“subtle” mistake, this will run forever

IntStream.iterate(0, i -> ( i + 1) % 2)
         .distinct()
         .limit(10)
         .forEach(i -> System.out.println(i));
```

Infinite stream

```
//“subtle” mistake, this will run forever

IntStream.iterate(0, i -> ( i + 1) % 2)
        .distinct()
        .limit(10)
        .forEach(i -> System.out.println(i));
```

```
//parallel makes it even worse!!!!

IntStream.iterate(0, i -> ( i + 1) % 2)
        .parallel()
        .distinct()
        .limit(10)
        .forEach(i -> System.out.println(i));
```



Watch out with parallel

Watch out with parallel in general

Parallel stream all use the common fork-join thread pool

Parallel can slow you down.

Streams

Don’t replace every loop with Stream

Be careful with infinite streams

Only uses streams once

Be very careful with parallel

A stream is not a data structure

Stream pipelines don’t do anything until consumed

Summary

Java is a Object Oriented Language

Java is not a Functional Language

There are some nice new “functional” code constructions you can use

Use them with care

Don’t be a cowboy

More ...

Java 8 Puzzlers: The Strange, the Bizarre, and the Wonderful
vJug Session by **Baruch Sadogursky** and **Viktor Gamov**
https://youtu.be/lu4UTY940tg

Optional - The Mother of All Bikesheds
Devoxx 2016 by **Stuart Marks**
https://youtu.be/Ej0sss6cq14

Brian Vermeer

@BrianVerm
brian@brianvermeer.nl

blue4IT