Java™ 8

```
Imperative.map.toFunctional.forEach( developer -> uprade() )
```

Lund&Bendsen

developing developers

Program

- 0900 - 0915 Velkommen, energizer, icebreaker, teams
- 0915 - 1000 Lambda expressions
- 1015 - 1100 Method references
- 1115 - 1200 DateTime API og Optional, API design
- 1200 - 1230 Frokost pause
- 1230 - 1430 Streams og Collections
- 1445 - 1530 Code clinic - upgrading the Keylane code base
- 1530 - 1600 Read the Code Team Challenge

What is new in Java 8?

Brian Goetz top 0.30% this year

Java Language Architect at Oracle

Java Language Architect at Oracle.

Author, *Java Concurrency in Practice.*

@BrianGoetz on Twitter.

43,014 REPUTATION

● 11 ● 75 ● 101

Hello, functional!
Rewrite of Collections
Streams
Date and Time
Goodbye NullPointer Exceptions, hello Optional!

New Syntax

New APIs

Teams

About 3 developers.
Team Work Challenge: 1 point per round: exercise, group activity
Compete for the prizes!

Image credit: http://natguy.net/programming.html

Lambda Expressions

- A *lambda* is a code block which can be referenced and passed to another piece of code for future execution one or more times.
- Java 8 introduced *lambda expressions*, which offer a simple syntax to create and use lambdas.

```
(Integer i1, Integer i2) -> i1 % 2 - i2 % 2;
```

How is it fitted into the language?

- Allow interfaces with methods - default and static
  - A class implementing the interface does not have to implement the default method
  - A **default** methods in an interface- is a method with an implementation
  - A **static** method in a an interface is shared by all instances of the implementing class

```
default String getDescription() {
    return "This is a default method";
}
```

Functional Interface

- @FunctionalInterface used to mark interfaces
- Single-abstract-method (SAM) requirement
- Replaces anonymous inner classes in Java 7

```java
public interface Calculator {

    double calculate(int a, int b);

    public default int subtract(int a, int b) {
        return a - b;
    }

    public default int add(int a, int b) {
        return a * b;
    }

    @Override
    public String toString();
}
```

Lambda Expression Syntax

- Override the S-A-M
- Syntax for oneliners: (parameter list) -> expression
- Syntax for longer methods: (parameter list) -> { statements}
- Types are optional

```
Calculator division  = (int a, int b) -> (double) a / b;
System.out.println(division.calculate(5, 2)); // prints 2.5

// or (types are optional!)
Calculator division = (a,b) -> a/b;
```

Example

```java
package java.lang;

@FunctionalInterface
public interface Runnable {

    public abstract void run();
}

// in main method
Runnable task = () -> System.out.println("Hello " +
Thread.currentThread().getName()) ;
```

Predefined Functional Interfaces

java.util.function package contains predefined Functional Interfaces, that make it easier to write lambda expressions

- Function - Models a function that can take one parameter and return a result. The result can be of a different type than the parameter.

- Predicate - A Function that takes a parameter and returns **true** or **false** based on the value of the parameter.

- Supplier - Represents a supplier of results, ie no parameters and a result

- Consumer - An operation that takes a parameter and returns no result.

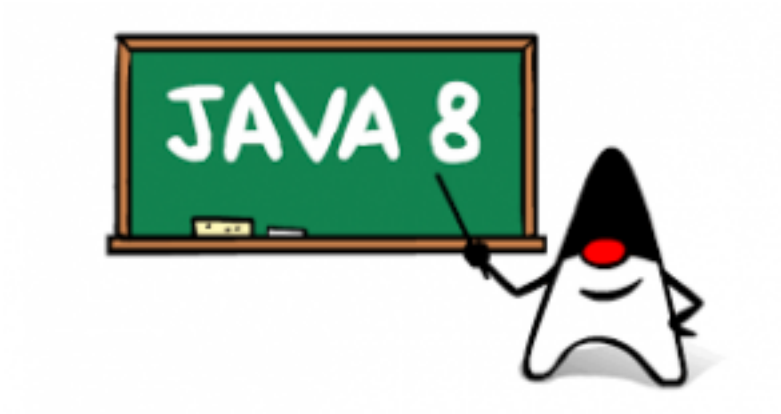
Example: Function Interface

- The **Function** interface is used to create a one-argument function that returns a result
- **Function** has one abstract method, **apply (S-A-M)**
- **See more on** https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

```
public interface Function<T, R>
R apply(T argument)
// in code
 Function<Integer, Double> milesToKms =  (input) -> 1.6 * input;
double kms = milesToKms.apply(miles);
```

Group Activity

Functional versus Imperative style



Programming Exercise!

Lesson_01_java_lambdas

Method References

- Reference a method that already exists
- Promotes reusable functional code
- Don’t write a new lambda each time - use method references

Method Reference explained

- A MethodReference is a way of providing a lambda
- You can use MethodReferences in API methods, that take a functional interface as an argument
- A method reference has shorter syntax than lambda expression - no definition - the method body is defined somewhere else
- You can use an existing method, thus promoting code reuse.
- Instead of using AN ANONYMOUS CLASS , use A LAMBDA EXPRESSION
- If a lambda expression just calls one method, useA METHOD REFERENCE
- Syntax: class name/object reference + double colon operator (::) + method name.

```
ClassName::staticMethodName
ContainingType::instanceMethod
objectReference::methodName
ClassName::new
```

```
//Lambda expression syntax
Consumer<String> c = s -> System.out.println(s);
//Method reference syntax
Consumer<String> c = System.out::println;
```

Method Reference Example

```
//Method in String class
public static String join(CharSequence delimiter, Iterable<? extends
CharSequence> elements)

@FunctionalInterface
    interface StringListFormatter {
        String format(String delimiter, List<String> list);
    }

    public static void formatAndPrint(StringListFormatter formatter,
            String delimiter, List<String> list) {
        String formatted = formatter.format(delimiter, list);
        System.out.println(formatted);
    }

    public static void main(String[] args) {
        List<String> names = Arrays.asList("Don", "King", "Kong");
        formatAndPrint(String::join, ", ", names);
    }
```

Static Method Reference

- Instead of: `(args) -> Class.staticMethod(args)`

- Use a static method reference: `Class::staticMethod`

- Instead of a lambda expression:
  `findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));`

- Use a method reference:
  `findNumbers(list, Numbers::isMoreThanFifty);`

Instance method of class

- An instance of an object is passed, and one of its methods is executed with some optional(s) parameter(s).

- Instead of lambda expression (obj, args) -> obj.instanceMethod(args)

- Use ObjectType::instanceMethod

```
//class
class Shipment {
  public double calculateWeight() {
    double weight = 0;
    // Calculate weight
    return weight;
  }
}
//method
public List<Double> calculateOnShipments(
  List<Shipment> l, Function<Shipment, Double> f) {
    List<Double> results = new ArrayList<>();
    for(Shipment s : l) {
      results.add(f.apply(s));
    }
    return results;
}
// Using a lambda expression
calculateOnShipments(l, s -> s.calculateWeight());

// Using a method reference
calculateOnShipments(l, Shipment::calculateWeight);
```

Instance method of existing object

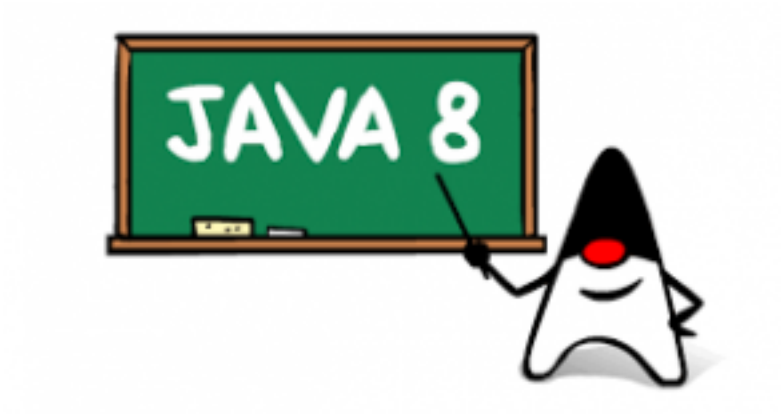
- Instead of lambda expression: (args) -> obj.instanceMethod(args)

- Use instance method reference obj::instanceMethod

```
class Car {
  private int id;
  private String color;}
class Mechanic {
  public void fix(Car c) {
    System.out.println("Fixing car " + c.getId());  }
}
// method accepts Functional Interface
public void execute(Car car, Consumer<Car> c) {
  c.accept(car);
}
// Using a lambda expression
execute(car, c -> mechanic.fix(c));

// Using a method reference
execute(car, mechanic::fix);
```

Constructor Reference

- A specialized form of method references, refers to constructor of a class
- Instead of lambda expression : `(args) -> new ClassName(args)`
- Use a Constructor References: `ClassName::new`
- Example: `Supplier<Integer> integerSupplier = Integer::new`



Programming Exercise!

Lesson_02_method_references

DateTime API

- JSR-310
- Immutable, Threadsafe
- Domain Driven Design
    - Extensible
    - Human readable
    - Machine readable
- Seperation of chronologies (different calendars)

Date and Time Classes

- LocalTime - local time from the context of the observer, like a clock on your wall
- LocalDate - local date, like a calendar on your desk
- LocalDateTime - composite class, pairing of `LocalDate` and `LocalTime`.
- TimeZone
- ZoneOffset

LocalTime, LocalDate, LocalDateTime

- Core classes in the `java.time` API
- Fluent factory methods
  - When constructing a value, the factory is called of
  - When converting from another type, the factory is called from
- parse methods that take strings as parameters

Examples

```
LocalDateTime timePoint = LocalDateTime.now(); // current date+time
LocalDate.of(2012, Month.DECEMBER, 12); // from values
LocalTime.of(17, 18); // the train I took home today
LocalTime.parse("10:15:30"); // From a String
```

TimeZone

- A *time zone* is a region of the earth where the same standard time is used.
- Each time zone is described by an identifier and usually has the format *region/city* (Asia/Tokyo) and an offset from Greenwich/UTC time.
- For example, the offset for Tokyo is +09:00.

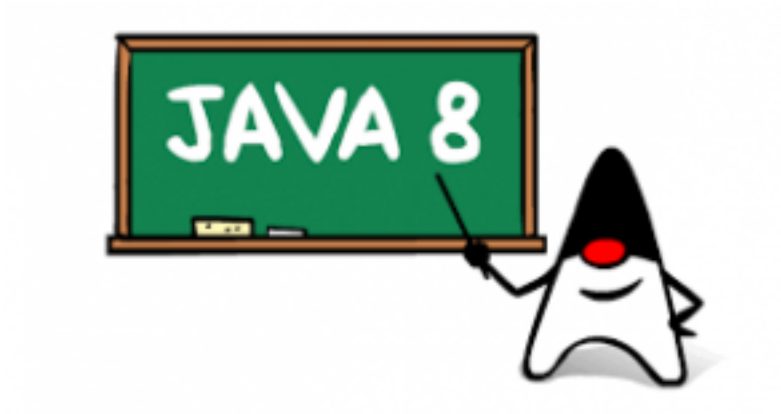
The Date-Time API provides three temporal-based classes that work with time zones:

- `ZonedDateTime` handles a date and time with a corresponding time zone with a time zone offset from Greenwich/UTC.
- `OffsetDateTime` handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.
- `OffsetTime` handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

Examples

```
// Flight is 10 hours and 50 minutes, or 650 minutes
ZoneId arrivingZone = ZoneId.of("Asia/Tokyo");
ZonedDateTime arrival = departure.withZoneSameInstant(arrivingZone)
                                 .plusMinutes(650);
```

```
Set<String> allZones = ZoneId.getAvailableZoneIds();
```



Programming Exercise!

Lesson_03_date_time

Optional

ARE YOU REALLY SURE THAT THIS VARIABLE CAN NEVER EVER BE NULL?
OF COURSE!!!

NullPointerException

- **Optional provides a way to get rid of null!** (Well, almost.)
- Say you have a NullPointerException
  - Where does the null reference come from ???
    - Maybe it is **missing initialization**?
    - Is it a **legal state** for that variable, ie **missing value**?
    - Maybe it's a return value of some crazy method aka **bug**?
- Optional helps you reveal your intention

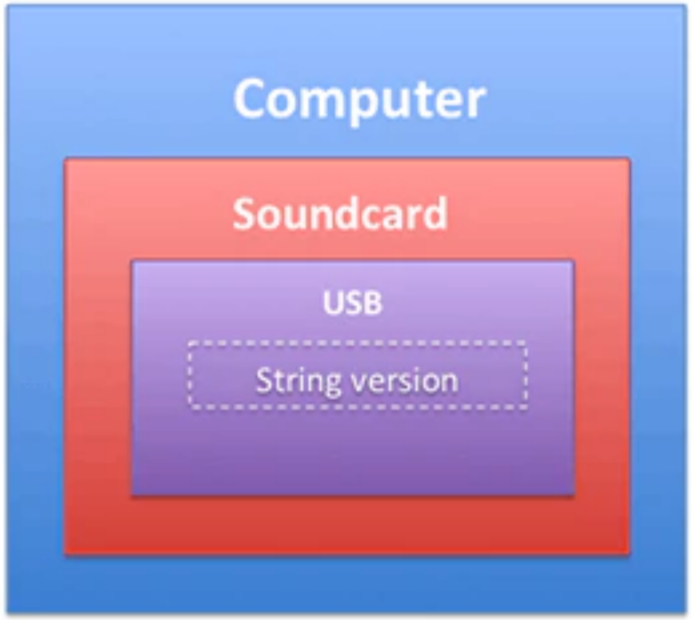
About Optional

- `Optional<T>` is a wrapper class for `T`
- May or may not contain a non-null value.
- Use it to design an API, that reveals your intentions
- If a value is present, `isPresent()` will return true
- `get()` will return the value.
- `orElse()` returns a default value, if value not present
- `ifPresent()` executes a block of code if the value is present

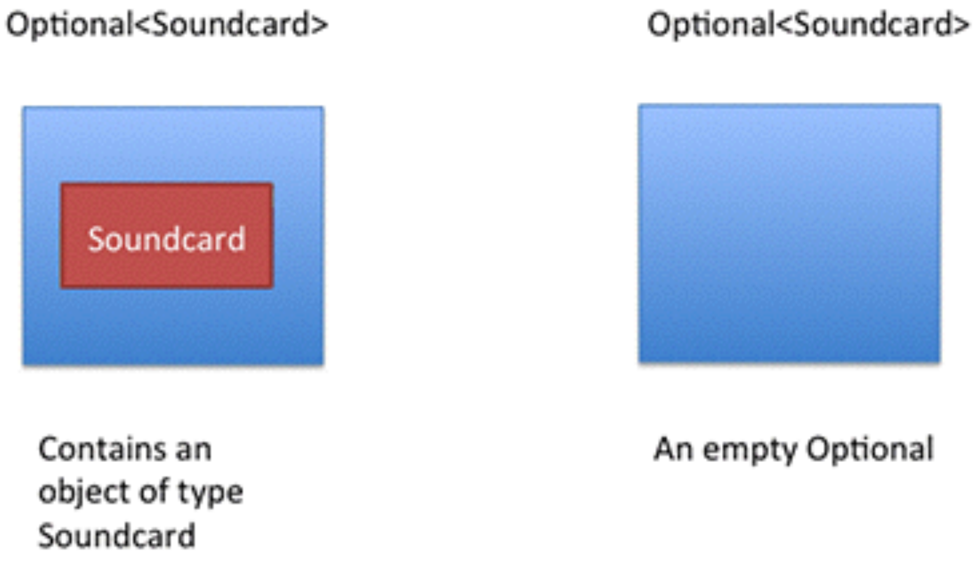
Construction

```
// an empty 'Optional';
// before Java 8 you would simply use a null reference here
Optional<String> empty = Optional.empty();

// an 'Optional' where you know that it will not contain null;
// (if the parameter for 'of' is null, a 'NullPointerException' is
thrown)
Optional<String> full = Optional.of("Some String");

// an 'Optional' where you don't know whether it will contain null or not
Optional<String> halfFull = Optional.ofNullable(someOtherString);
```

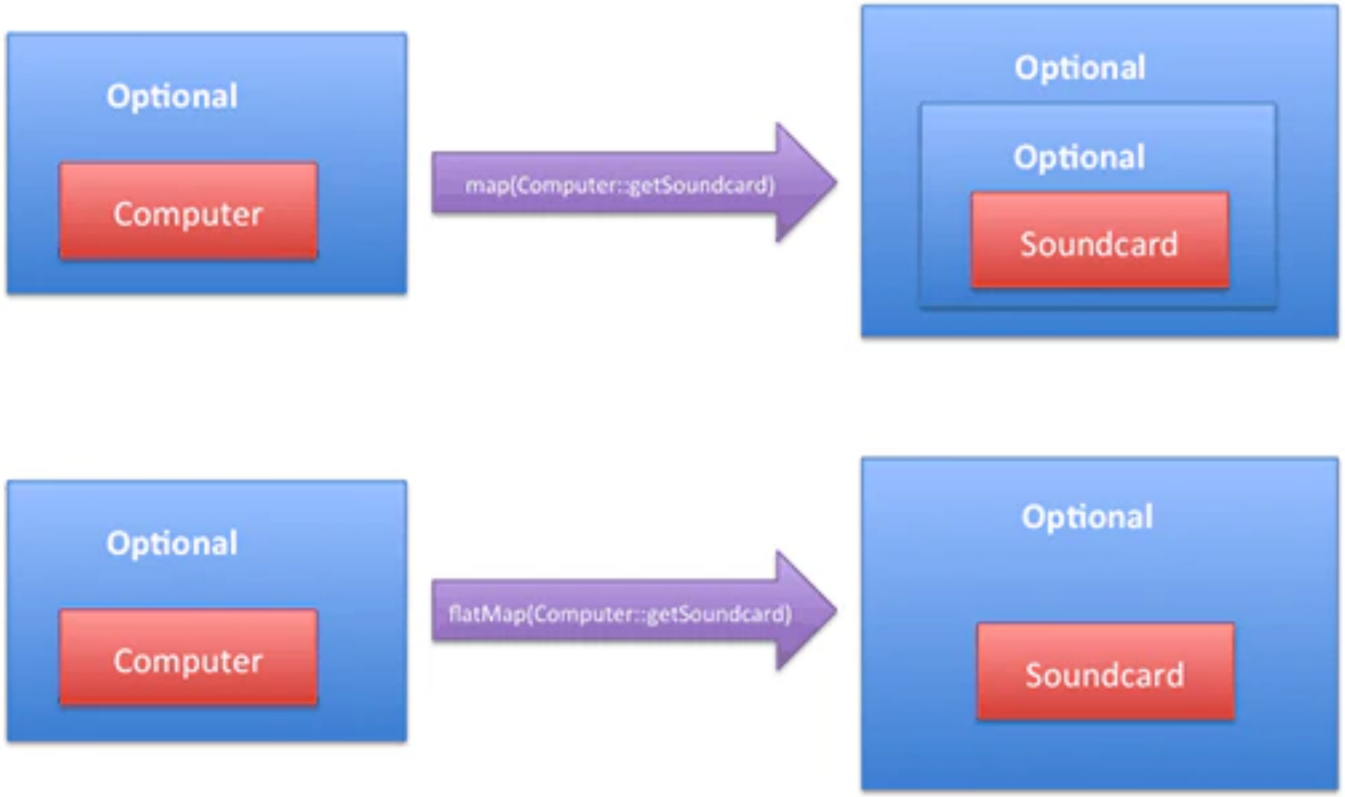
Example



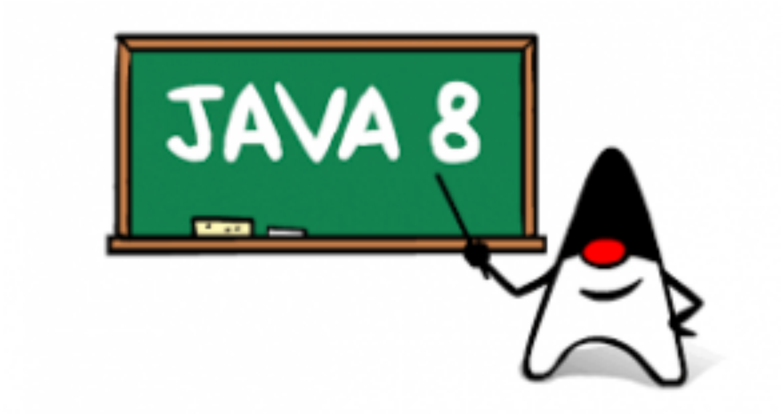
```
//Unsafe code
String version = computer.getSoundcard().getUSB().getVersion();

//With optional
public class Computer {
  private Optional<Soundcard> soundcard;
  public Optional<Soundcard> getSoundcard() { ... }
  ...
}
public class Soundcard {
  private Optional<USB> usb;
  public Optional<USB> getUSB() { ... }
}
public class USB{
  public String getVersion(){ ... }
}
```



Avoiding the null checks



```
String version = computer.flatMap(Computer::getSoundcard)
                         .flatMap(Soundcard::getUSB)
                         .map(USB::getVersion)
                         .orElse("UNKNOWN");
```



Programming Exercise!

Lesson_03_optional

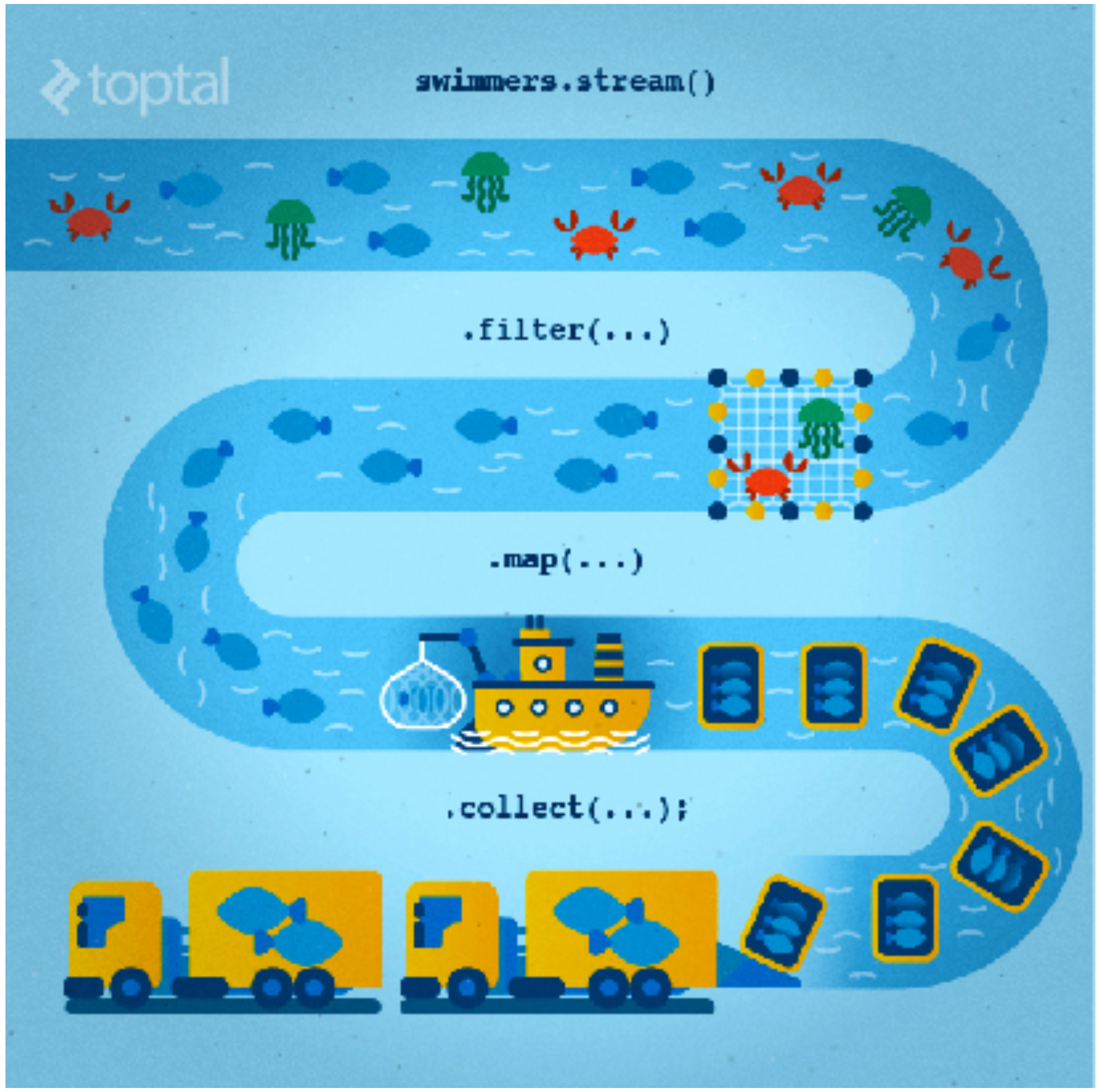
Java™ 8

Frokostpause!

Program

- 0900 - 0915 Velkommen, energizer, icebreaker, teams
- 0915 - 1000 Lambda expressions
- 1015 - 1100 Method references
- 1115 - 1200 DateTime API og Optional, API design
- 1200 - 1230 Frokost pause
- 1230 - 1430 Streams og Collections
- 1445 - 1530 Code clinic - upgrading the Keylane code base
- 1530 - 1600 Read the Code Team Challenge

Streams and Collections



```
int sum = swimmers.stream()
                      .filter(b -> b.type() == CONSUMABLE )
                      .mapToInt(b -> b.getWeight())
                      .sum();
```

Streams

- It is a pipe, that transports data from source to destination
- sequential or parallel
- not for storing objects
- easy to filter, sort & map elements
- functional nature - produces a result, but does not modify its source
- laziness seeking - intermediate vs terminal operations
- possibly unbounded
- consumable - elements of a stream are only visited once during the life of a stream

Image credit: https://www.toptal.com/java/why-you-need-to-upgrade-to-java-8-already

Stream Classes

- in the **java.util.stream** package
- can be used to transfer any type of objects
- Specialisations: **IntStream**, **LongStream**, **DoubleStream**
- See https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Stream operations

- operations are divided into source, intermediate and terminal
- combined to form stream pipelines
- A stream pipeline consists of
    - a **source** (such as a `Collection`, an array, a generator function, or an I/O channel)
    - followed by zero or more **intermediate** operations such as `Stream.filter` or `Stream.map`
        - *stateless* operations -retain no state from previously seen element, fx. `filter` and `map`
        - *stateful* - may incorporate state from previously seen elements fx. `distinct` and `sorted`
    - a **terminal** operation such as `Stream.forEach` or `Stream.reduce`.

Laziness seeking

- Intermediate operations return a new stream. They are always *lazy*;
- executing an intermediate operation such as `filter()` does not actually perform any filtering:
  - creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.
- Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.terminal or intermediate
  - terminal produces side effects
  - intermediate is not invoked (lazy) unless necessary
- Terminal operations, such as `Stream.forEach` or `IntStream.sum`, may traverse the stream to produce a result or a side-effect.
- After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used

Sequential or Parallel

- Streams facilitate parallel execution
- Computation as a pipeline of aggregate operations
- All streams operations can execute either in serial or in parallel
- Stream implementations in the JDK create serial streams unless parallelism is explicitly requested
- Careful with user behaviour - we have to **prevent *interference*** with the data source during the execution of a stream pipeline

```
int sum = swimmers.parallelstream()
                      .filter(b -> b.type() == CONSUMABLE )
                      .mapToInt(b -> b.getWeight())
                      .sum();
```

Obtaining a stream

- You can use the **of** static method in **Stream** to create a sequential stream
- you can pass an array to the **of** method
- The **java.util.Arrays** utility class now has a **stream** method for converting an array to a sequential stream
- **java.util.Collection** interface also has default methods named **stream** and **parallelStream** that return a sequential or a parallel stream

```
Stream<Integer> stream = Stream.of(100, 200, 300);
Stream<String> stream = Stream.of({"Bart", "Lisa", “Maggie”});
//From a Collection
Stream<Path> list = Files.list(Paths.get("."));
```

Filter

- Select the elements of the stream based on certain criteria
- return a new **Stream** with selected elements
- filter a stream by calling the **filter** method on a **Stream** object, passing a **Predicate**
- The **Predicate** determines whether or not an element will be included

```
Stream<T> filter(java.util.function.Predicate<? super T>
predicate)
```

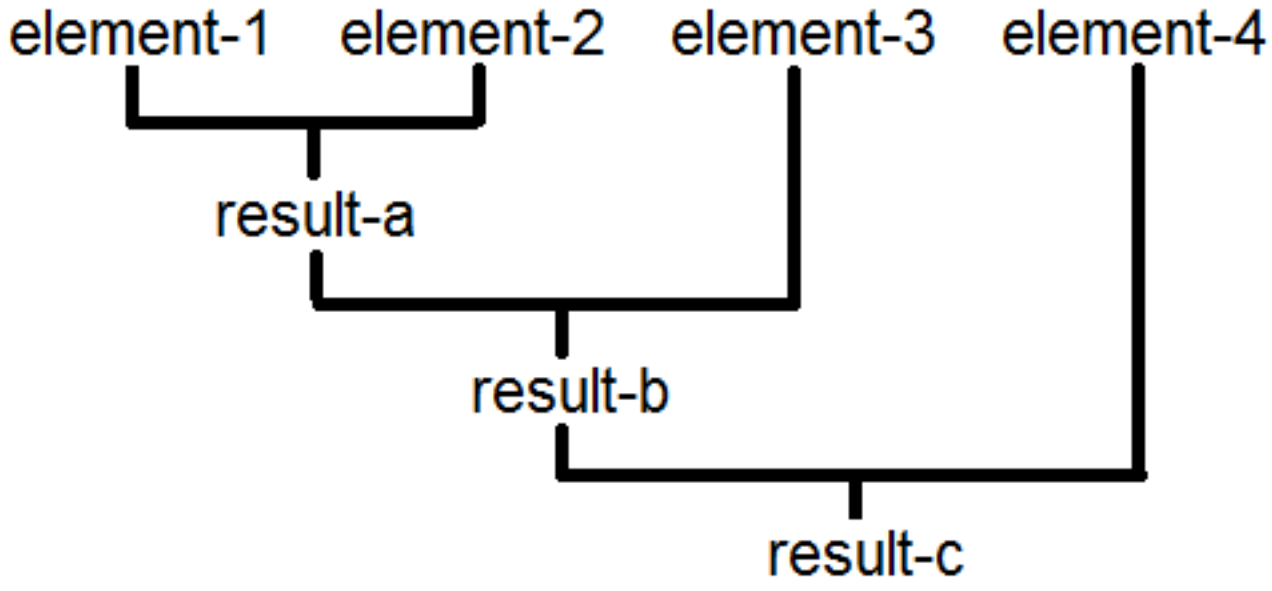
Map

- You have a stream of one kind of elements, but want to **transfmogrif** them to another kind of elements

- The map method **transfmogrifs** each element in a stream by passing it to a function

- The result is a new stream of **transmogriffed** elements

```
<R> Stream<R> map(java.util.function.Function<? super T,
        ? extends R> mapper)

tigers.stream.map( tiger -> transfmogrif(tiger));
```



Reduce

- Combines all elements of the stream into a single result
- For a parallel stream, operations can be done in parallel
- Reduces by combining pairs of elements into one single element
  (element-1 # element-2) # element-3) # element-4

```
java.util.Optional<T>
reduce(java.util.function.BinaryOperator<T>
        accumulator)
T reduce(T identity,
        java.util.function.BinaryOperator<T> accumulator)
```

Image credit: “Upgrading to Java 8”

Collect

- A mutable reduction mutable reduction operation accumulates a **Stream**’s elements into a container and returns the container
- The container is mutable
- The **collect** method does its job in 3 steps:
  - method handles its first argument, which is a **Supplier** that returns a container such as a **Collection** or a **StringBuilder**
  - Recall that a **BiConsumer** accepts two arguments of different types and do not return any value. Practically, the **BiConsumer** adds each stream element to the container or containers that the **Supplier** produced.
  - Another BiConsumer is only used in parallel streams

```
StringBuilder sb2 = stream2.collect(
              StringBuilder::new,
              StringBuilder::append,
              StringBuilder::append);
```

Collectors

- Implementations of `Collector` that implement various reductions (sum, average, count, group, put into collection, etc)

- predefined collectors to perform common mutable reduction tasks

- See more https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

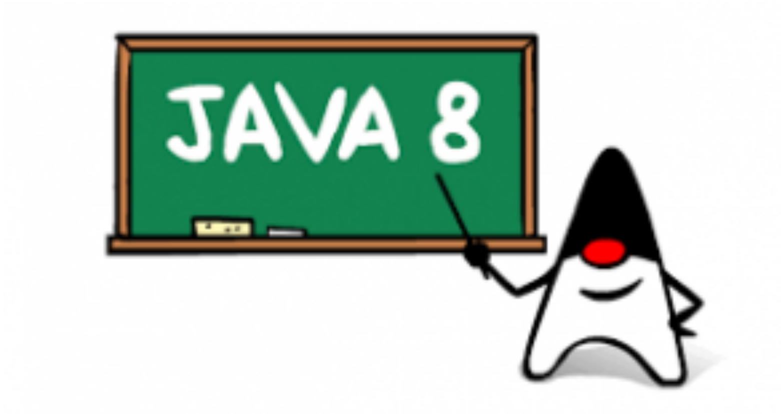
```
// Accumulate names into a List
List<String> list =
people.stream().map(Person::getName).collect(Collectors.toList());
```

Parallel Streams

- Streams can be executed in parallel to increase runtime performance on large amount of elements
- Parallel stream is more expensive to construct
- Parallel streams use a common ForkJoinPool
- Collections support the method parallelStream() to create a parallel stream of elements.
- Or, you can call the intermediate method parallel() on a given stream

Group Activity

Be the Stream



Programming Exercise!

Lesson_04_streams_and_collections

Additional ressources

- http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html#close
- https://blog.logentries.com/2017/01/java-8-lazy-argument-evaluation/
- https://blog.codefx.org/techniques/intention-revealing-code-java-8-optional/
- Upgrading to Java 8, by Kurniawan Budi; Brai 2015

Code Clinic

Game Time!