

# Higher-Order Functional Programming with Java 8 Streams

Florian Biermann  
fbie@itu.dk

IT University of Copenhagen & UCAS

2016-06-02

# Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

# Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- ▶ Brief review of Java's anonymous inner classes.

# Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- ▶ Brief review of Java's anonymous inner classes.
- ▶ Java 8 lambda expressions.

# Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- ▶ Brief review of Java's anonymous inner classes.
- ▶ Java 8 lambda expressions.
- ▶ Java 8 functional interfaces.

# Today's Plan

We will be looking at (parallel) higher order functional programming in Java.

- ▶ Brief review of Java's anonymous inner classes.
- ▶ Java 8 lambda expressions.
- ▶ Java 8 functional interfaces.
- ▶ Higher-order functional programming using lazy streams.

# Java 7 Anonymous Inner Classes

In Java, you can instantiate new classes on the fly:

```
Runnable r =  
    new Runnable() {  
        public void run() {  
            System.out.println("Inner class.");  
        }  
    };  
r.run(); // "Inner class."
```

# Java 7 Anonymous Inner Classes

In Java, you can instantiate new classes on the fly:

```
Runnable r =  
    new Runnable() {  
        public void run() {  
            System.out.println("Inner class.");  
        }  
    };  
r.run(); // "Inner class."
```

They can access global variables iff these are marked `final`:

```
final String s = "This is final!";  
Runnable r = new Runnable() {  
    public void run() {  
        System.out.println(s);  
    }  
};  
r.run(); // "This is final!";
```



# Runnable Is Just an Interface

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

# Runnable Is Just an Interface

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

There are other useful interfaces, e.g. `Callable<V>` from `java.util.concurrent`:

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

# Runnable Is Just an Interface

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

There are other useful interfaces, e.g. `Callable<V>` from `java.util.concurrent`:

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

```
final String s = "Callable."  
Callable<String> c = new Callable<String>() {  
    public String call() {  
        return s;  
    }  
};  
System.out.println(c.call()); // "Callable."
```

# Why Anonymous Inner Classes?

The Java answer to closures.

# Why Anonymous Inner Classes?

The Java answer to closures.

- ▶ Whenever we want to start a thread, we pass it a new instance of `Runnable`.

# Why Anonymous Inner Classes?

The Java answer to closures.

- ▶ Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- ▶ Short-running tasks, only used once.

# Why Anonymous Inner Classes?

The Java answer to closures.

- ▶ Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- ▶ Short-running tasks, only used once.
- ▶ Methods in Java are no *first-class citizens*, but objects are.

# Why Anonymous Inner Classes?

The Java answer to closures.

- ▶ Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- ▶ Short-running tasks, only used once.
- ▶ Methods in Java are no *first-class citizens*, but objects are.
- ▶ Anonymous inner classes are a *work-around* for this problem.



# Why Anonymous Inner Classes?

The Java answer to closures.

- ▶ Whenever we want to start a thread, we pass it a new instance of `Runnable`.
- ▶ Short-running tasks, only used once.
- ▶ Methods in Java are no *first-class citizens*, but objects are.
- ▶ Anonymous inner classes are a *work-around* for this problem.
- ▶ The syntax is ugly and hard to read, so Java 8 introduces *lambda expressions*.

# Our First Java 8 Lambda Expression

Equal to instantiating an anonymous `Runnable`:

```
Runnable r =  
    () -> { System.out.println("Lambda."); } ;
```

# Our First Java 8 Lambda Expression

Equal to instantiating an anonymous `Runnable`:

```
Runnable r =  
    () -> { System.out.println("Lambda."); } ;
```

No closing braces required if only one statement:

```
Runnable r =  
    () -> System.out.println("Lambda.");
```

# Our Second Java 8 Lambda Expression

Equal to instantiating an anonymous `Callable<String>`:

```
Callable<String> c =  
    () -> { return "Lambda."; };
```

# Our Second Java 8 Lambda Expression

Equal to instantiating an anonymous `Callable<String>`:

```
Callable<String> c =  
    () -> { return "Lambda."; };
```

No `return` statement required if only one statement:

```
Callable<String> c = () -> "Lambda.";
```

# Our Second Java 8 Lambda Expression

Equal to instantiating an anonymous `Callable<String>`:

```
Callable<String> c =  
    () -> { return "Lambda."; };
```

No `return` statement required if only one statement:

```
Callable<String> c = () -> "Lambda.";
```

Compare to Typed Racket, also no `return` statement:

```
(lambda () "Lambda.")
```

# This Is Illegal

```
Callable<String> c =  
    () -> {  
        System.out.println("Callable.call()");  
        "Lambda."  
    };
```

# This Is Illegal

```
Callable<String> c =  
    () -> {  
        System.out.println("Callable.call()");  
        "Lambda."  
    };
```

```
error: not a statement  
    "Lambda.";});  
    ^
```



# This Is Illegal

```
Callable<String> c =  
    () -> {  
        System.out.println("Callable.call()");  
        "Lambda."  
    };
```

```
error: not a statement  
    "Lambda.";});  
    ^
```

**Rules:**

# This Is Illegal

```
Callable<String> c =  
    () -> {  
        System.out.println("Callable.call()");  
        "Lambda."  
    };
```

```
error: not a statement  
    "Lambda."};  
    ^
```

## Rules:

- ▶ If you use more than one statement then you must use curly braces.

# This Is Illegal

```
Callable<String> c =  
    () -> {  
        System.out.println("Callable.call()");  
        "Lambda."  
    };
```

```
error: not a statement  
    "Lambda."};  
    ^
```

## Rules:

- ▶ If you use more than one statement then you must use curly braces.
- ▶ If you use curly braces then you *must* use `return` (except for `void` functions).

# Java Inner Classes Summary

The important points until here:

# Java Inner Classes Summary

The important points until here:

- ▶ Inner classes always implement an interface (or an abstract class).

# Java Inner Classes Summary

The important points until here:

- ▶ Inner classes always implement an interface (or an abstract class).
- ▶ There are two important interfaces already defined in Java 7.

# Java Inner Classes Summary

The important points until here:

- ▶ Inner classes always implement an interface (or an abstract class).
- ▶ There are two important interfaces already defined in Java 7.
- ▶ In Java 8, you can use *lambda expressions* instead of inner classes.

# Java Inner Classes Summary

The important points until here:

- ▶ Inner classes always implement an interface (or an abstract class).
- ▶ There are two important interfaces already defined in Java 7.
- ▶ In Java 8, you can use *lambda expressions* instead of inner classes.
- ▶ But what about more interesting functions that also take parameter arguments?



# Defining Your Own Functional Interfaces

A function of type `int`  $\rightarrow$  `String`:

```
@FunctionalInterface
public interface StringToInt {
    public int apply(String s);
}
```

# Defining Your Own Functional Interfaces

A function of type `int → String`:

```
@FunctionalInterface
public interface StringToInt {
    public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseInt =
    s -> Integer.parseInt(s);
```

# Defining Your Own Functional Interfaces

A function of type `int`  $\rightarrow$  `String`:

```
@FunctionalInterface
public interface StringToInt {
    public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseInt =
    s -> Integer.parseInt(s);
```

- Lambda expressions **have no types themselves!**

# Defining Your Own Functional Interfaces

A function of type `int`  $\rightarrow$  `String`:

```
@FunctionalInterface
public interface StringToInt {
    public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
    s -> Integer.parseInt(s);
```

- ▶ Lambda expressions **have no types themselves!**
- ▶ Instead, they have a *target function type*.

# Defining Your Own Functional Interfaces

A function of type `int`  $\rightarrow$  `String`:

```
@FunctionalInterface
public interface StringToInt {
    public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
    s -> Integer.parseInt(s);
```

- ▶ Lambda expressions **have no types themselves!**
- ▶ Instead, they have a *target function type*.
- ▶ Without target function type, the lambda expression does not work.

# Defining Your Own Functional Interfaces

A function of type `int`  $\rightarrow$  `String`:

```
@FunctionalInterface
public interface StringToInt {
    public int apply(String s);
}
```

Now, we can instantiate such a function using a lambda expression:

```
StringToInt parseToInt =
    s -> Integer.parseInt(s);
```

- ▶ Lambda expressions **have no types themselves!**
- ▶ Instead, they have a *target function type*.
- ▶ Without target function type, the lambda expression does not work.
- ▶ Here, one target function type is `StringToInt`.

# Java 8 Functional Interfaces

No need to define your own interfaces all the time.

`java.util.function` contains *many* different functional interfaces:

# Java 8 Functional Interfaces

No need to define your own interfaces all the time.

`java.util.function` contains *many* different functional interfaces:

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```



# Java 8 Functional Interfaces

No need to define your own interfaces all the time.

`java.util.function` contains *many* different functional interfaces:

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```

```
Predicate<String> isEmpty =  
    s -> s.isEmpty();
```

# Java 8 Functional Interfaces

No need to define your own interfaces all the time.

`java.util.function` contains *many* different functional interfaces:

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```

```
Predicate<String> isEmpty =  
    s -> s.isEmpty();
```

```
Consumer<String> print =  
    s -> System.out.println(s);
```

# Java 8 Functional Interfaces

No need to define your own interfaces all the time.

`java.util.function` contains *many* different functional interfaces:

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```

```
Predicate<String> isEmpty =  
    s -> s.isEmpty();
```

```
Consumer<String> print =  
    s -> System.out.println(s);
```

```
Supplier<String> genString =  
    () -> "Nihao.";
```

# Primitive Data Type Interfaces

To avoid *value boxing* (`int`  $\rightarrow$  `Integer`), there are special implementations for primitive data types:

# Primitive Data Type Interfaces

To avoid *value boxing* (`int`  $\rightarrow$  `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =  
    s -> Integer.parseInt(s);
```

# Primitive Data Type Interfaces

To avoid *value boxing* (`int`  $\rightarrow$  `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =  
    s -> Integer.parseInt(s);
```

```
LongBinaryOperator longMul =  
    (x, y) -> x * y;
```

# Primitive Data Type Interfaces

To avoid *value boxing* (`int`  $\rightarrow$  `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =  
    s -> Integer.parseInt(s);
```

```
LongBinaryOperator longMul =  
    (x, y) -> x * y;
```

```
DoubleToIntFunction round =  
    x -> (int)(x + 0.5d);
```

# Primitive Data Type Interfaces

To avoid *value boxing* (`int`  $\rightarrow$  `Integer`), there are special implementations for primitive data types:

```
ToIntFunction<String> parseToInt =  
    s -> Integer.parseInt(s);
```

```
LongBinaryOperator longMul =  
    (x, y) -> x * y;
```

```
DoubleToIntFunction round =  
    x -> (int)(x + 0.5d);
```

```
IntPredicate isEven =  
    x -> x % 2 == 0;
```



# Function References

This is redundant syntax:

```
ToIntFunction<String> parseToInt =  
    s -> Integer.parseInt(s);
```

# Function References

This is redundant syntax:

```
ToIntFunction<String> parseToInt =  
    s -> Integer.parseInt(s);
```

We call a function that has the same type as the target function type, `String`  $\rightarrow$  `int`. Instead, we can use a *function reference*:

# Function References

This is redundant syntax:

```
ToIntFunction<String> parseToInt =  
    s -> Integer.parseInt(s);
```

We call a function that has the same type as the target function type, `String`  $\rightarrow$  `int`. Instead, we can use a *function reference*:

```
ToIntFunction<String> parseToInt =  
    Integer::parseInt;
```

This makes the code much more concise!

# Functional Interfaces Summary

The important points until here:

# Functional Interfaces Summary

The important points until here:

- ▶ In Java 8, you can use *lambda expressions* instead of inner classes.

# Functional Interfaces Summary

The important points until here:

- ▶ In Java 8, you can use *lambda expressions* instead of inner classes.
- ▶ You can define your own *target function types* by defining interfaces.

# Functional Interfaces Summary

The important points until here:

- ▶ In Java 8, you can use *lambda expressions* instead of inner classes.
- ▶ You can define your own *target function types* by defining interfaces.
- ▶ Better: use pre-defined functional interfaces from `java.util.function.*`

# Functional Interfaces Summary

The important points until here:

- ▶ In Java 8, you can use *lambda expressions* instead of inner classes.
- ▶ You can define your own *target function types* by defining interfaces.
- ▶ Better: use pre-defined functional interfaces from `java.util.function.*`
- ▶ If you work on primitive data types, use specialized interfaces!



# Functional Interfaces Summary

The important points until here:

- ▶ In Java 8, you can use *lambda expressions* instead of inner classes.
- ▶ You can define your own *target function types* by defining interfaces.
- ▶ Better: use pre-defined functional interfaces from `java.util.function.*`
- ▶ If you work on primitive data types, use specialized interfaces!
- ▶ So what can we use these for?

# Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =  
    Stream.of(new Student("A", 89),  
              new Student("B", 44),  
              new Student("C", 62));
```

# Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =  
    Stream.of(new Student("A", 89),  
             new Student("B", 44),  
             new Student("C", 62));
```

We can map over streams:

```
students  
    .map(s -> new Student(s.name, s.grade + 10));
```

# Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =  
    Stream.of(new Student("A", 89),  
              new Student("B", 44),  
              new Student("C", 62));
```

We can map over streams:

```
students  
    .map(s -> new Student(s.name, s.grade + 10));
```

- *Functional*: `map` returns a new `Stream<Student>` instance!

# Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =  
    Stream.of(new Student("A", 89),  
              new Student("B", 44),  
              new Student("C", 62));
```

We can map over streams:

```
students  
    .map(s -> new Student(s.name, s.grade + 10));
```

- ▶ *Functional*: `map` returns a new `Stream<Student>` instance!
- ▶ *Lazy*: the computation is not executed directly.

# Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =  
    Stream.of(new Student("A", 89),  
             new Student("B", 44),  
             new Student("C", 62));
```

We can map over streams:

```
students  
    .map(s -> new Student(s.name, s.grade + 10));
```

- ▶ *Functional*: `map` returns a new `Stream<Student>` instance!
- ▶ *Lazy*: the computation is not executed directly.
- ▶ Requires a *terminal operation* to trigger computation.

# Java 8 Streams

Streams in Java 8 are lazy collections of values:

```
Stream<Student> students =  
    Stream.of(new Student("A", 89),  
             new Student("B", 44),  
             new Student("C", 62));
```

We can map over streams:

```
students  
    .map(s -> new Student(s.name, s.grade + 10));
```

- ▶ *Functional*: `map` returns a new `Stream<Student>` instance!
- ▶ *Lazy*: the computation is not executed directly.
- ▶ Requires a *terminal operation* to trigger computation.

```
students  
    .map(s -> new Student(s.name, s.grade + 10))  
    .forEach(System.out::println);
```

# What is Laziness?

Deferring computation until value is requested.

```
public class Lazy<T> implements Supplier<T> {  
    private final Supplier<T> f;  
    private volatile T t;  
  
    public Lazy(Supplier<T> f) {  
        this.f = f;  
    }  
  
    public T get() {  
        if (t == null)  
            t = f.get();  
        return t;  
    }  
}
```



# Laziness in Action

```
Lazy<String> s = new Lazy<String>(() -> {  
    System.out.println("Calling get()");  
    return "someString";  
}); // Not yet computed!
```

```
String s1 = s.get(); // "Calling get()"  
String s2 = s.get(); // Nothing printed.
```

# Laziness in Action

```
Lazy<String> s = new Lazy<String>(() -> {  
    System.out.println("Calling get()");  
    return "someString";  
}); // Not yet computed!
```

```
String s1 = s.get(); // "Calling get()"   
String s2 = s.get(); // Nothing printed.
```

**OBS:** If we use side-effects, this implementation is **not** thread-safe because `Lazy::get` might be called twice.

# How Does Laziness Help?

Laziness allows the `Stream` implementation to *fuse* successive applications of `map`. This is an in-place mapping over an array:

```
for (int i = 0; i < students.length; ++i)
    students[i] = f(students[i]);
for (int i = 0; i < students.length; ++i)
    students[i] = g(students[i]);
```

# How Does Laziness Help?

Laziness allows the `Stream` implementation to *fuse* successive applications of `map`. This is an in-place mapping over an array:

```
for (int i = 0; i < students.length; ++i)
    students[i] = f(students[i]);
for (int i = 0; i < students.length; ++i)
    students[i] = g(students[i]);
```

This is what loop fusion does:

```
for (int i = 0; i < students.length; ++i)
    students[i] = g(f(students[i]));
```

# How Does Laziness Help?

Laziness allows the `Stream` implementation to *fuse* successive applications of `map`. This is an in-place mapping over an array:

```
for (int i = 0; i < students.length; ++i)
    students[i] = f(students[i]);
for (int i = 0; i < students.length; ++i)
    students[i] = g(students[i]);
```

This is what loop fusion does:

```
for (int i = 0; i < students.length; ++i)
    students[i] = g(f(students[i]));
```

You can do it manually – we call it *function composition*:

```
Function<A, B> f = ...;
Function<B, C> g = ...;
Function<A, C> h = f.andThen(g);
```

# Computing Number of Primes

For-loop:

```
long count = 0;
for (long p = 0; p < n; ++p)
    if (isPrime(p)) ++count;
```

# Computing Number of Primes

For-loop:

```
long count = 0;
for (long p = 0; p < n; ++p)
    if (isPrime(p)) ++count;
```

Sequential stream:

```
LongStream.range(0, n)
    .filter(p -> isPrime(p))
    .count();
```

# Computing Number of Primes

For-loop:

```
long count = 0;
for (long p = 0; p < n; ++p)
    if (isPrime(p)) ++count;
```

Sequential stream:

```
LongStream.range(0, n)
    .filter(p -> isPrime(p))
    .count();
```

Parallel stream:

```
LongStream.range(0, n)
    .parallel()
    .filter(p -> isPrime(p))
    .count();
```

Modified from *P. Sestoft, Java Precisely, third ed., The MIT Press, Mar. 2016.*



# Java Streams Summary

Java streams are a great tool:

- ▶ Lazy, higher-order functional and declarative.
- ▶ Very well implemented, highly optimized.
- ▶ Easy to use when you are used to functional programming.
- ▶ Easy to parallelize (`.parallel()`).

# Java Streams Summary

Java streams are a great tool:

- ▶ Lazy, higher-order functional and declarative.
- ▶ Very well implemented, highly optimized.
- ▶ Easy to use when you are used to functional programming.
- ▶ Easy to parallelize (`.parallel()`).

But the downsides...

- ▶ Sometimes, performance is unpredictable because of laziness.
- ▶ OBS: Functions with side-effects will break!

# Building Streams

```
IntStream is = IntStream.of(2, 3, 5, 7, 11, 13);  
IntStream nats = IntStream.iterate(0, x -> x+1);
```

# Building Streams

```
IntStream is = IntStream.of(2, 3, 5, 7, 11, 13);  
IntStream nats = IntStream.iterate(0, x -> x+1);  
  
String[] a = { "Haidian", "Gulou", ...};  
Stream<String> beijingNeighbourhoods =  
    Arrays.stream(a);
```

# Building Streams

```
IntStream is = IntStream.of(2, 3, 5, 7, 11, 13);
IntStream nats = IntStream.iterate(0, x -> x+1);

String[] a = { "Haidian", "Gulou", ...};
Stream<String> beijingNeighbourhoods =
    Arrays.stream(a);

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

# Building Streams

```
IntStream is = IntStream.of(2, 3, 5, 7, 11, 13);
IntStream nats = IntStream.iterate(0, x -> x+1);

String[] a = { "Haidian", "Gulou", ...};
Stream<String> beijingNeighbourhoods =
    Arrays.stream(a);

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

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

Modified from *P. Sestoft, Java Precisely, third ed., The MIT Press, Mar. 2016.*

# Take-Aways From This Lecture

We have seen:

# Take-Aways From This Lecture

We have seen:

- Lambda expressions in Java 8.

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```



# Take-Aways From This Lecture

We have seen:

- ▶ Lambda expressions in Java 8.

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```

- ▶ Functional interfaces and how to define your own.

# Take-Aways From This Lecture

We have seen:

- ▶ Lambda expressions in Java 8.

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```

- ▶ Functional interfaces and how to define your own.
- ▶ Using streams in Java 8.

```
students.  
    map(s -> new Student(s.name, s.grade + 10));
```

# Take-Aways From This Lecture

We have seen:

- ▶ Lambda expressions in Java 8.

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```

- ▶ Functional interfaces and how to define your own.
- ▶ Using streams in Java 8.

```
students.  
    map(s -> new Student(s.name, s.grade + 10));
```

- ▶ Using *parallel* streams in Java 8.

```
students  
    .parallel()  
    .map(s -> new Student(s.name, s.grade + 10));
```

# Take-Aways From This Lecture

We have seen:

- ▶ Lambda expressions in Java 8.

```
Function<String, String> toUpperCase =  
    s -> s.toUpperCase();
```

- ▶ Functional interfaces and how to define your own.
- ▶ Using streams in Java 8.

```
students.  
    map(s -> new Student(s.name, s.grade + 10));
```

- ▶ Using *parallel* streams in Java 8.

```
students  
    .parallel()  
    .map(s -> new Student(s.name, s.grade + 10));
```

You will find them very useful!

# Projects

Please chose a project! You are allowed to work in groups of 2 – 3 students.

# Projects

Please chose a project! You are allowed to work in groups of 2 – 3 students.

TYPING AN UNTYPED RACKET LIBRARY Find an untyped Racket library on [pkgs.racket-lang.org](https://pkgs.racket-lang.org) and annotate it with types.

# Projects

Please chose a project! You are allowed to work in groups of 2 – 3 students.

**TYPING AN UNTYPED RACKET LIBRARY** Find an untyped Racket library on [pkgs.racket-lang.org](https://pkgs.racket-lang.org) and annotate it with types.

**PARALLEL IMMUTABLE FUNCTIONAL DATA STRUCTURES** Extend the Rope data structure with more functions and parallelize them.

# Projects

Please chose a project! You are allowed to work in groups of 2 – 3 students.

**TYPING AN UNTYPED RACKET LIBRARY** Find an untyped Racket library on [pkgs.racket-lang.org](https://pkgs.racket-lang.org) and annotate it with types.

**PARALLEL IMMUTABLE FUNCTIONAL DATA STRUCTURES** Extend the Rope data structure with more functions and parallelize them.

**A TINY MATH LANGUAGE IMPLEMENTATION** Implement an interpreter and a type checker for a math language with Racket-style syntax.



# Projects

Please chose a project! You are allowed to work in groups of 2 – 3 students.

**TYPING AN UNTYPED RACKET LIBRARY** Find an untyped Racket library on [pkgs.racket-lang.org](http://pkgs.racket-lang.org) and annotate it with types.

**PARALLEL IMMUTABLE FUNCTIONAL DATA STRUCTURES** Extend the Rope data structure with more functions and parallelize them.

**A TINY MATH LANGUAGE IMPLEMENTATION** Implement an interpreter and a type checker for a math language with Racket-style syntax.

**YOUR OWN IDEA?** If you have a personal project that you want to work on using Typed Racket or Java 8 Streams, you are very welcome to do so.

# Typing an Untyped Racket Library

- ▶ [pkgs.racket-lang.org](https://pkgs.racket-lang.org) is the official library repository for Racket.

# Typing an Untyped Racket Library

- ▶ [pkgs.racket-lang.org](https://pkgs.racket-lang.org) is the official library repository for Racket.
- ▶ Untyped code:

```
(define (id x) x)
```

# Typing an Untyped Racket Library

- ▶ [pkgs.racket-lang.org](https://pkgs.racket-lang.org) is the official library repository for Racket.
- ▶ Untyped code:

```
(define (id x) x)
```

- ▶ Your task:

```
(: id (All (A) (-> A A)))  
(define (id x) x)
```

# Typing an Untyped Racket Library

- ▶ [pkgs.racket-lang.org](https://pkgs.racket-lang.org) is the official library repository for Racket.

- ▶ Untyped code:

```
(define (id x) x)
```

- ▶ Your task:

```
(: id (All (A) (-> A A)))  
(define (id x) x)
```

- ▶ Please e-mail me which package you want to work on, so that I can check it is not too hard for you.

# Parallel Immutable Functional Data Structures

- ▶ Last lecture you saw the Rope data structure for parallel programming.

# Parallel Immutable Functional Data Structures

- ▶ Last lecture you saw the Rope data structure for parallel programming.
- ▶ Much more work to be done!
  - ▶ `rope-mapreduce`
  - ▶ `rope-reverse`
  - ▶ `rope-forall`
  - ▶ `rope-exists`
  - ▶ `rope-scan`

# Parallel Immutable Functional Data Structures

- ▶ Last lecture you saw the Rope data structure for parallel programming.
- ▶ Much more work to be done!
  - ▶ `rope-mapreduce`
  - ▶ `rope-reverse`
  - ▶ `rope-forall`
  - ▶ `rope-exists`
  - ▶ `rope-scan`
- ▶ Parallelize functions that are not parallel.



# A Tiny Math Language Implementation

- In Racket, you can easily implement other languages that look like Racket.

```
(let (x (neg 1)) (* x x))
```

# A Tiny Math Language Implementation

- ▶ In Racket, you can easily implement other languages that look like Racket.

```
(let (x (neg 1)) (* x x))
```

- ▶ You need not to implement the parser.

# A Tiny Math Language Implementation

- ▶ In Racket, you can easily implement other languages that look like Racket.

```
(let (x (neg 1)) (* x x))
```

- ▶ You need not to implement the parser.
- ▶ Tasks:
  - ▶ Implement the interpreter.
  - ▶ Implement a type checker.
  - ▶ Maybe implement a compiler?

# Project Descriptions

The project descriptions are available on

[github.com/fbie/parallel-functional-lectures](https://github.com/fbie/parallel-functional-lectures)

Hand-in deadline is

2016 - 07 - ??

# Questions?

Slides and code available at  
[github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming).

# Questions?

Slides and code available at  
[github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming).

Any questions? E-mail me at [fbie@itu.dk](mailto:fbie@itu.dk).

# Questions?

Slides and code available at  
[github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming).

Any questions? E-mail me at [fbie@itu.dk](mailto:fbie@itu.dk).

**Thank you for your attention!**