Functional Programming in Java

Prof. Dr. Ralf Lämmel
Msc. Johannes Härtel
Msc. Marcel Heinz

(C) 2018, SoftLang Team, University of Koblenz-Landau
Java without functional Constructs

101 Company Features
- OO-Model
- Cut
- Total

```java
package model;

import java.util.LinkedList;
import java.util.List;

/**
 * A company has a name and
 * consists of (possibly nested) departments.
 */

public class Company {

    private String name;
    private List<Department> depts = new LinkedList<Department>();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Department> getDepts() {
        return depts;
    }
}
```
Java without functional Constructs

101 Company Features
- OO-Model
- Cut
- Total

Getters and Setters are omitted on the slide for brevity.
Java without functional Constructs

101 Company Features
- OO-Model
- Cut
- Total

```
package model;

/**
 * An employee has a name, an address, and a salary.
 */
public class Employee {
    private String name;
    private String address;
    private double salary;
```

Getters and Setters are omitted on the slide for brevity.
Java without functional Constructs

101 Company Features
- OO-Model
- Cut
- Total

Typical imperative Java-code with for-loops.
Java without functional Constructs

101Company Features
- OO-Model
- Cut
- Total

```java
public double total(Company c) {
    double total = 0;
    for (Department d : c.getDepts())
        total += total(d);
    return total;
}

public double total(Department d) {
    double total = d.getManager().getSalary();
    for (Department sub : d.getSubdepts())
        total += total(sub);
    for (Employee e : d.getEmployees())
        total += total(e);
    return total;
}

public double total(Employee e) {
    return e.getSalary();
}
```

Typical imperative Java-code with for-loops.
Java History

- Java 8
- Java 9
- Java 10

See https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

- Language
  - “Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.”
  - “Method references provide easy-to-read lambda expressions for methods that already have a name.”
  - “Improved type inference.”
Java History

● Java 8
● Java 9
● Java 10

See https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Collections

○ “Classes in the new java.util.stream package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.”
Java History

- Java 8
- Java 9
- Java 10

See https://docs.oracle.com/javase/9/whatsnew/toc.htm#JSNEW-GUID-5B808B2F-E891-43CD-BF6E-78787E547071

Tools

- “The jshell tool provides an interactive command-line interface for evaluating declarations, statements, and expressions of the Java programming language. It facilitates prototyping and exploration of coding options with immediate results and feedback. The immediate feedback combined with the ability to start with expressions is useful for education—whether learning the Java language or just learning a new API or language feature.”
Java History

- Java 8
- **Java 9**
- Java 10

See https://docs.oracle.com/javase/9/whatsnew/toc.htm#JSNEW-GUID-5B808B2F-E891-43CD-BF6E-78787E547071

- Java Class Library
  - Changes in the core libraries that shall not be of interest for this course.
Java History

- Java 8
- Java 9
- Java 10

- Local Variable Type Inference
  - We will come to this later.

This course focuses on the changes introduced in Java 8.
Functional Programming
- in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

How can we code Cut and Total in a functional fashion?

Thus, it needs to resemble mathematical expressions instead of imperative commands.

Which functional constructs exist in Java?
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

```
public void cut(Company c) {
    for (Department d : c.getDepts())
        cut(d);
}
```

```
public void cut(Company c) {
    c.getDepts().forEach(d -> cut(d));
}
```

“Functional Programming”

Imperative Programming
We apply the function cut to every element of the department list.
Java8 introduces Lambda expressions. The symbol “λ” is not written. It looks like we’re applying an anonymous function.
Type Inferred at Lambda Expression

```java
public void cut(Company c) {
    c.getDepts().forEach(d -> cut(d));
}

public void cut(Department d) {
    cut(d.getManager());
    d.getSubdepts().forEach(e ->
        d.getEmployees().forEach(ee ->
            cut(ee));
    )
}
```

Hovering over d in Eclipse.
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Java 8 introduces Method References. Methods can be passed as arguments. Syntax: `<namespace>::<name>

- `<Namespace>` is a class for static methods or an object for instance methods.
- `<Name>` is the name of a method that is part of the namespace.
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Our current context serves as the namespace. Our cut() methods are instance methods of the class Cut. If they were static methods, we would write Cut::cut.
To demonstrate what happens underneath, let us rewrite the code in a more explicit manner.

Both times, an instance of Consumer is created, which is a **functional interface**. A functional interface is an **interface** with only one abstract method.
Changing Employees

We can write functions that can change any employee the way we want.

```java
public void changeE(Company c, Consumer<Employee> con) {
    c.getDepts().forEach(d -> changeE(d, con));
}

public void changeE(Department d, Consumer<Employee> con) {
    con.accept(d.getManager());
    d.getSubdepts().forEach(sub -> changeE(sub, con));
    d.getEmployees().forEach(con);
}

new ChangeEmployee().changeE(c, em -> em.setSalary(em.getSalary() / 2));
```
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

(Rolls eyes) And here comes the magic functional programmer turning this into one line...
Okay, I don’t get the code anymore. Let the magician maintain this part.

Besides, your TOC is gone, because of the long line.

```
public double total(Company c) {
    return c.getDepts().stream().map(d -> total(d)).reduce(0.0, (t1, t2) -> t1 + t2);
}
```
By now, there exist a lot of style guidelines on the issue of emphasizing understandability.

Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Haskell-way for comparison.

```java
public double total(Company c) {
    return c.getDepts().stream()
        .map(d -> total(d))
        .reduce(0.0, (t1, t2) -> t1+t2);
}
```
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Imperative way for comparison.
Streams offer the capability of processing elements in a sort of pipeline. A stream can either be linear or parallel.
After turning the list into a stream, we apply the lambda expression to every element in the stream.
Finally, we fold over the elements in the stream and compute their sum.

Example:

department salaries = [10.0, 20.0, 30.0]
0.0 + 10.0 = 10.0
10.0 + 20.0 = 30.0
30.0 + 30.0 = 60.0
We introduce Employee filtering as a new feature. We want to retrieve the employees that have a salary > 2000.0. An imperative solution may look like this:

```java
public List<Employee> filterE(Company c){
    List<Employee> es = new ArrayList<>();
    for(Department d : c.getDepts()) {
        es.addAll(filterE(d));
    }
    return es;
}
```

```java
public List<Employee> filterE(Department d){
    List<Employee> es = new ArrayList<>();
    for(Employee e : d.getEmployees()) {
        if(e.getSalary() > 2000)
            es.add(e);
    }
    for(Department sub : d.getSubdepts()) {
        es.addAll(filterE(sub));
    }
    return es;
}
```
A functional solution would rather use object streams.

```
public Stream<Employee> filterE(Company c) {
    return c.getDepts()
        .stream()
        .map(d -> filterE(d))
        .reduce(Stream.of(), Stream::concat);
}

public Stream<Employee> filterE(Department d) {
    Stream<Employee> stream1 = d.getEmployees()
        .stream()
        .filter(e -> e.getSalary() > 2000);
    Stream<Employee> stream2 = d.getSubdepts()
        .stream()
        .map(this::filterE)
        .reduce(Stream.of(), Stream::concat);
    return Stream.concat(stream1, stream2);
}
```
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

filter() takes a lambda expression that returns a boolean value.

```java
public Stream<Employee> filterE(Company c) {
    return c.getDepts()
        .stream()
        .map(d -> filterE(d))
        .reduce(Stream.of(), Stream::concat);
}
```

```java
public Stream<Employee> filterE(Department d) {
    Stream<Employee> stream1 = d.getEmployees()
        .stream()
        .filter(e -> e.getSalary() > 2000);
    Stream<Employee> stream2 = d.getSubdepts()
        .stream()
        .map(this::filterE)
        .reduce(Stream.of(), Stream::concat);
    return Stream.concat(stream1, stream2);
}
```

Internally the lambda expression is transformed to a predicate object that only takes a single argument.
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

```java
public Stream<Employee> filterE(Company c) {
    return c.getDepts()
        .stream()
        .map(d -> filterE(d))
        .reduce(Stream.of(), Stream::concat);
}

public Stream<Employee> filterE(Department d) {
    Stream<Employee> stream1 = d.getEmployees()
        .stream()
        .filter(e -> e.getSalary() > 1000);
    Stream<Employee> stream2 = d.getSubdepts()
        .stream()
        .map(this::filterE)
        .reduce(Stream.of(), Stream::concat);
    return Stream.concat(stream1, stream2);
}
```

We concatenate streams.
Now, this solution only hints at parallelization possibilities.
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

```
public Stream<Employee> filterE(Company c) {
    return c.getDepts()
        .stream()
        .map(d -> filterE(d))
        .reduce(Stream.of(), Stream::concat);
}

public Stream<Employee> filterE(Department d) {
    Stream<Employee> stream1 = d.getEmployees()
        .parallelStream()
        .filter(e -> e.getSalary() < 2000);
    Stream<Employee> stream2 = d.getSubdepts()
        .parallelStream()
        .map(sub -> filterE(sub))
        .reduce(Stream.of(), Stream::concat);
    return Stream.concat(stream1, stream2);
}
```

This solution uses “cheap” parallelization.

parallelStream() spawns several threads, which then execute what is to come.

Be careful! You do not have any control over thread objects.

(C) 2018, SoftLang Team, University of Koblenz-Landau
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Data in the stream is chunked. The succeeding operations like map and reduce from above are then performed in the chunks.

department salaries = [10.0, 20.0, 30.0, 40.0]

\[
\begin{align*}
0.0 + 10.0 + 20.0 & = 30.0 \\
0.0 + 30.0 + 40.0 & = 70.0
\end{align*}
\]
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Accessing values of a stream by transforming to a list.

```java
List<Employee> richEmployees = new FilterEmployee().filterE()
  .collect(Collectors.toList());
```
Collectors

```java
Collectors.toCollection(BuilderFactory, Collectors.<T, List<T>>)
```

### Function Signature:

- `Collectors.toCollection(Supplier<C> collectionFactory): Collector<T,?,C>`
- `Collectors.toList(): Collector<T,?,List<T>>` - `java.util.stream.Collectors`
- `Collectors.toList(Comparator): Collector<T,?,List<T>>`
- `Collectors.toMap(Function, Function): Collector<T,?,Map<K,V>>` - `java.util.stream.Collectors`
- `Collectors.toMap(Comparator, Comparator): Collector<T,?,Map<K,V>>`
- `Collectors.toSet(): Collector<T,?,Set<T>>` - `java.util.stream.Collectors`
- `Collectors.toArray() - convert collection to array`

### Returns:

Returns a Collector that accumulates the input elements into a new List. There are no guarantees on the type, mutability, serializability, or thread-safety of the List returned; if more control over the returned List is required, use `Collectors.toCollection` (Supplier).

### Type Parameters:

- `<T>` the type of the input elements

### Returns:

- a Collector which collects all the input elements into a List, in encounter order

(C) 2018, SoftLang Team, University of Koblenz-Landau
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Eclipse debugger assists at line-wise debugging.

```java
public double total(Company c) {
    return c.getDepts()
        .stream()
        .map(d -> total(d))
        .reduce(0.0, (t1, t2) -> t1 + t2);
}
```
Functional Programming - in Java

- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals

Eclipse debugger assists at line-wise debugging.

```java
public double total(Company c) {
    return c.getDepts()
        .stream()
        .map(d -> total(d))
        .reduce(0.0, (t1, t2) -> t1 + t2);
}
```

(C) 2018, SoftLang Team, University of Koblenz-Landau
Let us assume that a manager may not always be available, but we still want to implement cut. This is typical code.

```java
if (d.getManager() != null)
    cut(d.getManager());
```

Java8 introduces Optionals that kind of correspond to Maybe values from Haskell. If the value is not null, then a Consumer is applied.

```java
Optional.ofNullable(d.getManager())
    .ifPresent(this::cut);
```

 Functional Programming - in Java
- ForEach
- Lambda Calculus
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals
An operation applied to the possibly null value could also return a null value.

Metaphorically speaking, the optional provides a protected context (it is a monad). As soon as a null value is produced, the operations on the context are not executed. We can define default values for this case.
Accessing single values in a stream

```java
Optional<Employee> oRichEmployee1 = new FilterEmployee().filterE(c).findFirst();
Employee richEmployee1 = oRichEmployee1.get();
System.out.println(richEmployee1);
```

We retrieve the first element in the stream. This returns an instance of Optional.
Accessing single values in a stream

```java
Optional<Employee> oRichEmployee1 = new FilterEmployee().filterE(c).findFirst();
Employee richEmployee1 = oRichEmployee1.get();
System.out.println(richEmployee1);
```

We assume a value to be present and print it to the console. Thus, at least one element exists in the filtered stream.
Instead of "brutally" retrieving the value using get, we can execute a Consumer instance if the value is present.

```java
Optional<Employee> oRichEmployee1 = new FilterEmployee().filterE(c).findFirst();
oRichEmployee1.ifPresent(System.out::println);
```
Where is the major impact of the new functional concepts?
Advanced Concepts

- Interface Instantiation
- Fluent API

```java
cutButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        controller.cutCompanyClicked();
    }
});
```

Good old times solution with a lot of boilerplate code.

Oh oh... Incoming one-liner!
Advanced Concepts

- **Interface Instantiation**
- **Fluent API**

A lambda expression, where the event is actually not processed.

```java
cutButton.addActionListener(e -> controller.cutCompanyClicked());
```

This may be confusing to some as we tend to always aim at processing every input.

\((\lambda x.42)\ 0 = 42\)
Advanced Concepts

- Interface Instantiation
- Fluent API

```java
subDeptList.addListSelectionListener(e -> {
    if (subDeptList.getSelectedIndex() == newIndex)
        controller.deptClicked(dept);
});
```

Code blocks are still possible inside of a lambda term.
Advanced Concepts

- Interface Instantiation
- Fluent API

```
subDeptList.addListSelectionListener(e -> {
    if (subDeptList.getSelectedIndex() == newIndex) {
        controller.deptClicked(dept);
    }
});
```

- Code blocks are still possible inside of a lambda term.

- If this anonymous code block gets bigger, you should rather think about extracting a separate method.
As soon as there need to be additional fields, lambdas cannot be used.
What is boilerplate code when, for example, instantiating an object model and how can we reduce it?
Advanced Concepts

- Interface Instantiation
- Fluent API

module Company.Data where

type Company = (Name, [Department])

data Department = Department Name Manager [Department] [Employee]
deriving (Eq, Read, Show)

type Employee = (Name, Address, Salary)
type Manager = Employee
type Name = String
type Address = String
type Salary = Float

The company model in Haskell.
Advanced Concepts

- Interface Instantiation
- Fluent API

```haskell
sampleCompany :: Company
sampleCompany =
  ( "Acme Corporation",
    Department "Research"
      ("Craig", "Redmond", 123456)
      [
      ("Erik", "Utrecht", 12345),
      ("Ralf", "Koblenz", 1234)],
    Department "Development"
      ("Ray", "Redmond", 234567)
      [
      Department "Dev1"
        ("Klaus", "Boston", 23456)
        [
        Department "Dev1.1"
          ("Karl", "Riga", 2345)
          [
          ("Joe", "Wifi City", 2344)]
          []
        ]
      ]
    ]
  )
```

Instantiation in Haskell. This is a composite expression.
Company Instantiation in Java

```java
Company c = new Company();
c.setName("ACME corporation");
Department research = new Department();
research.setName("Research");
Employee craig = new Employee();
craig.setName("Craig");
craig.setAddress("Redmond");
craig.setSalary(123456);
research.setManager(craig);
Employee erik = new Employee();
erik.setName("Erik");
erik.setAddress("Utrecht");
erik.setSalary(12345);
research.addEmployee(erik);
Employee ralf = new Employee();
ralf.setName("Ralf");
ralf.setAddress("Koblenz");
ralf.setSalary(1234);
research.addEmployee(ralf);

Department dev = new Department();
dev.setName("Development");
Employee ray = new Employee();
ray.setName("Ray");
ray.setAddress("Redmond");
ray.setSalary(234567);
dev.setManager(ray);

Department dev1 = new Department();
dev1.setName("Dev1");
Employee klaus = new Employee();
klaus.setName("Klaus");
klaus.setAddress("Boston");
klaus.setSalary(23456);

So many lines and we are still not done.
```
Advanced Concepts

- Interface Instantiation
- Fluent API

```java
Company c = new Company("ACME corporation");
Department research = new Department("Research",
    new Employee("Craig", "Redmond", 123456));
Employee erik = new Employee("Erik", "Utrecht", 12345);
research.addEmployee(erik);
Employee ralf = new Employee("Ralf", "Koblenz", 1234);
research.addEmployee(ralf);
Department dev = new Department("Development",
    new Employee("Ray", "Redmond", 234567));
Department dev1 = new Department("Dev1",
    new Employee("Klaus", "Boston", 23456));
Department dev11 = new Department("Dev1.1",
    new Employee("Karl", "Riga", 2345));
Employee joe = new Employee("Joe", "Wifi City", 2344);
dev11.addEmployee(joe);
dev1.addSubdept(dev11);
dev.addSubdept(dev1);
c.addDept(research);
c.addDept(dev);
return c;
```
Company Instantiation in Java

```java
Company c = company("ACME corporation")
    .addDepartment("Research")
    .setManager("Craig", "Redmond", 123456, "Research")
    .addEmployee("Erik", "Utrecht", 12345, "Research")
    .addEmployee("Ralf", "Koblenz", 1234, "Research")
    .addDepartment("Development")
    .setManager("Ray", "Redmond", 234567, "Development")
    .addSubDepartment("Dev1", "Development")
    .setManager("Klaus", "Boston", 23456, "Dev1")
    .addSubDepartment("Dev1.1", "Dev1")
    .setManager("Karl", "Riga", 2345, "Dev1.1")
    .addEmployee("Joe", "Wifi City", 2344, "Dev1.1");
```

This is an instantiation that uses a fluent API.

The amount of boilerplate code is reduced.
Company Instantiation in Java

Company c = company("ACME corporation")
    .addDepartment("Research")
    .setManager("Craig", "Redmond", 123456, "Research")
        .addEmployee("Erik", "Utrecht", 12345, "Research")
        .addEmployee("Ralf", "Koblenz", 1234, "Research")
    .addDepartment("Development")
        .setManager("Ray", "Redmond", 234567, "Development")
        .addSubDepartment("Dev1", "Development")
            .setManager("Klaus", "Boston", 23456, "Dev1")
            .addSubDepartment("Dev1.1", "Dev1")
                .setManager("Karl", "Riga", 2345, "Dev1.1")
                .addEmployee("Joe", "Wifi City", 2344, "Dev1.1");

This illustrates the concept of “Method Chaining”
Company Instantiation in Java

Company c = company("ACME corporation")
  .addDepartment("Research")
  .setManager("Craig", "Redmond", 123456, "Research")
  .addEmployee("Erik", "Utrecht", 12345, "Research")
  .addEmployee("Ralf", "Koblenz", 1234, "Research")
  .addDepartment("Development")
  .setManager("Ray", "Redmond", 234567, "Development")
  .addSubDepartment("Dev1", "Development")
  .setManager("Klaus", "Boston", 23456, "Dev1")
  .addSubDepartment("Dev1.1", "Dev1")
  .setManager("Karl", "Riga", 2345, "Dev1.1")
  .addEmployee("Joe", "Wifi City", 2344, "Dev1.1");

When creating employees, we always have to tell in which department we want to add them, since we do not have a flat structure.
Advanced Concepts

- Interface Instantiation
- Fluent API

Name to department mappings facilitates instantiating the composite structure.

```java
private Map<String, Department> depts = new HashMap<>();

public Company addDepartment(String name) {
    depts.put(name, new Department(name));
    return this;
}

public Company addSubDepartment(String name, String toDep) {
    Department sub = new Department(name);
    depts.put(name, sub);
    depts.get(toDep).addSubdept(sub);
    return this;
}

public static Company company(String name) {
    return new Company(name);
}
```

Returning the root object “this” allows us writing the complete method chain.

We consistently avoid constructors by adding a static method.
Advanced Concepts

- Interface Instantiation
- Fluent API

More examples from https://en.wikipedia.org/wiki/Fluent_interface#Java

```java
Author author = AUTHOR.as("author");
create.selectFrom(author)
    .where(exists(selectOne())
        .from(BOOK)
        .where(BOOK.STATUS.eq(BOOK_STATUS.SOLD_OUT))
        .and(BOOK.AUTHOR_ID.eq(author.ID)));

JPanel p = new JPanel();
Packer pk = new Packer( p );

JLabel l = new JLabel("Name:");
JTextField nm = new JTextField(10);

pk.pack( l ).gridx(0).gridy(0);
pk.pack( nm ).gridx(1).gridy(0).fillx();
```
Java 10

- Local Variable Type Inference

See http://openjdk.java.net/jeps/286

```java
private Map<String, Department> depts = new HashMap<>();

Map<String, Department> cDepts = company.getDepts();
```

In Java < 10 code we always have to write down the type in the declaration of a variable.
Java 10

- Local Variable Type Inference

See http://openjdk.java.net/jeps/286

```java
private Map<String, Department> depts = new HashMap<>();

public Map<String, Department> getDepts() {
    return depts;
}

var cDepts = company.getDepts();
```

We already know the type from the return type of the method `getDepts()`.
Java 10

- Local Variable Type Inference

See http://openjdk.java.net/jeps/286

```java
private Map<String,Department> depts = new HashMap<>();

public Map<String, Department> getDepts() {
    return depts;
}

var cDepts = company.getDepts();
```

New Eclipse versions recognize inferred types as well.
Summary

- ForEach
- Lambda Expressions
- Method Reference
- Streams
  - Map
  - Reduce
  - Filter
  - Concat
  - Parallel
  - Collect
  - Debugging
- Optionals
- Type Inference
- Object Initialization
- Fluent APIs

Questions?
Thank you for listening!