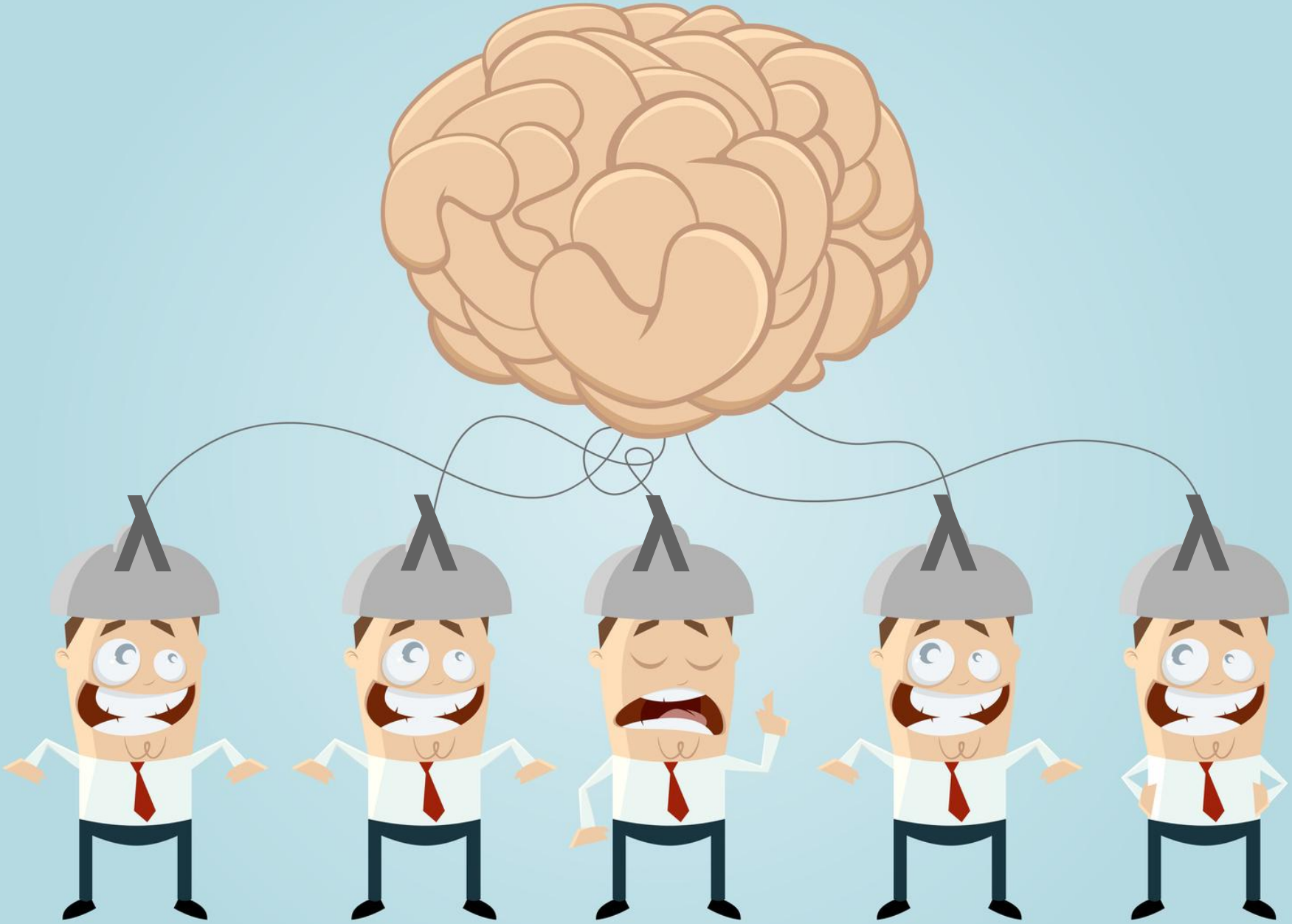
ConSol Consulting & Solutions

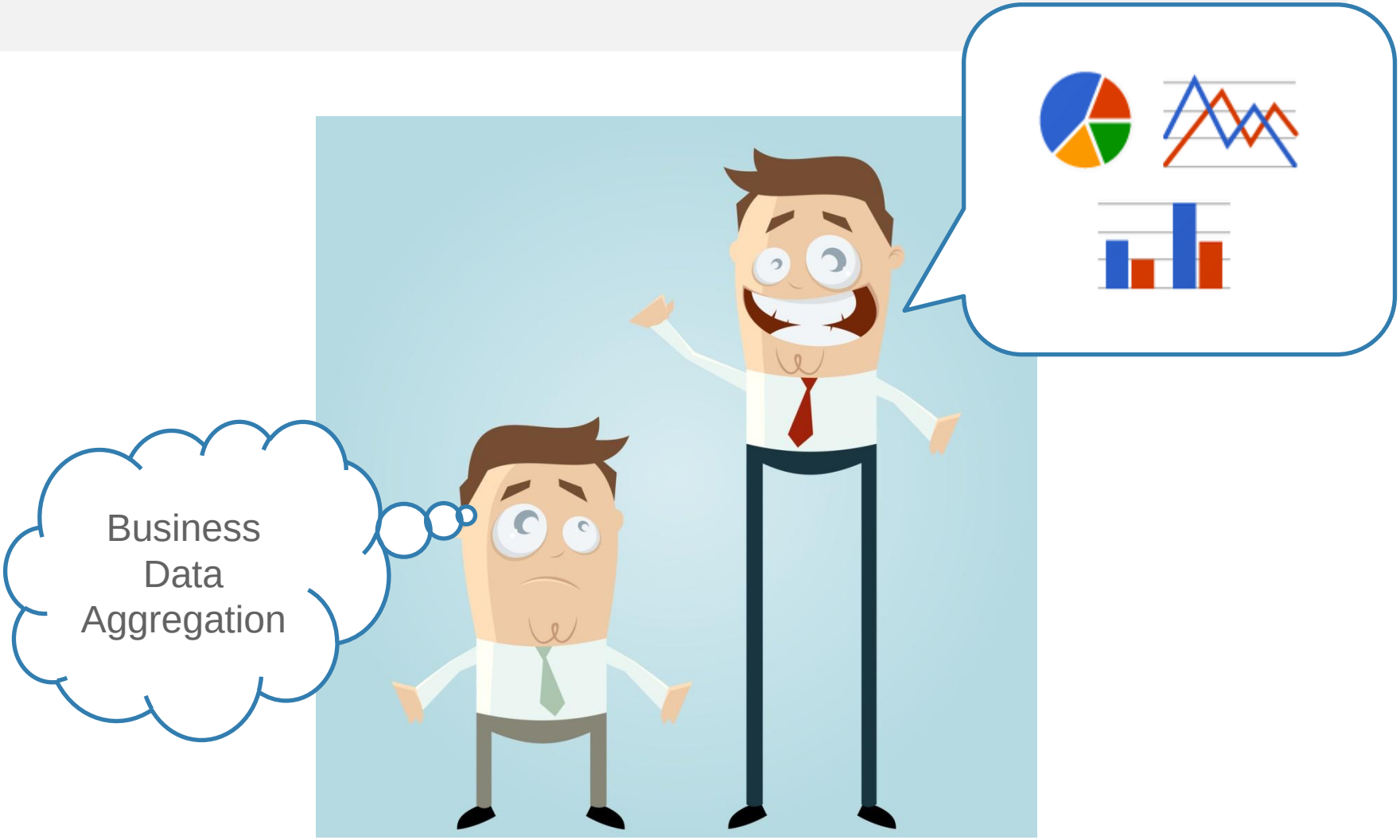
Wir unternehmen IT.

Efficient use of multi-core processors with lambdas and streams

Dr. Fabian Stäber


λ
λ
λ
λ
λ

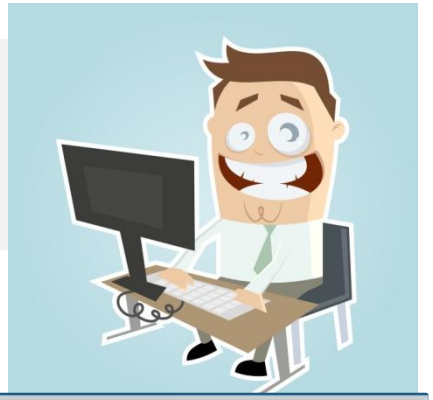
Example



Example

- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestDataGenerator
- View
  - JavaScript, HTML
- Controller
  - Dashboard.

Example

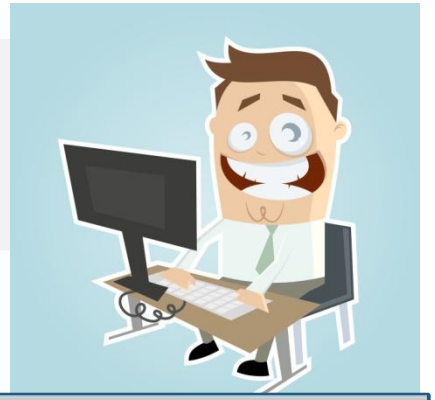


- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestDataGenerator
- View
  - JavaScript, HTML
- Controll
  - Dash

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>de.consol.research</groupId>
    <artifactId>lambda</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>war</packaging>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <failOnMissingWebXml>false</failOnMissingWebXml>
    </properties>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
                ee-api</artifactId>
            ersion>

                finalName>
```

```
<dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>7.0</version>
</dependency>
```

Example



- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestDat
- View
  - JavaSc
- Controller
  - Dashboard.

```
package de.consol.research;

import ...

@ApplicationPath("api")
public class RestConfig
        extends Application
{}
```

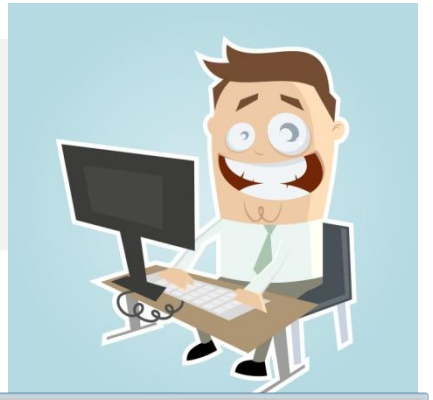
```
@ApplicationPath("api")
public class RestConfig
        extends Application
{}
```

Example

- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestD
- View
  - Java
- Controll
  - Dashboard.

```
package de.consol.research;

import java.time.LocalDate;

public class Contract {

    private final State state;
    private final int priceInCent;
    private final LocalDate date;

    public Contract(State state, int priceInCent, LocalDate date)
    {
        this.state = state;
        this.priceInCent = priceInCent;
        this.date = date;
    }

    public State getState() {
        return state;
    }

    public int getPriceInCent() {
        return priceInCent;
    }

    public LocalDate getDate() {
        return date;
    }
}
```

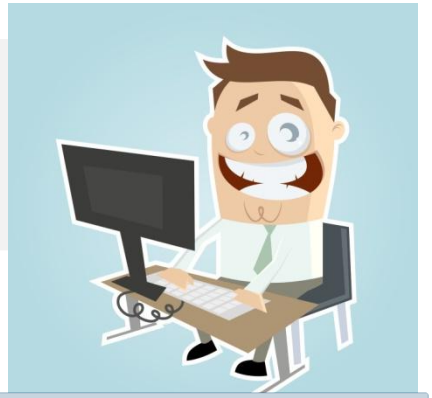
Example



- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestDataGen
- View
  - JavaScript, HTML
- Controller
  - Dashboard.

```
package de.consol.research;

public enum State {
    BW, // Baden-Württemberg
    BY, // Bayern
    BE, // Berlin
    BB, // Brandenburg
    TH  // Thüringen
}
```

Example



- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestDataGenerator
- View
  - Jav
- Contro
  - Das

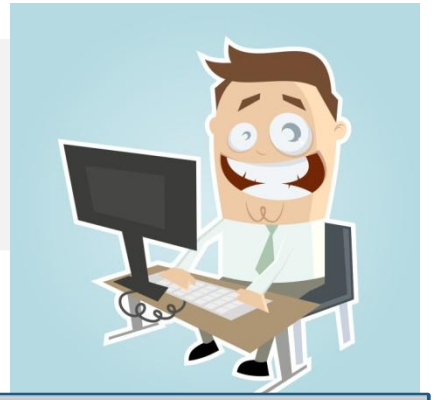
```
package de.consol.research;

import ...

public class TestDataGenerator {

    public static List<Contract> makeTestData() {
        return Arrays.asList(
            new Contract(BY, 2000, now()),
            new Contract(BY, 3500, now().minusDays(1)),
            new Contract(NW, 7000, now().minusMonths(1))
                                        .minusDays(2))
        );
    }
}
```

```
public class TestDataGenerator {

    public static List<Contract> makeTestData() {
        return Arrays.asList(
            new Contract(BY, 2000, now()),
            new Contract(BY, 3500, now().minusDays(1)),
            new Contract(NW, 7000, now().minusMonths(1)),
```

Example

- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestDataGenerator
- View
  - JavaScript, HTML
- Controller
  - Dashboard.

Example



- Boilerplate
  - pom.xml
  - RestConfig (modern web.xml)
- Model
  - Contract
  - State enum
  - TestD
- View
  - Java
- Controll
  - Dash

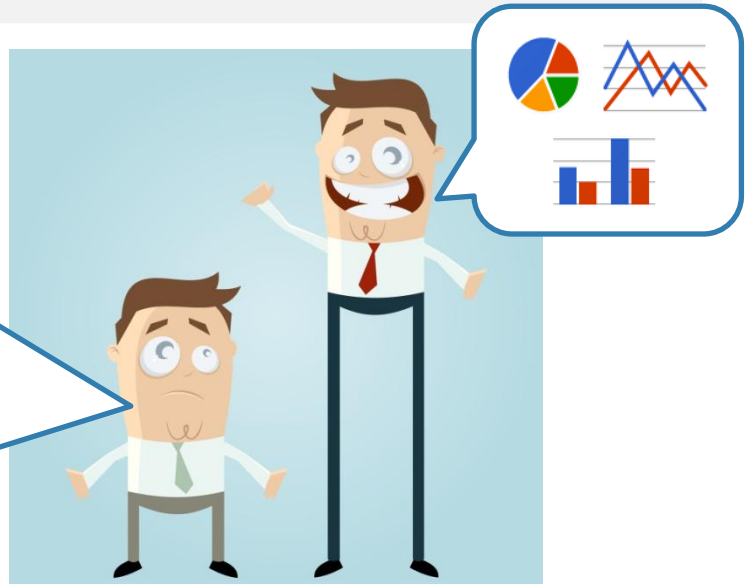
```
package de.consol.research;

import ...

@Path("dashboard")
public class Dashboard {

    private List<Contract> contracts =
        TestDataGenerator.makeTestData();
}
```


```
@Path("dashboard")
public class Dashboard {

    private List<Contract> contracts =
        TestDataGenerator.makeTestData();
}
```

Stream API

Business
Data
Aggregation
with

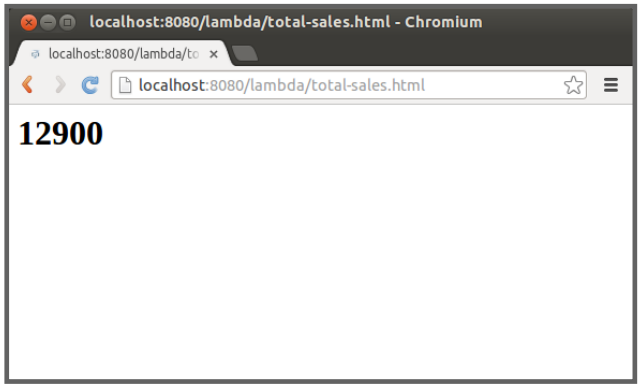
Stream API



Stream API

Task 1:
Total Sales

GET /lambda/api/dashboard/total-sales



12900

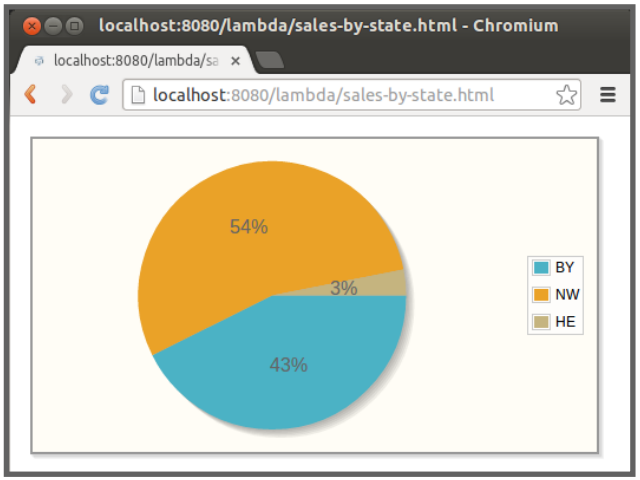
Stream API

```
@GET
@Path("total-sales")
@Produces(APPLICATION_JSON)
public int totalSales() {
    return contracts
        .stream()
        .filter(contract -> contract.getDate().isAfter(now().minusYears(1)))
        .mapToInt(Contract::getPriceInCent)
        .sum();
}
```

Stream API

Task 2:
Sales by State

GET /lambda/api/dashboard/sales-by-state



```
{
    "BY":  5500,
    "NW": 7000,
    "HE":  400
}
```

Stream API

```
@GET
@Path("sales-by-state")
@Produces(APPLICATION_JSON)
public Map<State, Integer> salesByState() {
    return contracts
        .stream()
        .filter(contract -> contract.getDate().isAfter(now().minusYears(1)))
        .collect(Collectors.groupingBy(
                    Contract::getState,
                    Collectors.summingInt(Contract::getPriceInCent)));
}
```

Stream API

```
@GET
@Path("sales-by-state")
@Produces(APPLICATION_JSON)
public Map<State, Integer> salesByState() {
    return contracts
        .stream()
        .filter(contract -> contract.getDate().isAfter(now().minusYears(1)))
        .collect(Collectors.groupingBy(
                Contract::getState,
                Collectors.summingInt(Contract::getPriceInCent)));
}
```

Can we re-use Lambda expressions?

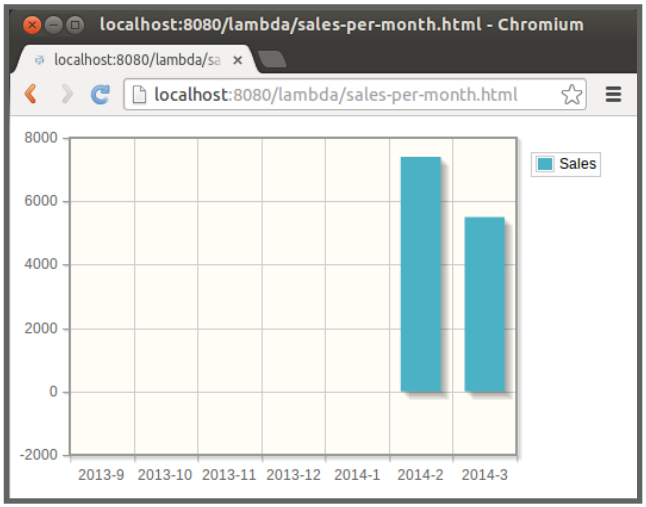
Stream API

```
private Predicate<Contract> last12months =
    contract -> contract.getDate().isAfter(now().minusYears(1));

@GET
@Path("sales-by-state")
@Produces(APPLICATION_JSON)
public Map<State, Integer> salesByState() {
    return contracts
        .stream()
        .filter(last12months)
        .collect(Collectors.groupingBy(
                    Contract::getState,
                    Collectors.summingInt(Contract::getPriceInCent)));
}
```

Stream API

Task 3:
Sales per Month

GET /lambda/api/dashboard/sales-per-month



```
{
    “2014-03": 5500,
    “2014-02": 7400
}
```

Stream API

```
@GET
@Path("sales-per-month")
@Produces(APPLICATION_JSON)
public Map<String, Integer> salesPerMonth() {
    return contracts
        .stream()
        .filter(last12months)
        .collect(Collectors.groupingBy(
              c -> c.getDate().getYear() + "-" + c.getDate().getMonthValue(),
              Collectors.summingInt(Contract::getPriceInCent)));
}
```

Stream API

Get things done in few lines of code…

…really ?

Stream API

```
public int totalSales() {
    int resultInCent = 0;
    for (Contract contract : contracts) {
        resultInCent += contract.getPriceInCent();
    }
    return resultInCent;
}
```

Java 7

Java 8

```
public int totalSales() {
    return contracts
        .stream()
        .mapToInt(Contract::getPriceInCent)
        .sum();
}
```

Stream API

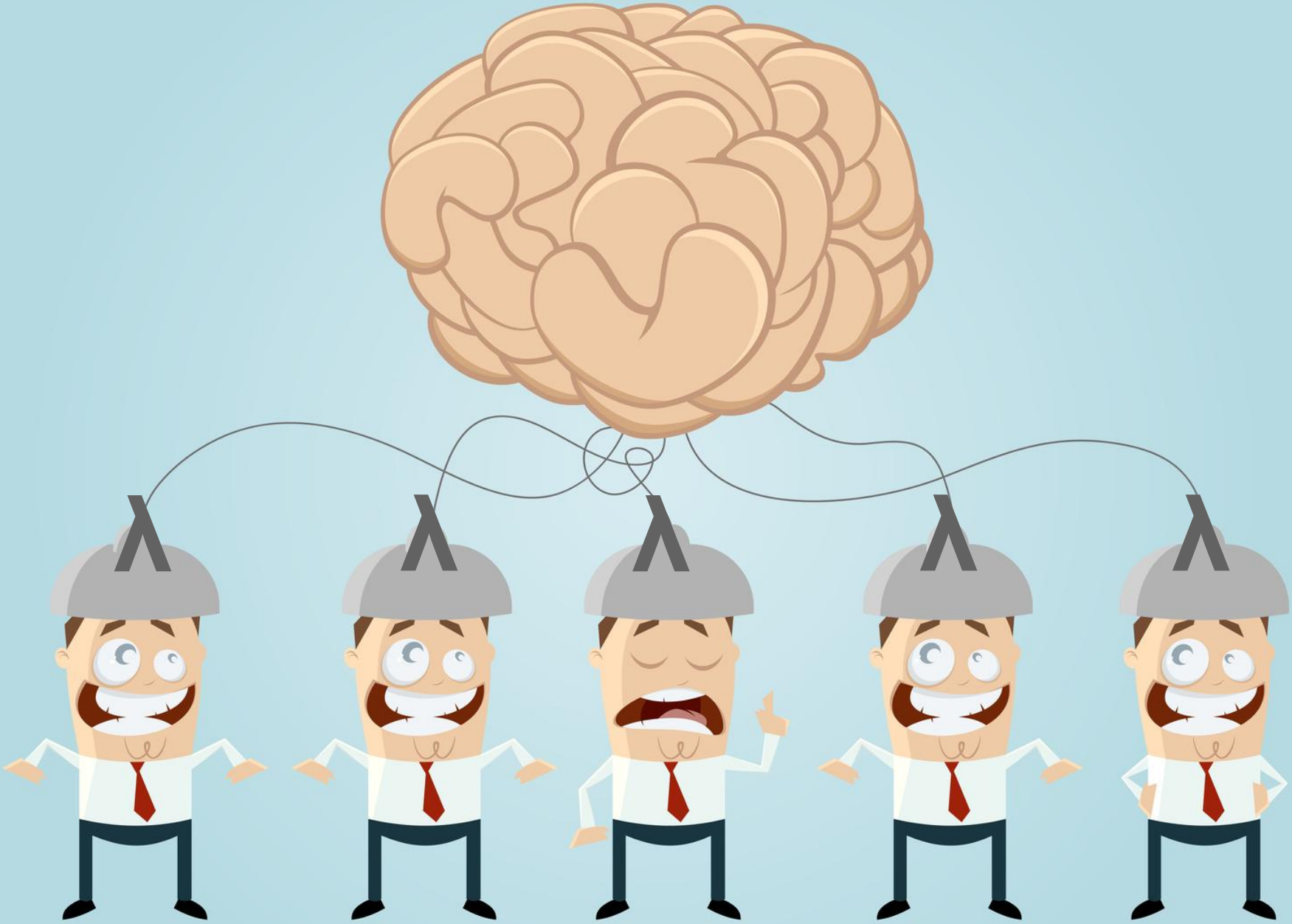
Why all that hype?

Stream API

How To Make Things Faster

- Keep Data in RAM ✓
- Avoid DB Roundtrips ✓
- Snappy Lightweight AJAX Front-End ✓

- Use Multiple Cores ?


λ
λ
λ
λ
λ

Use Multiple Cores

parallelize

Use Multiple Cores

```
public int totalSales() {
    int resultInCent = 0;
    for (Contract contract : contracts) {
        resultInCent += contract.getPriceInCent();
    }
    return resultInCent;
}
```

Java 7

Java 8

```
public int totalSales() {
    return contracts
        .stream()
        .mapToInt(Contract::getPriceInCent)
        .sum();
}
```

Use Multiple Cores

```
public int totalSales() {
    int resultInCent = 0;
    for (Contract contract : contracts) {
        resultInCent += contract.getPriceIn
    }
    return resultInCent;
}
```

Do It Yourself

synchronized, volatile, Lock, Executor, Callable, …

Let the Library Do It For You

```
public int totalSales() {
    return contracts
            .stream()
            .mapToInt(Contract::getPriceInCent)
            .sum();
}
```

Use Multiple Cores

```
public int totalSales() {
    int resultInCent = 0;
    for (Contract contract : contracts) {
        resultInCent += contract.getPriceIn
    }
    return resultInCent;
}
```

Do It Yourself

synchronized, volatile, Lock, Executor, Callable, …

Let the Library Do It For You

```
public int totalSales() {
    return contracts
        .parallelStream()
        .mapToInt(Contract::getPriceInCent)
        .sum();
}
```

Use Multiple Cores

Thread Safe Operations

```
// Set of all states in contract data
Set<State> states = new HashSet<State>();
contracts
    .parallelStream()
    .map(Contract::getState)
    .forEach(s -> states.add(s));
```

Not Thread Safe

Use Multiple Cores

Thread Safe Operations

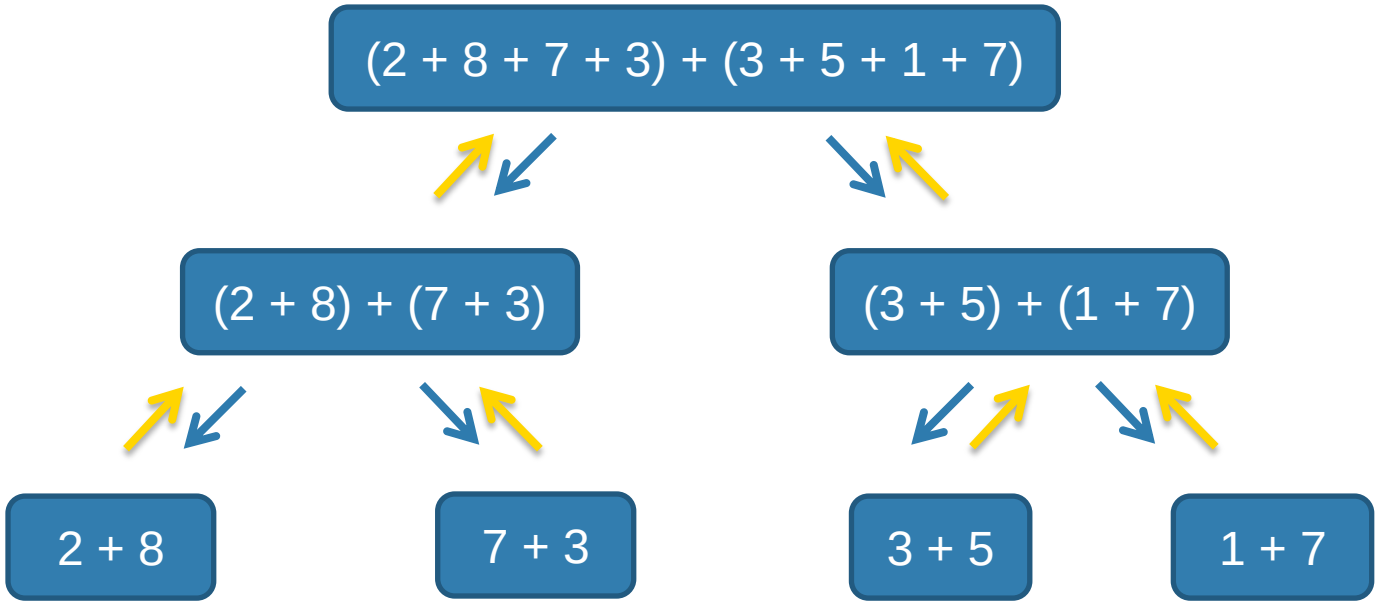
```
// Set of all states in contract data
Set<State> states = contracts
    .parallelStream()
    .map(Contract::getState)
    .collect(Collectors.toSet());
```

Thread Safe ✓

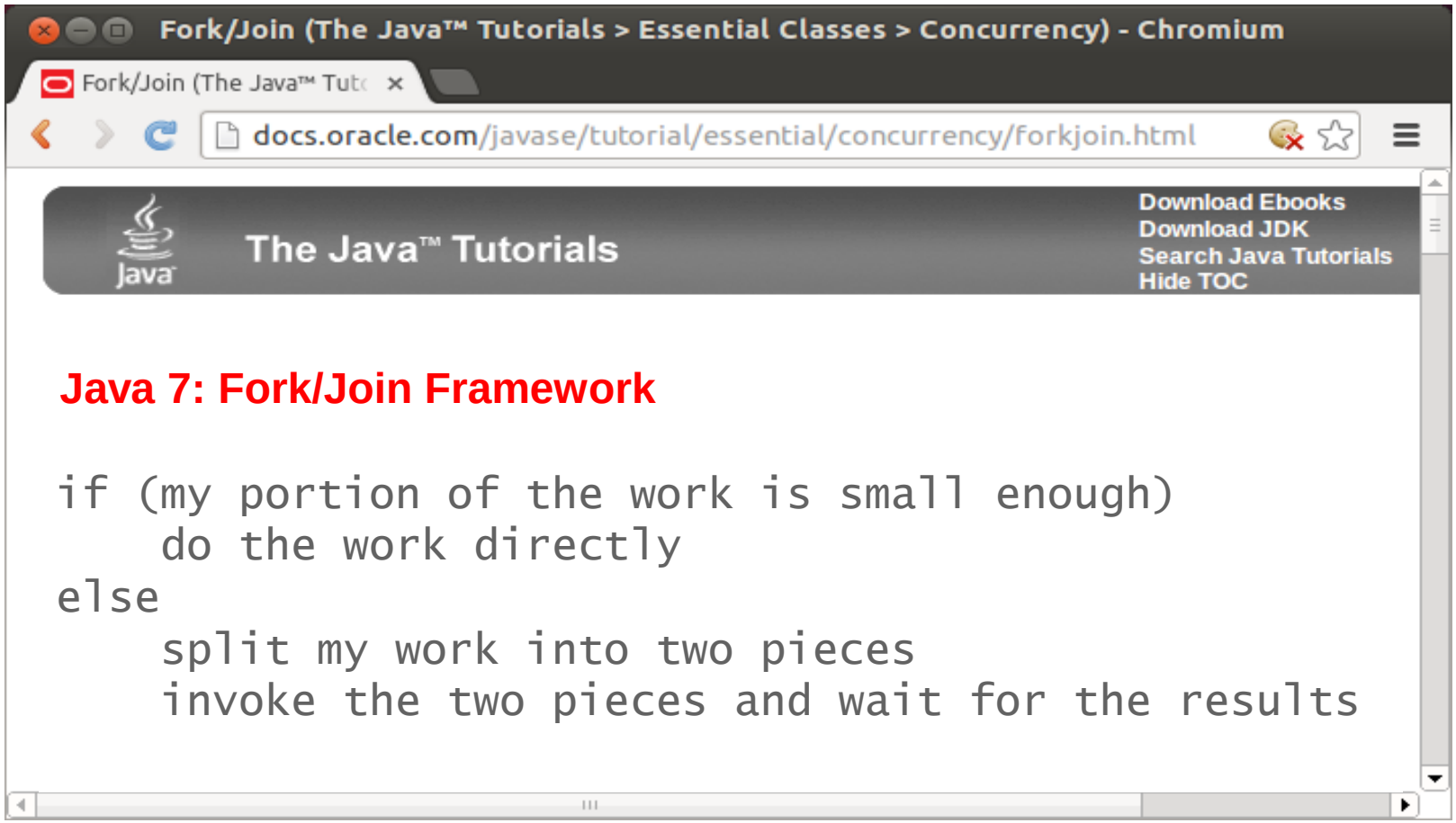
Under the Hood

(internal stream API implementation)

Under the Hood



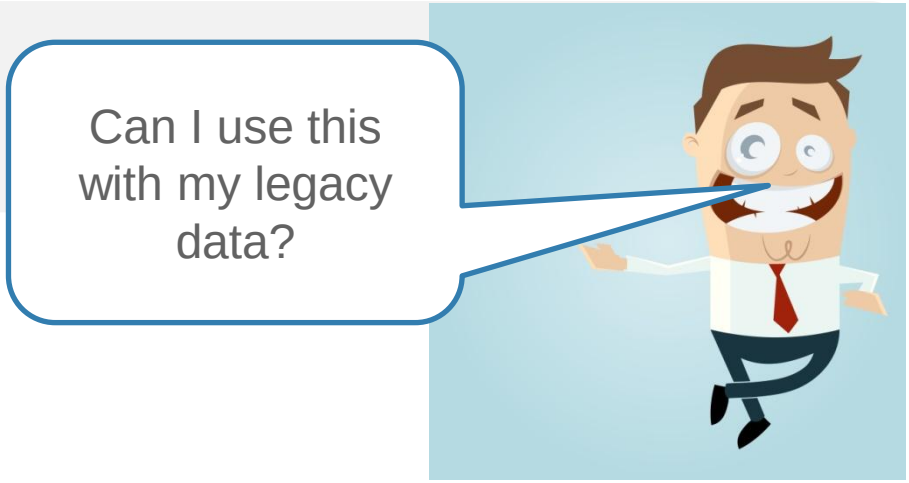
Under the Hood



Under the Hood


```
public class LegacyContractsData implements Iterable<Contract> {

    private State[] states;
    private int[] pricesInCent;
    private LocalDate[] dates;

    // ...
}
```

Under the Hood



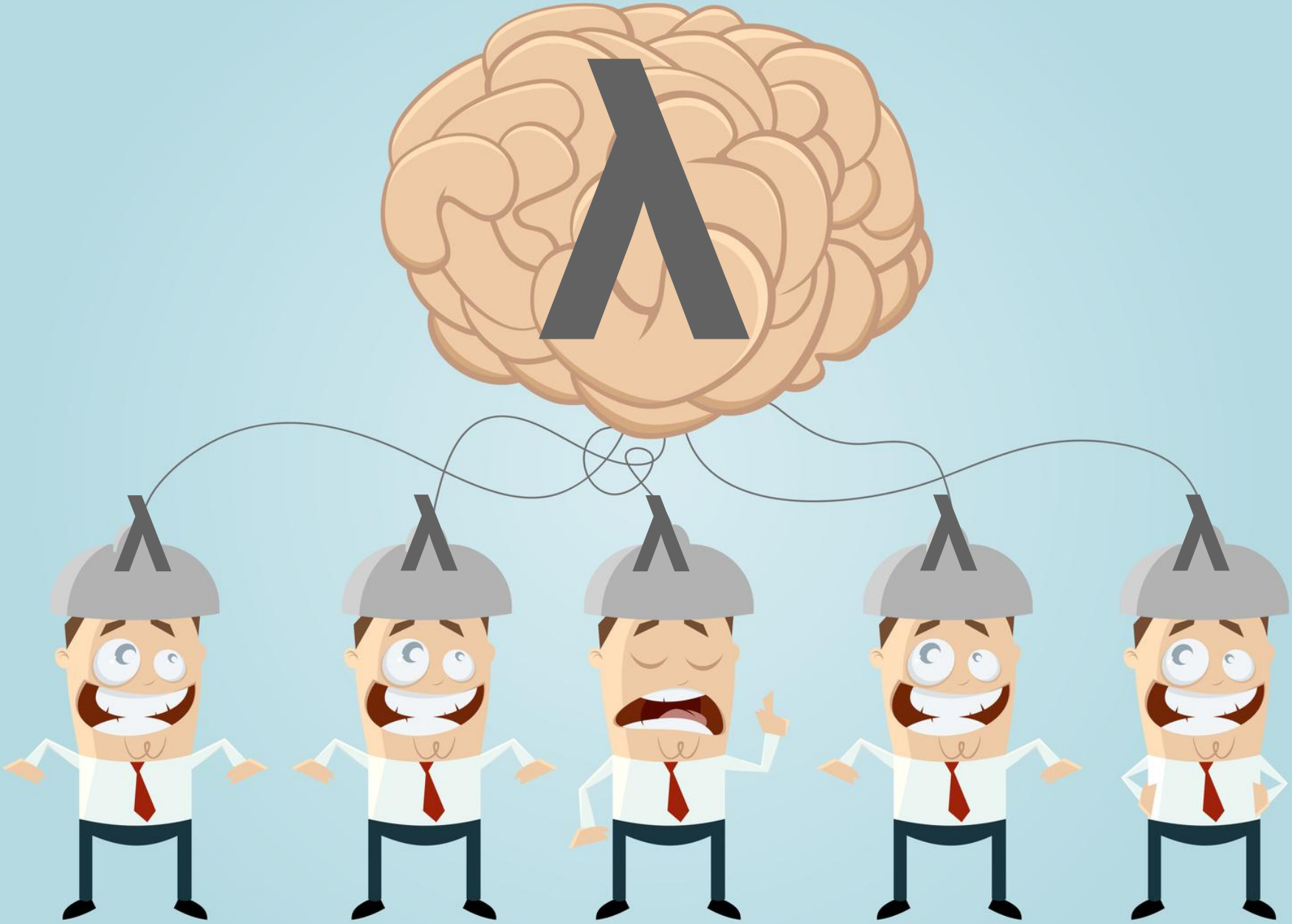
Tell the stream API:

- how to split your data into chunks
- how to iterate over a chunk.

Implement a Spliterator.

Under the Hood

```
public class MyOwnSpliterator implements Spliterator<Contract> {

    @Override
    public Spliterator<Contract> trySplit() {
        // split data into chunks
    }

    @Override
    public boolean tryAdvance(Consumer<? super Contract> action) {
        // iterate over a chunk
    }

    @Override
    public long estimateSize() {
        // optional metadata
    }

    @Override
    public int characteristics() {
        // optional metadata
    }
}
```


λ
λ
λ
λ
λ
λ

Summary

Efficient use of multi-core processors with lambdas and streams:

- Library does it for you
- Use thread safe operations
- Implement Spliterators