Java Programming 3

Week 1

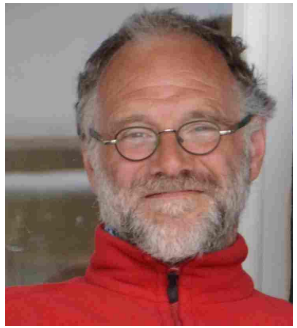
KdG

Welcome!

Who are we?

- Hans Vochten
  - Java Programming 3
  - Integration 3

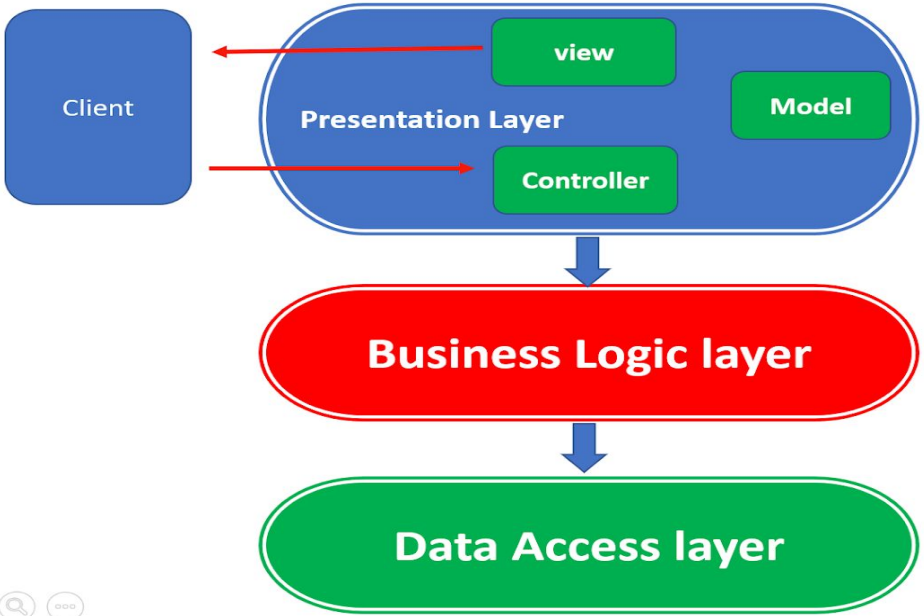
- Jan de Rijke
  - Java Programming 1
  - Java Programming 2
  - Java Programming 3
  - Java Programming 4

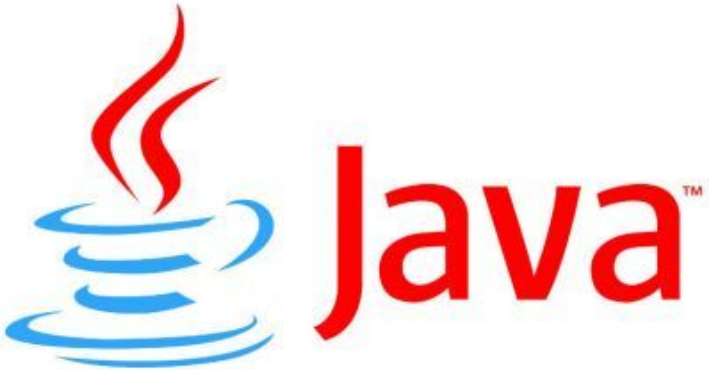


KdG

Programming 3?

- Using Java to build enterprise applications
  - Using the **Spring Framework**
  - Using **Advanced Java Features and Libraries**
  - Using **3-tier architecture**





KdG

Software?

- IDE: Intellij 2023.2 or later
- Java: Oracle JDK 17
- Git

LTS and stable…!

→ Instructions and licenses: see Canvas

KdG

How?

- 12 weeks - 6 hours/week
- Every week
  - Learn new Spring Framework features
  - Learn new Java Language features or Java libraries
  - Study demos and examples
  - Apply in exercises
  - Apply in your *individual project*
- Every 2 weeks
  - Sprint deliverable of the next part of the *individual project*

KdG

Individual project

- Based on your own dataset
- Web application with Java back-end connected to a database
- Possibility to **C**reate, **R**ead, **U**pdate, **D**elete items
- Implemented using a 3-tier architecture
- Every week you get a new assignment:
  - Apply the topics of that weeks to the project
- Every 2 weeks you tag a release in gitlab
- The application “*grows”* towards a final product

KdG

Individual project: dataset

- You choose your dataset: see Canvas for specs
- Submit your dataset on MS Teams
  - We will review and approve
  - Everyone has different dataset!

KdG

Evaluation?

- 6 release tags → for feedback
- Final deliverable @ end of second period
- Oral exam:
  - Demonstration and explanation of your personal CRUD application
  - Oral interrogation about the topics of the course

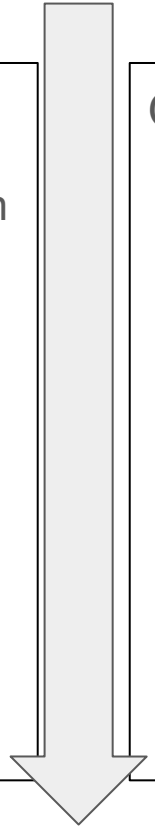
→ Important: **you need a project to be able to participate in the oral exam!**

KdG

Agenda (more or less…)

Spring Framework:

- Spring basics: Dependency Injection - the Application Context
- Spring Boot
- Spring MVC with Thymeleaf
- Validation
- JDBC Templates
- Object Relational Mapping with Hibernate
- ...



Other topics:

- Gradle/Git
- What is a 3-tier architecture?
- Annotations
- Streams
- XML - JSON
- Generics
- Logging
- ...

KdG

Agenda this week

Introduction
Where are we heading to…?
Java project with Gradle - Git
Lambdas and streams
Annotations
JSON - Gson

KdG

Agenda this week

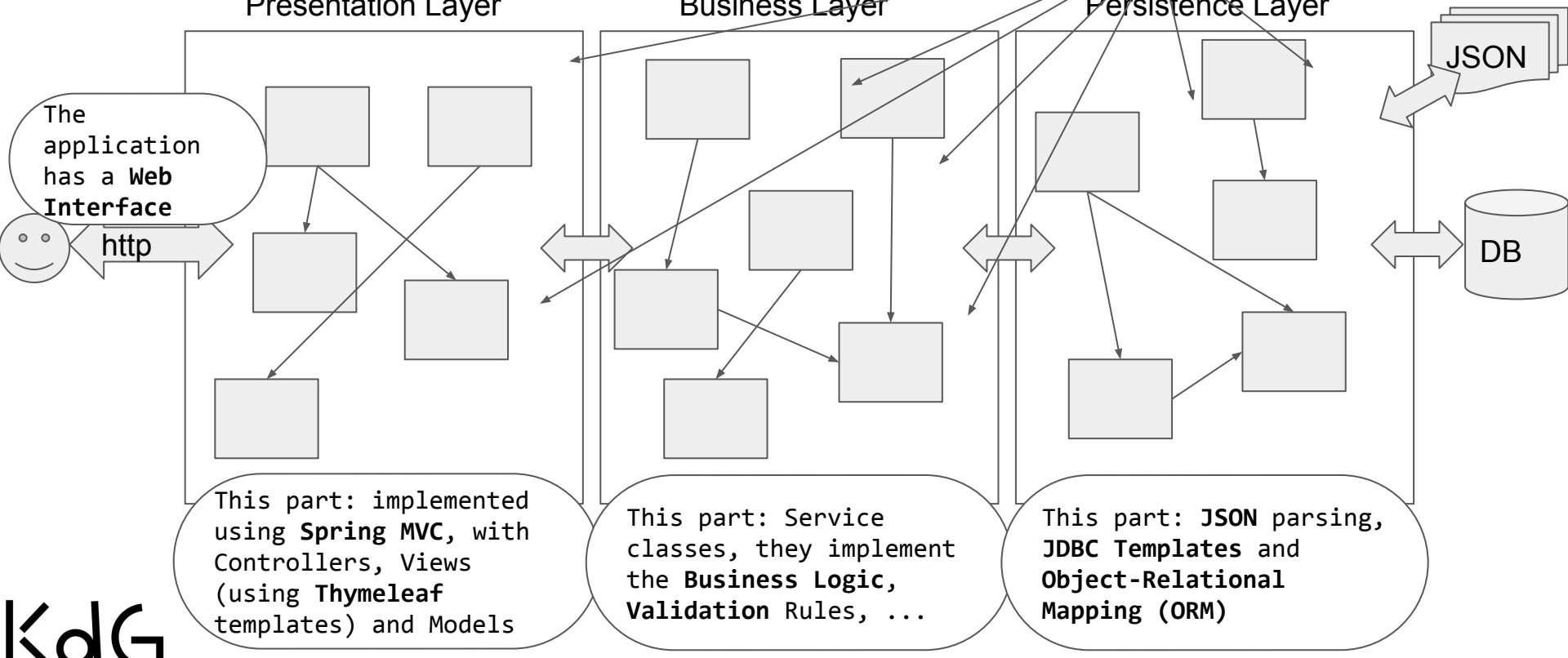
Introduction
Where are we heading to…?
Java project with Gradle - Git
Lambdas and streams
Annotations
JSON - Gson

KdG

A 3-t… using the Ja… ork

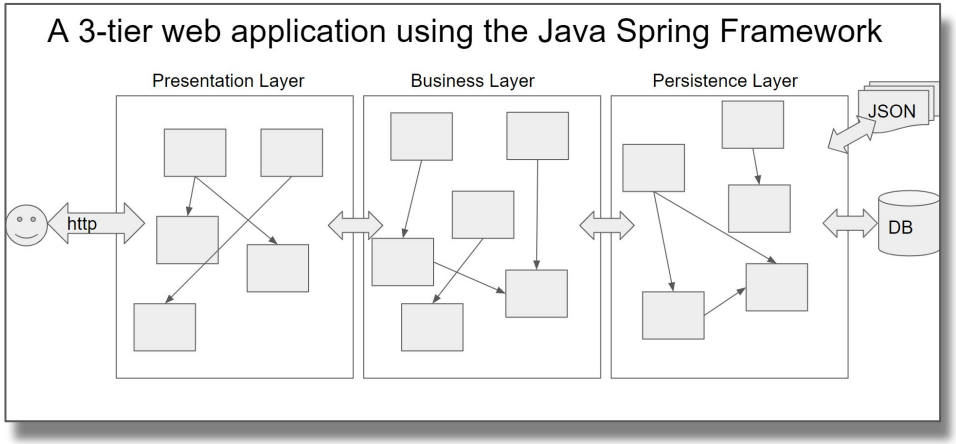
We use a **3-layered architecture** and think about the dependencies between the layers...

A lot of *Components*! (aka *Beans* which are Java Classes). They are managed by a **Spring Container**



KdG

Before we can start with →



We need some extra Java tooling:

- **Gradle**: build automation
- **Git**: version control
- **Annotations**: metadata to configure stuff
- **Lambdas and Streams**: introduce functional programming and easy collections handling
- **JSON and Gson**: simple data interchange format and how to parse it

KdG

Agenda this week

Introduction
Where are we heading to…?
Java project with Gradle - Git
Lambdas and streams
Annotations
JSON - Gson

KdG

Gradle: build automation tool

- Configure and automate the build process
- Gradle defines *tasks* to
  - Download the dependent libraries (“dependencies”)
  - Compile the source code to class files
  - Make jar files
  - Run tests
  - Deploy to the server
  - ...
- Very good integration with IntelliJ!
- Read these tutorials:
  - What is Gradle?
  - Building Java Applications Sample (just read this one and try to understand…)

KdG

Gradle: build.gradle.kts file (1/2)

- Configuration of the build process for the application
  - Specify the main class
  - Specify the dependencies
  - …
- Uses the Kotlin syntax…
- Example:

```
plugins {
   id("java")
   id("application")
}

application {
   mainClass.set("be.kdg.programming3.StartApplication")
}

group = "be.kdg.java3"
version = "1.0-SNAPSHOT"
```

Gradle uses plugins, you specify them here. We add the “application” plugin to be able to run as a Java application

Here we configure the “application” plugin: we tell it where to find the main class…

KdG

Gradle: build.gradle.kts file (2/2)

- Configuration of the build process for the application
  - Specify the main class
  - **Specify the dependencies**
  - …
- Uses the Kotlin syntax…
- Example:

```
…
repositories {
   mavenCentral()
}

dependencies {
   testImplementation(platform("org.junit:junit-bom:5.9.1"))
   testImplementation("org.junit.jupiter:junit-jupit
}

…
```

Here you can specify what repository to use for the dependencies.

Intellij already added 2 dependencies: the junit libs for testing. We can add more dependencies here…

KdG

demo_gradle

- In this demo we will
  - Create a Java - Gradle project in Intellij
  - Add a StartApplication class with a main method → “Hello World with Gradle!”
  - Modify the gradle.build file to be able to run the application as a Gradle task
  - Run from Gradle view (Intellij) - run from Terminal

KdG

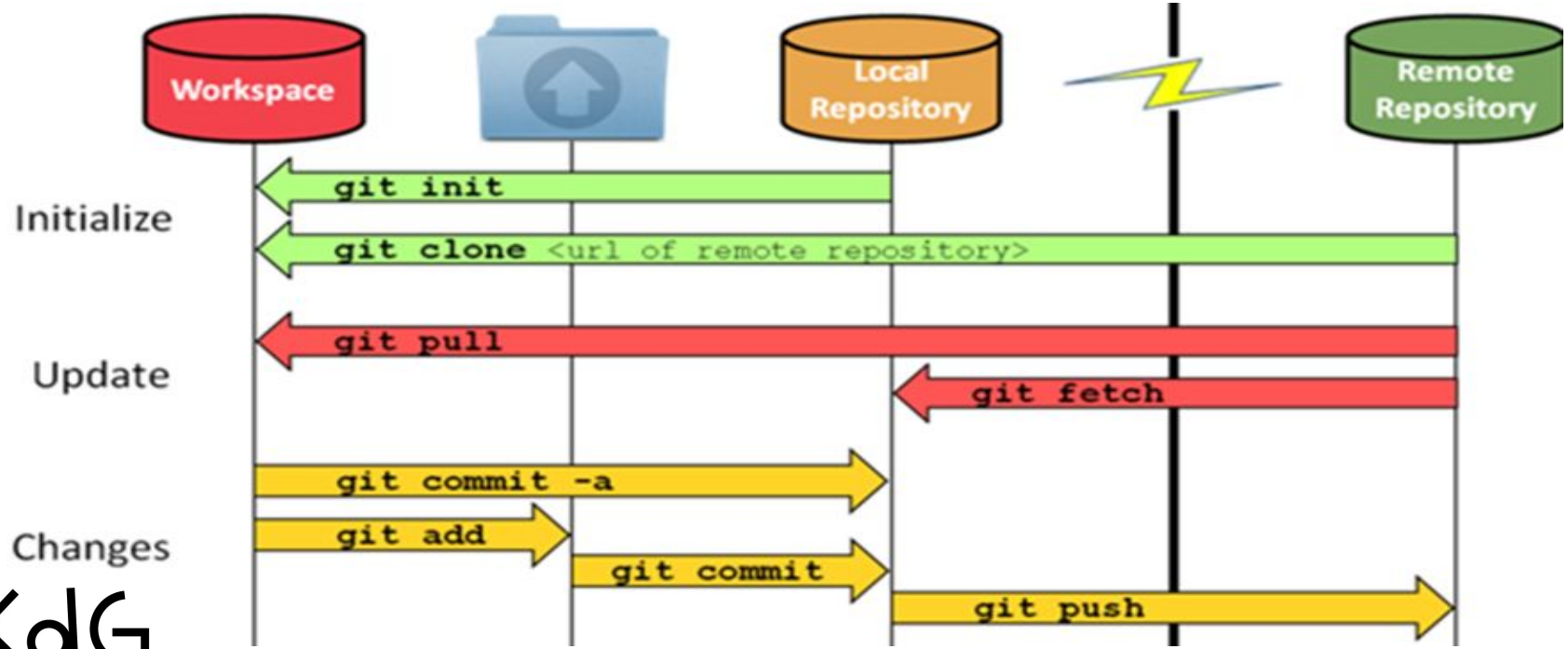
Exercise 1

- Create a new project “persons” project (group: be.kdg.programming3)
- Modify the build.gradle to be able to run as a Java Application
- Add a StartApplication class with a main method
- Add a Person class, a person has a tellJoke method
- Add a dependency to the random-joke-crawler (look up @ https://search.maven.org/)
- Implement the tellJoke method in a Person class, run from main method
- Run application from a terminal as a Gradle task

KdG

Git: version control

- Basic commands:



KdG

demo_git

- In this demo we will
  - Create an online git repository on gitlab
  - Clone the repository on our local machine
  - Add a Java Gradle project to the repository
  - Add and commit our files
  - Push the files to the online repository

KdG

Exercise 2

- Push exercise 1 to a gitlab repository
  - Make an account on gitlab (use your kdg email account!)
  - Create a project “persons” (you can first create a subgroup for it?)
  - Clone the project on your local machine
  - Copy your local project from exercise 1 into this empty cloned folder
  - Add all files
  - Commit the files
  - Push the files to gitlab
- Make a change to the project
  - Make some changes to the project
  - Add the change to the local git repository
  - Commit the changes to the local git repository
  - Push the changes to the online repository
  - Review the changes on gitlab

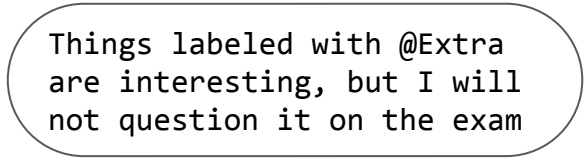
KdG

Agenda this week

Introduction
Where are we heading to…?
Java project with Gradle - Git
Lambdas and streams
Annotations
JSON - Gson

KdG

Lambdas



@Extra

- Introduced in JDK 8 (2014)
- Introduces *functional programming* in Java
- Makes it easy to pass a function as a parameter to another function (functions are *first-class citizens*)
- Such functions are called *Lambda expressions*
- Lambdas make use of the arrow syntax: `->`
- Advantages of lambdas
  - More readable code (for example event handling code)
  - Processing of collections is more intuitive (no more for-loops…)
  - Parallel processing is easier

KdG

Example of Lambda

```
button.setOnAction(event -> System.out.println("You clicked"));
```

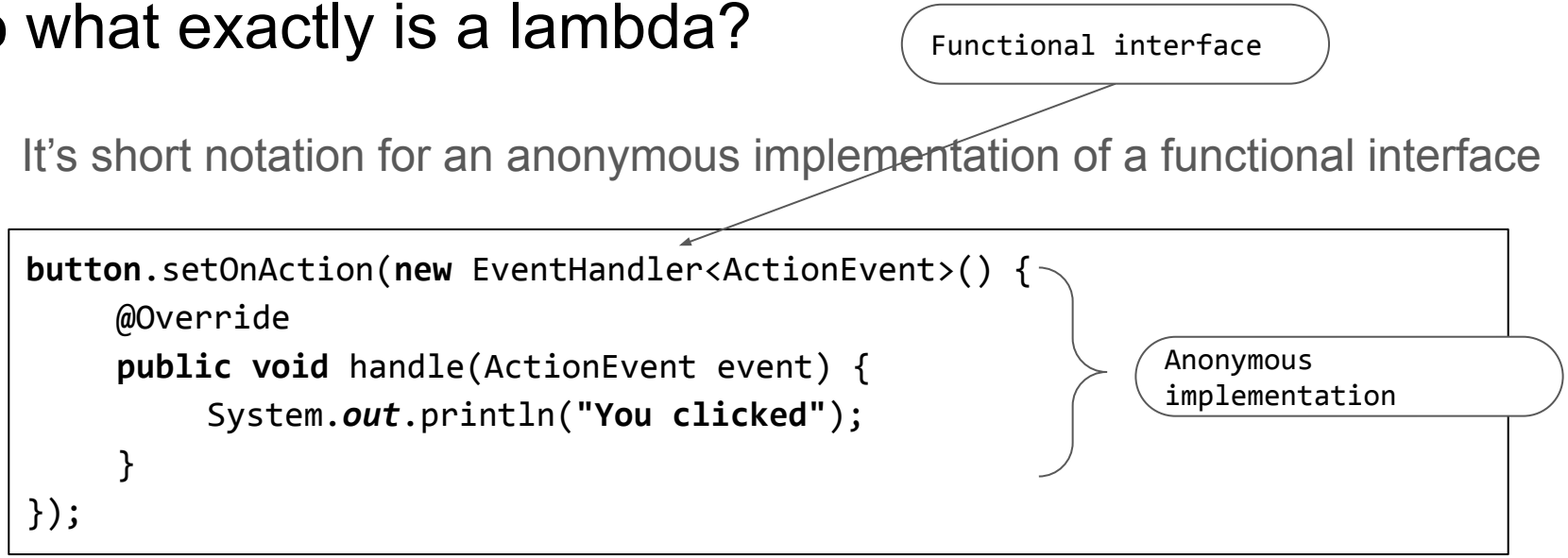
- You used lambas in JavaFX event handling!
- Before lambdas this would be:

```
button.setOnAction(new EventHandler<ActionEvent>() {
     @Override
     public void handle(ActionEvent event) {
          System.out.println("You clicked");
     }
});
```

KdG

So what exactly is a lambda?

- It's short notation for an anonymous implementation of a functional interface



KdG

General syntax

- Replace:

```
new SomeInterface() {
     @Override
     public SomeType someMethod(args) {
          body
     }
}
```

- With

```
(args) -> { body }
```

KdG

Other example: `Comparator` interface

```
Arrays.sort(testStrings, new MyComparator());
class MyComparator implements Comparator <String>() {
     @Override
     public int compare(String s1, String s2) {
     return s1.length() - s2.length();
     }
});
```

Not anonymous implementation

```
Arrays.sort(testStrings, new Comparator<String>() {
     @Override
     public int compare(String s1, String s2) {
     return s1.length() - s2.length();
     }
```

Anonymous implementation

```
Arrays.sort(testStrings,
    (String s1, String s2) -> {
        return s1.length() - s2.length();
    }
);
```

Lambda syntax

```
Arrays.sort(testStrings, (s1, s2) ->
s1.length()-s2.length());
```

Even shorter!

Lambdas: remarks

- Implementation only returns statement: omit the {} and the return
- Only one parameter: omit the ()

```
button.setOnAction(e -> System.out.println("Action required: " + e));
```

- No parameter: ()
- Lambdas can only be used with *functional interfaces*: interfaces **with only one method**!
  - Comparator and EventHandler are functional interfaces
- Lambdas are very powerful when used in combination with *Streams*

KdG

demo_lambda

- In this demo we will
  - Show how to implement the Comparator interface with an anonymous inner class
  - Replace that implementation with a lambda

KdG

Lambdas: remarks

- You can create your own functional interfaces: add the @FunctionInterface annotation.
- A functional interface can only have 1 method
- You seldom create your own functional interfaces: you can find a list of functional interfaces in java.util.function

```
@FunctionalInterface
public interface MyFunctionalInterface {
   double myMethod(int myParam);
}
```

KdG

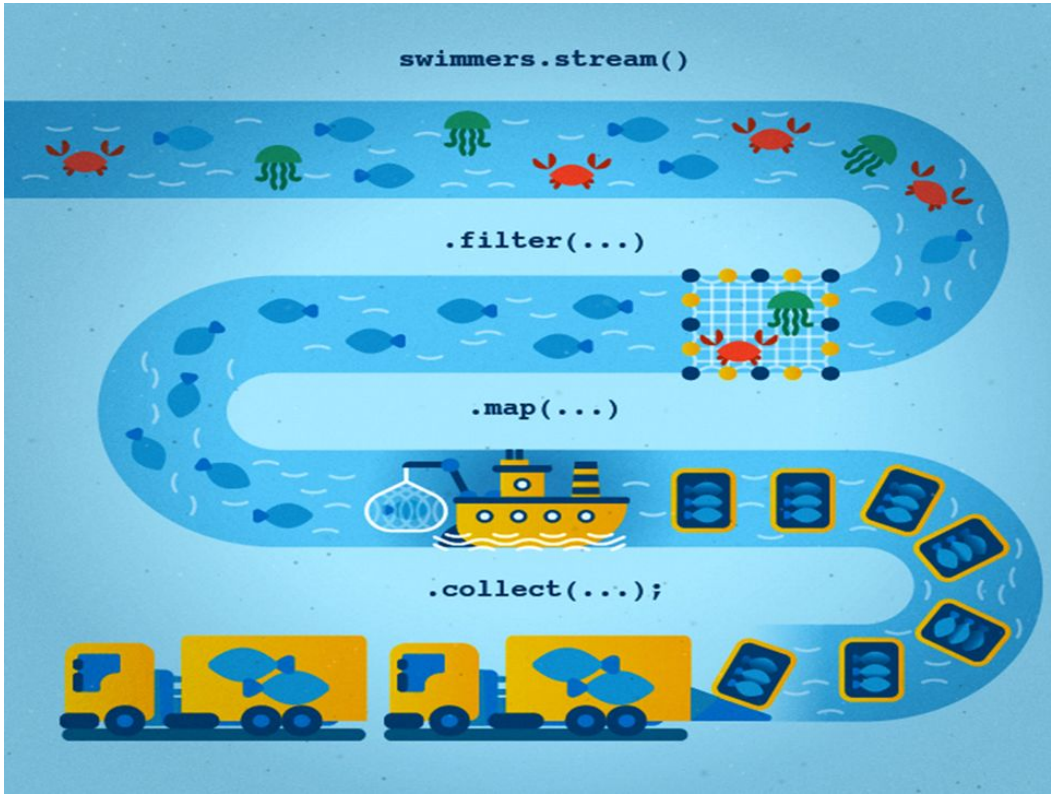
Exercise 3: lambda

- Can you write a void function named `calculator` that has a lambda as a parameter. This lambda has one input (int) and one output (double). Create your own functional interface for this!
- The `calculator` function uses the lambda with the numbers from 0 to 10 as parameters and prints out “Calculating…” and then the result of the lambda
- Now use this `calculator` function with
  - a lambda that calculates 10 to the power of n.
  - a lambda that calculates the factorial of a number
- Make a List of 10 lambda’s: the first calculates 0 to the power of n, the second 1 to the power and so on.
- Run through the List and pass all lambdas to the calculator function.

KdG

Streams

- Collections can be turned into a *stream*
- You can define *operations* on the elements of a stream using lambdas
- Type of operations:
  - **Filtering** the stream of elements
  - **Transforming** the stream of elements
  - **Collecting** the stream of elements



Good streams tutorial: https://stackify.com/streams-guide-java-8/

KdG

Working with streams

1. Create a **stream**
2. Specify 0 or more **“intermediate operations”**
   - Operation on stream (filter, transform), use lambda
   - Result is new stream
3. Specify a **“terminal operation”**
   - Consumes the stream: turns it into a final object

KdG

Functional interfaces are added to Java to use for the operations

Functional Interface	Single method
`Predicate<T>`	`boolean test (T t)`
`Function<T,R>`	`R apply (T t)`
`BiFunction<T,U,R>`	`R apply (T t, U u)`
`Consumer<T>`	`void accept (T t)`
`BiConsumer<T,U>`	`void accept (T t, U u)`
`Supplier<T>`	`T get ()`
`BinaryOperator<T>`	`T apply(T t1, T t2)`
`UnaryOperator<T>`	`T apply(T t)`

All these functional interfaces make use of **generics**!
This will be a topic one of the following lessons...

Example

Count the number of articles that cost less than €400

```
List<Article> articles = Articles.getArticles();
```

Classic loop:

```
long count = 0;
for (Article article: articles) {
     if (article.getPrice() > 400.0) {
          count++;
     }
}
System.out.println(count);
```

With stream:

```
long count =
   articles.stream()
     .filter(a -> a.getPrice() > 400.0)
     .count();
System.out.println(count);
```

This lambda uses the **Predicate** functional interface

KdG

Examples of creation of streams


```
List<T> list = …;
list.stream()
    .intermediate
    .intermediate
    ...
    .terminal
```

```
T[] array = … ;
 Stream.of(array)
 .intermediate
 .intermediate
  …
 .terminal
```

```
Stream<Integer> numberStream = Stream.generate(()->random.nextInt(900) + 100)
```

```
Stream<Integer> numberStream = Stream.iterate(1, n -> n * 2)
```

KdG

Intermediate operations

- `map:` transform the data. Uses a Function
- `filter:` some elements pass through. Uses a Predicate.
- `sorted:` sorts the stream. Uses a Comparator.
- `limit:` limits to a certain number. No lambda used.
- ...

```
Arrays.asList( 1, 2, 3, 4, 5);
    .stream()
    .map(i -> i * i)
    .forEach(e -> System.out.print(e + " "));
```

```
Arrays.stream({1,2,3})
     .filter(n -> n % 2 == 0)
     .filter(n -> n < 40)
     .forEach(a -> System.out.print(a + " "));
```

KdG

Terminal operations

- `forEach:` runs the lambda on all elements. Use a Consumer.
- `findFirst:` returns Optional of first element.
- `collect:` collect into a List, Set, Map, String, ...
  - Uses Collectors.toList(), Collectors.toSet(), ... as parameter.
- `toList`: same as `collect(Collectors.toList());`
- `reduce`
  - Combines elements in single result.
- `min, max, average, sum, count`
- `...`

KdG

Streams: remarks

- Streams are *lazy*: intermediate operations are not evaluated until the terminal operation is invoked
- Better alternative for for-loops. We will try to avoid for-loops in combination with collections.
- Automatic parallelization possible: you can make a stream parallel and it will try to perform the operations in parallel!
- Stream are not IO Streams...
- If the lambda is a single method invocation, in some cases you can use a *method reference*.

KdG

Method reference examples

```
articles.stream()
    .filter(a -> a.getPrice() > 400.0)
    .forEach(a -> System.out.println(a))
```

```
articles.stream()
    .filter(a -> a.getPrice() > 400.0)
    .forEach(System.out::println)
```

```
Stream.of("Dirk", "Annelies", "Thomas")

.sorted((s1,s2)->s1.compareTo(s2))
        .forEach(System.out::println);
```

```
Stream.of("Dirk", "Annelies", "Thomas")
        .sorted(String::compareTo)
        .forEach(System.out::println);
```

KdG

demo_streams

- In this demo we will
  - Create a stream of Articles
  - Demonstrate intermediate operations like filter, map, sorted
  - Demonstrate terminal operations like forEach, sum, collect, ..

KdG

Exercise 4

- Create a class Actor. An actor has a name, gender (m,f) and a birth year.
- Create a static factory method createRandomActor(), that creates a random Actor (names are “keanu1,2,3...” or “reece1,2,3...”, birth year between 1920 and 2010)
- Use streams to (NO classic loops!)
  - Print out a list of 20 random actors, save in a List<Actor>
  - Filter out all female actors and print
  - Filter out all male actors that are older than 50 and print
  - Make a sorted list, sorted by age and print
  - Give the total age of all female actors
  - Make a string with all the first letters of the male actors

KdG

Agenda this week

Introduction
Where are we heading to…?
Java project with Gradle - Git
Lambdas and streams
Annotations
JSON - Gson

KdG

Annotations

- Labels you can add to your code
- Added since JDK5
- Can be used by the compiler, frameworks or tools
- Examples
  - `@Override`: the following method overrides a method. Compiler will now check if that is true.
  - `@Deprecated`: the following method should no longer be used. Compiler will now check if it is used and gives a warning.
  - `@SuppressWarnings` : suppress compiler warnings (for example: unused variable)
  - `@FunctionalInterface` : the following interface is a functional interface. Compiler checks if it has just one method.
- Java Code Geeks Annotation tutorial (https://www.javacodegeeks.com/2014/11/java-annotations-tutorial.html)

KdG

Annotations are used by frameworks and libraries

- In the following lessons we will meet some frameworks and libraries that make use of their own annotations
- Examples
  - Spring: @Bean, @Component, @Controller, ...
  - GSon: @SerializedName, @Expose, ...
  - JUnit: @Test, @Before, @After, ...
  - ...

KdG

Annotations parameters

- Annotations can have parameters
  - No parameter: it's called a **Marker** annotation
    - example: `@Override`
  - One parameter: **Single-value** annotation
    - Example: `@SuppresWarnings(value = "deprecation")`
    - Or shorter (if the parameter is called 'value'):
      `@SuppresWarnings("deprecation")`
  - More parameters: **Full** annotation
    - Example: `@Deprecated(since = "4.5", forRemoval = true)`

KdG

Create your own Annotations

- You can write your own annotations
- For example: you make a performance testing framework that will run all methods in a class which have the annotation @Heavy
- The annotation will have one parameter: the maxtime the method can take
- Your framework will run all the @Heavy methods and report the ones that took more time than the maxtime

KdG

Create your own Annotations

The annotation has annotations itself!

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Heavy {
   int maxTime() default 100;
}
```

It is a single-value annotation with a default value

`@Retention` determines at what point annotation should be discarded.

- SOURCE will be retained only with source code, and discarded during compile time.
- CLASS will be retained till compiling the code, and discarded during runtime.
- RUNTIME will be available to the JVM through runtime.

`@Target` determines where the annotation can be used. Examples are: FIELD, METHOD, CONSTRUCTOR...

KdG

Use your own Annotation

```
public class MyHeavyMethodsClass {
    @Heavy(maxTime = 100)
    public static void slowMethod(){
        System.out.println("slow method starting");
        try {
            Thread.sleep(new Random().nextInt(500));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void normalMethod(){
        System.out.println("This is a normal method");
    }
}
```

KdG

Our little “framework” code...


```
for (Method method: MyHeavyMethodsClass.class.getDeclaredMethods()) {
   Heavy heavy = method.getAnnotation(Heavy.class);
   if (heavy!=null) {
       System.out.println("Found heavy method:" +  method.getName());
       try {
           long time = System.currentTimeMillis();
           method.invoke(null);
           long deltaTime = System.currentTimeMillis() - time;
           if (deltaTime> heavy.maxTime()) {
               System.out.println("The method was to slow!");
               System.out.printf("It took %d ms, while maxtime was %d ms!\n",
deltaTime, heavy.maxTime());
           }
       } catch (IllegalAccessException | InvocationTargetException e) {
           e.printStackTrace();
       }
   }
}
```

This code uses the Java Reflection API.
This API is not part of this course...

demo_annotations

- In this demo we will
  - Create a Heavy annotation, it is Single-valued
  - Use this annotation on some methods
  - Use reflection to read out the annotation and run the methods

KdG

Exercise 5

- Create a new gradle project.
- Write an annotation @NotFinished. It should be used by developers to label methods that they have not yet finished.
- Use it in a demo class Demo that has at least 4 methods. Some are @NotFinished.
- Write a StartApplication class that uses reflection to make a small report of all the @NotFinished methods in the Demo class.
- Add a parameter to the Annotation: developer (a String).
- Add the developers name to the corresponding unfinished methods in your report.

KdG

Agenda this week

Introduction
Where are we heading to…?
Java project with Gradle - Git
Lambdas and streams
Annotations
JSON - Gson

KdG

JSON - Gson

- JSON is an acronym for **J**ava**S**cript **O**bject **N**otation
- It is data format widely used for data exchange
- It is a human readable data format
- It is easy to parse and generate by machines

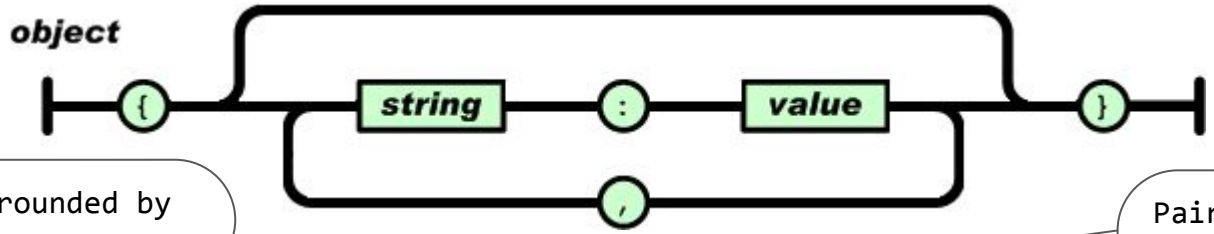
employees.json

```
{
  "employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter","lastName":"Jones"}
  ]
}
```

KdG

JSON object

- A JSON object consists out of multiple name - value pairs



Object is surrounded by {}

Pairs are separated by a comma

```
{"firstName":"John","lastName":"Doe"}
```

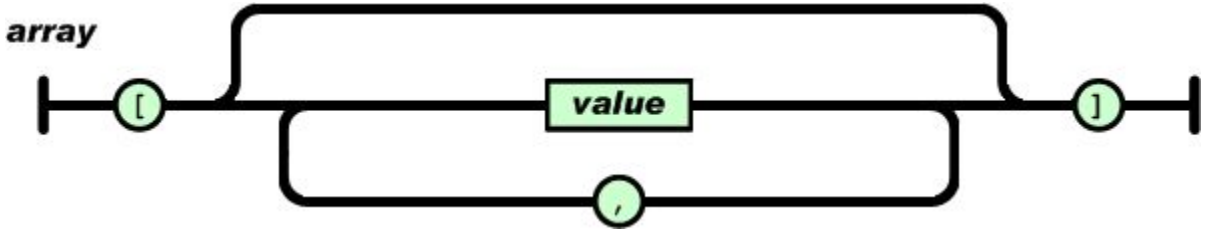
The name always between double quotes.
We use camelCase as naming convention, but snake_case is also often used

Value can be
- Number
- String (double quotes)
- Boolean: true or false
- Array: between []
- Object: between {}
- null

KdG

JSON array

- JSON objects can be grouped in an array:



The name of the array followed by a :

[] surround the elements of the array.

```
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter","lastName":"Jones"}
]
```

KdG

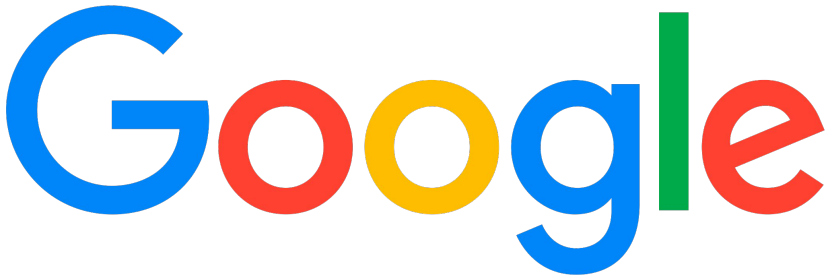
JSON and Gson

Converting a java object to a file and back is called **serialization** and **deserialization**

- Gson is one of the most popular libraries to convert Java to JSON and back
- It was created by Google
- It is freely available, you can find it on the maven repository:

  https://search.maven.org/artifact/com.google.code.gson/gson

Tutorial:
https://github.com/google/gson/blob/master/UserGuide.md



KdG

Java → JSON: the Java class

```
public class Person {
   private String name;
   private int age;

   public Person(String name, int age) {
       this.name = name;
       this.age = age;
   }

   @Override
   public String toString() {
       return "Person{" + "name='" + name + '\'' + ", age=" + age +
'}';
   }
}
```

KdG

Java → JSON

```
Person person = new Person("Jack", 27);
GsonBuilder gsonBuilder = new GsonBuilder();
Gson gson = gsonBuilder.create();
String jsonString = gson.toJson(person);
System.out.println(jsonString);
```

Result:

```
{"name":"Jack","age":27}
```

KdG

Java → JSON: add a friend...

```
public class Person {
   private String name;
   private int age;
   private Person friend;

   public Person(String name, int age) {
       this.name = name;
       this.age = age;
   }

   public void setFriend(Person friend) {
       this.friend = friend;
   }

   //...
}
```

KdG

Java → JSON

```
Person person = new Person("Jack", 27);
person.setFriend(new Person("John", 34);
GsonBuilder gsonBuilder = new GsonBuilder();
Gson gson = gsonBuilder.create();
String jsonString = gson.toJson(person);
System.out.println(jsonString);
```

Result:

```
{"name":"Jack","age":27,"friend":{"name":"John","age":34}}
```

KdG

Java → JSON: `SerializedName` annotation

```
public class Person {
    private String name;
    private int age;
    @SerializedName("only_friend")
    private Person friend;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setFriend(Person friend) {
        this.friend = friend;
    }

    //...
}
```

The annotation can be used if you want the name of the attribute to be different from the name in the json file

```
{"name":"Jack","age":27,"only_friend":{"name":"John","age":34}}
```

KdG

Java → JSON: add enemies...

```
public class Person {
   private String name;
   private int age;
   private Person friend;
   private List<Person> enemies = new ArrayList<>();

   public Person(String name, int age) {
       this.name = name;
       this.age = age;
   }

   public void addEnemy(Person enemy) {
       this.enemies.add(enemy);
   }

   //...
}
```

```
{"name":"Jack0","age":31,"friend":{"name":"Jack1","a
ge":80,"enemies":[]},"enemies":[{"name":"Jack2","age
":43,"enemies":[]},{"name":"Jack3","age":34,"enemies
":[]},{"name":"Jack4","age":9,"enemies":[]}]}
```

KdG

Java → JSON: transient keyword

```
public class Person {
   private String name;
   private int age;
   private Person friend;
   private transient List<Person> enemies = new ArrayList<>();

   public Person(String name, int age) {
       this.name = name;
       this.age = age;
   }

   public void addEnemy(Person enemy) {
       this.enemies.add(enemy);
   }

   //...
}
```

If you want a certain field not to be serialized to the JSON file, you can add the **transient** keyword.

```
{"name":"Jack0","age":31,"friend":{"name":"Jack1","age":80}}
```

KaG

Write the JSON to a file

setPrettyPrinting will add indentation to the JSON file.

```
gsonBuilder.setPrettyPrinting();
//...
//... create the JSON string here
//...
try (FileWriter jsonWriter = new FileWriter("jack.json")) {
   jsonWriter.write(jsonString);
} catch (IOException e) {
   e.printStackTrace();
}
```

This will create the json file in the working directory, which is fine for now...

KdG

JSON → Java

```
GsonBuilder gsonBuilder = new GsonBuilder();
Gson gson = gsonBuilder.create();
try (BufferedReader data = new BufferedReader(new FileReader("jack.json"))) {
   Person person = gson.fromJson(data, Person.class);
   System.out.println(person);
} catch (IOException e) {
   //...
}
```

KdG

JSON → Java: read an array

persons.json

```
[
 {
   "name": "Jack0",
   "age": 39
 },
 {
   "name": "Jack5",
   "age": 39
 },
 {
   "name": "Jack8",
   "age": 39
 }
]
```

To arrays is simple, to collection type needs extra step...

```
Person[] persons = gson.fromJson(data,  Person[].class);
Stream.of(persons).forEach(System.out::println);
```

```
Type listType = new TypeToken<List<Person>>(){}.getType();
List<Person> persons = gson.fromJson(data,  listType );
Stream.of(persons).forEach(System.out::println);
```

KdG

demo_gson

- In this demo we will
  - Create a small class (Person) that will be used for serialization
  - Convert this class to a JSON String using Gson
  - Write this string to a file
  - Use de @SerializedName annotation
  - Add a List<Person> attribute to the class and try to convert it
  - De-serialize a JSON file to Java object. What happens with arrays?

KdG

Exercise 6

- Create a new gradle project.
- Copy the Actors class from previous exercise.
- Generate a new json file “actors.json” that contains a list of 20 random actors
- We would like to have the attribute birthYear as birth_year in the JSON: make the necessary changes
- Make a class Movie. A Movie has a title, a release year and a director (String)
- Change Actor: an actor has a List of Movies he or she played in
- Change the factory method: it adds a random number of random movies to the actor. Use streams!
- Generate the “actors.json” again, check if the movies are there!
- Use this generated “actors.json” in the rest of the exercise:
  - Show small (ascii) menu: user can filter on age or gender
  - List of actors that match the chosen value for age or gender is printed
  - User can enter a name and the movies of this actor are saved to a new json file with the actors name.

```
To be able to read from the standard
input you need to add some extra lines
to your build.gradle file!
```

KdG

Read from standard input in Gradle project

- If you want to use a Scanner to read input
- Add following lines to the build.gradle.kts:

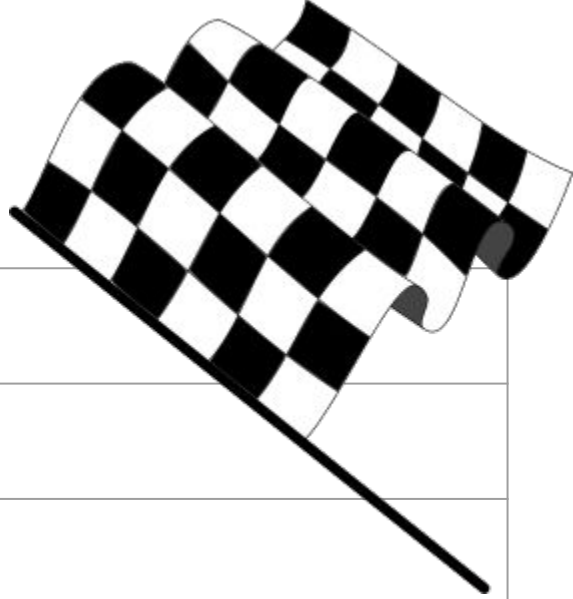
```
…

tasks.getByName("run", JavaExec::class) {
   standardInput = System.`in`
}

…
```

KdG

Finished!



Introduction
Where are we heading to…?
Java project with Gradle - Git
Lambdas and streams
Annotations
JSON - Gson

KdG