

(6) Java Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

- (6) Java Abstractions
 - Local [classes](#)
 - Anonymous [classes](#)
 - Initializer Blocks
 - A Look at Lambda Expressions and Method References
 - Callbacks
- Cited Literature:
 - Just Java, Peter van der Linden
 - Bruce Eckel, Thinking in Java

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Local Classes

- In another lecture, we have already discussed nested UDTs. Those are UDTs, which are defined within another UDT.
- Additionally, Java allows the definition of local classes. Those are classes, which are defined within the code of a method.
 - The class within the method is called local class.
 - The method, in which the local class is defined, is called enclosing method.
 - The enclosing method can access the local classes private members.
 - The type of the class, in which the local class is defined, is called outer type/class.
- Limitations of local classes:
 - We cannot have static members in local classes.
 - We cannot have local interfaces or enums.

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        class LocalClass {
            public void testMethod() {
                System.out.println("in LocalClass.testMethod()");
            }
        }

        LocalClass localClassInstance = new LocalClass();
        localClassInstance.testMethod();
        // >in LocalClass.testMethod()
    }
}
```

Local Classes as Sub Classes

- A crucial point about local **classes**: they can inherit from other **classes** and implement **interfaces** like non-local **classes**:

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        class LocalCar extends Car implements GasAcceptor {
            private double liters;
            public void testMethod(double liters) {
                System.out.println("in LocalClass.testMethod()");
            }
            @Override
            public void fillGas(double liters) {
                this.liters = liters;
                System.out.printf("Refueled with %.2f liters%n.", liters);
            }
        }

        LocalCar localClassInstance = new LocalCar();
        localClassInstance.testMethod();
        // >in LocalClass.testMethod()
        joesStation.refuel(localClassInstance, 20.4);
        // >Refueled with 20.40 liters.
    }
}
```

- Now, *LocalClass* **extends** *Car*, **implements** *GasAcceptor* and **@Overrides** its method *GasAcceptor.fillGas()*!

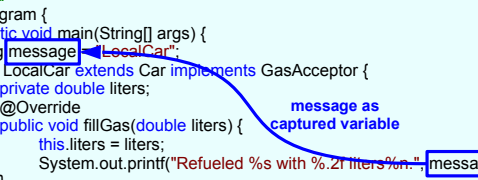
- We can also define **abstract** or **final** local **classes**.

The Code of Local Classes

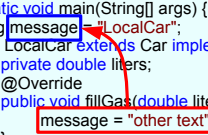
- The code of local **classes**:
 - (1) is allowed to read local variables of the enclosing method's local scope and fields, also **non-public ones**, of the enclosing **class**.
 - If a variable of the enclosing method's local scope is used in a local **class**, it is said to be a **captured variable**.
 - (2) is allowed to call methods, also **non-public ones**, of the enclosing UDT.
 - (3) is allowed to write fields of the enclosing UDT.
- Local **classes** are not allowed to write captured variables. Captured variables are effectively final.
 - Effectively **final** means, that they need not to be declared **final**, but as a matter of fact they are **final** as soon as they are captured.
 - Effectively **finals** are also **final** outside of the local **class** in the enclosing methods.

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        String message = "LocalCar";
        class LocalCar extends Car implements GasAcceptor {
            private double liters;
            @Override
            public void fillGas(double liters) {
                this.liters = liters;
                System.out.printf("Refueled %s with %.2f liters\n", message, liters);
            }
        }

        joesStation.refuel(new LocalCar(), 20.4);
        // >Refueled LocalCar with 20.40 liters.
    }
}
```



```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        String message = "LocalCar";
        class LocalCar extends Car implements GasAcceptor {
            private double liters;
            @Override
            public void fillGas(double liters) {
                message = "other text";
            }
        }
    }
}
```



6

- Before Java 8 it was required to define captured variables of the enclosing method's local scope as **final** explicitly. Since Java 8, local variables, which are only read after initialization are implicitly **final**, i.e. eventually **final**.

Introducing anonymous Classes

- This is a very important topic in Java! Esp. to create so called event handlers, which we'll introduce in a future lecture.
- Java even allows to create local anonymous classes and immediate instances thereof:

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        Car localCarInstance = new Car() {
            @Override
            public void startEngine() {
                System.out.println("start a local Car");
            }
        };
        localCarInstance.startEngine();
        // >start a local Car
    }
}
```

anonymous class

Notice

Different from the definition of a local class, the definition of an anonymous class is a statement, which must be terminated with a semicolon!

- Several things are happening in this compact code.
 - (1) On the right side of the assignment
 - (a) a new anonymous class is created, which inherits from Car
 - (b) an instance of that anonymous class is created.
 - => An anonymous class is a sub class UDT without a name.
 - (2) On the left side of the assignment
 - a reference of type Car holds the new instance.

```
// (1) ... gets assigned an instance of an anonymous class, which inherits Car:
... new Car() {
    @Override
    public void startEngine() {
        System.out.println("start a local Car");
    }
};
```

```
// (2) An instance of type Car ...:
Car localCarInstance = ...
```

Calling super Ctors of anonymous Classes

- It is syntactically not directly obvious, but calling `super` ctors along with creating anonymous `classes` is possible.
 - Assume `Car` with a ctor accepting a car's vehicle id:

```
public class Car { // (other members omitted)
    private final String vehicleID;

    public Car(String vehicleID) {
        this.vehicleID = vehicleID;
    }
    public String getVehicleID() {
        return this.vehicleID;
    }
    public void startEngine() {
    }
}
```

- Principally, the `super classes` ctor is just called during the initialization of the anonymous `classes` instance.

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        Car localCarInstance = new Car("W0L000051T2123456");
        @Override
        public void startEngine() {
            System.out.printf("starting \"%s\"%n", getVehicleID());
        }
    };

    localCarInstance.startEngine();
    // >starting 'W0L000051T2123456'
}
}
```

----- calls the ctor of the anonymous classes super class (`Car` in this case)

Local (incl. anonymous) Classes and the Scope of this and super

- The **this** reference of a local **class** refers to its current instance, e.g. we can call a **private** method on **this**:

```
public static void main(String[] args) {  
    class LocalClass {  
        private void anotherTestMethod() { System.out.println("in LocalClass.anotherTestMethod()"); }  
        public void testMethod() {  
            this.anotherTestMethod();  
        }  
    }  
  
    LocalClass localClassInstance = new LocalClass();  
    localClassInstance.testMethod();  
    // >in LocalClass.anotherTestMethod()  
}
```

- The **super** reference of a local **class** refers to the super classes part of the current instance:

```
public class Car { // (other members omitted)  
    private String vehicleID;  
  
    public Car(String vehicleID) {  
        this.vehicleID = vehicleID;  
    }  
  
    public String getVehicleID() {  
        return this.vehicleID;  
    }  
  
    public void startEngine() {  
        System.out.println("in Car.startEngine()");  
    }  
}
```

```
public static void main(String[] args) {  
    class LocalCar extends Car {  
        @Override  
        public void startEngine() {  
            super.startEngine();  
        }  
    }  
  
    LocalCar localCarInstance = new LocalCar();  
    localCarInstance.startEngine();  
    // >in Car.startEngine()  
}
```

Anonymous Classes from Interfaces

- In Java we can go yet another step: We can create an anonymous class from an interface!

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        GasAcceptor aGasAcceptor = new GasAcceptor() {
            @Override
            public void fillGas(double liters) {
                System.out.printf("filling %.2f of gas%n", liters);
            }
        };

        joesStation.refuel(aGasAcceptor, 34.8);
        // >filling 34.80l of gas
    }
}
```

«interface»
GasAcceptor
+ fillGas(liters : double)

- This code is similar to the creation of an anonymous class on the last slide, but this time we implement an interface.

- On the right side of the assignment

- (a) a new anon. class is created, which implements `GasAcceptor`
- (b) an instance of that anonymous class is created.

```
// (1) ... gets assigned an instance of a new UDT, which implements GasAcceptor:
... new GasAcceptor() {
    @Override
    public void fillGas(double liters) {
        System.out.printf("filling %.2f of gas%n", liters);
    }
}
```

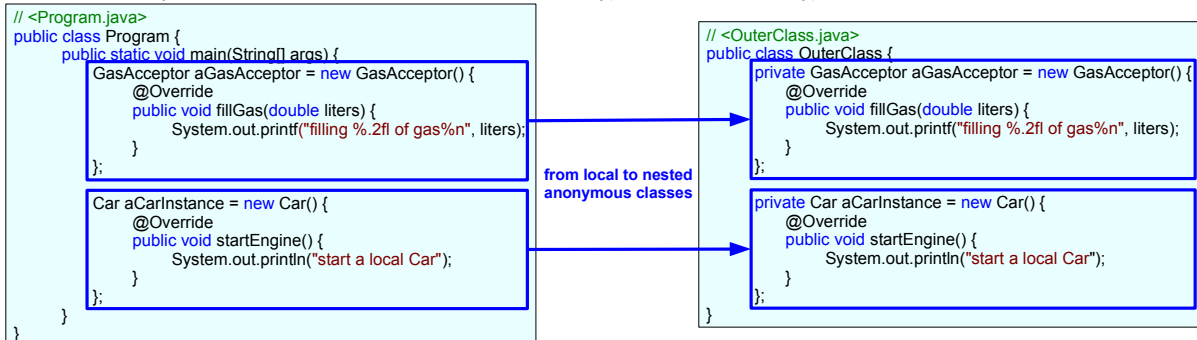
- On the left side of the assignment

- a reference of type `GasAcceptor` holds the new instance.

```
// (2) An instance of type GasAcceptor ...:
GasAcceptor aGasAcceptor = ...
```

Using anonymous Classes in Fields

- Up to now we've seen local anonymous classes, but anonymous classes can also be nested!
 - I.e. anonymous classes can also be created and used as types of fields of outer types!



- The code of nested anonymous classes:
 - (1) is allowed to read and write and capture fields of the outer class (*OuterClass* in this case), also private ones
 - (2) is allowed to call methods of the outer class (*OuterClass* in this case), also private ones.

Using anonymous Classes – Limitations

- Anonymous **classes** must either implement an **interface** or **extend** a **class** during creation.
 - It can only directly implement one interface at creation time.
 - It cannot implement an **interface** and additionally extend a **class** at creation time.
- Instead we can, e.g., create **local classes** to build "mini-hierarchies" to come over such limitations:

```
public void method() {  
    class LocalCarImpl extends Car implements GasAcceptor {  
        public LocalCarImpl(String vehicleID) {  
            super(vehicleID);  
        }  
        @Override  
        public void fillGas(double liters) {  
            System.out.printf("filling %.2f of gas%n", liters);  
        }  
    }  
    Car localCarInstance = new LocalCarImpl("W0L000051T2123456") {  
        @Override  
        public void startEngine() {  
            System.out.printf("starting '%s'%n", getVehicleID());  
        }  
    }  
  
    localCarInstance.startEngine();  
    // >starting 'W0L000051T2123456'  
}
```

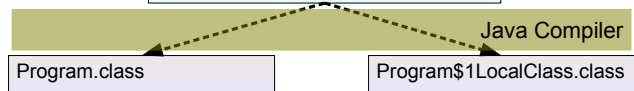
- LocalCarImpl* was just added as "relay" **class** to get a **class** extending *Car* and implementing *GasAcceptor*.
- ... then the following code in can furthermore **extend** *LocalCarImpl* and **@Override** inherited methods.

Local and anonymous Classes and the Java Compiler

- Like with **static** nested **classes**, the compiler creates class-files for local and anonymous **classes** with **structured names**.
 - Per **local (named) class**, we get a class-file with the enclosing class name and after the '\$' a uniquely numbered local **class** name:

- The unique number is needed, because we could have local classes with the same name in different methods of the same enclosing **class**.

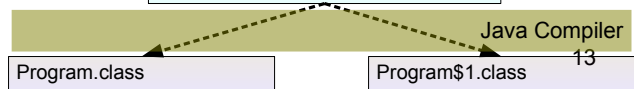
```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        class LocalClass {
        }
    }
}
```



- Per **anonymous class**, we get a class-file with the enclosing **class** name and after the '\$' a unique number as **class** name:

- This is also true for non-local anonymous **classes** (e.g. in field definitions), all anonymous **classes** of an enclosing **class** are just consecutively numbered.

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        Car localCarInstance = new Car() {
            // An anonymous class
        };
    }
}
```



Initializer Blocks

- Sometimes it is required to do initialization of an anonymous **class**, e.g. to initialize fields.
 - But how can fields be initialized, when fields are usually **private**? – Sure, with ctors! But there is a problem...
- Because an anonymous **class** has no name, we cannot define a ctor!
 - Therefor Java allows to write so called initializer blocks. They allow to create a ctor in an anonymous **class**!
 - Initializer blocks are sometimes called instance initializer blocks.

```
// <Program.java>
public class Program {
    public static void main(String[] args) {
        Car localCarInstance = new Car() {
            {
                System.out.println("in initializer block");
            }
        };
        // >in initializer block
    }
}
```

- Initializer blocks represent code, which is written in curly braces on an (anonymous) **classes member level**.
 - Initializer blocks can access members and read/write fields of (anonymous) **classes**.
 - Yes, we can also define initializer blocks in named (i.e. "non-anonymous"/"normal") **classes**!
 - Like ctors, initializer blocks are executed, when an instance of the **class** is created.
 - Internally, initializer blocks are executed, after the **super classes** ctor returns.
 - We can have several initializer blocks in a **class**, which are executed in the order they are written. -> This is very error prone!

Excursus: Static Initializer Blocks

- After we have introduced initializer block, we'll also introduce **static initializer blocks** right now.
 - If, for whatever reason, a **static** field cannot be assigned "inline", it can be done in a **static** initialization block:

```
public class Date { // (members hidden)
    public static final double S_PER_GREGORIAN_YEAR;

    static {
        SECONDS_PER_GREGORIAN_YEAR = 31556952.2;
    }
}
```

- **static** initializer blocks cannot be used in non-**static** classes (incl. anonymous **classes**)!
 - This is really just an excursus put in because of the functional and syntactic similarity to non-**static** initializer blocks.
- Typically, the code in the **static** block will be executed exactly once, when a program using *Date* is started.
 - Warning: The code in **static** initializer blocks is executed, when a **class** is initialized, i.e. there is no *main()*-method required!
 - Harmless-looking code put into a **static** initializer block can cover difficult to explain effects or bugs:

```
public class Date { // (members hidden)
    static { // Will always be executed, when the class Date is initialized.
        System.out.println("Hello World!!!");
    }
    // >Hello World!!!
}
```

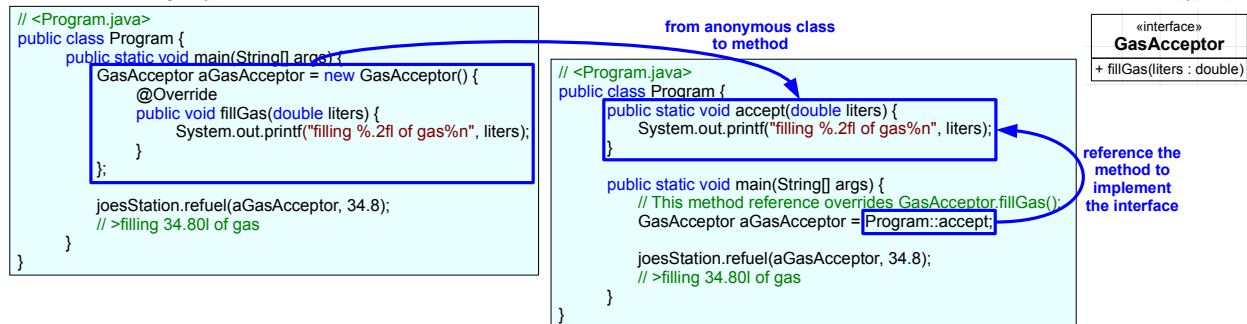
- **static** initialization blocks are used rarely.

15

- Questions about when a **static** block is executed are difficult to answer. Most sources tell, that this happens, when a **class** is initialized. Sometimes it is said, that it is executed, when a **class** is loaded by the VM (btw., this seems to be wrong).
- Like with non-**static** initializer blocks, we can have several **static** initializer blocks in a **class**, which are executed in the order they are written. This is also very error prone, because it is tempting to rely on any orders of execution!

Anonymous Classes from Interfaces – Method References

- As alternative to anonymous **classes**, we can use **method references** to implement **single-method-interfaces**.
 - The **target type** of method references must be **interfaces** with **one method**. – We call those **interfaces** **functional interface** (types).



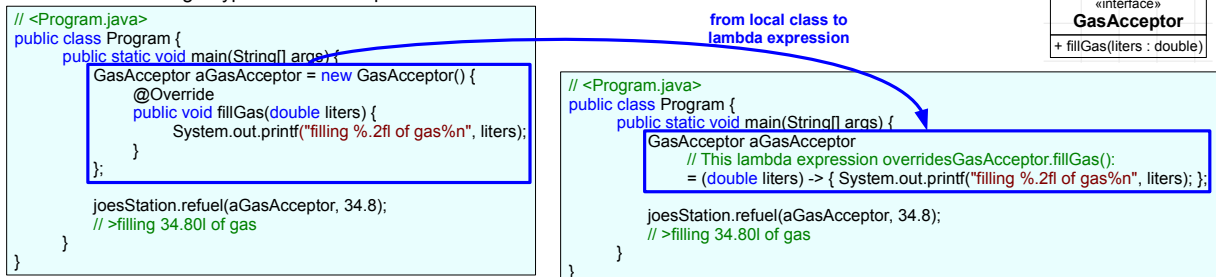
- If we have a **method**, which matches the **signature of a functional interface**, we can use a **reference to this method** as **valid implementation** of that **interface** w/o creation of an **anonymous class**!
- Attention! Underneath **method references** don't create **anonymous classes**! Instead Java's **invokedynamic** is used.
- But Java provides yet another **shortcut** to implement **functional interfaces** with so called **lambda expressions**...

16

- To be a functional **interface**, an **interface** is only allowed to have exactly one **abstract** method! – It means, that other methods **must be default** methods (or **static** methods, but they don't contribute to the "contract-nature" of an **interface**).
- Functional **interface** types are sometimes also called **Single Abstract Method (SAM)** types.
- No anonymous **class** is generated for method references, notice that no class-file like one for an inner **class** is created.

Anonymous Classes from Interfaces – Lambda Expressions

- As alternative to anonymous **classes**, we can use **lambda expressions** to implement functional **interfaces**.
 - Read: The target type of lambda expressions must also be a functional **interface**.



- A lambda expression, aka "lambda", is a compact way to create an ad hoc implementation of a functional **interface**.
 - Lambdas are very powerful**, because they allow writing **powerful, yet compacter** code.
 - As a matter of fact, the presented definition of the lambda expression can be written even more compact!
 - Since its introduction with Java 8, the **lambda-idiom changed programming in Java profoundly**.
 - We'll discuss lambdas in depth and in a **future lecture**. Then we'll understand, that lambdas are a **game-changer** in Java.
 - => **Lambdas are a deep key enabler for Java's streams**.

17

- Attention! Underneath **lambdas don't create anonymous classes**! Instead Java's **invokedynamic** is used.

- Captured symbols must also be effectively **final** in lambdas.
- No anonymous **class** is generated for lambdas: notice that no class-file like one for an inner **class** is created.

Anonymous Classes – Summary

- The definition of anonymous **classes** is tied to present UDTs .
 - Anonymous **classes** must extend a **class** or implement an **interface**!
- Implementation side of anonymous **classes**:
 - Anonymous **classes** cannot be **abstract** (can't define **abstract** methods), indeed, they are implicitly **final** .
 - Therefor, anonymous **classes** have to **@Override** or implement all methods of its ancestors to become concrete.
 - Anonymous **classes** can also contain non-**static** fields, other non-**static** methods and even further non-**static** nested **classes**.
 - Anonymous **classes** have no name, therefor we can't formulate a call to a **static** member.
 - Code in anonymous **classes** can access and modify fields of the outer **class** , but it cannot modify locals of the enclosing method .
 - Locals of the enclosing method are effectively **final** for the anonymous **class**.
 - Anonymous **classes** can defined as local **classes**, **static** nested **classes** or inner **classes**.
- Usage side of anonymous **classes**:
 - It is required to create an instance of the anonymous **class** directly with its definition!
 - As compile time type of that instance, we usually use the type of the UDT being **extended** or **implemented**.

Callbacks – Part I

- Up to here, we have only dealt with code, that we use explicitly, e.g. we call own or 3rd party/JDK's methods.
- We can also program code, that is called from other code, e.g. from the operating system or the JVM!
- Consider following code, which tells the JVM to call the method *MyTimerTask.run()* to be called every 10s:

```
// <Program.java>
import java.util.*;

public class Program {
    public static void main(String[] args) {
        class MyTimerTask extends TimerTask {
            @Override
            public void run() {
                System.out.println("Invoked");
            }
        }
        TimerTask timerTask = new MyTimerTask();
        Timer timer = new Timer(true);
        timer.scheduleAtFixedRate(timerTask, 10_000);
        Thread.sleep(60_000);
        timer.cancel();
    }
}
```

- Since the method *MyTimerTask.run()* is called back from the JVM, it is called call back method or simply "callback".

Callbacks – Part II

- The `classes` *TimerTask* and *Timer* are Java's connection to the operating system's timer system.
- We can use those `classes` to get notifications about timer-events from the operating system.
- So, an event from the operating system signals, that something has happened.
 - *Timer* allows us to react on an event, esp. it allows us to react on periodically sent events from the operating system.
 - Instead, that our code "directly" reacts on such events, we register a *TimerTask*-object, that is called, when the event was emitted.
 - The *TimerTask*-object we use as callback is not directly called, instead it is invoked at a later point in time: when the event was emitted.
- The take away: a callback is an object/method, which is called "somewhen by the system".
 - When it is called is out of our control, we just provide the callback.

Callbacks – Part III

- So called event-driven programming is another programming paradigm.
 - The idea is, that a program is not executed by imperative flow, but driven by events.
- The code, which we want to be called, when the event in question is emitted must be registered as callback.
 - The callback is executed at any point in future. I.e. the time the callback is registered is different from the time it is actually called!
 - We say, the callback is called asynchronously. – It means it is called asynchronously related to the time the callback was registered.
 - Events and callback are the basis of yet another programming paradigm: asynchronous programming.
 - Callbacks, i.e. objects, which are called when an event is emitted are usually called event handlers in Java.
- Event-driven programming means that code is invoked by own or external events, which are emitted asynchronously.
- Callbacks represent an own programming principle: the Hollywood principle.
 - The Hollywood principle says "Don't call us, we call you!".
 - In our case with the timer: "Please call me back every 10s, here is my number, i.e. the *timerTask* reference".

```
// Registers timerTask as event handler:  
timer.scheduleAtFixedRate(timerTask, 0, 10_000);
```


Callbacks – Part IV

- One of Java's crucial applications of anonymous `classes` is to define callbacks in a simple way. Consider:

```
class MyTimerTask extends TimerTask {  
    @Override  
    public void run() {  
        System.out.println("Invoked");  
    }  
}  
TimerTask timerTask = new MyTimerTask();  
Timer timer = new Timer(true);  
timer.scheduleAtFixedRate(timerTask, 0, 10_000);  
Thread.sleep(60_000);  
timer.cancel();
```

```
TimerTask timerTask = new MyTimerTask();  
Timer timer = new Timer(true);  
timer.scheduleAtFixedRate(new TimerTask() {  
    @Override  
    public void run() {  
        System.out.println("Invoked");  
    }  
}, 0, 10_000);  
Thread.sleep(60_000);  
timer.cancel();
```

- As can be seen, we just pass the anonymous class-expression as argument to `Timer.scheduleAtFixedRate()`.
 - The code is functionally equivalent.
- Timer events with `TimerTask` are simple examples how to use anonymous `classes` as callbacks.
 - Callbacks are used in many other places in Java, not only as event handlers.
 - Callbacks are also used as objects transporting a piece of code to an algorithm.
 - An important example are *Comparators*, which allow to control sorting algorithms.
 - As lambdas and method references together with *Streams*, callbacks support functional programming as higher order functions.
- Callbacks can – of course – also be objects of top level `classes`!



Thank you!