(2) Java Advanced: Lambdas

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Literature:
  - Java 8 in Action, Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft
  - http://www.angelikalanger.com/Lambdas/LambdaTutorial/lambdatutorial_4.html#_Toc353459794

Initial Words

Yes, my slides are heavy.

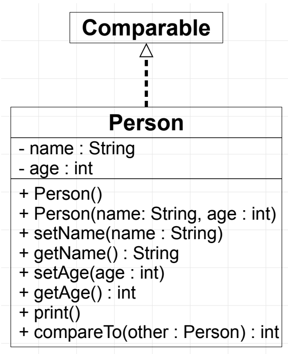
I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

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!

The Class Person for our Discussion

- For our following discussion, we'll assume the presence of this class *Person*:




```java
// <Person.java>
public class Person implements Comparable<Person> {
        private String name;
        private int age;
        public Person() {
        }
        public Person(String name, int age) {
              this.name = name;
              this.age = age;
        }
        public String getName() {
              return name;
        }
        public void setName(String name) {
              this.name = name;
        }
        public int getAge() {
              return age;
        }
        public void setAge(int age) {
              this.age = age;
        }
        public void print() {
              System.out.println("name = " + getName() + ", age = " + getAge());
        }
        @Override
        public int compareTo(Person other) {
              return this.getName().compareTo(other.getName());
        }
}
```

Passing Code to Methods: Instances of Classes

- With *Arrays.sort()* we can sort a *Person[]* in a very simple way:

```
Person[] persons = {new Person("Bonnie", 24), new Person("James", 78), new Person("Clyde", 24), new Person("Archie", 46)};
Arrays.sort(persons);
// persons = {Person(name = "Archie", age = 46), Person(name = "Bonnie", age = 24), Person(name = "Clyde", age = 24), Person(name = "James", age = 78)}
```

- *Arrays.sort()* sorts the passed array based on the relative order of the contained elements.
  - Java allows to express the relative order for a UDT, by having the UDT implement *Comparable*.
  - To cut the story short: it means, that a UDT does itself define, how it is to be sorted!

- In case we want to sort *persons* for the *persons*' ages, we have to define a dedicated *Comparator*:

```
public class PersonAgeComparator implements Comparator<Person> {
        @Override
        public int compare(Person lhs, Person rhs) {
              return Integer.compare(lhs.getAge(), rhs.getAge());
        }
}
```

```
Arrays.sort(persons, new PersonAgeComparator());
// persons = {Person(name = "Bonnie", age = 24), Person(name = "Clyde", age = 24), Person(name = "Archie", age = 46), Person(name = "James", age = 78)}
```

  - The idea of *Comparator*s is that comparing of objects is no longer the objects' business, but a specific *Comparator* handles this.
  - The pattern we see here is an object, namely a *Comparator* instance represents the comparison algorithm.
  - => We have delegated comparison to the *Comparator* instance.

Passing Code to Methods: anonymous Classes and Lambdas

- In case we don't want to reuse the special age-*Comparator*, we can instead use an anonymous class.
  - Following code creates an "ad hoc" age-*Comparator* and passes it to *Arrays.sort()*:

```
Array.sort(persons, new Comparator<Person>() {
                        @Override
                        public int compare(Person lhs, Person rhs) {
                            return Integer.compare(lhs.getAge(), rhs.getAge());
                        }
                   });
// persons = {Person(name = "Bonnie", age = 24), Person(name = "Clyde", age = 24), Person(name = "Archie", age = 46), Person(name = "James", age = 78)}
```

- Before Java 8, this was the shortest way to pass "ad hoc code" to another method.

- Without further ado, I will show Java 8's way to pass code to another method with a so called lambda expression:

```
Array.sort(persons, (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge()));
// persons = {Person(name = "Bonnie", age = 24), Person(name = "Clyde", age = 24), Person(name = "Archie", age = 46), Person(name = "James", age = 78)}
```

```
Comparator<Person> personAgeComparator
    = new Comparator<>() {
        @Override
        public int compare(Person lhs, Person rhs) {
            return Integer.compare(lhs.getAge(), rhs.getAge());
        }
    };
```

```
// Voilá, the lambda:
Comparator<Person> personAgeComparator
    = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

Objects to transport Code

- Like any kind of object, anonymous classes and the more compact lambdas allow transportation of code to other code.
  - Mind, that also a *String* is just an object, on which we can execute code to calculate something.
  - E.g. assume a *String* we pass to a method, that asks the *String*, if it contains a certain sub string:

```
String text = "Montana is The Treasure State";

public static boolean hasOs(String in) {
	// Just calls code defined in the class String:
	return in.contains("o") || in.contains("O");
}
boolean containsOs = hasOs(text);
// containsOs = true
```

- But there are important differences to *Comparator*:
  - (1) The methods in *String* relate to the data encapsulated in the *String* object.
  - (2) The *Comparator* object basically encapsulates an algorithm, which needs objects as arguments.

- We can say, that *Comparator* is mainly an object, that encapsulates or holds code.
  - It should be said, that a *Comparator*, that has a state can be useful, e.g. a *Comparator* counting *compare()*-calls:
  - We could create a *Comparator* featuring the decorator pattern as reusable variant!

```
public class PersonAgeComparatorWithCounting implements Comparator<Person> {
	private int countOfComparisons;
	public int getCountOfComparisons() {
		return this.countOfComparisons;
	}
	@Override
	public int compare(Person lhs, Person rhs) {
		++countOfComparisons;
		return Integer.compare(lhs.getAge(), rhs.getAge());
	}
}
```

Objects to transport Code – with Lambdas

- In many cases transporting code can be accomplished just by providing a single method!
  - On the other hand in Java there is also a pattern, which usually requires the implementation of a set of methods: event listeners.
  - However, the topic of this lecture focusses on functional programming, which does not (yet) concern event listeners in Java.

- Java 8 makes transporting code simpler than implementing an anonymous class with lambdas.

- If an implementation of an interface requires only one method to be a concrete class, this can be expressed as lambda.
  - Let's discuss the applicability of lambdas along some examples to unleash the idea behind this mighty tool.

Functional Interfaces – Part 1

- Any interface, which only demands one abstract method to be implemented can be used as target type for a lambda.
  - Actually, there are many JDK interfaces, which fulfill this requirement: *Iterable*, *Closeable*, *Comparable*, *Comparator* and more.

```
Comparator<Person> personAgeComparator = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

```
int result = personAgeComparator.compare(new Person("Bonnie", 24), new Person("James", 78));
// result < 0
```

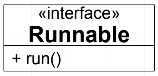
  - An interface can also be used as type of a lambda expression, if it has one abstract method, but also default and static methods.
  - As mentioned above, types, which can hold lambdas are called target types (for lambdas).

- Only interfaces with one abstract method can be uses as target types, no abstract classes.

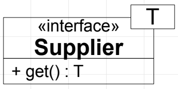
- Java's libraries provide interfaces, that were only designed to act as target type in the package *java.util.function*:
  - *Function<T, R>*, *Predicate<T>*, *Supplier<T>*, *Consumer<T>*, *UnaryOperator<T>*, *BiFunction<T, U, R>*, *IntSupplier<T>* and more.
    - One could say we have the core interfaces *Function<T, R>*, *Predicate<T>*, *Supplier<T>* and *Consumer<T>*.
  - The names of those interfaces indicate the kind of lambda they can hold: i.e. the returned values and accepted arguments.
  - interfaces, which are specially designed as target types for lambdas are called functional interfaces in Java.

- Functional interfaces can also be used apart from lambdas, e.g. for explicit implementation or anonymous classes.
  - And functional interfaces can also be used for so called method references, which will be discussed soon.

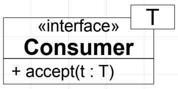
Functional Interfaces – Part 2



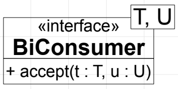
- A *Runnable* accepts no arguments and returns no value.
  - Its code (i.e. the code in the implementation of *Runnable.run()*) can only perform side effects.
  - A legacy interface (pre Java 8).



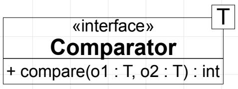
- A *Supplier* only returns a *T*.



- A *Consumer* just accepts one argument of type *T*.
  - Its code can only perform side effects.

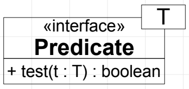


- A *BiConsumer* accepts two arguments, one of type *T*, one of type *U*.
  - Its code can only perform side effects.

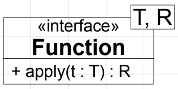


- A *Comparator* accepts two arguments of type *T* and returns an int.
  - It should only be used to implement specific equivalence-comparison algorithms for specific *T*s.
  - A legacy interface (pre Java 8).

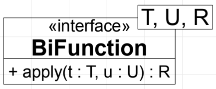
Functional Interfaces – Part 3



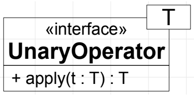
- A *Predicate* accepts an argument of type *T* and returns a boolean.



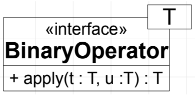
- A *Function* accepts an argument of type *T* and returns an *R*.
  - This functional interface can hold code, that maps a *T* to an *R* like a mathematical function.



- A *BiFunction* accepts an argument of type *T* and an argument of type *U* and returns an *R*.
  - This functional interface can hold code, that maps a *T* and a *U* to an *R* like a math. binary function.



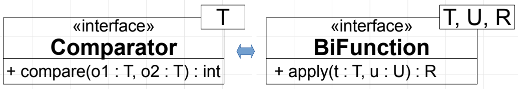
- A *UnaryOperator* accepts an argument of type *T* and returns a *T*.
  - This functional interface can hold code, that maps a *T* to another *T* like a mathematical function.



- A *BinaryOperator* accepts two arguments of type *T* and returns a *T*.
  - This functional interface can hold code, that maps two *T*s to another *T* like a math. binary function.

Functional Interfaces – Part 4

- Instead of *Comparator<Person>*, we can use a functional interface from *java.util.function* as target type for our lambda.
  - Namely, we can use *BiFunction<Person, Person, Integer>*:




```
BiFunction<Person, Person, Integer> twoPersonsToInt
        = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());

int result = twoPersonsToInt.apply(new Person("Bonnie", 24), new Person("James", 78));
// result < 0
```

  - *BiFunction<T, U, R>* provides only one method: *apply()*.
  - The signatures of *"Integer BiFunction.apply(Person t, Person u)"* and *"int Comparator.compare(Person lhs, Person rhs)"* match.

- Let's fix some terminology: the assigned lambda is invoked by calling the only (abstract) method in the target type.
  - Depending on the target type, these methods have varying (but schematic) names (such as *apply()*), signatures and return values.
  - In this course, we call the target type's method, that invokes the lambda the invocation method.
  - The invocation method of *BiFunction* is *BiFunction.apply()*:

```
BiFunction<Person, Person, Integer> twoPersonsToInt = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

target type

lambda expression

invocation

```
int result = twoPersonsToInt.apply(new Person("Bonnie", 24), new Person("James", 78));
```

call the invocation method

- As a further shortening, we call the signature of the invocation method also the signature of the target type.
  - So, in short: the signature of *BiFunction.apply()* is the signature of *BiFunction*.

Target Types – Part 1

- Java requires a lambda being bound to an explicit target type. So lambdas cannot be initialized with the implicit type:

```
var personAgeComparator = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge()); // Invalid! Lambda expression needs an explicit target type!
```

  - Even assignment to an *Object* is not possible:

```
Object personAgeComparator = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge()); // Invalid! Object is not a functional interface!
```

  - These facts could be interpreted like "Lambdas have no type at all in Java!", which is fairly correct! – They have no explicit type.

- Back to the idea, that the same lambda expression can be held by different target types, which are functional interfaces:

```
Comparator<Person> personAgeComparator =
BiFunction<Person, Person, Integer> twoPersonsToInt =
(lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

  - *Comparator<Person>* and *BiFunction<Person, Person, Integer>* have the same target type signature. This feature is called generalized target type inference.

- Generalized target type inference is generally fine, but there is a problem: different target types are incompatible:

```
personAgeComparator = twoPersonsToInt; // Invalid! Incompatible types: ToIntBiFunction<Person, Person> cannot be converted to Comparator<Person>
```

  - We could solve this compiler error with a cast, but then we'll have a run time error:

```
personAgeComparator = (Comparator<Person>) twoPersonsToInt; // Invalid! Will throw a ClassCastException!
```

- Usually, functional interface types appear as parameter type.
  - This means, the have not to care for the target type, but just pass the lambda literally, the compiler cares for it (inference)!
  - Nevertheless, we have to discuss these (seldom) situations, when we have to care for target types explicitly right now ...

Target Types – Part 2

- Assume following overloads of *MyClass.callIt()*, that have potentially different matching target types for the same lambda:

```
// <MyClass.java>
public class MyClass {
        public static int callIt(Comparator<Person> personComparator, Person lhs, Person rhs) {
            return personComparator.compare(lhs, rhs);
        }

        public static int callIt(BiFunction<Person, Person, Integer> twoPersonsToInt, Person lhs, Person rhs) {
            return twoPersonsToInt.apply(lhs, rhs);
        }
}
```

- We cannot pass the lambda we have discussed up to here like this:

```
MyClass.callIt((lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge()), new Person("Bonnie", 24), new Person("James", 78));
// Invalid! reference to callIt is ambiguous both method callIt(Comparator<Person>, Person, Person) in MyClass and method callIt(BiFunction<Person,
// Person, Integer>, Person, Person) in MyClass match
```

  - The reason is not obvious, but understandable after our discussion of target types: which overload should the compiler select?

- We must help the compiler to select a specific overload by using a cast to the desired target type:

```
MyClass.callIt((Comparator<Person>)(lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge()), new Person("Bonnie", 24), new Person("James", 78)); // OK!
```

  - This is an interface cast, i.e. an upcast, in this case it can also be called discretization cast.

How Lambdas are treated by the Java Compiler

- Different from anonymous classes, the Java compiler does not create class-files for each lambda!
  - Everything is done at run time.
- Instead, the Java compiler places the byte code mnemonic invokedynamic (since Java 7), when a lambda is specified.
  - invokedynamic enables the JVM to create the so called synthesized lambda type, it is created on the first time of usage.
  - The synthesized type is an implementation of the target type, just like with anonymous classes.
  - The JVM has enough information to perform aggressive optimization at run time.
- Java's type system was not changed in Java 8 to support lambdas.
  - There was the idea to introduce first class function-types, but people resorted to the concept of functional interfaces.
  - Since Java 1, there are already many functional interfaces, i.e. such interfaces offering only one method, in the JDK.
  - Functional interfaces were formerly called Single Abstracts Methods (SAMs).
  - The new view of lambdas is: a lambda is a simple way to implement a functional interface.
- As to James Gosling, lambdas were planned for Java since version, but it wasn't put into effect for time pressure reasons.
  - Anonymous classes were added in Java 1.1 as a compromise (but mainly to support the new event handler system).
  - And for Java 8, lambdas were designed in a way to replace anonymous classes in present code easily.

15

- Alternative design proposals for lambdas in Java:
  - CICE (Concise Instance Creation Expression): Basically a syntactically simpler way to create anonymous classes, it set the basis for ARM (Automatic Resource Management, i.e. try-with-resource-blocks) in Java.
  - FCM (First Class Methods): Made methods a first class concept and lambdas are reflected as local anonymous methods.
  - BGGA (Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé): The idea was to introduce a function type incl. lexical binding of this, return, continue and break.

The Lambda Concept

- The concept of lambdas is fairly old, it has its origins in maths as the basis of the lambda calculus.
- In other programming languages there exist other names for the concept: blocks, closures, arrow-functions
- The basic idea is to have an anonymous function, i.e. a function without a name or a "function literal".
- Lambdas were introduced into Java:
  - To eliminate verbosity of anonymous classes.
  - To allow clutter-free transportation of code as method arguments or return values.
    - This concept is also known as "code as data".
    - If a parameter is a functional interface, it is often called behavioral parameter.
    - Functions/methods can be treated as objects!
    - A function/method accepting or returning other functions/methods is often called higher order function.
  - To support functional programming, esp. for Java's *Stream* API.

Excursus: The Lambda Calculus

- The idea to use a generalized notation for functions emerged in the 1930ies.
- The exist many different ways to express "a calculation, which is bound to variables".
  - The set builder notation in maths: $\{x \mid f(x)\}$
  - Functions in maths: $f(x)=x+2$
  - Methods in Java:

```java
public static double square(double x) {
	return x * x;
}
```

- Alonzo Church developed a formal way to express "functions and bound parameters" with the lambda calculus:

$\{x \mid f(x)\}$ as lambda $\lambda x.f\,x$

$f(x)=x+2$ as lambda $\lambda x.x\;+\;2$

Good to know:
The term "lambda" stems from the greek letter λ (lambda), which was chosen to stand for "unnamed function".

  - All bound variables are written between the lambda-symbol and the dot, other symbols right from . are unbound constants.
  - This common notation allows simple and formal ways to express, combine and analyze mathematical expressions.

- Java just uses a "Java-syntactically-fine" form of the lambda calculus' notation:

```java
public static double square(double x) {
	return x * x;
}
```

as lambda $\boxed{\texttt{x -> x * x}}$ → $\lambda x.x{\cdot}x$

  - With lambdas in Java, Java hits the ground of mathematics and makes functional programming more approachable.

- Lambdas can be used to analyze the "computability" of calculations. Computable functions can be expressed and even evaluated using the lambda calculus. It is, as far as the expressibility goes, equivalent to Turing machines, but it is more abstract from an actual implementation.
- But Java as a programming language took over following important feature of lambdas: the idea to write transformation rules with a syntax, in which functions have no name.

Lambda's Syntax Aspects – Part 1

- Let's reconsider the lambda, with which we've started our discussion, it shows the (almost) shortest lambda syntax:

```
Comparator<Person> personAgeComparator = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

  - The symbol-combination – and >, namely "minus" and "greater than" separates the lambda's parameters from its body.

- With the fully blown lambda syntax, we get this variant (the optional parts are pale in color):

```
Comparator<Person> personAgeComparator = (Person lhs, Person rhs) -> {
        return Integer.compare(lhs.getAge(), rhs.getAge());
};
```

  - So, the shorter form allows to leave the parameter types, the braces, the return keyword and the semicolon away.
  - There are situations, in which have to write some of those optional parts. – We'll discuss those cases now.

- If we specify no parameter types, they will be inferred by the compiler.
  - However, we have either to specify all parameter types of the parameter list, or none of them. A mix is not allowed:

```
Comparator<Person> personAgeComparator                   // Invalid lambda parameter declaration! Cannot mix implicitly
        = (Person lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());   // typed and explicitly typed parameters.
```

- If a lambda has only one parameter, we can leave the parentheses surrounding this parameter (*person*) away:

«interface» **Predicate** (T)
+ test(t : T) : boolean

```
Predicate<Person> isFullAgedInGermany = (person) -> 18 <= person.getAge();
boolean result = isFullAgedInGermany.test(new Person("Bonnie", 24));
// result = true
```

  - But if the lambda has no parameters, we still have to write a pair of empty parentheses:

«interface» **DoubleSupplier**
+ getAsDouble() : double

```
DoubleSupplier rnd = () -> Math.random();
double result = rnd.getAsDouble();
// result = 0.6111252041948116
```


- Before Java 11: When the lambda parameters are declared with types, they can be declared final and can be annotated. We will discuss this aspect in a minute ...

Lambda's Syntax Aspects – Part 2

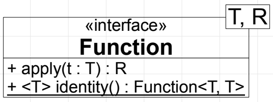
- A special case of function is the identity function, it just returns its single parameter. The lambda-syntax for this is easy:

```
Function<String, String> stringIdentity = s -> s;
String result = stringIdentity.apply("Echo");
// result = "Echo"
```

  - Identity is also simple in mathematical notations as named function or lambdas:

$$identity(x) = x \quad \Longleftrightarrow \quad \lambda x.x$$

  - Identity is so important in fp, that Java provides the simple factory *Function.identity()* to create an identity function for a certain type:




```
Function<String, String> stringIdentity = Function.identity();
```

    - The service of this simple factory is, that it will infer the correct type of *T* and *R* for us.

- There is no direct way to express recursive calls of a lambda.

```
Function<Integer, Integer> factorial = n -> (n <= 1) ? 1 : n * factorial.apply(n - 1);
// Invalid! variable factorial might not have been initialized
```

  - The separation of declaration and assignment solves the compile time error above, but does also not work:

```
Function<Integer, Integer> factorial = null;
factorial = n -> (n <= 1) ? 1 : n * factorial.apply(n - 1);
// Invalid! local variables referenced from a lambda expression (factorial) must be final or effectively final
```

  - That Java doesn't offer direct support signals that recursive lambdas are not very relevant for Java.
    - (The lambda calculus can be used with so called Y-combinators to define recursive lambdas.)

Lambda's Syntax Aspects – Part 3

- A lambda is not required to return a value, instead it can only contain statements performing side effects:

«interface» T
Consumer
+ accept(t : T)

```
Consumer<Person> personToConsoleWriter = person -> System.out.println("name: " + person.getName() + ", age: " + person.getAge());
personToConsoleWriter.accept(new Person("Clyde", 24));
// > name: Clyde, age: 24
```

  - The functional interface *Consumer<T>* can be used to hold a lambda accepting one argument, but not returning a value.
  - If we need more than one statement to express the lambda's algorithm, its code must be put into braces like a method body's block:

```
Consumer<Person> adultPersonToConsoleWriter = person -> {
        if (18 <= person.getAge()) {
                String message = "name: " + person.getName() + ", age: " + person.getAge();
                System.out.println(message);
        }
};
adultPersonToConsoleWriter.accept(new Person("Quaden", 8)); // Will print nothing to the console!
```

  - Such lambdas are called statement lambdas.
  - Statement lambdas can also be used to program empty lambdas:

«interface»
Runnable
+ run()

```
Runnable emptyLambda = () -> {};
emptyLambda.run(); // Will do nothing.
```

- A statement lambda can also return a value, but then an explicit return statement must be written:

«interface» T, R
Function
+ apply(t : T) : R

  - The functional interface *Function<T, R>* can be used to hold a lambda accepting one argument and returning a value.

```
Function<Person, String> createTextRepresentation = person -> {
        if (40 >= person.getAge()) {
                return "name: " + person.getName() + ", age: " + person.getAge();
        }
        return "name: " + person.getName() + ", age: is a secret";
};
String theText = createTextRepresentation.apply(new Person("James", 78));
// theText = "name: James, age: is a secret"
```

Lambda's Syntax Aspects – Part 4

- Since Java 11 we can also use var to type all parameters implicitly:

```
Comparator<Person> personAgeComparator = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

```
Comparator<Person> personAgeComparator = (var lhs, var rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

  - Both syntax variants leads to the compiler inferring the correct parameter types.
  - Mixes with explicit types and var are not allowed in a lambda's parameter list.

- On a first look, defining implicitly typed parameters with var seems really fluffy: we can leave it away and have the same!
  - This is not quite correct, implicitly typed lambda parameters with var enable some interesting features:
    - (1) We can define final implicitly typed lambda parameters:

```
Comparator<Person> personAgeComparator = (var lhs, final var rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

    - (2) We can add annotations to implicitly typed lambda parameters:

```
Comparator<Person> personAgeComparator = (@Test var lhs, var rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

- Not so obvious: a lambda's return type and the throws declaration are always inferred and cannot be explicitly specified.

The Scope of Lambdas – Part 1

- Now, we have to discuss another relevant difference between lambdas and anonymous classes: scope.

- Each class in Java defines its own scope.
  - The scope can be the scope of names shared between instances (class scope) or the scope of each instance (instance scope).

- An anonymous class only has an instance scope of its single instance:
  - The scope of the field *counter* in the anonymous class is different from *OuterClass.counter*, therefor we have no name clash.
  - The scope of *OuterClass.myMethod()*'s local *result* is different from the scope of *compare()*'s local *result*, therefor we have no name clash.
  - The scope of this and super in the anonymous class is the scope of the anonymous class, therefor we have no name clash.
  - *OuterClass.this.counter* refers the field counter of the current *OuterClass* instance (*OuterClass.super* the super class part of *OuterClass*).

```java
// <OuterClass.java>
public class OuterClass {
	private int counter;

	public void myMethod() {
		int result = 0;
		Comparator<Person> personAgeComparator = new Comparator<>() {
			private int counter;

			@Override
			public int compare(Person lhs, Person rhs) {
				++this.counter;
				++OuterClass.this.counter;
				int result = Integer.compare(lhs.getAge(), rhs.getAge());
				return result;
			}
		};
	}
}
```

The Scope of Lambdas – Part 2

- A lambda expression has a completely different scope, the so called lexical scope.
  - Lexical scope means, that a lambda has no own scope, instead it just uses the enclosing scope lexically.

- Let's dissect a lambda's lexical scope:
  - The scope of this and super in the lambda is the scope of the enclosing class.
    - *this.counter* refers the field *OuterClass.counter* of the current *OuterClass* instance.

```java
// <OuterClass.java>
public class OuterClass {
	private int counter;
	public void myMethod() {
		int result;

		Comparator<Person> personAgeComparator = (lhs, rhs) -> {
			++this.counter;
			return Integer.compare(lhs.getAge(), rhs.getAge());
		};
	}
}
```

- But if we define a variable *result* in the lambda, while we still have a local *result* in *myInvalidMethod()*, we'll end in a name clash.
  - *result* is "lexically bound in the lambda's scope".

```java
// <OuterClass.java>
public class OuterClass {
	public void myInvalidMethod() {
		int result;

		Comparator<Person> personAgeComparator = (lhs, rhs) -> {
			// Invalid! result is already defined in method myInvalidMethod()!
			int result = Integer.compare(lhs.getAge(), rhs.getAge());
			return result;
		};
	}
}
```

Capturing Variables in Lambdas – Part 1

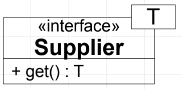
- Lambdas support transportation of code. Code can be passed to methods and also returned from methods.

- Returning lambdas from a method implies, that the return type of this method must be a functional interface, consider:

```
public static Supplier<String> provideFunctionalObject() {
        String s = "Hello World!";

        return () ->  s;
}
```

  - Let's call *provideFunctionalObject()* to get the object holding the lambda. Then call the invocation method *Supplier.get()*:




```
Supplier<String> functionalObject = provideFunctionalObject();
String result = functionalObject.get();
// result = "Hello World!"
```

  - *Supplier<T>* is a functional interface, which returns a *T*, but does not accept any arguments.

- Mind, that the local *s* is captured by the lambda and also transported to the caller of *functionalObject*.
  - And effectively *s* is returned, when *functionalObject.get()* is called.
  - Lambdas are to be defined in one context, but executed in other contexts: capturing is required to get around their lexical scope.

Capturing Variables in Lambdas – Part 2

- As we know it from anonymous classes, captured variables in lambdas are effectively final and cannot be set:

```
public static Supplier<String> provideFunctionalObject() {
        String s = "Hello World";

        return () -> {
              s = "other text"; // Invalid! Local variables referenced from a lambda expression must be final or effectively final.
              return s;
        };
}
```

- From within a lambda we cannot set a captured variable, but we can modify the state of the object it refers to, consider:

```
public static Supplier<String> provideFunctionalObject() {
        StringBuffer sb = new StringBuffer();

        return () -> {
              sb.append("the text ");
              return sb.toString();
        };
}
```

  - When we call *functionalObject.get()* for multiple times, we see, that the captured *StringBuffer* is actually modified:

```
Supplier<String> functionalObject = provideFunctionalObject();
String result = functionalObject.get();
// result = "the text "
String result2 = functionalObject.get();
// result2 = "the text the text "
```

- Lambdas can capture variables, whose state can be modified on each call of the target types invocation methods
  - It means: a lambda can have a modifiable state. – But so called stateful lambdas can be harmful with parallelized algorithms.

Lambdas and checked Exceptions – Part 1

- We stated, that throws declarations of lambdas get inferred, but this only works for unchecked *Exceptions*!

- Actually, lambdas are working really badly with checked *Exceptions*:
  - When we create a lambda with code, that may throw an unchecked *Exception*, we have to handle it. Following code won't compile:

```
Function<Path, String[]> fileToClauses = path -> Files.readString(path).split(","); // Invalid! Unreported IOException must be caught or declared to be thrown.
String[] pieces = fileToClauses.apply(Path.of("/Users/nico/Homers_Troy.txt"));
```

  - The reason for this not to compile is simple: *Files.readString()* could throw an *IOException* and the code doesn't handle it!

- At first view, it is quite strange, that following variant does also not compile:

```
try {
	Function<Path, String[]> fileToClauses = path -> Files.readString(path).split(","); // Invalid! Unreported IOException must be caught or declared to be thrown.
	String[] pieces = fileToClauses.apply(Path.of("/Users/nico/Homers_Troy.txt"));
} catch (IOException e) { // Invalid! IOException is never thrown in body of corresponding try statement!
	System.out.println(e);
}
```

  - Reason 1: We still do not handle the *IOException*, which could be thrown from *Files.readString()*.
  - Reason 2: The try-block wants to handle an *IOException*, but it doesn't enclose code, which throws this *Exception*.

- The problem we are seeing here: lambdas have no own scope, but they have their own stack trace at run time!

Lambdas and checked Exceptions – Part 2

- When we only concentrate on the lambda of the last snippet, we can see, why we cannot handle *Exceptions*:

```
Function<Path, String[]> fileToClauses = null;
try {
        fileToClauses = path -> Files.readString(path).split(","); // Invalid! Unreported IOException must be caught or declared to be thrown.
} catch (IOException e) {
        System.out.println(e);
}
```

  - The important point is that *File.readString()* is not called in this code!
  - Instead we only define a lambda, which will call *Files.readString()*, when the lambda is called (*fileToClauses.apply()*):

```
String[] pieces = fileToClauses.apply(Path.of("/Users/nico/Homers_Troy.txt"));
```

  - This means, that the *IOException* thrown from *fileToClauses* would never reach to the point where *fileToClauses* is defined.
  - Instead, *fileToClauses* could be executed in a part of the code far away from the code it was defined!
  - => This effect is called deferred execution. The code is executed or can be executed timely deferred from its definition.

- Alas, neither Java's runtime nor the compiler "forward" the potential *IOException* to the place where the lambda is called:

```
try {
        String[] pieces = fileToClauses.apply(Path.of("/Users/nico/Homers_Troy.txt"));
} catch (IOException e) { // Invalid! IOException is never thrown in body of corresponding try statement!
        System.out.println(e);
}
```

- So, how can we get out of this situation?

Lambdas and checked Exceptions – Part 3

- Without further ado, here is the correct implementation of *fileToClauses*:

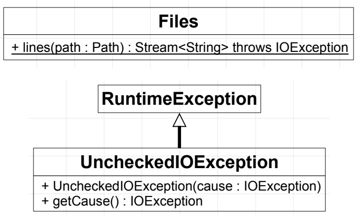
```
Function<Path, String[]> fileToClauses = path -> {
	try {
		return Files.readString(path).split(",");
	} catch (IOException ioex) {
		System.out.println(ioex);
		// We handle the IOException in the lambda and return null in this case:
		return null;
	}
};
```

```
String[] pieces = fileToClauses.apply(Path.of("/Users/nico/Homers_Troy.txt"));
// ... and consider, that pieces could be null:
String reJoined = null != pieces ? String.join(",", pieces) : "";
```

- The solution is simple: a lambda must handle checked *Exception*s on its own!
  - The consequence: if a lambda contains code, which may throw unchecked *Exception*s, it must be a statement lambda!
- This is really bad!
  - Currently, there is no simpler solution to deal with checked *Exception*s in lambdas.
  - The obtrusiveness of checked *Exception*s are matter of many discussions in the Java community.
- The good news: many new APIs in Java 8 where designed using unchecked *Exception*s, to be usable with lambdas.

Example: UncheckedIOException

- Some I/O APIs like *File.lines()* provide *Streams*, which wrap *IOExceptions* of subsequent operations into *UncheckedIOExceptions*.
  - Functionally, *File.lines()* transforms the specified text-file into a *Stream* of lines/*Strings*, that are read after the file was opened!
    - I.e. *File.lines()* opens the file and might still throw *IOExceptions*, e.g. if the specified text-file does not exist.
    - But when the *Stream<String>* returned by *File.lines()* is actually read other *IOExceptions* may appear, which are wrapped into *UncheckedIOExceptions*:




```
public static void printLinesToConsole(Stream<String> lines) {
        // Follow-up operations on Files.lines()'s result might throw UncheckedIOExceptions, which need no handling:
        lines.forEach(line -> System.out.println(line));
}

// Files.lines() might throw IOException on opening the specified file:
try (Stream<String> lines = Files.lines(Paths.get("/Users/nico/Homers_Troy.txt"))) {
        printLinesToConsole(lines);
} catch (IOException ioex) {
        System.out.println("Couldn't open file!");
}
```

- Let's use *UncheckedIOException* to make *fileToClauses* sneakier by wrapping *IOExceptions* into unchecked *Exceptions*:

```
Function<Path, String[]> fileToClauses = path -> {
        try {
                return Files.readString(path).split(",");
        } catch (IOException ioex) {
                System.out.println(ioex);
                return null; // OK, but could get ignored!
        }
};
```

```
Function<Path, String[]> fileToClauses = path -> {
        try {
                return Files.readString(path).split(",");
        } catch (IOException ioex) {
                System.out.println(ioex);
                throw new UncheckedIOException(ioex); // Excellent!
        }
};
```

```
// The call will throw UncheckedIOException in case of IOException:
String[] pieces = fileToClauses.apply(Path.of("/Users/nico/Homers_Troy.txt"));
String reJoined = String.join(",", pieces);
```


- Alternatively, we can also use the 3rd party library jOOλ, which adds some "missing functionality" to Java's *Streams*. In this case we can use the wrappers of the *Unchecked* class. *Unchecked.function()* wraps an exception-handler transforming checked *Exceptions* into unchecked *Exceptions* (e.g. *UncheckedIOException* if an *IOException* is thrown) around the passed *Function* exactly as we have done here explicitly:

```
Function<Path, String[]> fileToClauses2
        = Unchecked.function(path -> Files.readString(path).split(","));
```

```
String[] pieces = fileToClauses2.apply(Path.of("/Users/nico/Homers_Troy.txt"));
```

A first practical Lambda Application

- After so much theoretical information about lambdas we'll discuss some mighty applications.

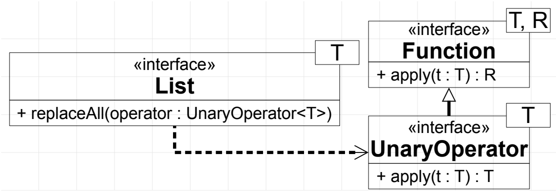
- Consider following code snippet, which just manipulates each element in a *List<String>*:

```
List<String> names = new ArrayList(List.of("Ashley", "Lisa", "Samuel", "Pat", "Marion"));
for (int i = 0; i < names.size(); ++i) {
        names.set(i, names.get(i).toUpperCase());
}
// names = { "ASHLEY", "LISA", "SAMUEL", "PAT", "MARION" }
```

  - Fair enough, this code in not spectacular. It is just an iteration over a *List<String>* and performing some actions.
  - A specialty is that *String*s cannot be modified, instead we replace them in the *List<String>*.

- In Java 8, the interface *List<T>* was extended by the method *List.replaceAll()*, which does the same, but with less code:

```
names.replaceAll(name -> name.toUpperCase());
// names = { "ASHLEY", "LISA", "SAMUEL", "PAT", "MARION" }
```



First practical Lambda Application – External and internal Iteration – Part 1

- External iteration means the usage of classical loop control structures like for, while and do-while:

```
List<String> names = new ArrayList(List.of("Ashley", "Lisa", "Samuel", "Pat", "Marion"));
for (int i = 0; i < names.size(); ++i) {
        names.set(i, names.get(i).toUpperCase());
}
// names = { "ASHLEY", "LISA", "SAMUEL", "PAT", "MARION" }
```

  - This is not only "classical" in a traditional sense, it is classical, because it "just works".

- Internal iteration is another way to get the same: via methods accepting functional interface references, passing lambdas:

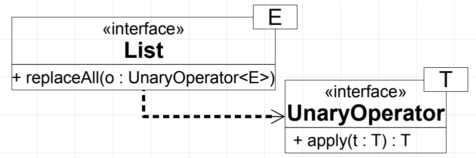
```
names.replaceAll(name -> name.toUpperCase());
// names = { "ASHLEY", "LISA", "SAMUEL", "PAT", "MARION" }
```

  - Internal iteration exploits Java's functional interfaces as a JDK feature and lambdas as a language feature.
  - It saves code, because the iteration logic is completely encapsulated in *List<T>.replaceAll()*.
  - From an engineering standpoint this is useful: it separates iteration "fluff" from the actually relevant operation logic in the lambda.

- With internal iteration programmers are only in charge to deliver code to be applied to *List*'s elements.
  - This enables internal optimizations of how the iteration is actually performed:
    - Internal iteration could be deferred to a later point in time.
    - Internal iteration could be done in a parallelized manner, e.g. split to several processing units (cores or CPUs).
    - Those powerful features, and many more, are available via Java's *Stream* API, we'll discuss in a future lecture.

First practical Lambda Application – External and internal Iteration – Part 2

- Let's go one step back, to understand the full picture of how passing lambdas to methods work.



(1) pass the lambda

```
// somewhere in the JDK
public interface List<E> { // simplified
        default void replaceAll(UnaryOperator<E> operator) {
                Objects.requireNonNull(operator);
                ListIterator<E> li = this.listIterator();
                while (li.hasNext()) {
                        li.set(operator.apply(li.next()));
                }
        }
}
```

(2) invoke the lambda's code

```
names.replaceAll(name -> name.toUpperCase())
```

- The lambda *name -> name.toUpperCase()* is inferred to a signature, that accepts a *String*, and returns a *String*.
  - Methods accepting a *String*, and returning a *String* can be assigned to references of the functional interface *UnaryOperator<String>*
  - The type parameter *T* (or *E* from *List<E>*) in *UnaryOperator<T>* is of course also implicitly inferred to *String* from the lambda.
  - We already know this feature called generalized target type inference.
  - Btw. this is also the time to honor Java's type inference features based on generics, which are doing a lot of work for us!

- When *List.replaceAll()* calls the invocation method *operator.apply()*, the code in the lambda is executed.
  - So, actually, the lambda passed to *List.replaceAll()* is called back during the internal iteration.

First practical Lambda Application – External and internal Iteration – Part 3 – Debugging

- Code with lambdas can be <u>challenging for IDEs</u>, esp. when debugging lambdas. Consider this code:

```
/*(1)*/ List<String> names = new ArrayList(List.of("Ashley", "Lisa", "Samuel", "Pat", "Marion"));
/*(2)*/ names.replaceAll(name -> name.toUpperCase());
```

- If we put a breakpoint on line (2), <u>when will the code's execution halt at debug time</u>?
  - Will it halt when *names.replaceAll()* is called, i.e. will it halt on the <u>statement</u>?
  - Will it halt <u>each time *name.toUpperCase()* is called</u>, i.e. will it halt on the <u>lambda</u>?

- E.g. in IntelliJ IDEA, we have the luxury to decide, which kind of breakpoint we want to set:

  - If we select it to be a "line breakpoint", the code's execution will halt at the statement *names.replaceAll()* at debug time.
  - If we instead select the λ-symbol, the code's execution will halt each time the lambda's code is called at debug time:

    - In this case, the "λ breakpoint" will be hit for five times.

First practical Lambda Application – External and internal Iteration – Part 4

- Many interfaces of the Collection API were extended to support internal iteration with lambdas w/o breaking compatibility:
  - *Iterable.forEach()*, *Collection.removeIf()*, *List* (*replaceAll()*, *sort()*), *Map* (*computeXXX()*, *replaceAll()*, *merge()*, *forEach()* (useful, because *Map* is not *Iterable*).

- But, how can this work? If interfaces get extended after their first publishing implementing classes get invalid!
  - (Those classes would not implement the added methods of the extended interfaces, automatically getting abstract classes!)

- The solution is simple: the Collection API's interfaces were extended with default methods:
  - The idea of default methods follows this thought: "Do the added methods make sense for every implementor? – If not, we could provide reasonable default implementations."
  - => default methods and some static methods were added to add new functionality to existing interfaces w/o breaking compatibility:

```java
// somewhere in the JDK
public interface List<E> { // Aha!
	default void replaceAll(UnaryOperator<E> operator) {
		Objects.requireNonNull(operator);
		ListIterator<E> li = this.listIterator();
		while (li.hasNext()) {
			li.set( operator.apply(li.next()) );
		}
	}
}
```

```java
// somewhere in the JDK
public interface Iterable<E> { // Aha!
	default void forEach(Consumer<? super T> action) {
		Objects.requireNonNull(action);
		for (T t : this) {
			action.accept(t);
		}
	}
}
```

Method References

- Using lambdas as an argument being passed to methods as behavioral parameter is a big topic in Java.
- Before we discuss more examples to introduce important Java APIs, we have to introduce method references.
- Reconsider following code:

```
names.replaceAll(name -> name.toUpperCase());
```

  - One might think, this is already dense enough, but in Java we can write this even less obtrusive like this:

```
names.replaceAll(String::toUpperCase);
```

- The expression *String::toUpperCase* is a so called method reference, it safes us from explicit mentioning parameters.
- Instead of passing method references to methods accepting functional interface references, we can keep it in a variable:

```
UnaryOperator<String> operationOnString = String::toUpperCase;
Consumer<String> operationOnString = String::toUpperCase;
```

  - Notice: the method reference can be inferred to different target types as with lambdas!

Good to know:
A programming style using method references instead of explicit method-calls is called tacit programming or also "point-free" programming. This is because parameters, which are mapped to arguments need no mention.

- This simplification of lambdas is also present in the formal lambda calculus: the η-reduction (greek letter eta):

```
names.replaceAll(name -> name.toUpperCase());
```

$\lambda x . f\, x$

η-reduction

```
names.replaceAll(String::toUpperCase);
```

f

35

- The concept of method references is more or less directly derived from C/C++, where we have function pointers.
- η-reduction is also called η-conversion.

A Method Reference can represent different Overloads

- What happens, if a method reference could represent many overloads of a method? Consider this class:

```
// <MyClass.java>
public class MyClass {
        public static String somethingWithString(String s) {
            return "in somethingWithString(String)";
        }

        public static String somethingWithString(String s, int i) {
            return "in somethingWithString(String, int)";
        }

        public static String somethingWithString(String s, int i, double d) {
            return "in somethingWithString(String, int, double)";
        }
}
```

- This is no conflict! – The compiler infers the correct overload, because it regards the signature of the target type:

```
// UnaryOperator<String> leads to resolving somethingWithString(String):
UnaryOperator<String> operationStringResult = MyClass::somethingWithString;
String result = operationStringResult.apply("test");
// result = "in somethingWithString(String)"

// BiFunction<String, Integer, String> leads to resolving somethingWithString(String, int):
BiFunction<String, Integer, String> operationStringResult2 = MyClass::somethingWithString;
String result2 = operationStringResult2.apply("test", 42);
// result = "in somethingWithString(String, int)"
```

Method References to Instance Methods – Part 1

- Back to the method reference *String::toUpperCase*:
  - It is "just a method" of a certain type.
  - The call `names.replaceAll(String::toUpperCase)` basically says "apply *String::toUpperCase* to each passed *String* instance".

- Consider *stringLengthSupplier*. It refers to the method *String.length()*:

```
Function<String, Integer> stringLengthSupplier = String::length;
```

  - When we invoke *stringLengthSupplier* we have to pass a *String* object as argument:

```
int result = stringLengthSupplier.apply("Trish");
// result = 5
```

  - Why that? *stringLengthSupplier* refers to the method *String.length()*, but not to a certain *String* instance's *length()*-method!

- We can also get a method reference from a specific instance, consider *nameLengthSupplier*:

```
String name = "Trish";
Supplier<Integer> nameLengthSupplier = name::length;
```

  - When we invoke *nameLengthSupplier* we have nothing to pass:

```
int result = nameLengthSupplier.get();
// result = 5
```

  - Here the situation is different: the *String* instance, on which *length()* is called is bound to *nameLengthSupplier* together with *String.length()*.

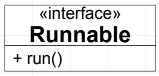
- In practice all situations are working fine. However the need to deal with different target types is strange.

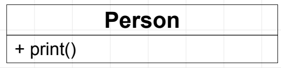
Method References to Instance Methods – Part 2

- A method reference to an instance method can be set to a reference of a target type with <u>the same signature</u>:

```
Person bonnie = new Person("Bonnie", 24);
Runnable consolePrinter = bonnie::print;
```

```
consolePrinter.run();
// > name = Bonnie, age = 24
```



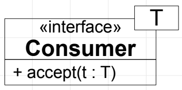


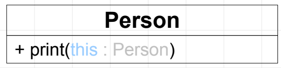
  - The signature of the invoke method *Runnable.run()* is equivalent to the signature of *Person.print()*.

- Alternatively, a method reference to an instance method can be set to a reference of a target type with a signature <u>holding the implicit this as extra parameter</u>:

```
Consumer<Person> consolePrinter = Person::print;
```

```
consolePrinter.accept(bonnie);
// > name = Bonnie, age = 24
```





  - This time, we draw the <u>method reference from the type *Person*</u>, <u>not from an instance of type *Person*</u>.
  - The signature of the invoke method *Consumer.accept()* matches to *Person.print()* <u>plus an extra parameter to hold the implicit this</u>.
  - Now, we have to pass the <u>formerly implicit this</u> as <u>explicit argument as *consolePrinter.accept(bonnie)*</u>.

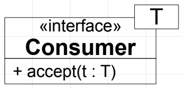
- An instance method can not only be assigned to different target types, but <u>even to target types of different signature</u>!
  - At first view, there is very unexpected/strange conversion is going on, but this required for some features to work!

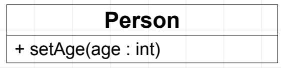
Method References to Instance Methods – Part 3

- If we have an instance method with one parameter, Java offers two target type signatures:

```
Consumer<Integer> ageSetter = bonnie::setAge;
```

```
ageSetter.accept(30);
// bonnie.age = 30
```



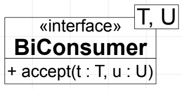


  - => The signature of the target type in *Consumer.accept()* is equivalent to the signature of *Person.setAge()*.

- Alternatively, we can use a target type accepting two parameters: the original parameter and the implicit this as parameter

```
BiConsumer<Person, Integer> ageSetter = Person::setAge;
```

```
ageSetter.accept(bonnie, 25);
//  bonnie.age = 25
```



  - Once again, we draw the method reference from the type *Person*, not from an instance of type *Person*.
  - => The signature of the target type in *BiConsumer.accept()* is equivalent to the signature of *Person.setAge()* plus the implicit this.

Creating own Target Types to hold Instance Methods

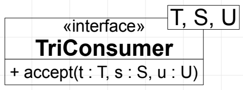
- If an instance method accepts two or more parameters, we have to create our own target type, but this is very simple.
  - We just create a new functional interface accepting, e.g., three arguments to hold the implicit this and two other arguments:

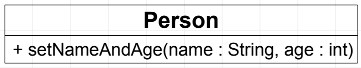
```java
// <TriConsumer.java>
@FunctionalInterface
public interface TriConsumer<T, S, U> {
	void accept(T t, S s, U u);
}
```

  - Our new *TriConsumer<Person, String, Integer>* is a target type for *Person.setNameAndAge()*:

```java
TriConsumer<Person, String, Integer> nameAndAgeSetter
	= Person::setNameAndAge;
```

```java
nameAndAgeSetter.accept(bonnie, "Bonnie-May", 26);
// > name = Bonnie-May, age = 26
```





- As can be seen, we used the annotation *@FunctionalInterface* on our target type.
  - This annotation can be put on interfaces. When set, the compiler ensures, that the annotated interface has only one abstract method.

- Definition: a functional interface is an interface, that can be fully implemented by lambdas and method references.

Method References to static Methods – Part 1

- References to static methods can only be assigned to target types with the exactly matching signature:

```
Runnable action = Utilities::printTime;
action.run();
// > 17:54:32
```

«interface»
Runnable
+ run()

```
// <Utilities.java>
public class Utilities {
	public static void printTime() {
		System.out.printf("%tT%n", new Date());
	}
}
```

  - The signature of the invoke method *Runnable.run()* is equivalent to the signature of *Utilities.printTime()*:
  - This makes perfect sense: <u>static methods have no implicit this</u>, therefor we have exactly matching signatures.

- For the static method *Utilities.dumpLogToConsole()*, the situation is different: <u>it throws the checked *IOException*</u>.

```
Runnable consolePrinter = Utilities::dumpLogToConsole;
// Invalid! Incompatible thrown types IOException in functional expression.
```

```
// <Utilities.java>
public class Utilities {
	public static void dumpLogToConsole() throws IOException {
		String logText = Files.readString(Path.of("/Users/nico/applications.log"));
		System.out.println(logText);
	}
}
```

  - The problem: *Runnable.run()* has no throws declaration, thus it s no match for *Utilities.dumpLogToConsole()*!

- <u>Takeaway: the invocation method of the target type must also have a compatible throws declaration to match!</u>

Method References to static Methods – Part 2

- Because Java offers static imports, the syntax using method references can be somewhat streamlined.

```
// <Utilities.java>
public class Utilities {
        public static void printTime() {
            System.out.printf("%tT%n", new Date());
        }
}
```

- We can add a static import for all of *Utilities* static methods and leave the *Utilities::*-prefix away:

```
Runnable action = Utilities::printTime;
action.run();
// > 17:54:32
```

```
static import Utilities.*;

Runnable action = printTime;
action.run();
// > 17:54:32
```

Method References to exceptional Methods

- In this case, we have to program a lambda instead, which handles the *IOException* appropriately:

```
Runnable consolePrinter = () -> {
        try {
             Utilities.dumpLogToConsole();
        } catch (IOException e) {
            System.out.println("Cannot read log: "+e);
        }
};
```

  - We have already talked about this solution, when we discussed lambdas and checked *Exception*s.

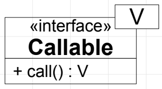
- Alternatively, we can create our own target type, whose invocation method has a matching throws declaration:

```
ExceptionalRunnable consolePrinter = Utilities::dumpLogToConsole; // OK!
try {
        consolePrinter.run();
} catch (Exception e) {
        System.out.println("Cannot read log: "+e);
}
```

```
// <ExceptionalRunnable.java>
@FunctionalInterface
public interface ExceptionalRunnable {
      void run() throws Exception;
}
```

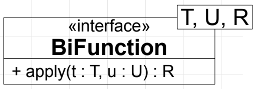
  - But this solution only works, if we have the freedom to create a new target type, instead of relying on a *Exception* unaware one.

- Currently, the JDK does not provide target types throwing checked *Exception*s for all cases.
  - The only target type is the functional interface *java.util.concurrent.Callable*<*V*>, which accepts no arguments, but returns *V*.



Primitive Functional Interface Specializations avoid Boxing

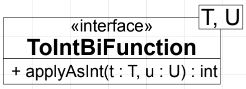
- The first lambda we programmed had *Comparator<T>* as target type, then we switched over to *BiFunction<T, U, R>*:




```
BiFunction<Person, Person, Integer> twoPersonsToInt = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

- Assume, we have to compare millions of *Person*s with *twoPersonsToInt*, it requires millions of *BiFunction.apply()* calls.
  - We must have a closer look at the target types, which is the filled in generic type *BiFunction<Person, Person, Integer>*.
  - So, the return type of the target type is *Integer*! – We cannot use int, because primitive types cannot be used as type arguments.
  - The problem: on each call, the int returned by *Integer.compare()* in the lambda will be boxed to an *Integer* object!

- This is a serious problem in Java, esp. because it can lead to a silent performance penalty.
  - To remedy the problem, the JDK provides functional interfaces with primitive "specializations".
  - Instead of *BiFunction<Person, Person, Integer>* we can use *ToIntBiFunction<Person, Person>*:

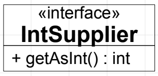



```
ToIntBiFunction<Person, Person> twoPersonsToInt = (lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge());
```

```
int result = twoPersonsToInt.applyAsInt(new Person("Bonnie", 24), new Person("James", 78));
// result < 0
```

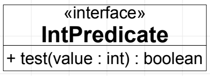
  - The return type of *ToIntBiFunction*'s invocation method is no longer generic, but fixed to int, which avoids boxing operations!
  - The package *java.util.functions* provides a set of functional interface specializations featuring Java's primitive types.

Functional Interface Specializations

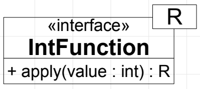


- An *IntSupplier* only returns an int.
  - Also available: *DoubleSupplier*, *LongSupplier*, *BooleanSupplier*

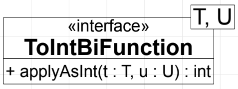
- An *IntConsumer* just accepts one argument of type int and returns no value.
  - Its code can only perform side effects.
  - Also available: *DoubleConsumer, LongConsumer*, *ObjDoubleConsumer*, *ObjIntConsumer* ...



- An *IntPredicate* accepts an argument of type int and returns a boolean.
  - Also available: *DoublePredicate*, *LongPredicate*



- An *IntFunction* accepts an argument of type int and returns an *R*.
  - This functional interface can hold code, that maps an int to an *R* like a mathematical function.
  - Also available: *DoubleFunction, LongFunction* ...

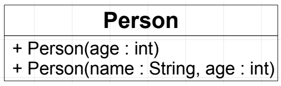


- A *ToIntBiFunction* accepts an argument of type *T* and an argument of type *U* and returns an int.
  - This functional interface can hold code, that maps a *T* and a *U* to an int like a math. binary function.
  - Also available: *ToDoubleFunction, ToDoubleBiFunction, ToLongFunction*, *ToLongBiFunction* ...

Constructor References

- We can also handle references to constructors, logically, we call them constructor references.
  - The constructor reference is selected by using the pseudo-method named new:

```
Supplier<Person> personCreator = Person::new;
Person newPerson = personCreator.get();
// newPerson = {name = null, age = 0}
```



  - Here, we used *Supplier<Person>* as target type, this leads the compiler to select *Person* dctor for the reference.
  - If we want to have the compiler selecting another ctor overload, we have to use another target type:

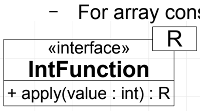
```
BiFunction<String, Integer, Person> personCreator2 = Person::new;
Person newPerson2 = personCreator2.apply("Clara", 19);
// newPerson2 = {name = "Clara", age = 19}
```

  - A target type for a constructor reference must return the created type and its signature must match the ctors parameter list.
    - *BiFunction<String, Integer, Person>* matches the ctor *Person(String, int)*.

- We can also get a constructor reference to an array "constructor", the syntax follows that of constructor references:

```
Function<Integer, Person[]> personCreator = Person[]::new;
Person[] persons = personArrayCreator.apply(3);
// person = {null, null, null}
```

  - The target type for array constructor references must return the created array type and accept an int for the arrays' length.
  - For array constructor references we can use the primitive specialized target type *IntFunction<R>* to avoid boxing of length values:




```
IntFunction<Person[]> personArrayCreator2 = Person[]::new;
Person[] persons2 = personArrayCreator2.apply(3);
// persons2 = {null, null, null}
```

Method References to Methods of the own Instance

- We can also define a method reference to the own instance, therefor we use the this keyword:

```
// <Person.java>
public class Person { // simplified
	private String name;

	public String getName() {
		return name;
	}
	public void referToGetName() {
		Supplier<String> nameGetter = this::getName;
	}
}
```

- The method reference is syntax is only available for methods! – There exist no "field references" in Java.

The "execute-around-Method" Pattern

- Lambdas add a mighty syntactic means to Java, which are reflected in many usage patterns.
- Now we will discuss the execute-around-method pattern, which is a kind of base pattern for many other of those patterns.
  - The patterns have their commonality, in that they separate parts, which are relatively stable from parts which are relatively flexible.
- The method *measureAndPrint()* applies the execute-around-method pattern:

```
public static void measureAndPrint(String actionTitle, Runnable action) {
        Instant then = Instant.now();
        action.run();
        System.out.printf("Action %s took %dms%n", actionTitle, Duration.between(then, Instant.now()).toMillis());
}
```

  - It accepts an *actionTitle*, giving the measured piece of code a name (for the "protocol").
  - And it accepts a behavioral parameter of type *Runnable*.
  - *measureAndPrint()* executes the passed code in action and does the measurement/logging-activity around the activity to measure.
- We can pass any action to measure to *measureAndPrint()*:

```
measureAndPrint("loop_100_000", () -> {
            long sum = 0L;
            for (long i = 0; i < 100_000; ++i) {
                sum += i;
            }
        });
```

Terminal

```
Action loop_100_000 took 1ms
```

```
measureAndPrint("loop_100_000_000", () -> {
            long sum = 0L;
            for (long i = 0; i < 100_000_000; ++i) {
                sum += i;
            }
        });
```

Terminal

```
Action loop_100_000_000 took 49ms
```

Thank you!