Introduction to Programming

L06 Object Orientation III

Stephan Krusche

6 December 2023
Technical University of Munich

Schedule

#	Date	Subject
1	18.10.23	Introduction
1b	25.10.23	Central exercise
	01.11.23	**No lecture**
2	08.11.23	Control Structures
3	15.11.23	Data Types
4	22.11.23	Object Orientation I
5	29.11.23	Object Orientation II
6	**06.12.23**	**Object Orientation III**
7	13.12.23	Algorithms
	20.12.23	**No lecture**
8	10.01.24	Programming Languages
9	17.01.24	Graphical User Interfaces
10	24.01.24	Recursion
11	31.01.24	Beyond Programming
12	07.02.24	Course Review

Roadmap

- **Context**
  - Apply object oriented programming
  - Use control structures (**`if`**, **`switch`**, **`for`**, **`while`**) and basic data types (**`List`**, **`Stack`**, **`Queue`**)
  - Apply abstraction, encapsulation, inheritance and polymorphism
  - Use generics, collections and apply error handling
- **Learning goals**
  - Program for each loops using iterators and collections
  - Implement switch expressions with enum types
  - Explain the differences between anonymous inner classes and lambda expressions
  - Implement lambda expressions
  - Apply streams to write more concise code

Outline

- **Iterators and collections**
- Enum types and switch expressions
- Lambda expressions
- Streams

Motivation

- There are many ways to implement ordered collections of objects (or base values)
  - Strings are collections of characters
  - Lists and arrays are ordered collections of objects of a defined type
- When accessing the individual elements, it is important to
  - Know if there is another element
  - Jump to the next element
  - Remove an element after processing
- This is defined by the interface **`Iterable<T>`** with corresponding constructors as well as methods **`hasNext()`**, **`next()`** and **`remove()`**

The interface **`Iterable<T>`**

- The interface **`Iterable<T>`** defines the method **`public Iterator<T> iterator()`**
- The implementing class must provide the method **`iterator()`** that returns an **`Iterator<T>`**
- Example: a class **`IterableString`**, where it is possible to iterate over the individual **`Character`** of the **`String`**

```
import java.util.Iterator;

public class IterableString implements Iterable<Character> {
    private final String str;
    public IterableString(String str) {
        this.str = str;
    }

    public Iterator<Character> iterator() {
        return new IterableStringIterator(str);
    }
}
```

`Iterable<T>` and `Iterator<T>`

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

public class IterableStringIterator implements Iterator<Character> {
    private final String str;
    private int position = 0;
    public IterableStringIterator(String str) {
        this.str = str;
    }

    public boolean hasNext() {
        return position < str.length();
    }

    public Character next() {
        if (position == str.length()) {
            throw new NoSuchElementException();
        }
        return str.charAt(position++);
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

String is immutable

`var` uses type inference: the compiler will infer the type of the variable, you do not need to declare it

Usage

```java
var s = new IterableString("Hello World");
Iterator<Character> it = s.iterator();
while (it.hasNext()) {
    System.out.print(it.next());
}
System.out.println();
```

Shorter and easier to understand alternative

```java
var s = new IterableString("Hello World");
for (Character c : s) {
    System.out.print(c);
}
System.out.println();
```

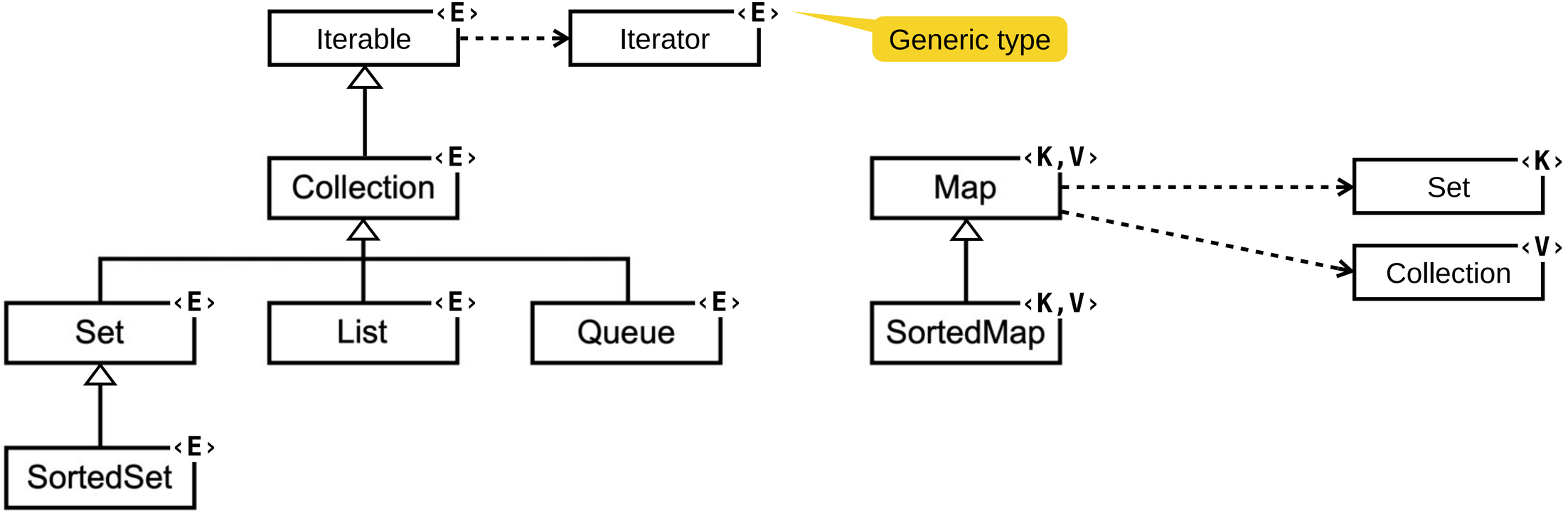
Enhanced for loop (**for each** loop)

Output (for both)

```
Hello World
```

Java collections framework - interfaces

- Main interfaces of the collection framework



- The main interfaces encapsulate different types of collections
- Enable implementation-independent manipulation of collections

The class **`Collections`**: algorithms (code overview)

```
import java.util.Comparator;

public class Collections {

    // true, if no element is contained in both c1 and c2
    public static boolean disjoint(Collection<?> c1, Collection<?> c2) { ... }

    // number of elements in c that are, according to equals, identical to o
    public static int frequency(Collection<?> c, Object o) { ... }

    // Reverses the order of the elements in list
    public static void reverse(List<?> list) { ... }

    // Replaces every element in list identical to oldV with newV
    public static <T> boolean replaceAll(List<T> list, T oldV, T newV) { ... }

    // Sorts list according to the method compareTo of type T
    public static <T extends Comparable<? super T>> void sort(List<T> list) { ... }

    // Sorts list according to the comparator c
    public static <T> void sort(List<T> list, Comparator<? super T> c) { ... }
}
```

The class **`Collections`**: algorithms

- Example usage

```
import java.util.*;

public class Playground {

    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(18, 46, 18, 12);
        Set<Integer> set = new HashSet<>(list);

        System.out.println(list);       // [18, 46, 18, 12]
        Collections.reverse(list);
        System.out.println(list);       // [12, 18, 46, 18]
        Collections.sort(list);
        System.out.println(list);       // [12, 18, 18, 46]

        System.out.println(set);        // [12, 18, 46]
        System.out.println(Collections.disjoint(list, set));   // false
        System.out.println(Collections.frequency(list, 18));   // 2
        System.out.println(Collections.frequency(set, 18));    // 1

        Collections.replaceAll(list, 18, 22);
        System.out.println(list);       // [12, 22, 22, 46]
    }
}
```

The interface **`Comparator<T>`**

- The interface **`Comparator<T>`** defines the method **`int compare(T o1, T o2)`** and thus a total order on objects of type **`T`**
- Return value analogous to **`compareTo`** of the interface **`Comparable`**
  - **`< 0`**, if **`o1`** is smaller than **`o2`**
  - **`== 0`**, if **`o1`** is equal to **`o2`**
  - **`> 0`**, if **`o1`** is greater than **`o2`**
- Example: descending sorting of integers

```
import java.util.*;

public class Playground {
    public static void main(String[] args) {
        var list = Arrays.asList(18, 46, 18, 12);
        Collections.sort(list, new Comparator<Integer>() {
            @Override
            public int compare(Integer i1, Integer i2) {
                return i2 - i1;
            }
        });
        System.out.println(list);       // [46, 18, 18, 12]
    }
}
```

Creates an **anonymous inner class** that implements **`Comparator<Integer>`** and instantiates this class once

Anonymous inner classes

- Make your code more concise
- Declare and instantiate a class at the same time
- Like local classes except that they do not have a name
- Use them if you need to use a local class only once
- Sneak preview: shorter and even more concise: **lambda** expressions

```
var list = Arrays.asList(18, 46, 18, 12);
Collections.sort(list, (i1, i2) -> i2 - i1);
System.out.println(list);          // [46, 18, 18, 12]
```

Lambda expression

Easier to understand alternative

```
var list = Arrays.asList(18, 46, 18, 12);
list.sort(Comparator.reverseOrder());
System.out.println(list);          // [46, 18, 18, 12]
```

Predefined comparator to sort descending

Example

- Each interface can be implemented directly in an **anonymous inner class**

```
interface Runnable {
    void run();
}

public class Playground {

    public static void main(String[] args) {

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("Run was invoked");
            }
        };

        runnable.run();
    }
}
```

Output

```
Run was invoked
```

Example

- **Anonymous inner classes** can also implement multiple methods

```
interface Executable {
    void execute();
    void executeAsync();
}

public class Playground {
    public static void main(String[] args) {

        Executable myExecutable = new Executable() {
            @Override
            public void execute() {
                System.out.println("execute");
            }
            @Override
            public void executeAsync() {
                System.out.println("execute asynchronously");
            }
        };

        myExecutable.execute();
        myExecutable.executeAsync();
    }
}
```

Output

```
execute
execute asynchronously
```

L06E02 String Sorting

Not started yet.

Start exercise

Easy

Due date tonight

10 min

3 pts

- Problem statement
  - Create a list of the following strings: “BMW”, “Audi”, “Mercedes”, “Seat”, “Volkswagen”, “Hyundai”, “Tesla”
    - Feel free to add your favorite car brands (or any other kind of strings)
  - Sort the list after their length using the **`Collections.sort(...)`** and an anonymous inner class
    1. The longest string should come first
    2. If two strings have the same length, sort them alphabetically
  - **Hints**
    - Use **`s2.length() – s1.length()`** in the compare method
    - Use the method **`String.compareTo(...)`** for alphabetical comparison

Example solution

```
var list = Arrays.asList("BMW", "Audi", "Mercedes",
        "Seat", "Volkswagen", "Hyundai", "Tesla");

Collections.sort(list, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        } else {
            return s2.length() - s1.length();
        }
    }
});
System.out.println(list);
```

Alternative

```
list.sort((s1, s2) -> {
    if (s1.length() == s2.length()) {
        return s1.compareTo(s2);
    }
    else {
        return s2.length() - s1.length();
    }
});
System.out.println(list);
```

Output

```
[Volkswagen, Mercedes, Hyundai, Tesla, Audi, Seat, BMW]
```

Example solution (with comments)

```java
var list = Arrays.asList("BMW", "Audi", "Mercedes",
        "Seat", "Volkswagen", "Hyundai", "Tesla");

Collections.sort(list, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        } else {
            return s2.length() - s1.length();
        }
    }
});
System.out.println(list);
```

Static method **`sort()`** on **`Collections`** with an anonymous inner class

Alphabetical comparison

Sort from longest to shortest string

Alternative

Use the object method **`sort()`** on **`List`**

```java
list.sort((s1, s2) -> {
    if (s1.length() == s2.length()) {
        return s1.compareTo(s2);
    }
    else {
        return s2.length() - s1.length();
    }
});
System.out.println(list);
```

Lambda expression

Output

```
[Volkswagen, Mercedes, Hyundai, Tesla, Audi, Seat, BMW]
```

Break

10 min

The lecture will continue at **15:00**

Outline

- Iterators and collections
- **Enum types and switch expressions**
- Lambda expressions
- Streams

Enum

- The type Enum is an enumeration type with a finite range of values
- Example

```
public enum PizzaStatus {
    ORDERED,
    READY,
    DELIVERED
}
```

- Usage

```
PizzaStatus status = PizzaStatus.ORDERED;
System.out.println(status); // output: ORDERED
```

- Schema

```
public enum EnumName {
    Value1,
    Value2,
    Value3
}
```

- Enum values are constants
- Use enums, if a fixed number of constants is necessary (e.g. days of the week, points of the compass, positions, . . .)

Enum

- Since enum types ensure that only one instance of the constants exists in the JVM, we can safely use the **`==`** operator to compare two variables
- The **`==`** operator provides **compile-time** and **runtime safety**
- Example

```
PizzaStatus status = PizzaStatus.ORDERED;

if (status == PizzaStatus.DELIVERED) {
    System.out.println("Delivered");
}
```

- Get all values of an enum (e.g. to use it in for loops)
- **`Enum.values()`**
- Example

```
PizzaStatus[] allStates = PizzaStatus.values();
System.out.println(Arrays.asList(allStates));
```

Enum

- You can define attributes, constructors and methods in enums (as in classes)

Invokes the constructor below

```
public enum PizzaStatus {
    ORDERED(5) {
        @Override
        public boolean isOrdered() { return true; }
    },
    READY(2) {
        @Override
        public boolean isReady() { return true; }
    },
    DELIVERED(0) {
        @Override
        public boolean isDelivered() { return true; }
    };

    private final int timeToDelivery;
    public boolean isOrdered() { return false; }
    public boolean isReady() { return false; }
    public boolean isDelivered() { return false; }
    public int getTimeToDelivery() { return timeToDelivery; }
    PizzaStatus(int timeToDelivery) {
        this.timeToDelivery = timeToDelivery;
    }
}
```

Usage

```
var status = PizzaStatus.READY;
System.out.println(status.getTimeToDelivery());
System.out.println(status.isReady());
```

Output

```
2
true
```

Switch **statements** with enum

- Example

```
public int getDeliveryTimeInDays(PizzaStatus status) {
    switch (status) {
        case ORDERED: return 5;
        case READY: return 2;
        case DELIVERED: return 0;
    }
    return -1;
}
```

- Problems with switch statements
  - The compiler does **not** show an error if we forget an enum value
  - We still need to provide a **`default`** handling or handle the return after the switch statement

Switch **expressions**

- Since Java 14, there is a more elegant syntax for case distinctions

```
public int getDeliveryTimeInDays(PizzaStatus status) {
    return switch (status) {
        case ORDERED -> 5;
        case READY -> 2;
        case DELIVERED -> 0;
    };
}
```

- In case we add a new enum value in the future, the method above will lead to a compile error, so we will not forget to handle the case
- **Important**: switch expressions **do not** fall through (difference to switch statements)

Switch **expressions**

- Compiler error when a new enum value was added

```
public int getDeliveryTimeInDays(PizzaStatus status) {
    return switch (status) {
        case ORDERED -> 5;
        case READY -> 2;
        case DELIVERED -> 0;

    };
}
```

Compile error

```
Compile error:

'switch' expression does not
cover all possible input values
```

```
public enum PizzaStatus {
    ORDERED, READY, DELIVERED, DELAYED
}
```

New value

Switch **expressions**

- This allows you to fix the issue during compile time and no runtime issues occur

```
public int getDeliveryTimeInDays(PizzaStatus status) {
    return switch (status) {
        case ORDERED -> 5;
        case READY -> 2;
        case DELIVERED -> 0;
        case DELAYED -> 10;
    };
}
```

```
public enum PizzaStatus {
    ORDERED, READY, DELIVERED, DELAYED
}
```

- **Note:** the old style switch statement would not lead to a compile error

Switch expressions - more examples

```
String aString = "test";
switch (aString) {
    case "Test" -> System.out.println("Ok");
    case "Hello", "World" -> aString = "Error?";
    default -> {
        System.out.println("This is impossible!");
        System.out.println("Never!");
    }
}
```

```
String str = "test";
String str2 = switch (str) {
    case "Test" -> "O" + "k!";
    case "Hello", "World" -> str = "Error?";
    default -> {
        System.out.println("This is impossible!");
        System.out.println("Never!");
        yield "Nonsense";
    }
};
```

Switch expressions - more examples (with comments)

```
String aString = "test";
switch (aString) {
    case "Test" -> System.out.println("Ok");
    case "Hello", "World" -> aString = "Error?";
    default -> {
        System.out.println("This is impossible!");
        System.out.println("Never!");
    }
}
```

: is replaced with **->** to separate the **case** from the **alternative** to be executed (**break** is no longer required)

Several cases may be combined into a comma-separated list

All cases must be handled (potentially using **default**)

```
String str = "test";
String str2 = switch (str) {
    case "Test" -> "O" + "k!";
    case "Hello", "World" -> str = "Error?";
    default -> {
        System.out.println("This is impossible!");
        System.out.println("Never!");
        yield "Nonsense";
    }
};
```

The individual alternatives may now consist of expressions whose values are returned

The assignment is considered as an expression whose value (after the assignment to the left-hand side) is returned

An alternative may also consist of a **block {...}** of statements ending with a **yield** statement which signifies the return value

Switch expressions - more examples

```
enum DayOfWeek {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}

public class Playground {
    public static void main(String[] args) {
        DayOfWeek day = DayOfWeek.FRIDAY;

        int numCharacters = switch (day) {
            case MONDAY, FRIDAY, SUNDAY -> 6;
            case TUESDAY -> 7;
            case THURSDAY, SATURDAY -> 8;
            case WEDNESDAY -> 9;
        };

        System.out.println(day + " has " + numCharacters + " characters");
    }
}
```

L06E03 Switching the Months

Not started yet.

Start exercise

Easy

Due date tonight

10 min

2 pts

- Problem statement
  - Create a switch expression to return the number of days for a given month
  - Implement the method
    `int daysOfMonth(int month)`
  - **Optional challenge:** create an `enum Month` and use it instead
    `int daysOfMonth(Month month)`

Example solution

```java
static int daysOfMonth(int month) {
    return switch (month) {
        case 1, 3, 5, 7, 8, 10, 12 -> 31;
        case 2 -> 28;
        case 4, 6, 9, 11 -> 30;
        default -> throw new IllegalStateException("Unexpected month: " + month);
    };
}
```

Optional challenge

```java
enum Month {
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
}

public class Playground {

    static int daysOfMonth(Month month) {
        return switch (month) {
            case JANUARY, MARCH, MAY, JULY, AUGUST, OCTOBER, DECEMBER -> 31;
            case FEBRUARY -> 28;
            case APRIL, JUNE, SEPTEMBER, NOVEMBER -> 30;
        };
    }
}
```

Example solution (with comments)

```java
static int daysOfMonth(int month) {
    return switch (month) {
        case 1, 3, 5, 7, 8, 10, 12 -> 31;
        case 2 -> 28;
        case 4, 6, 9, 11 -> 30;
        default -> throw new IllegalStateException("Unexpected month: " + month);
    };
}
```

Combine all cases with the same result

All other invalid cases lead to an exception

Optional challenge

A similar enum already exists: **`java.time.Month`**

```java
enum Month {
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
}

public class Playground {

    static int daysOfMonth(Month month) {
        return switch (month) {
            case JANUARY, MARCH, MAY, JULY, AUGUST, OCTOBER, DECEMBER -> 31;
            case FEBRUARY -> 28;
            case APRIL, JUNE, SEPTEMBER, NOVEMBER -> 30;
        };
    }
}
```

Advantages of enums: no **`default`** is needed, because the data type is limited to only correct values, and the compiler warns if an enum value is missing

java.time.Month

```java
public enum Month implements TemporalAccessor, TemporalAdjuster {

    //...

    /**
     * Gets the length of this month in days. This takes a flag to determine whether to
     * return the length for a leap year or not. February has 28 days in a standard year
     * and 29 days in a leap year. April, June, September and November have 30 days.
     * All other months have 31 days.
     *
     * @param leapYear  true if the length is required for a leap year
     * @return the length of this month in days, from 28 to 31
     */
    public int length(boolean leapYear) {
        switch (this) {
            case FEBRUARY:
                return (leapYear ? 29 : 28);
            case APRIL:
            case JUNE:
            case SEPTEMBER:
            case NOVEMBER:
                return 30;
            default:
                return 31;
        }
    }
    //...
}
```

30 min

The lecture will continue at **16:10**

Outline

- Iterators and collections
- Enum types and switch expressions
- **Lambda expressions**
- Streams

Lambda expressions

- Provide a clear and concise way to represent one **method interface** using an **expression**
- Particularly useful when dealing with collections: **iterate**, **filter** and **extract** data
- Save boilerplate code
- Treated as pure functions, so the compiler does not create a class file
- Provide an implementation to the **functional interface**
  - A **functional interface** has only one abstract object method
  - You can enforce this using the annotation **`@FunctionalInterface`**
- Lambda expressions cannot be used if an interface has multiple abstract methods

Syntax

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

1. **Argument list**: can be empty or contain one or more arguments
2. **Arrow token**: used to link arguments-list and body of expression
3. **Body**: contains expressions and statements for a lambda expression

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

No arguments

```
p1 -> {body}
```

One argument → use without parentheses

```
(p1, p2, ...) -> {body}
```

Multiple arguments

Example

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
list.forEach(number -> System.out.println(number));
```

Examples

```java
import java.util.*;

public class Lambda {

    static int sum = 0;
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

        List<Integer> filteredList = new ArrayList<Integer>();
        list.forEach(n -> {
            if (n % 3 == 0) {
                filteredList.add(n);
            }
        });
        System.out.println(filteredList); // [3, 6, 9]

        // local variables cannot be used within a lambda expression (without body)
        filteredList.forEach(n -> sum += n);
        System.out.println(sum); // 18
    }
}
```

Find all numbers from **`list`** that are divisible by 3 and add them to **`filteredList`**

Calculate the sum of all values in **`filteredList`**

Functional interface

- Any interface with a **single abstract method** is a **functional interface**, and its implementation may be treated as lambda expressions
- Follows the **interface segregation principle**: developers should never be forced to implement an interface that is not used or depend on methods that are not used
- Example

```
@FunctionalInterface
interface Square {
    int area(int length);
}
```

```
// Lambda expression to define the area method and instantiate an object s of type Square
Square square = length -> length * length;
// The type of the parameter and the return type must be same as defined in the functional interface
int area = square.area(5);
System.out.println(area); // 25
```

Method references

```
var list = Arrays.asList("BMW", "Seat", "Audi", "Mercedes", "Volkswagen", "Hyundai", "Tesla");
list.forEach(System.out::println);
```

Method reference

Output

```
BMW
Seat
Audi
Mercedes
Volkswagen
Hyundai
Tesla
```

- Compact and easy way to refer to one method of a functional interface
- The argument is passed automatically
- 4 kinds of references to
  1. A static method (**`ClassName::methodName`**)
  2. An instance method of one specific object (**`objectName::methodName`**)
  3. An instance method of an arbitrary object with a specific type (**`ClassName::methodName`**)
  4. A constructor (**`ClassName::new`**)

Without method reference

```
var numbers = Arrays.asList(5, 3, 50, 24);
numbers.sort((a, b) -> a.compareTo(b));  // 3, 5, 24, 50
numbers.sort(Integer::compareTo);        // 3, 5, 24, 50
```

With method reference (kind 3)

Example: sorting points after x or y values

```java
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }

    @Override
    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
```

```java
import java.util.*;

public class Playground {
    public static void main(String[] args)  {
        var p1 = new Point(5.1, 7.8);
        var p2 = new Point(10.2, -5.1);
        var p3 = new Point(2.3, 8.6);
        var p4 = new Point(3.5, 5.9);
        var points = Arrays.asList(p1, p2, p3, p4);

        points.sort(Comparator.comparing(Point::getX));
        System.out.println(points);

        points.sort(Comparator.comparing(Point::getY).reversed());
        System.out.println(points);
    }
}
```

Ascending after x value

Descending after y value

- ***Comparator.comparing*(...)** - static method with a lambda expression that extracts the value of an object to be sorted by and uses **compareTo(...)** of the type in the list (in this case **Double**)
- **Comparator.reversed()** - default method reversing the order defined in the comparator

Difference between lambda and inner class

- The two concepts are different in an important way: **scope**
- An **inner class** creates a **new scope**
  - We can hide local variables from the enclosing scope by instantiating new local variables with the same names
  - We can also use the keyword **`this`** inside our inner class as a reference to its instance
- **Lambda** expressions, however, work with **enclosing scope**
  - We cannot hide variables from the enclosing scope inside the lambda's body
  - In this case, the keyword **`this`** is a reference to an enclosing instance

Best practices

- Keep lambda expressions (lambdas) **short** and **self-explanatory**
- Avoid blocks of code in lambdas, ideally they have **one line of code**
- Avoid specifying parameter types
  - Type inference automatically determines the type
  - Use descriptive variable names
- Avoid parentheses around a single parameter, return statement, and braces

Effectively final variables

- Accessing a non final variable inside lambdas will cause a compile-time error
- Does not mean that we should mark every variable used in lambdas as final
- Effectively final concept: a compiler treats every variable as final if it is assigned only once
- It is safe to use such variables inside lambdas because the compiler will control their state and trigger a compile-time error immediately after any attempt to change them
- Example: the following code will not compile

```
public void test() {
    String value = "value1";
    value = "value2";
    Function<String, String> fun = key -> {
        return key + ": " + value;
    };
    String result = fun.apply("test");
}
```

If you remove this statement, the code works

```
Compile error: Variable used in lambda expression
should be final or effectively final
```

→ The effectively final concept simplifies the process of making lambda execution thread safe

https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

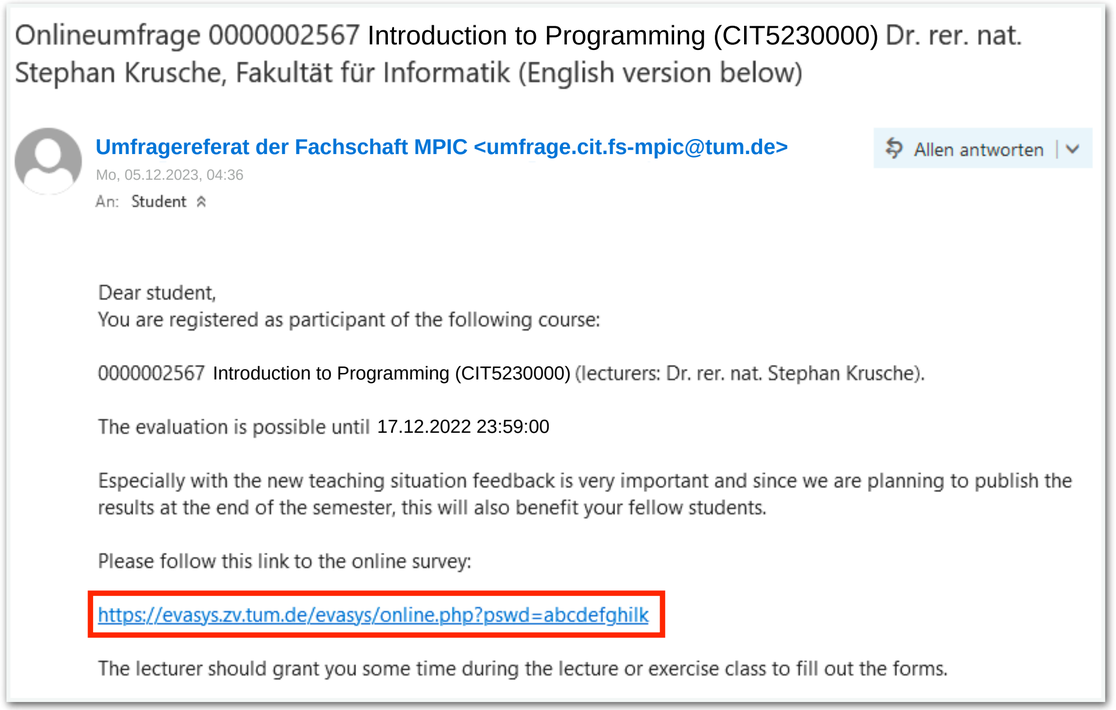
Covered later

University course evaluation

- We put a lot of effort and passion into creating **a great learning atmosphere** and providing you with the **latest concepts, tools, and workflows**
- We tailor the course specifically for the heterogeneous and international student group of management students
- We hope you **appreciate our effort** 😊 and comment on issues, that we can improve in the future semesters
- **Your feedback is valuable to us and to the university!**
- You should have received an email by the Department Student Council MPIC (”Fachschaft”) to evaluate **CIT5230000**
- **You now have 15 minutes to fill out the anonymous online survey**

University course evaluation (15 min)

- Find the email with a link to https://evasys.zv.tum.de/... for **CIT5230000**
- Fill out the survey now



- Alternative participation using Moodle

Break

10 min

The lecture will continue at **16:50**

Outline

- Iterators and collections
- Enum types and switch expressions
- Lambda expressions
- **Streams**

Streams

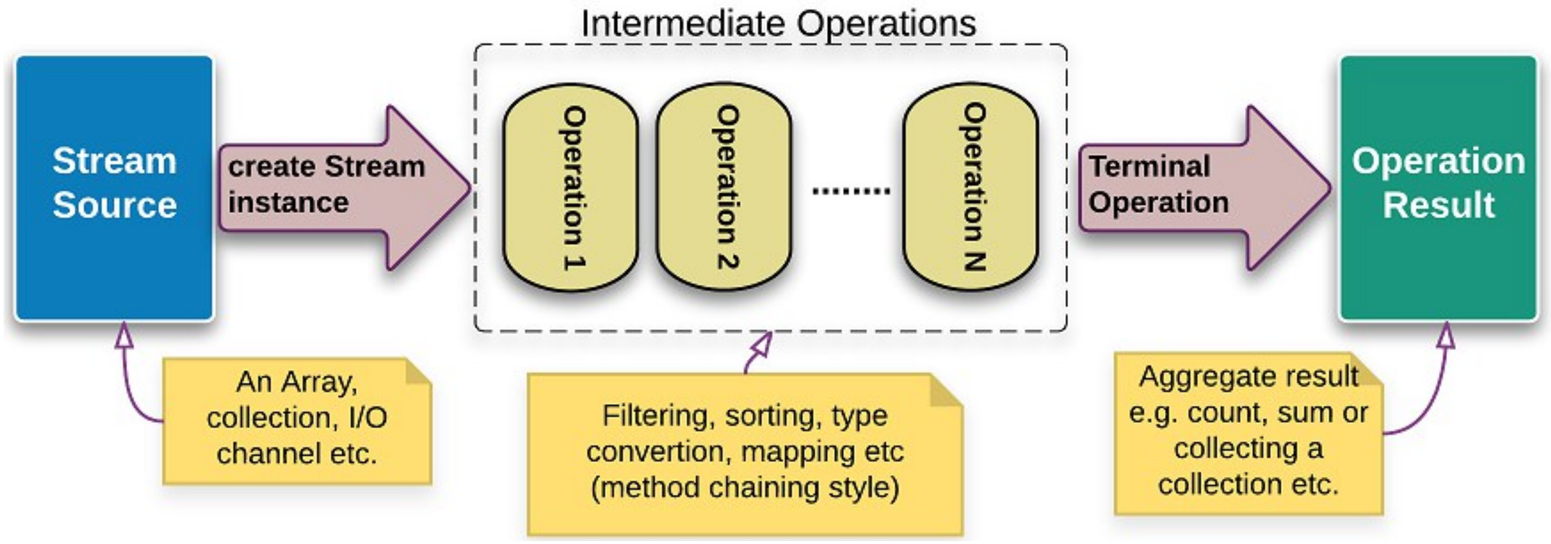
- Lambda expressions come from theoretical computer science: lambda calculus is an abstract "programming language" used to compute all computable functions
- Functional programming is strongly influenced by lambda calculus
- The basic idea is to understand computations as mathematical functions; the concept of state does not exist directly at first
- The central data structure of functional programming is the `list`
  - Many programming patterns are used again and again
  - The most used ones are **`map`** and **`fold`**
    - **`map`** applies a function to each list element
    - **`fold`** combines all list elements into a new expression (called **`reduce`** in Java)

  → Program without loops

→ Java: lambda expressions with streams

Streams

- Functional style operations on discrete sequence of data elements
- Operate on the provided source and produce results rather than modifying the source
- A stream life cycle (also called pipeline) can be divided into three types of operation
  1. Create the stream instance
  2. Zero or more intermediate operations which transform a stream into another stream, such as filtering, sorting, element transformation (mapping)
  3. A terminal operation which produces a result, such as count, sum or a new collection



Streams: **`map`** and **`filter`**

- **`map()`** produces a new stream after applying a function to each element of the original stream (the new stream could be of different type)

```
var list = Arrays.asList("BMW", "Seat", "Audi", "Mercedes", "Volkswagen", "Hyundai", "Tesla");
var lowerCaseList = list.stream().map(String::toLowerCase).toList();
System.out.println(lowerCaseList);
```

Output: [bmw, seat, audi, mercedes, volkswagen, hyundai, tesla]

- **`filter()`** produces a new stream that contains elements of the original stream that pass a given test (specified by a predicate)

```
var list = Arrays.asList("BMW", "Seat", "Audi", "Mercedes", "Volkswagen", "Hyundai", "Tesla");
var filtered = list.stream().filter(s -> s.length() == 4).toList();
System.out.println(filtered);
```

Output: [Seat, Audi]

Streams: **`map`** and **`filter`** (with comments)

- **`map()`** produces a new stream after applying a function to each element of the original stream (the new stream could be of different type)

```
var list = Arrays.asList("BMW", "Seat", "Audi", "Mercedes", "Volkswagen", "Hyundai", "Tesla");
var lowerCaseList = list.stream().map(String::toLowerCase).toList();
System.out.println(lowerCaseList);
```

Output: [bmw, seat, audi, mercedes, volkswagen, hyundai, tesla]

Simplified collect operator (**immutable**), mutable alternative: **`collect(Collectors.toList())`**

- **`filter()`** produces a new stream that contains elements of the original stream that pass a given test (specified by a predicate)

```
var list = Arrays.asList("BMW", "Seat", "Audi", "Mercedes", "Volkswagen", "Hyundai", "Tesla");
var filtered = list.stream().filter(s -> s.length() == 4).toList();
System.out.println(filtered);
```

Output: [Seat, Audi]

Only two strings have a **length** of 4 characters and remain in the list

Streams: **`reduce`**

- **`reduce()`** takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation

```
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Playground {

    public static void main(String[] args) {

        int sum = IntStream.range(1, 10).reduce(0, Integer::sum);
        System.out.println(sum);        // 45


        int product = Stream.of(5, 9, 15).reduce(1, (a, b) -> a * b);
        System.out.println(product);    // 675
    }
}
```

Combination of 3 operations **map**, **filter** and **reduce** in a parallel stream

```
int magic = Stream.of(1, 2, 3, 4, 4, 5, 6).parallel().map(i -> i * 3).filter(i -> i < 15).reduce(0, Integer::sum);
System.out.println(magic);    // 42
```

Streams: **`reduce`** (with comments)

- **`reduce()`** takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation

```java
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Playground {

    public static void main(String[] args) {

        int sum = IntStream.range(1, 10).reduce(0, Integer::sum);
        System.out.println(sum);          // 45


        int product = Stream.of(5, 9, 15).reduce(1, (a, b) -> a * b);
        System.out.println(product);      // 675
    }
}
```

Create a stream with all int values from 1 to 10

Reduce the stream by taking the sum of all values starting from 0

Create a stream with just 3 concrete int values

Reduce the stream by taking the product of all values starting from 1

Combination of 3 operations **map**, **filter** and **reduce** in a parallel stream

```java
int magic = Stream.of(1, 2, 3, 4, 4, 5, 6).parallel().map(i -> i * 3).filter(i -> i < 15).reduce(0, Integer::sum);
System.out.println(magic);    // 42
```

Allows to perform operations in parallel on multiple elements at the same time

Streams: **`collect`**

- **`collect()`** performs mutable fold operations on data elements held in the stream instance
  - Repackage elements to some data structures and apply some additional logic, concatenate them, etc.
  - The strategy for this operation is provided via the **`Collector`** interface implementation
  - Example: the **`toList`** collector collects all stream elements into a **`List`** instance

```
var list = Arrays.asList("BMW", "Seat", "Audi", "Mercedes", "Volkswagen", "Hyundai", "Tesla");
List<String> filtered = list.stream()
        .filter(s -> s.length() == 4)
        .collect(Collectors.toList());
System.out.println(filtered);  // [Seat, Audi]
```

Streams: **`collect`** (with comments)

- **`collect()`** performs mutable fold operations on data elements held in the stream instance
  - Repackage elements to some data structures and apply some additional logic, concatenate them, etc.
  - The strategy for this operation is provided via the **`Collector`** interface implementation
  - Example: the **`toList`** collector collects all stream elements into a **`List`** instance

```
var list = Arrays.asList("BMW", "Seat", "Audi", "Mercedes", "Volkswagen", "Hyundai", "Tesla");
List<String> filtered = list.stream()
        .filter(s -> s.length() == 4)
        .collect(Collectors.toList());
System.out.println(filtered);  // [Seat, Audi]
```

Collect all elements in the stream and store them in a **mutable** list

Lambda expressions and streams

- Lambda expressions are functions without names
  - We need them because in Java functions are generally not allowed as parameters (except with the `::` notation)
- Streams are sequences of values on which functions like **`map`**, **`filter`**, and **`reduce`** can be applied in the manner of functional programming
  - Often use lambda expressions and can be "stacked" (pipelines) → elegant

SOLID: 5 principles of object oriented design

- **`S` - Single responsibility principle**: a class should have one and only one reason to change, meaning that a class should have only one job
- **`O` - Open closed principle**: objects should be open for extension but closed for modification
- **`L` - Liskov substitution principle**: objects of a superclass shall be replaceable with objects of its subclasses without breaking the application
- **`I` - Interface segregation principle**: developers should never be forced to implement an interface that is **not** used or to depend on methods that are **not** used
- **`D` - Dependency inversion principle**: high level components should **not** depend on low level components; both should depend on abstractions → loose coupling

→ Software projects that follow **SOLID** principles can be shared with collaborators, extended, modified, tested, and refactored with fewer complications

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

L06E04 Comma Separator

Not started yet.

Start exercise

Easy

Due date tonight

10 min

2 pts

- Problem statement
  - Combine multiple strings into one string using a comma as separator
  - Use streams and the collect function with **`Collectors.joining(...)`**
    - Input: `["Hello", "World", "Heilbronn"]`
    - Output: `"Hello,World,Heilbronn"`
  - **Optional challenges**
    - Filter all words that begin with “H”
    - Append “!” to the end of each string using the map function

Example solution

```
var words = Arrays.asList("Hello", "World", "Heilbronn");
var combined = words.stream().collect(Collectors.joining(","));
System.out.println(combined);
```

Output: Hello,World,Heilbronn

Optional challenges

```
var words = Arrays.asList("Hello", "World", "Heilbronn");
var combined = words.stream()
        .filter(word -> word.startsWith("H"))
        .map(word -> word + "!")
        .collect(Collectors.joining(","));
System.out.println(combined);
```

Output: Hello!,Heilbronn!

Example solution (with comments)

```
var words = Arrays.asList("Hello", "World", "Heilbronn");
var combined = words.stream().collect(Collectors.joining(","));
System.out.println(combined);
```

Join all strings with “,”

Output: Hello,World,Heilbronn

Optional challenges

```
var words = Arrays.asList("Hello", "World", "Heilbronn");
var combined = words.stream()
        .filter(word -> word.startsWith("H"))
        .map(word -> word + "!")
        .collect(Collectors.joining(","));
System.out.println(combined);
```

Filter all words starting with **H**

Transform all words

Output: Hello!,Heilbronn!

Next steps

- **Tutor group** exercises
  - T06E01 - The Iterator Games
  - T06E02 - Streamception
- **Homework** exercises
  - H06E01 - The Baggagebusters
  - H06E02 - Stream Wars - A New Hope
- Read the following articles
  - https://www.javatpoint.com/java-lambda-expressions
  - https://www.baeldung.com/java-8-streams

→ Due until **Wednesday, December 13, 13:00**

Summary

- Iterators allow to quickly process data in collections using e.g. for each loops
- Enum types represent data with a finite range of constant values
- Switch expressions provide a modern and compile safe way to handle cases and alternatives without unintuitive fall through
- Lambda expressions enable functional programming and allow the use of functions as objects and arguments
- Streams are functional-style operations on a discrete sequence of data elements (e.g. collections) using a pipeline to create, transform and collect

References

- https://www.baeldung.com/java-iterator
- https://www.baeldung.com/a-guide-to-java-enums
- https://docs.oracle.com/en/java/javase/13/language/switch-expressions.html
- https://www.baeldung.com/java-8-lambda-expressions-tips
- https://www.javatpoint.com/java-lambda-expressions
- https://www.baeldung.com/java-8-functional-interfaces
- https://www.baeldung.com/java-8-streams
- https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- http://tutorials.jenkov.com/java/annotations.html
- https://docs.oracle.com/javase/tutorial/java/annotations

Intermediate exam 2 information

- **Date:** Monday, 11 December 2023, 7:00 pm - 8:40 pm
- **Time**: 90 min + 10 min, **points**: 100
- **Content**: everything until the end of lecture week 05 (Object Orientation II)
- **Location**: Garching or Munich (city campus)

You will receive an email with the actual lecture hall until Monday morning

- **Onsite**: you **must** participate in the assigned lecture hall
  - If you have a conflict with an exam in the city campus
    → fill out https://collab.dvb.bayern/x/8oHWDg until **Thu, Dec 7**

In this case, make sure to try out the computer in the “Rechnerhalle" beforehand

- **Setup**: use your own notebook
  - If you do **not** have a proper notebook, you can use a computer in the “Rechnerhalle”
    → fill out https://collab.dvb.bayern/x/8oHWDg until **Thu, Dec 7**
- **Open book**: use any resources (except AI)
- **Important: work alone, no communication is allowed!**

Rules

Same rules as for the intermediate examination I

- You must work on the exam **on your own**
  - Do **not** use chat applications (→ keep them **completely** closed all time)
  - Do **not** use artificial intelligence (OpenAI, ChatGPT, GitHub Copilot, or any similar systems are forbidden → **uninstall** or **disable** them!)
  - Do **not** post exam questions online
- You **must not** participate in the exam from home

You **must** participate in the assigned lecture hall

- You may only use one monitor (no second monitor allowed)
- You must turn off all secondary devices (smartphone, tablet, etc.)
- Suspicious behavior, plagiarism and communication with other students is classified as cheating ("Unterschleif") and leads to consequences as mentioned in the APSO ("Allgemeine Prüfungs- und Studienordnung")
- In particular, the corresponding module in TUMonline will be marked as failed (w. cheating)