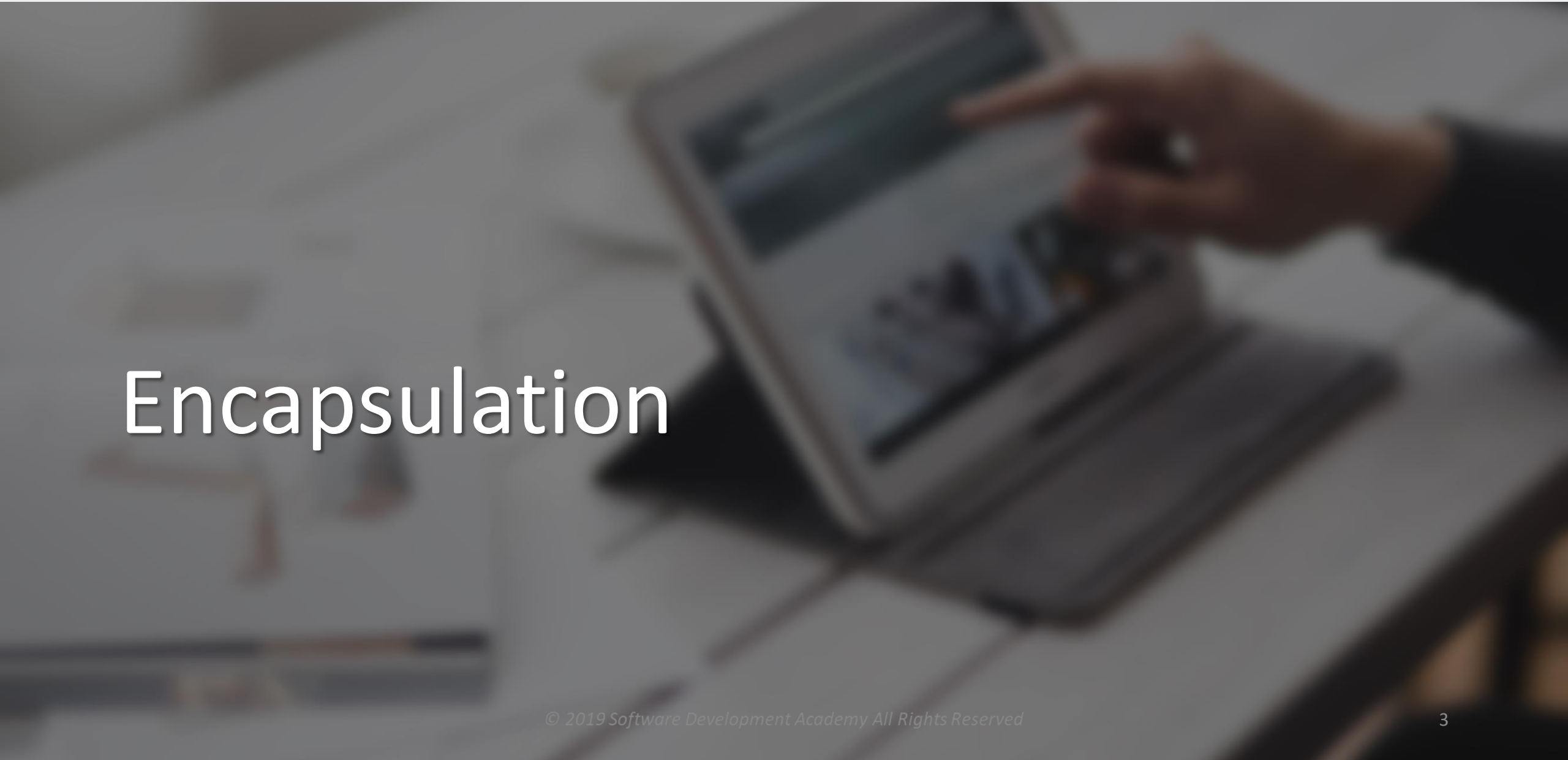
Java Advanced Features

Agenda

1. Encapsulation
2. Inheritance
3. Composition
4. Enums
5. Abstarct class
6. Interface
7. Exceptions
8. Generic types
9. Collections (and Maps)
10. Annotations
11. Input/Output
12. Concurrency
13. Lambda expressions
14. Optionals
15. Streams
16. Nested Classes



Encapsulation

Encapsulation

Encapsulation is **wrapping fields** and **methods** that will work together as **a single unit**.

Other classes will **not** be able to **access fields/methods** that they are **not supposed to**. A java class is encapsulated by applying **private modifiers** to the **fields** and adding **public methods** or **constructors** that are necessary **for other classes to use**.

Encapsulation – getters and setters

Example

```
public class Person {
    private String firstName;
    private boolean male;
    ...
    public String getFirstName() {
        return firstName;
    }
    public boolean isMale() {
        return male;
    }
    ...
}
```

```
public class Person {
    private String firstName;
    private boolean male;
    ...
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public void setMale(boolean male) {
        this.male = male;
    }
    ...
}
```

Encapsulation – constructor

Example

```
public class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

Encapsulation – other methods

Example

```
public class Person {
    private String firstName;
    ...
    private int age;
    ...
    // We do not want to assign null or empty value
    // to firstName field.
    public void setFirstName(String firstName) {
        if(firstName != null && !"".equals(firstName)) {
            this.firstName = firstName;
        }
    }
    // We only allow increasing age value by one at a time.
    public void growOlder() {
        age++;
    }
}
```

Exercises – Encapsulation

1. Create class Dog.
    a) Add private fields to the class, like name, age, gender, race, weigth.
    b) Create constructor that accepts all of the class fields.
    c) Create additional constructor, that will accept only gender and race. It should call main constructor with default values.
    d) Create getters and setters for age and weigth.
    e) Create object of class Dog. Verify if everything works as expected.
    f) Add verification for all arguments passed to the setters. E.g. setWeigth method should not accept values below or equal to 0.
2. Create class Pocket.
    a) Add field „money”, create constructor, getter and setter.
    b) Add verification for both getter and setter. Getter should result in returning as much money, as the user asked for. It should return 0 if money <= 10.
    c) Setter should not accept values below 0 and greater than 3000. It may print a message like „I don’t have enough space in my pocket for as much money!”

Inheritance

Inheritance

- A class can **inherit** from another one by adding ***extends*** **keyword** in the **class declaration**.
- If a class **inherits** from another one, it in other words **extends it**.
- The extended class is called **the parent class** (or **the base class**).
- The class that extends is called **the child class** (or **the derived class**).
- In Java a class can **extend only one class**.

Inheritance

Example

```
public class Vehicle {
    private int maxSpeed;

    public Vehicle(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }
}
```

```
public class Car extends Vehicle {
    private boolean convertible;

    public Car(int maxSpeed, boolean convertible) {
        super(maxSpeed);
        this.convertible = convertible;
    }

    public boolean isConvertible() {
        return convertible;
    }
}
```

Inheritance – assigning to parent type variable

Example

```
Car myCar = new Car(130, false);
Vehicle myCarVehicle = new Car(210, true);
```

Inheritance – overriding methods

Example

```
public class Vehicle {
    ...
    @Override
    public String toString() {
        return "Fields values: maxSpeed=" + maxSpeed;
    }
}
```

```
Vehicle vehicle = new Vehicle(85);
// System.out.println prints object's toString value,
// so it will print "Fields values: maxSpeed=85"
System.out.println(vehicle);
```

Inheritance – polymorphism

Example

```
Vehicle vehicle = new Vehicle(85);
Vehicle myCarVehicle = new Car(210, true);
vehicle.toString(); // invokes Vehicle#toString()
myCarVehicle.toString(); // invokes Car#toString()
```

Inheritance – calling parent's implementation

Example

```
public class Car extends Vehicle {
    ...
    @Override
    public String toString() {
        return super.toString()
                + ", convertible=" + convertible;
    }
}
```

Inheritance – hiding parent's fields

Example

```
public class Vehicle {
    protected int maxSpeed;
    ...
}
```

```
public class Car extends Vehicle {
    private double maxSpeed;
}
```

```
public class Car extends Vehicle{
    ...
    public int getParentsHiddenField() {
        return super.maxSpeed;
    }
}
```

Inheritance – passing parameters

Example

```
public static void main(String[] args) {
    Car myCar = new Car(120);
    // passing a Car as a parameter - it is a Vehicle as well.
    printMaxSpeed(myCar);
    // again - this time passing a Car as an Object
    // (a Car is also an Object).
    printWithPrefix("My car: ", myCar);
}

private static void printMaxSpeed(Vehicle vehicle) {
    System.out.println(vehicle.getMaxSpeed());
}

private static void printWithPrefix(String prefix, Object object) {
    System.out.println(prefix + object);
}
```

Object - universalus tipas (klases).
Negali but primityvus

Exercises – Inheritance

1. Create a Shape class.
    a) Add fields, create constructor, getters and setters.
    b) Create classes Rectangle and Circle. Both of them should inherit class Shape. Which fields and methods are common?
2. Create classes Dog and Cat.
    a) Move common methods and fields to the class Animal.
    b) Create method „yieldVoice”.
    c) Create simple array of type Animal, that will contain one object of type Dog and one object of type Cat.
    d) Using for-each loop show which animal gives what kind of voice. How to print a name of an object?

Composition

Composition

Composition and **aggregation** means **usage of objects** as values of fields of **another objects**.

```
public class Monitor {
    private int displaySize;
    ...
}
```

```
public class Mouse {
    private int buttonCount;
    private boolean optical;
    ...
}
```

```
public class ComputerSet {
    private Mouse mouse;
    private Monitor monitor;
    ...
}
```

Exercises – Composition

1. Create a Muzzle class.
    a) Create a Muzzle object.
    b) Assing the created object to a Dog object.
    c) How to use a Dog object to show all attributes of a Muzzle object?

Enums

Enums

An **enum** is a type that has a **predefined set of values**.

Enum types implicitly extend **the Enum class**.

```
public enum LengthUnit {
    METER,
    CENTIMETER,
    FOOT,
    INCH
}
```

Example

```
LengthUnit meterUnit;
```

```
LengthUnit meterUnit = LengthUnit.METER;
```

```
System.out.println(LengthUnit.METER); // prints "METER"
```

Enums – switch-case

Example

```
LengthUnit lengthUnit;
...
switch(lengthUnit) {
    case FOOT:
        System.out.println("Foot unit is selected");
        break;
    case METER:
        System.out.println("Meter unit is selected");
        break;
    case INCH:
        System.out.println("Inch unit is selected");
        break;
    case CENTIMETER:
        System.out.println("Centimeter unit is selected");
        break;
}
```

Enums – constructors, fields, methods

Example

```
public enum LengthUnit {
    METER(1),
    CENTIMETER(0.01),
    FOOT(0.3),
    INCH(0.025);

    double value;

    LengthUnit(double value) {
        this.value = value;
    }

    // returns value - length in meters
    double convertToMeters() {
        return value;
    }
}
```

Enums – methods

Example

```
// prints "0.3"
System.out.println(LengthUnit.FOOT.convertToMeters());
```

```
for (LengthUnit lengthUnit : LengthUnit.values()) {
    System.out.println(lengthUnit);
}
```

Enums – overriding methods

Example

```
public enum LengthUnit {
    METER(1, "Meter unit"),
    CENTIMETER(0.01, "Centimeter unit");

    double value;
    String prettyName;
    LengthUnit(double value, String prettyName) {
        this.value = value;
        this.prettyName = prettyName;
    }

    @Override
    public String toString() {
        return prettyName;
    }
}
```

```
// prints "Centimeter unit"
System.out.println(LengthUnit.CENTIMETER);
```

Exercises – Enums

1. Create enum Planets.
    a) Override toString method. It should print relative size of a planet and it's name. E.g. „Huge Jupiter", „Small Pluto".
    b) Create distanceFromEarth method.
    c) Verify both methods for multiple planets.

Abstract class

Abstract class

- **Abstract class** cannot be **instantiated**.
- **It has to** be **extended** to be used.
- Abstract class **might** include **abstract methods** – methods **without implementation**.
- Fields, non-abstract and static methods **work in the same way**.

Abstract class

Example

```
public abstract class Vehicle {
    private int maxSpeed;

    public Vehicle(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public abstract void move();
}
```

```
// Invalid code - abstract class cannot be instantiated
Vehicle myVehicle = new Vehicle();
```

Abstract class

Example

```
public class Car extends Vehicle {

    public Car(int maxSpeed) {
        super(maxSpeed);
    }

    @Override
    public void move() {
        System.out.println("Car moved");
    }
}
```

```
Car myCar = new Car(120);
myCar.move(); // prints "Car moved"
Vehicle myVehicle = myCar;
// Calling Car's implementation
// (myVehicle refers to a Car object).
myVehicle.move(); // prints "Car moved"
```

Exercises – Abstract class

1. Change Shape and Animal classes to be abstract. Which methods should/may be abstract?
2. Add getPerimeter and getArea methods declaration to the Shape abstract class. Implement and verify those methods for both Circle and Rectangle classes.

Interface

Interface

- **Interfaces** cannot include **fields**.
- Methods are **public** and **abstract by default**.
- **Might** include **static methods** and **fields**.
- **A class** can implement **many interfaces**.
- **Might** include ***default*** **implementation** (Java 8).
- **Interfaces** can be **extended**.

Example

```
public interface Shape {
    // public and abstract by default
    double getArea();
    double getPerimeter();
}
```

```
// Invalid code - interface cannot be instantiated
Shape myShape = new Shape();
```

Interface – implementation

Example

```
class Rectangle implements Shape {
    private double a;
    private double b;
    ...
    @Override
    public double getArea() {
        return a * b;
    }

    @Override
    public double getPerimeter() {
        return 2 * a + 2 * b;
    }
}
```

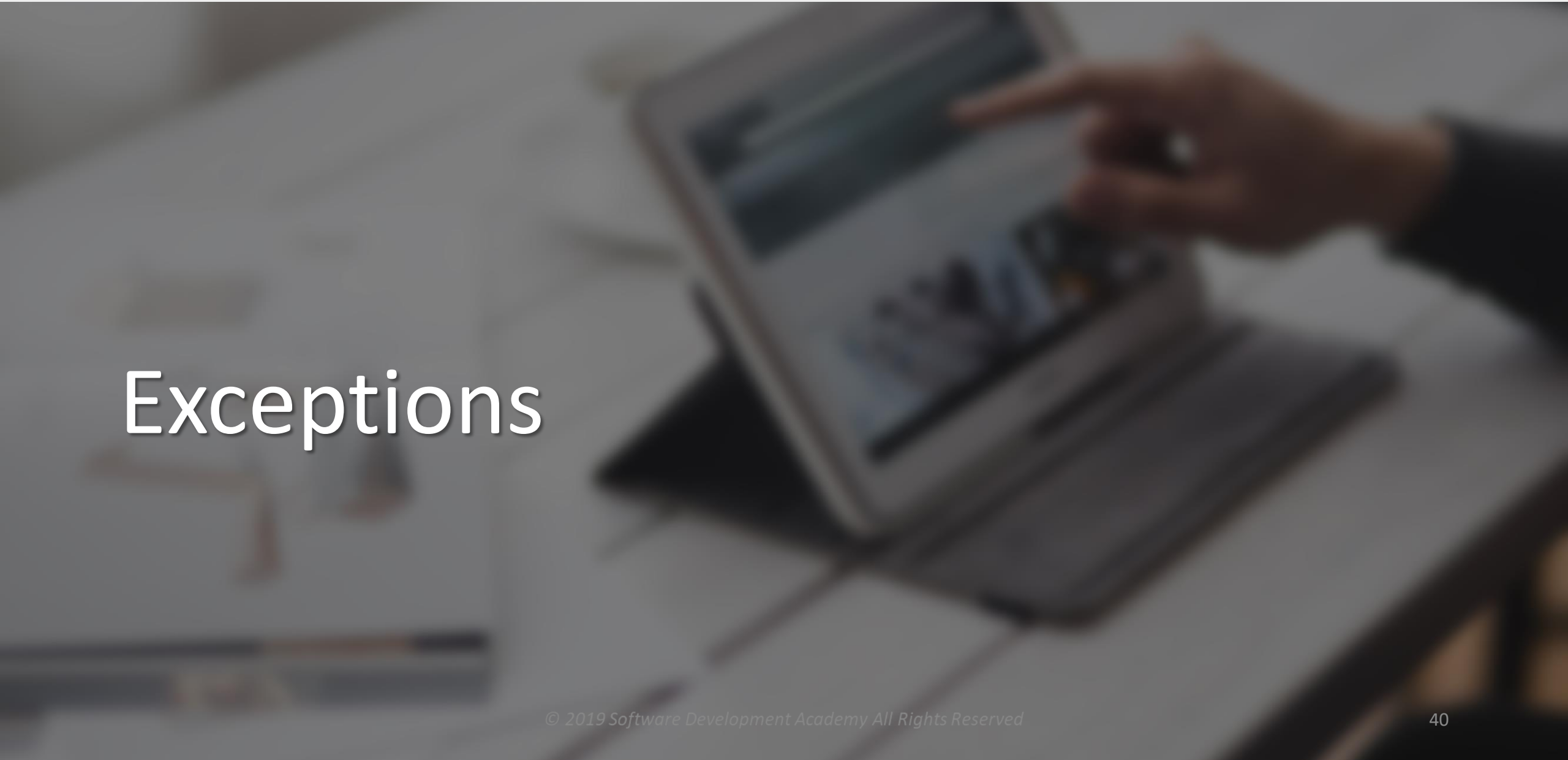
```
Rectangle myRectangle = new Rectangle(4, 5);
System.out.println(myRectangle.getArea());
Shape myShape = myRectangle;
// Calling Rectangle#getPerimeter implementation.
System.out.println(myShape.getPerimeter());
```

Interface – default method

Example

```
public interface Shape {
    double getArea();
    double getPerimeter();

    default void print() {
        System.out.println("Shape: " + this);
    }
}
```

```
Rectangle myRectangle = new Rectangle(4, 5);
// prints "Shape: " + myRectangle#toString methods value
myRectangle.print();
```



Exceptions

Exceptions

- If a program's execution is disrupted **an exception is thrown**.
- **Exceptions** can be **caught** and **handled**.
- **Exceptions** are represented **as objects**.

Exceptions – base hierarchy

Base type for **all exceptions** is the **Throwable class**.

Two types extend **Throwable**:

- **Error** – errors are caused **by environment** in which application is running - program **does not want to handle them**.
- **Exception** – an exception is thrown due to **program's fault**. It makes sense to **handle them**. **Every exception** caught by a program will derive **from Exception class**.

Exception – try-catch

Example

```
try {
    int x = 5 / 0;
} catch(Exception e) {
    System.out.println("Exception is caught and handled!");
}
```

Exception – try-catch-finally

Example

```
try {
    x = y / 0;
} catch(Exception e) {
    System.out.println("Exception is caught and handled!");
} finally {
    System.out.println("This will be printed no matter what the value of y is")
}
```

Exception – multiple catch

Example

```
try {
    int x = intArray[10] / y;
} catch(ArithmeticException e) {
    System.out.println("ArithmeticException caught!");
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("ArrayIndex(...)Exception caught!");
} catch(Exception e) {
    System.out.println("Another exception caught!");
}
```

Exception – catch with mutiple parameter types

Example

```
try {
    int x = intArray[10] / y;
} catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
    System.out.println("ArithmeticException or a " +
            "ArrayIndexOutOfBoundsException caught!");
}
```

Exception – throwing an exception

Example

```
throw new AnExceptionType(arguments);
```

```
throw new Exception("Some program failure");
```

Exception – „throws” in method signature

Example

```
int[] someIntArray = {3, 4, 2};
// invalid code - printArrayElement throws Exception, so it needs
// to be handled (putting it into try-catch or adding throws Exception
// to main method declaration will fix it).
printArrayElement(someIntArray, 3);
```

```
private static void printArrayElement(int[] intArray, int index) throws Exception {
    if (index < 0 || index >= intArray.length) {
        throw new Exception("Incorrect argument!");
    }
    System.out.println(intArray[index]);
}
```

Exception – custom exceptions

Example

```
public class CarCrashedException extends Exception {
    public CarCrashedException(Car car) {
        // calling Exception(String message) constructor
        super("Car " + car + " has crashed!");
    }
}
```

```
public class Car {
    private int speed;

    public void increaseSpeed() throws CarCrashedException {
        speed += 70;
        if(speed > 200) {
            throw new CarCrashedException(this);
        }
    }
}
```

Exception – autoclosing

Example

```
try(Scanner scanner = new Scanner(System.in)) {
    // code
}
```

Exercises – Exception

1. Write an application that will read the input and print back value that user provided, use try-catch statements to parse the input, e.g. I/O:
   Input: 10
   Output: int -> 10

   Input: 10.0
   Output: double -> 10.0

   Input: „Hello!”
   Output: „Hey! That’s not a value! Try once more.”
2. What do you think about raising an exception: when should we raise an exception and when return Null on method „failure”? Try to implement one method for each situation.

Generic types

Generic types

A **generic type** is **a generic class** or **interface** that is **parametrized over types**.

Generic types

Example

```
public class GenericBox<T> {
    private T item;

    public Box(T item) {
        this.item = item;
    }

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}
```

Example

```
Car car = new Car();
// new GenericBox<Car>(car) is valid as well, but there
// is no need to put the parametrized type again
// in the diamonds
GenericBox<Car> boxWithACarInIt = new GenericBox<>(car);
```

Generic types – „extends” keyword

Example

```
public abstract class Vehicle {
    public abstract void repair();
}
```

```
public class Car extends Vehicle {
    public void repair() {
        System.out.println("Car is repaired");
    }
}
```

```
public class Garage<T extends Vehicle> {
    private T vehicle;

    public Garage(T vehicle) {
        this.vehicle = vehicle;
    }

    public void repairVehicle() {
        vehicle.repair();
    }
}
```

Generic types – „extends” keyword usage

Example

```
Car car = new Car();
Garage<Car> garage = new Garage(car);
garage.repair();
```

Generic types – generic interface

Example

```
public interface Comparable {
    public int compareTo(T o);
}
```

```
public class Car implements Comparable<Car> {
    private int maxSpeed;

    public Car(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    @Override
    public int compareTo(Car otherCar) {
        return this.maxSpeed - otherCar.maxSpeed;
    }
}
```

Generic types – generic interface usage

Example

```
Car car1 = new Car(200);
Car car2 = new Car(150);

if(car1.compareTo(car2) > 0) {
    System.out.println("car1 is faster!");
}
```

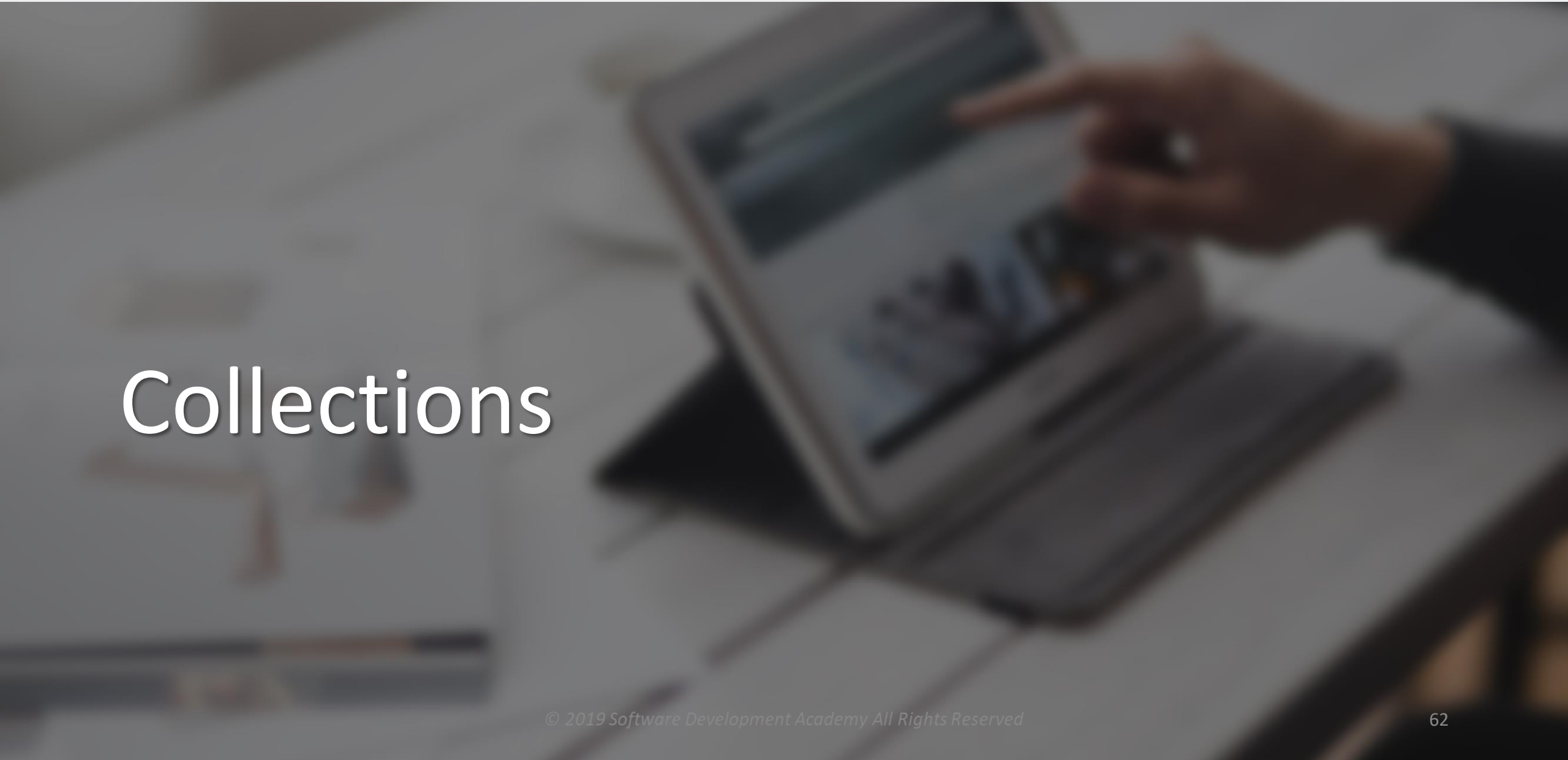
Generic types

There is a convention that the **parametrized type name is a single uppercase letter**. Also there is a convention specifying **which letter should be used**:

- **E, T** - element type
- **K** - key type
- **V** - value type
- **T** - type
- **N** - number type
- **S, U, V** if there are more parametrized types

Exercises – Generic types

1. Create a Person class that will implement a Comparable interface. Person class should implement compareTo method, that will verify if one person is taller than another.
2. Create a simple Generic class, that will give a possibility, to store any kind of value within. Add object of type String, Integer and Double to array of that Generic type. Print all values of the array within a loop.



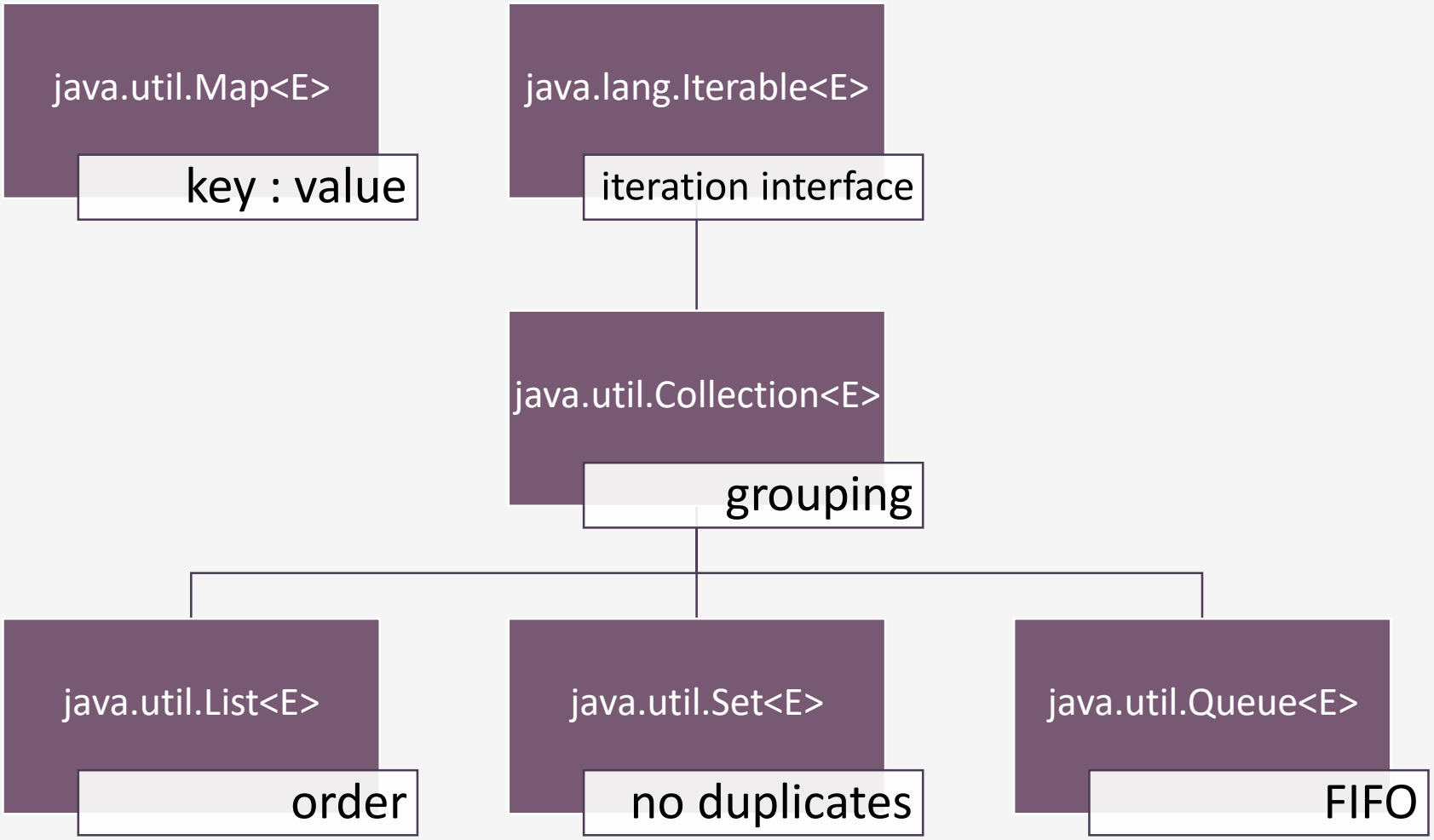
Collections

Collections

Java offers very useful types that represent ***collections*** (sets, lists etc.).
Three main types are these interfaces:

- *Collection* represents a collection.
- **List** – extends *Collection*, represents an ordered list of elements.
- **Set** – extends *Collection*, represents a set of elements
  (a set does not contain duplicates, order is not crucial).
- **Map** represents **a dictionary** – storage of pairs of **keys** and **values**.

Collections – hierarchy



List

A List type object is very similar to an **array object** - it’s elements have **0-based indices**. List is a **generic type**, parametrized type represents the type of it's elements. It includes many **useful methods**.

List is an interface that extends the *Collection* interface. To assign a value to a variable of such type you have to use **it's implementation**.
There are **various implementation** of the **List interface**.
One of them is **an *ArrayList***.

Collections – List

Example

```
List<String> visitedCountries = new ArrayList<>();
visitedCountries.add("Germany");
visitedCountries.add("France");
visitedCountries.add("Spain");
visitedCountries.remove("France");

// prints Germany Spain, as "France" was removed
for (String country : visitedCountries) {
    System.out.print(country + " ");
}
```

Collections – List using Iterator

Example

```
Iterator<String> iterator = visitedCountries.iterator();
// prints Germany Spain, as "France" was removed
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}
```

Exercises – List

1. Create a List and display its result (data should be provided by the user - console):
    a) Purchases to be made. *If an element already exists on the list, then it should not be added.
    b) *Add to the example above the possibility of "deleting" purchased elements
    c) Display only those purchases that start with „m” (e.g. milk)
    d) * View only purchases whose next product on the list starts with „m” (e.g. eggs, if milk was next on the list)
2. Ratings received. Display their average. The numbers can not be less than 1 and greater than 6.
3. * List of lists - multiplication table

Set

Set is **an interface** that extends *Collection* interface. A set either contains **an element** or **it doesn't** – an element **cannot** be present in a set **multiple times**.

Set is a **generic interface** (parametrized type is the type of elements). If we want to get a Set instance, we need it's **implementation**. We will use the ***HashSet*** class.

Collections – Set

Example

```
Set<String> travelRoute = new HashSet<>();
travelRoute.add("Berlin");
travelRoute.add("Paris");
travelRoute.add("Madrid");
travelRoute.add("Paris");
travelRoute.add("Berlin");
travelRoute.remove("Paris");

// prints Berlin Madrid, order may differ
// as set is not ordered
for (String country : travelRoute) {
    System.out.print(country + " ");
}
```

Exercises – Set

1. Create a set consisting of colors - given from the user.
2. Present the removal of individual elements from the set.
3. Display the collection before and after sorting.

Map

A **Map** is **an object** that maps **keys** to **values**. A map **cannot contain duplicate keys**: each key can map **at most** to **one value**.

Collections – Map

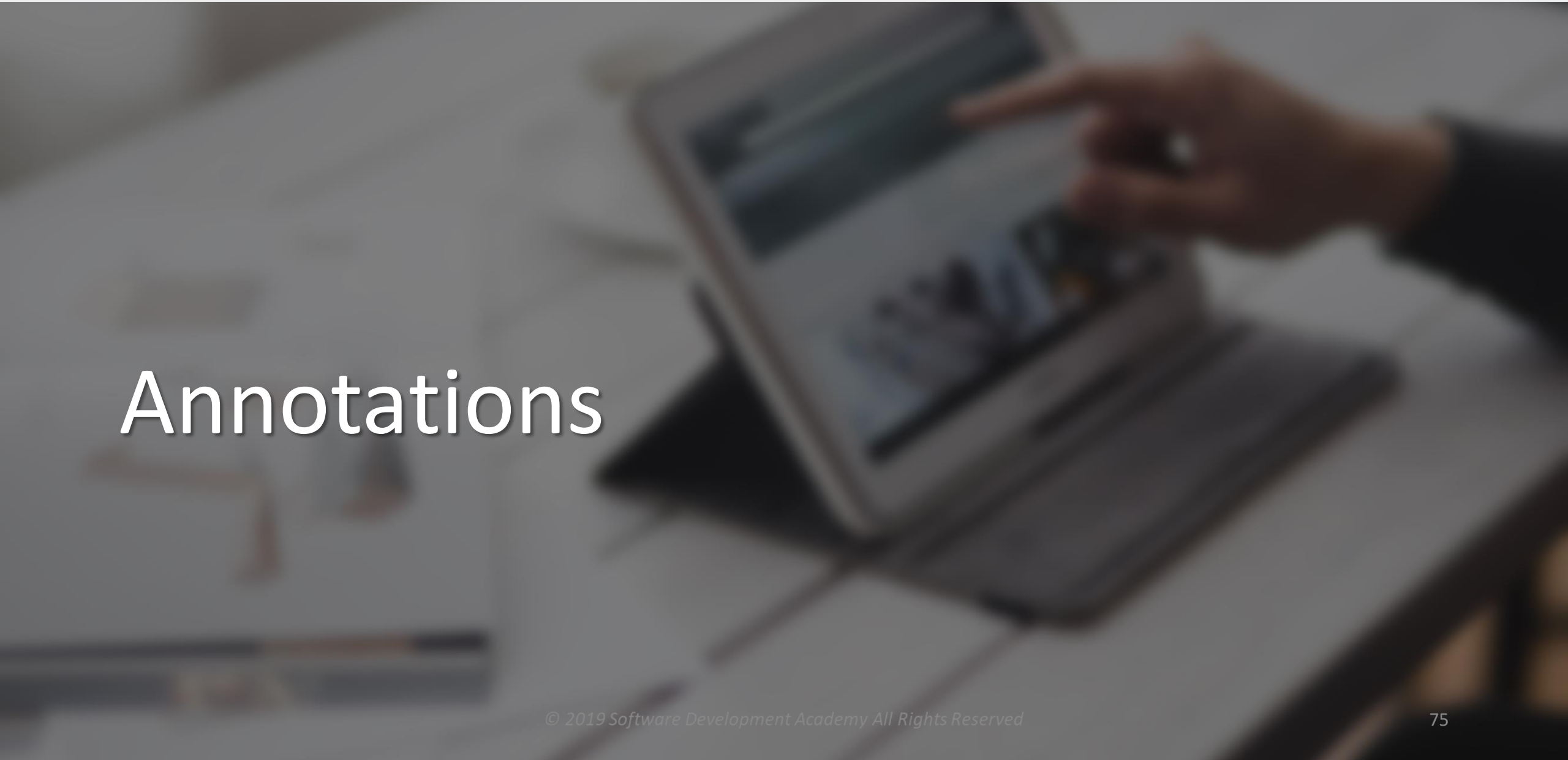
Example

```
Map<String, String> countries = new HashMap<>();
countries.put("Poland", "Warsaw");
countries.put("Germany", "Berlin");
for(Map.Entry<String, String> dictionary: countries.entrySet()) {
    String country = dictionary.getKey();
    String capital = dictionary.getValue();
    System.out.printf("%s : %s\n", country, capital);
}
```

Exercises – Map

1. Create a map and display its result (data should be provided by the user - console):
    a) Names and surnames
    b) Names and ages.
    c) Names and lists of friends (other names).
    d) * Names and details (map of maps), e.g.

    „Mike”:
    „ID”: „...”,
    „birthPlace” : „…”
    …



Annotations

Annotations

- Inform **the compiler** about **code structure**.
- Tools can **process annotations** to generate **code**, **documentation**.
- Annotations can be **processed during runtime**, e.g. classes can be found by **their annotations**.

Annotations

Annotation is a **java type**. It is used by writing "**@**" character followed by annotation's **type name**.

Annotation ***@Override*** informs compiler that method is declared in either **parent class** or in **implemented interface**. It helps to avoid mistakes when **overriding methods**.

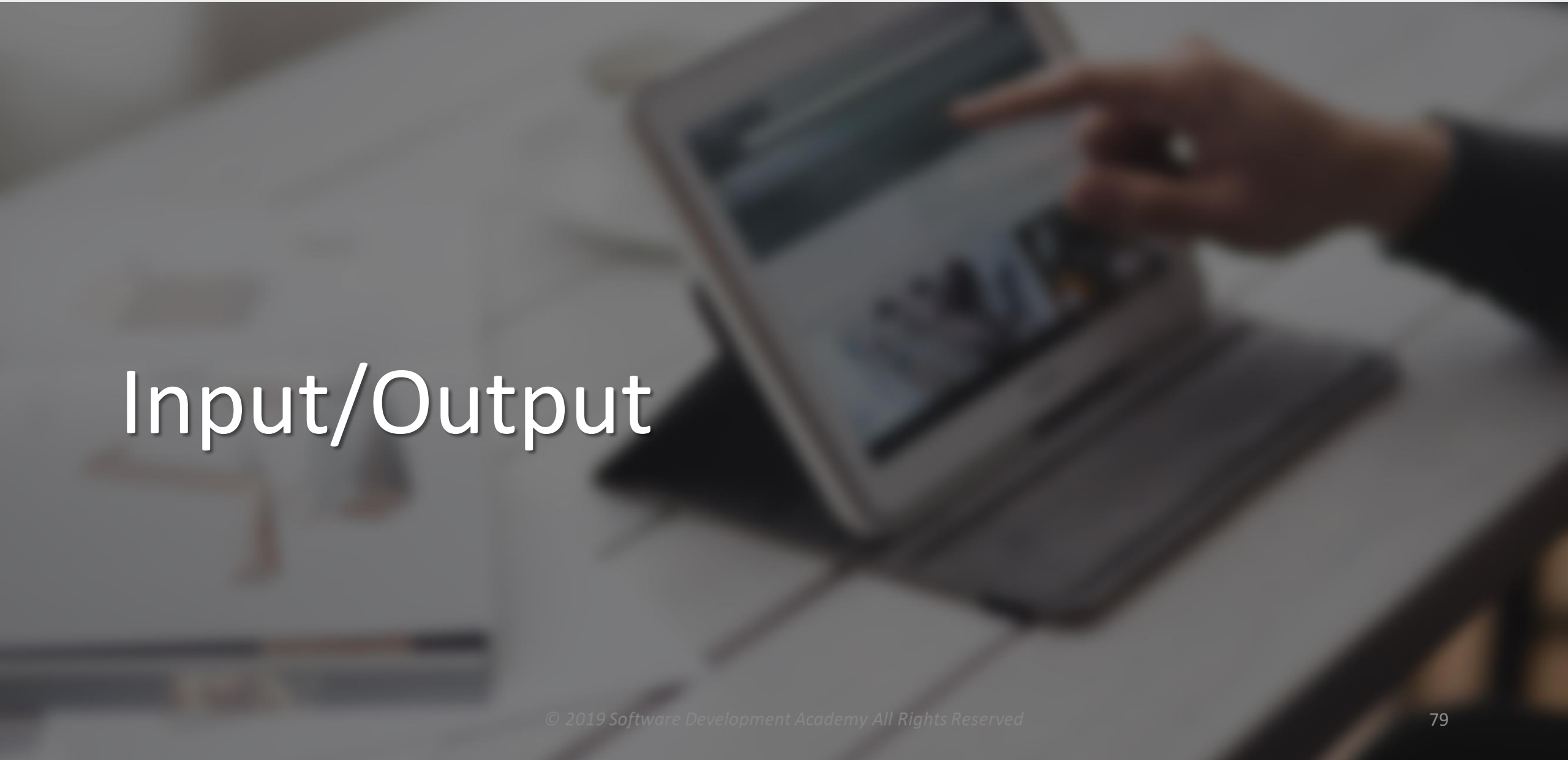
```
@Override
public String toString() {
    //...
}
```

Annotations

Annotations can be **parametrized**.

In the example annotation will make the compiler assume, that **the code is correct** (it will **not result** in **showing warnings**).

```
@SuppressWarnings("unchecked")
public static void addToList(List list, String string) {
    list.add(string);
}
```



Input/Output

I/O

A File object represents a file. It can be created passing the path to it's constructor. Path can be either absolute or relative.

```
File absoluteFile = new File("C:/myDirectory/myFile.txt");
File relativeFile = new File("myFile.txt");
```

I/O – reading and writing files

Example

```
try (BufferedReader bufferedReader = new BufferedReader(new FileReader(file))) {
    String fileLine;
    while ((fileLine = bufferedReader.readLine()) != null) {
        System.out.println(fileLine);
    }
}
```

```
try (BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(file))) {
    String fileLine = "file line";
    bufferedWriter.write(fileLine);
}
```

```
try (BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(file, true)) {
    String fileLine = "\nappended file line";
    bufferedWriter.write(fileLine);
}
```

I/O – Serializable

Example

```
public class Person implements Serializable {
    private String firstName;
    private String lastName;

    // ... constructors and other methods
}
```

```
Person person = new Person("Michael", "Dudikoff");
try (FileOutputStream fileOutputStream = new FileOutputStream(file);
     ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream)) {
        objectOutputStream.writeObject(person);
}
```

```
Person person;
try (FileInputStream fileInputStream= new FileInputStream(file);
     ObjectInputStream objectInputStream =
        new ObjectInputStream(fileInputStream)) {
    person = (Person) objectInputStream.readObject();
}
```

New I/O

Java 8 introduced **new classes** for **reading/writing files**.

They are put in the *java.nio* package.

```
Path absolutePath = Paths.get("C:/a.txt");
Path relativePath = Paths.get("a.txt");
```

New I/O – reading and writing files

Example

```
List<String> fileLines = Files.readAllLines(path);
List<String> fileLines = Files.readAllLines(absolutePath,
                                            Charset.forName("UTF-8"));
```

```
List<String> fileLines = Arrays.asList("first line", "second line");
Files.write(path, fileLines);
```

```
Files.write(absolutePath, fileLines, StandardOpenOption.APPEND);
```

New I/O – directories

Example

```
Files.createDirectory(path);
```

```
Files.isDirectory(relativePath);
```

Exercises – I/O and New I/O

1. Create a file with a „lorem ipsum” paragraph within – it can be done by copy-pasting existing paragraph or generating it dynamically using Java library. Read that file.
    a) Count words.
    b) *Count special signs (like comma, dot, spaces).
    c) *Select one word and print it’s number of occurences.

Concurrency in Java

Concurrency

Concurrency is ability to perform **multiple operations simultaneously**. Maintainer and resources needed to perform a set of operations is called **thread**. Every application has **at least one thread**.

Concurrency – main thread

If we run an application with the *public static void main(String[] args)* method, **a thread is created**. We will refer to this thread as **the main thread**.

Concurrency – sleep method

Example

```
public static void main(String[] args) throws InterruptedException{

    System.out.println("Main thread starts");
    Thread.sleep(5000);
    System.out.println("Main thread is still running");
    Thread.sleep(5000);
    System.out.println("Main thread ends");

}
```

Concurrency – extending Thread class

Example

```
public class StopWatchThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Stop watch: " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Concurrency – extending Thread class

Example

```
public static void main(String[] args) throws InterruptedException{

    StopWatchThread stopWatchThread = new StopWatchThread();
    stopWatchThread.start();
    System.out.println("Main thread starts");
    Thread.sleep(5000);
    System.out.println("Main thread is still running");
    Thread.sleep(5000);
    System.out.println("Main thread ends");

}
```

Concurrency – extending Thread class

Example

```
class StopWatchThread extends Thread {
    private String prefix;

    StopWatchThread(String prefix) {
        this.prefix = prefix;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(prefix + ": " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Concurrency – extending Thread class

Example

```
public static void main(String[] args) throws InterruptedException{

    StopWatchThread stopWatchThread1 = new StopWatchThread("SW1");
    StopWatchThread stopWatchThread2 = new StopWatchThread("SW2");
    stopWatchThread1.start();
    stopWatchThread2.start();
    System.out.println("Main thread starts");
    Thread.sleep(5000);
    System.out.println("Main thread is still running");
    Thread.sleep(5000);
    System.out.println("Main thread ends");

}
```

Concurrency – implementing Runnable interface

Example

```
class StopWatchThread implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Stop watch: " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
Thread stopWatchThread = new Thread(new StopWatchThread());
stopWatchThread.start();
```

Concurrency – synchronization

Example

```
public class Bench {
    private int availableSeats;

    public Bench(int availableSeats) {
        this.availableSeats = availableSeats;
    }

    public void takeASeat() {
        if (availableSeats > 0) {
            System.out.println("Taking a seat.");
            availableSeats--;
            System.out.println("Free seats left " + availableSeats)
        } else {
            System.out.println("There are no available seats. :(");
        }
    }
}
```

Concurrency – synchronization

Example

```
public class SeatTakerThread extends Thread {
    private Bench bench;

    public SeatTakerThread(Bench bench) {
        this.bench = bench;
    }

    @Override
    public void run() {
        bench.takeASeat();
    }
}
```

Example

```
public static void main(String[] args) throws InterruptedException{

    Bench bench = new Bench(1); // creating bench with one free seat
    SeatTakerThread seatTaker1 = new SeatTakerThread(bench);
    SeatTakerThread seatTaker2 = new SeatTakerThread(bench);
    seatTaker1.start();
    seatTaker2.start();

}
```

```
> Taking a seat.
> Taking a seat.
> Free seats left -1
> Free seats left -1
```

Concurrency – synchronization

Example

```
public synchronized void takeASeat() {
    if(availableSeats > 0) {
        System.out.println("Taking a seat.");
        availableSeats--;
        System.out.println("Free seats left " + availableSeats;
    } else {
        System.out.println("Cannot take a seat. :(");
    }
}
```

```
> Taking a seat.
> Free seats left 0
> Cannot take a seat. :(
```

Concurrency – synchronization

Example

```
public void methodWithSyncedCodeBlock() {
    System.out.println("Unsynced part");
    synchronized (this) {
        System.out.println("Synced part");
        //...
    }
}
```

Exercises – Concurrency

1. Create a class implementing the Runnable interface (implementing the run method):
    a) Inside the run method display „Hello!”
    b) Create a class object.
    c) Start the thread receiving the created object as a parameter (new Thread (<object>).start ())
    d) Create several objects, run a separate thread for each of them.
    e) Add the constructor to the created class, that accepts the int value.
    f) For the displayed data inside the run method, add the received value (Hello + value).
    g) Add a method to the class that will modify the int value.
    h) Add a while loop to the run method, inside which it will print the modified int value every few seconds.
    i) Add the ability to disable (gracefully shutdown) the thread. Why shouldn’t we just „kill” the thread?

Exercises – Concurrency

1. *You are the manager. You have 5 employees. Simulate the situation in which each of them comes at a different time to work.
    a) Every employee, after getting to work, displays the information „<name: I came to work at <time HH:MM>.”
    b) Every 10 seconds, the employee displays „name: I’m still working!”
    c) Every 30 seconds, we release one of the employees to home (remember about stopping the thread!) and remove the employee from the „active employees list”
    d) When you release your employee to home, print „<name: <time HH:MM>, it's time to go home!”
    e) * When you release a given employee, all of the others speed up. From that moment, display the information about work 2 seconds faster.
    f) ** The manager decides in which order release employees (e.g. through an earlier defined list)

Lambda Expression

Lambda Expression

Java 8 introduced **lambda expressions**. A lambda expression is a **representation** of **an implementation** of **an interface** which consists of only **one method**.

```
(parameter names) -> expression to evaluate
```

Lambda expression – predicate (functional interface)

Example – without lambda

```
public class AdultPersonTest implements Predicate<Person> {
    @Override
    public boolean test(Person person) {
        return person.getAge() >= 18;
    }
}
```

```
Person andyMurray = new Person("Andy", "Murray", 30);
Predicate<Person> adultPersonTest = new AdultPersonTest();
System.out.println(adultPersonTest.test(andyMurray));
```

Example – with lambda

```
Person andyMurray = new Person("Andy", "Murray", 30);
Predicate<Person> adultPersonTest = p -> p.getAge() > 30;
System.out.println(adultPersonTest.test(andyMurray));
```

Lambda expression

Example

```
public static void main(String[] args){

    Runnable myRunnable = () -> System.out.println("Running a runnable");
    myRunnable.run();
    Predicate<String> startsWithABCTest = s -> s.startsWith("ABC");
    System.out.println(startsWithABCTest.test("ABCDEF"));

}
```

Lambda expression – function interface

Example

```
public static void main(String[] args){

    // Method apply returns Integer and the type of the parameter is String
    Function<String, Integer> stringLengthFunction = s -> s.length();
    System.out.println(stringLengthFunction.apply("ABCDE"));

    // Method apply returns String and the type of the parameter is String
    Function<String, String> replaceCommasWithDotsFunction = s -> s.replace(',', '.');
    System.out.println(replaceCommasWithDotsFunction.apply("a,b,c"));

}
```

Lambda expression – method reference

Example

```
Person johnSmith = new Person("John", "Smith", 15);
Predicate<Person> adultPersonTest = Person::isAdult;
System.out.println(adultPersonTest.test(johnSmith));
```

Lambda expression – supplier, consumer

Example

```
Supplier<Integer> randomNumberSupplier = () -> new Random().nextInt();
int randomNumber = randomNumberSupplier.get();
```

```
Consumer<Double> printWithPrefixConsumer = d -> System.out.println("Value: " + d);
printWithPrefixConsumer.accept(10.5);
```

Lambda expression – operator

Example

```
UnaryOperator<Integer> toSquareUnaryOperator = i -> i * i;
System.out.println(toSquareUnaryOperator.apply(5));
```

Lambda expression – block of code

Example

```
UnaryOperator<Integer> toSquareUnaryOperator = i -> {
    int result = i * i;
    System.out.println("Result: " + result);
    return result;
};
toSquareUnaryOperator.apply(4);
```

Exercises – Lambda expression

1. Create and present the usage of lambda expressions:
    a) Addition, subtraction, multiplication, division.
    b) The sum of elements (int type) of the list.
    c) Number of words in the input expression (list containing elements of type String).
    d) * List before and after sorting (use the Arrays class and lambda expressions: String :: compareToIgnoreCase as a comparator)

Optionals

Optional class

Optional **class** is **a wrapper** that can be used if we **are not sure** if **a value is present**. It contains a useful **set of methods**.

```
// of method does not allow null value
Optional<String> stringOptional1 = Optional.of(stringVariable);
// ofNullable method does allow null value
Optional<String> stringOptional2 = Optional.of(stringVariable);
```

Example

```
// returns true if value is present
stringOptional.isPresent();
```

```
if(stringOptional.isPresent()) {
    String value = stringOptional.get();
}
```

```
// Prints value if it is present
stringOptional.ifPresent(System.out::println);
```

```
String value = stringOptional.orElse("default value");
```

Streams

Streams

Streams (available in Java > 8) can be used to perform **operations on collections** with the usage of **lambda expressions**.

Let's assume we have a list of people. We would like to get the average age of people named "Thomas".

Streams make such operations **very easy to implement**.

Streams – collect, findFirst, findAny

Example

```
List<String> names = Arrays.asList("Andrew", "Brandon", "Michael");
Stream<String> namesStream = names.stream();
```

```
List<String> names = Arrays.asList("Andrew", "Brandon", "Michael");
List<String> namesCopy = names.stream()
        .collect(Collectors.toList());
```

```
// we use Optional, because stream may have no values
Optional<String> firstName = names.stream()
        .findFirst();
```

```
Optional<String> firstName = names.stream()
        .findAny();
```

Streams – filter, map

Example

```
List<String> names = Arrays.asList("Andrew", "Brandon", "Michael");
List<String> namesStartingWithA = names.stream()
        .filter(n -> n.startsWith("A"))
        .collect(Collectors.toList());
```

```
List<String> names = Arrays.asList("Andrew", "Brandon", "Michael");
List namesLengths = names.stream()
        .map(String::length)
        .collect(Collectors.toList());
```

```
List<String> names = Arrays.asList("Andrew", "Brandon", "Michael");
OptionalDouble averageNameLengthOptional = names.stream()
        .mapToInt(String::length)
        .average();
averageNameLengthOptional.ifPresent(System.out::println);
```

Streams – allMatch and anyMatch

Example

```
boolean allNamesLengthIsGtThan3 = names.stream()
        .allMatch(n -> n.length() > 3);
```

```
boolean thereIsANameWhichLengthIsGtThan3 = names.stream()
        .anyMatch(n -> n.length() > 3);
```

Streams – reduce method

Example

```
// Initial value is empty string.
// If current result value is an empty string we simply add element to an empty string,
// otherwise we concatenate current value, ", " and element.
String namesConcatenation = names.stream()
        .reduce("",
                (currValue, element) -> (currValue.equals("") ? "" : currValue + ", ") + element);
```

Streams – forEach, sorted

Example

```
// The same as names.forEach(System.out::println);
names.stream()
        .forEach(System.out::println);
```

```
List<Person> people = Arrays.asList(
        new Person("John", "Smith", 20),
        new Person("Sarah", "Connor", 30)
);
people.stream()
        .sorted((p1, p2) -> p1.getLastName().compareTo(p2.getLastName()))
        .forEach(p -> System.out.println(p.getLastName()));
// alternative way:
// people.stream().sorted(Comparator.comparing(Person::getLastName)) ...
```

Exercises – Streams

1. Using streams, for a given lists:
   - [„John”, „Sarah”, „Mark”, „Tyla”, „Ellisha”, „Eamonn”]
   - [1, 4, 2346, 123, 76, 11, 0, 0, 62, 23, 50]
   a) Sort the list.
   b) Print only those names, that start with „E” letter.
   c) Print values greater than 30 and lower than 200.
   d) Print names uppercase.
   e) Remove first and last letter, sort and print names.
   f) *Sort backwards by implementing reverse Comparator and using lambda expression.

Nested classes

Nested classes – non-static

A **nested class** can be defined **in body of another class**.
We will call them **inner** and **outer classes**.

Object of **non-static inner type** is bound
to **an object of outer type**.

Nested classes – non-static

Example

```
public class Bicycle {
    private int maxSpeed = 40;

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public class Wheel {
        public void damage() {
            // we can refer to outer class's field
            // (Wheel type object will be created
            // for an Bicycle instance)
            maxSpeed *= 0.5;
        }
    }
}
```

Nested classes – non-static

Example

```
Bicycle bicycle = new Bicycle();
// Prints 40
System.out.println(bicycle.getMaxSpeed());
Bicycle.Wheel wheel = bicycle.new Wheel();
wheel.damage();
// Prints 20
System.out.println(bicycle.getMaxSpeed());
```

Nested classes – static

Static nested classes work similarly, the difference is that an instance of **inner static class** **is not bound** to an instance **of outer class**.

Nested classes – static

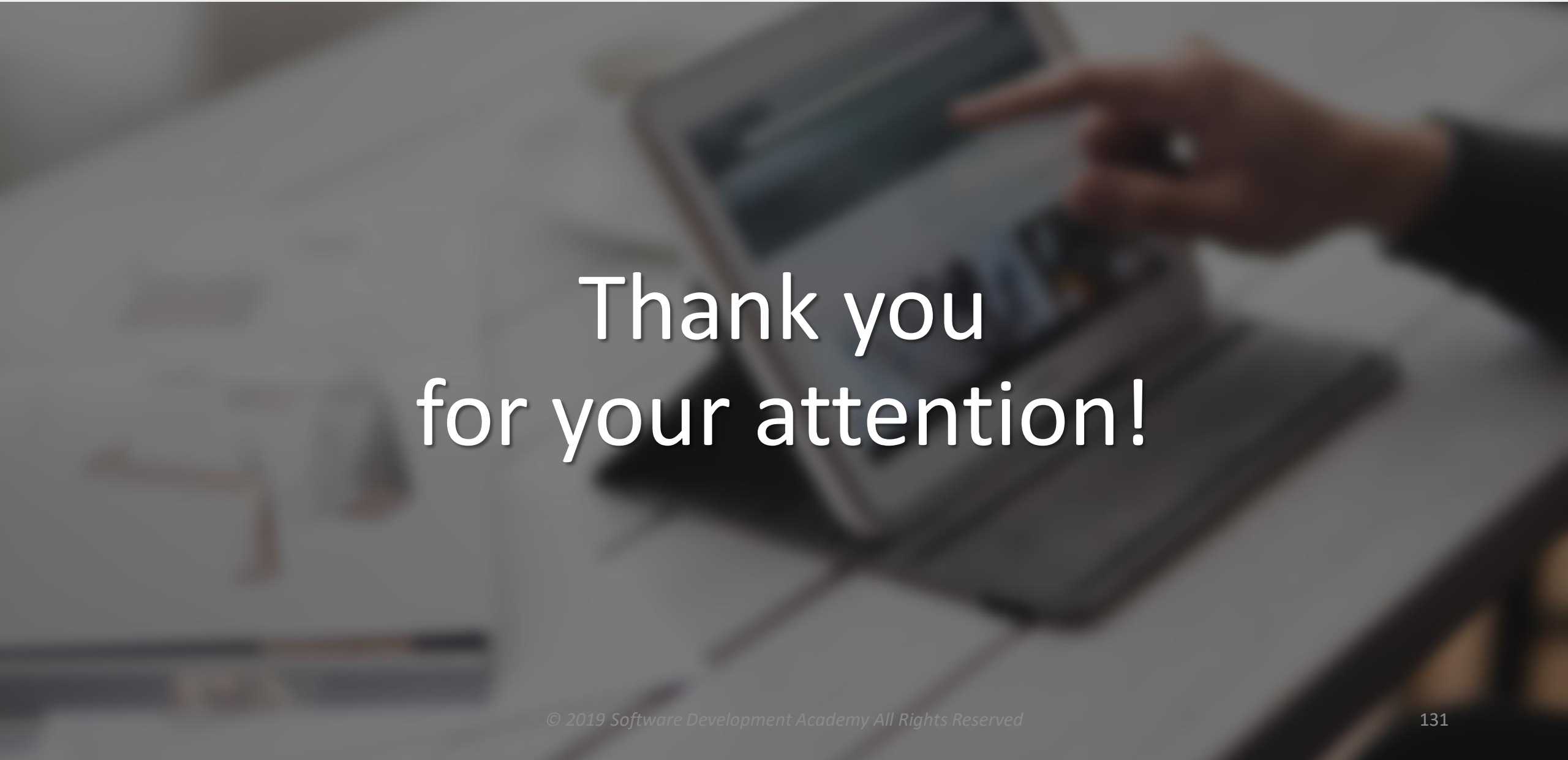
Example

```
public class Bicycle {
    private int maxSpeed = 25;

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public static class Mechanic {
        public void repair(Bicycle bicycle) {
            // nested static class can refer
            // private field of outer class
            bicycle.maxSpeed += 15;
        }
    }
}
```

Nested classes – static

Example

```
Bicycle bicycle = new Bicycle();
// Prints 25
System.out.println(bicycle.getMaxSpeed());
Bicycle.Mechanic mechanic = new Bicycle.Mechanic();
mechanic.repair(bicycle);
// Prints 40
System.out.println(bicycle.getMaxSpeed());
```



Thank you for your attention!