Java Fundamentals

Agenda

1. History of Java
2. Few words about Java
3. Our first application – *„Hello, World!”*
4. Data types and variables
5. Operators and casts
6. Strings
7. Control Flow
8. Loops
9. Arrays
10. Object-oriented programming
11. Varargs
12. Date, time
13. Regular expressions

History of Java

- James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project (initially called „Oak”) in 1991.
- The small team of sun engineers called Green Team.
- Sun changed the name of the Oak language to Java (from Java coffee), after a trademark dispute from Oak Technology.
- In 1995 Sun Microsystems released the first public implementation as JDK Alpha and Beta. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms
- In 1996 At the first-ever JavaOne developer conference, more than 6000 attendees gather to learn more about Java technology. Sun licenses java to operating systems vendors, including Microsoft, Apple, IBM, and others.
- JDK 1.1 was released in 1997. It includes JavaBeans API and Java Database Connectivity (JDBC).
- In 1999 HotSpot 1.0 was released and became the default Sun JVM in Java1.3.
- 2017 brought JDK 9 with jshell and reactive streams on board.

Design goals of the Java

1. Simple, Object Oriented, and Familiar
2. Robust and Secure
3. Architecture Neutral and Portable
4. High Performance
5. Interpreted, Threaded, and Dynamic

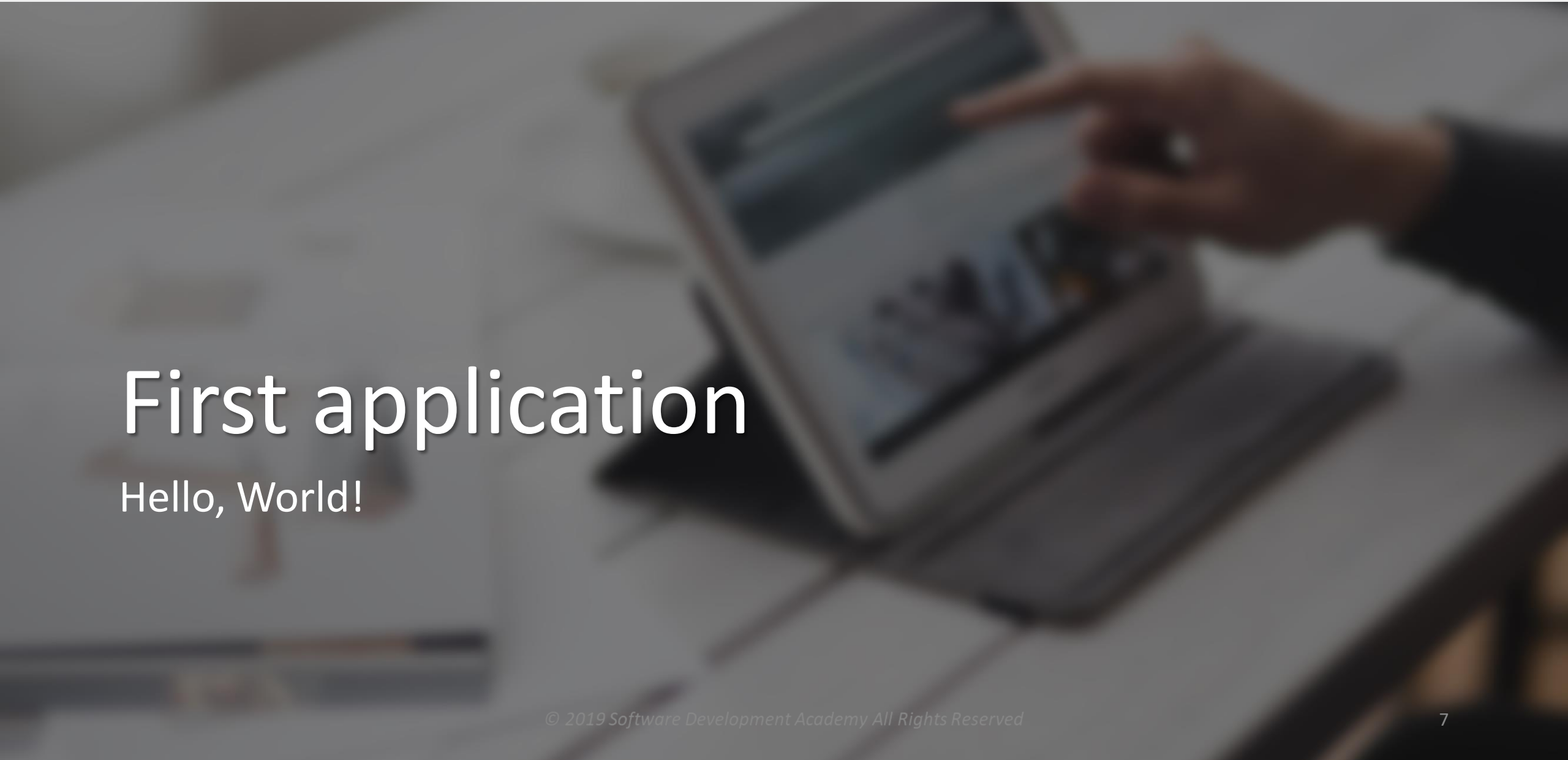
Source: http://www.oracle.com/technetwork/java/intro-141325.html

Basic assumptions of language

1. Architecture neutral
2. Distributed
3. Dynamic
4. High Performance
5. Interpreted
6. Multithreaded
7. Object-Oriented
8. Platform independent
9. Portable
10. Robust
11. Secured
12. Simple

Java Environment

- JDK (Java Development Kit) – the software for programmers who want to write Java programs
- JRE (Java Runtime Environment) – the software for consumers who want to run Java programs
- IDE (Integrated Development Environment) – a software application which enables users to more easily write and debug Java programs



First application

Hello, World!

Hello, World!

```java
public class Application {

    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }

}
```

```
$ javac Application.java
```

```
$ java Application
Hello, World!
```

Exercises – first application modification

1. Change default text, that is printed on the console. E.g. „Hello, Mike!”.

```
Hello, Mike!
```

2. Print the same text twice.

```
Hello, World!
Hello, World!
```

3. Print different text in multiple lines.

```
Hello, World!
It’s a great day, to learn something new.
```

4. *Split line in the middle – use only one `System.out.println` method.

```
Hello,
World!
```

Data types

Data types

- Java is strongly typed language
- Every variable must have a declared type. There are eight primitive types:
  - four are integer types: **byte, short, int, long**
  - two are floating-point number types: **float, double**
  - one is character type char for individual characters: **char**
  - one is a boolean type for truth values: **boolean**

Data type examples

- 56 – **int** literal
- 523342.5432 – **double** literal
- 'g' – **char** literal
- true – **boolean** literal

Variables

Variables

- A variable is **a storage location** in a computer program.
- Each variable has **a name** and **holds a value.**
- In Java, every variable has **a type.**
- Good practice – use a **short, descriptive, meaningful** variable **name**!
- There are four types of variables in java: **block, local, instance, static**.

Variables

Declaration

```
int width;
boolean done;
double factor;
```

Declaration with initialization

```
int width = 1920;
boolean done = false;
double factor = 4.127;
```

Variables

- A variable name **must begin with a letter** and **must be a sequence** of **letters** or **digits**.
- A letter is defined as 'A'–'Z', 'a'–'z', '_', '$', or any Unicode character that denotes a letter in a language.
- Similarly, digits are '0'–'9' and any Unicode characters that denote a digit in a language.
- The **first** letter **should be lowercase**, and then normal **CamelCase** rules should be used.
- All characters in the name of a variable are significant and case is also significant.

Java keywords

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

Exercises – data types and variables

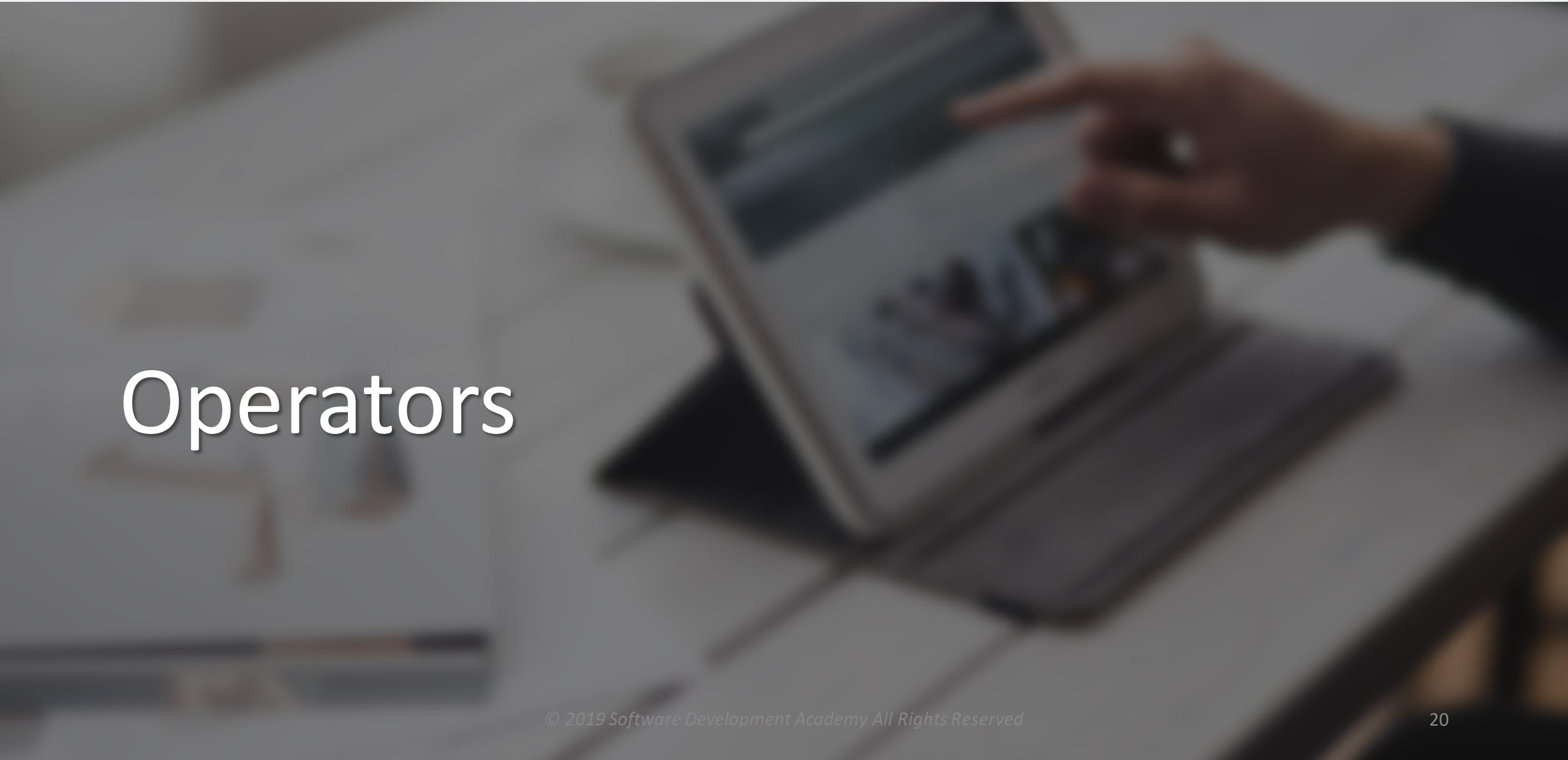
1. Define (declare and initialize) two variables: one of type „int” and second of type „double”. Print theirs values.

Output:
19
187.2342

2. Define variable of type int. What is a maximum and minimum value, that you are able to store within that variable?
3. Do the same as above for other numeric types (long, double, byte..). Check results.
4. Define the int type variable with maximum value. Add 1 to it. What do you think will happen?

Exercises – data types and variables

1. Create two variables of type int with initial values of 6 and 11.
   Print sum of those variables.
2. Read any double literal from the console. Print that value rounded to the second decimal place.
   Input: 3.23523
   Output: 3.23
3. *Print values: 192, 168, 1, 10 in HEX format XX:XX:XX:XX.
   Use *System.out.printf*() method.
   Input: 192, 168, 1, 10
   Output: „C0:A8:01:0A”



Operators

Operators

- Data in Java is manipulated using **operators**.
- Java operators produce **new values** from **one** or **more operands**.
- **Operands** are the things **on the right or left side** of the **operator**.
- The result of **most operations** is either **a boolean or numeric** value.

Operators

ASSIGNMENT OPERATORS

=, +=, -=, *=, /=

RELATIONAL OPERATORS

<, <=, >, >=, ==, !=

ARITHMETIC OPERATORS

+, -, *, /, %, ++, --

LOGICAL OPERATORS

&, |, ^, !, &&, ||

BITWISE OPERATORS

&, |, ^, ~

BIT-SHIFTING OPERATORS

<<, >>, >>>

CONDITIONAL OPERATOR

? :

INSTANCEOF OPERATOR

instanceof

Exercises – operators

1. Write in a comment on each line what result you expect.
   Launch it and verify the results.

```
int x =  4;
System.out.println(x++);
System.out.println(--x);
System.out.println(x % 3);
System.out.println(11 % 2);
System.out.println(7 % x++);
System.out.println(x == 4);
System.out.println(x != 4);
x = 10;
int y = 5;
System.out.println(x == 10 && y <= 5);
System.out.println(x <= y && y > 5);
System.out.println(„abc” instanceof String);
```

Casts

Casts

- Numeric conversions are possible in Java.
- Conversions in which loss of information is possible are done by means of casts.
- The syntax for casting is to give the target type in parentheses, followed by the variable name.

```
double y = 89.832;
int x = (int) y;
```

Strings

Strings

- A string is a **sequence of characters**.
- Java strings are **sequences of Unicode characters**.
- Java **does not have** a built-in **string type**, **strings** are **objects.**
- The **standard library** contains a predefined **class** called **String**.
- Strings are **immutable objects**!

Strings

Definition

```
String a = "abc";
String b = new String("abc");
```

Equality

```
System.out.println(a == b); // false
System.out.println(a.equals(b)); // true
```

Concatenation

```
String h = "Hello";
String w = "World!";
String text = h + ", " + w;
```

Useful methods from API

- char charAt(int index)
- int compareTo(String other)
- boolean endsWith(String suffix)
- **boolean equals(Object other)**
- **boolean equalsIgnoreCase(String other)**
- int indexOf(String str)
- int lastIndexOf(String str)
- **int length()**
- **String replace(CharSequence oldString, CharSequence newString)**
- **boolean startsWith(String prefix)**
- String substring(int beginIndex)
- **String toLowerCase()**
- String toUpperCase()
- String trim()

StringBuilder

- Should be used when you have to make **a lot of modifications** to strings of characters.
- Every time you concatenate strings, a **new String object is constructed**.
- This is **time-consuming** and **wastes memory** – use StringBuilder class to avoid this problem.
- Prefer **StringBuilder** to StringBuffer.

Exercises – strings

1. Create variable of type String. Initialize it with value *„Lorem ipsum dolor sit amet, consectetur adipiscing elit”*.
   a) ***Convert it to lower case.***
   b) ***Convert it to upper case.***
   c) ***Replace „o” with „z”.***
   d) ***Check if your variable ends with „elit”.***
2. Write in a comment on each line what result you expect.
   Launch it and verify the results.

```
String a = "abc";
String b = "abc";
String c = new String("abc");
System.out.println(a == b);
System.out.println(a.equals(b));
System.out.println(b == c);
System.out.println(b.equals(c));
```

Control flow

If statement

- **The if statement** is commonly referred to as **decision statements**
- Rules for using **else and else if**:
  - You can have **zero or one else** for a given **if**, and it must come **after** any **else ifs.**
  - You can have **zero to many else ifs** for a given **if** and they must come **before the (optional) else.**
  - Once an **else if succeeds**, **none** of the remaining **else ifs nor** the **else will be tested**.

If statement

```
if (booleanExpression) {
    // statement or block of code
}
```

```
if (booleanExpression) {
    // statement or block of code
} else {
    // statement or block of code
}
```

```
if (booleanExpression) {
    // statement or block of code
} else if (booleanExpression) {
    // statement or block of code
    ...
} else
    //
}
```

If statement

Example

```
if (points >= 100) {
    System.out.println("You win!");
}
```

```
if (age < 18) {
    System.out.println("You are teenager!");
} else {
    System.out.println("You are adult!");
}
```

```
if (age < 18) {
    System.out.println("You are teenager!");
} else if (age > 100) {
    System.out.println("You are very old!");
} else {
    System.out.println("You are adult!");
}
```

Exercises – if statement

1. Modify the sample application so that the retrieved number (age) comes from the console. Verify the application for each case (number smaller, equal to or greater than …).
2. Pick from the console a value from 0 to 5. On the basis of the obtained value, display any sign. For example, for number 0, display "*", for 1 display "$" (or any other).
3. * As above, but instead of values, operate on strings.
   E.g. for the word "star", display "*".

Switch statement

- The if/else construct can be **cumbersome** when you have to deal **with multiple selections** with many **alternatives**.
- The **switch statement** provides a cleaner way to handle **complex** decision **logic**.

```
switch (expression) {
    case constant1:
        // statement or block of code
        break;
    case constant2:
        // statement or block of code
        break;
    ...
    default:
        // statement or block of code
}
```

Switch statement

Example

```
switch (direction) {
    case 'n':
        System.out.println("You are going North!");
        break;
    case 's':
        System.out.println("You are going South!");
        break;
    case 'e':
        System.out.println("You are going East!");
        break;
    case 'w':
        System.out.println("You are going West!");
        break;
    default:
        System.out.println("Bad direction!");
}
```

Switch statement

- **A switch's expression** must evaluate to a **char, byte, short, int, an enum, and a String**.
- You **won't be able** to **compile** if you use types of **long, float, and double**.
- A **case constant** must evaluate to **the same type** that the **switch expression can use**.
- A **case constant** must be a **compile-time constant**!
- The **default keyword** should be used in a **switch statement** if you want to run some code **when none** of the case values **match the conditional value**.

Exercises – switch statement

1. Modify the sample application so that the retrieved direction comes from the console. Verify the application for each case (‚e', ‚w'…).
2. Pick from the console a value from 0 to 5. On the basis of the obtained value, display any sign. For example, for number 0, display "*", for 1 display "$" (or any other).
3. * As above, but instead of values, operate on strings.
   E.g. for the word "star", display "*".

Loops

Loops

- Loops let **repeat a block of code** as long as some **condition is true**, or for a specific number of **iterations**:
  - **while** loop,
  - **do while** loop,
  - **for** loop,
  - **enhanced for** loop.

While loop

- The while loop executes a block or statement as long, as some condition is true.
- Loop will never execute if the condition is false at the outset.

```
while (expression) {
    // statement or block of code
}
```

While loop

Example

```
int x = 3;
while (x > 1) {
    System.out.println(x);
    x--;
}
```

```
while (true) {
    System.out.println("Endless loop...");
}
```

Exercises – while loop

Every exercise below should be done using while loop. Always add *System.out.println(„…”)* inside the loop, to check, if it works as expected.

1. Print your name 5 times.
2. Create while loop that will never execute.
3. Create while loop that will print the same value, to the console, as long, as application will be active.
4. Within a loop read text from console and print it back (simple „echo”).
5. Within a loop read text from console and print it backwards.

Do While loop

- The **do while loop** is quite similar **to the while loop**.
- The code in a do loop **is guaranteed** to execute **at least once**.
- The expression **is not evaluated** until after the do loop's code **is executed.**

```
do {
    // statement or block of code
} while (expression);
```

Do While loop

Example

```
do {
    System.out.println("Greetings from do while loop!");
} while (false);
```

```
do {
    System.out.println("Endless loop...");
} while (true);
```

Exercises – do-while loop

Every exercise below should be done using do while loop. Always add *System.out.println(„…”)* inside the loop, to check, if it works as expected.

1. Print your name 5 times.
2. Create do-while loop that will execute only once.
3. Create do-while loop that will print the same value, to the console, as long, as application will be active.
4. Within a loop read text from console and print it back (simple „echo”).

For loop

- Is especially useful for flow control when you already know how many times you need to execute the statements in the loop's block.
- Has three main parts:
  - Declaration and initialization of variables,
  - the boolean expression (conditional test),
  - the iteration expression.

```
for (initialization; condition; iteration) {
    // statement or block of code
}
```

For loop

Example

```
for (int x = 0; x < 10; x++) {
    System.out.println("x is " + x);
}
```

```
for ( ; ; ) {
    System.out.println("Endless loop...");
}
```

Exercises – for loop

Every exercise below should be done using for loop. Always add *System.out.println(„…”)* inside the loop, to check, if it works as expected.

1. Print your name 5 times.
2. The same as above, but your application should also print the actual value of the index.
   Output:
   Mike: 0
   Mike: 1
   ..
   Mike: 4
3. The same as above, but index should be printed from the biggest value (5 included) to the smallest one.
4. *Calculate sum of index value from 10 to 30, using for loop.
5. *Create nested for loop. Print actual values of the iterators.
   E.g.:
   i=5 : j=0
   i=5 : j=1
   i=5 : j=2

   …

Enhanced For loop

- The enhanced for loop is a specialized for loop that simplifies looping through an array or a collection
- Has two main parts:
  - Declaration - the newly declared block variable
  - Expression - must evaluate to the array or collection (instance of *java.lang.Iterable*)

```
for (declaration : expression) {
    // statement or block of code
}
```

Enhanced For loop

Example

```
for (Animal a: animals) {
    System.out.println(a);
}
```

```
int[] arrayOfInts = {1, 2, 3, 4, 5, 6};
for (int n : arrayOfInts) {
    System.out.println(n);
}
```

Loop control flow

Code in loop	Behaviour
break	execution jumps immediately to the first statement after the loop
continue	stops just the current iteration (jumps to the next iteraton)
return	execution jumps immediately back to the calling method
System.exit()	all program execution stops; the VM shuts down

Exercises – loops

Choose the best loop for every task. Always add *System.out.println(„…”)* inside the loop, to check, if it works as expected.

1. Do simple „echo” application. Your application should work as long, as you won't write „quit”.
2. The same as above, but if you'll write „continue” – your application should go back to the beginning of your loop, without printing back your text.
3. *Draw rectangle from stars
   Use nested for loops – parent loops iterator should be called „row”, child one – „column”.
   Output:
   ****
   ****
   ****
4. **Draw rectangle empty inside (only edges).

Arrays

Arrays

- Arrays are the **fundamental mechanism** in Java for collecting **multiple values**.
- Arrays can hold **primitives or objects**, but the array itself is **always an object**.
- You access **each individual** value through an integer **index**.
- Arrays are **indexed** beginning with **zero**.
- An *`ArrayIndexOutOfBoundsException`* occurs if you use a **bad index value**.
- Arrays have a **length** attribute whose value is the **number** of array **elements**.

Example

```
dataType[] array; // recommended  dataType []array;
dataType []array;
dataType array[];
```

```
int[] arrayOfInts;
String[] arrayOfStrings;
```

Example

```
dataType[] array = new dataType[size]; // recommended
dataType []array = new dataType[size];
dataType array[] = new dataType[size];
```

```
int[] arrayOfInts = new int[5];
// initialization
arrayOfInts[0] = 10;
arrayOfInts[1] = 15;
arrayOfInts[2] = 20;
arrayOfInts[3] = 25;
arrayOfInts[4] = 30;
```

```
String[] arrayOfStrings = new String[2];
// initialization
arrayOfStrings[0] = "Tree";
arrayOfStrings[1] = "Forest";
```

Example

declaration, instantiation and initialization

```
dataType[] array = new dataType[]{el1, el2, ... , eln};
dataType[] array = {el1, el2, ... , eln};
```

```
int[] arrayOfInts = new int[]{10, 15, 20, 25, 30};
int[] arrayOfInts = {10, 15, 20, 25, 30};

String[] arrayOfStrings = new String[]{"Tree", "Forest"};
String[] arrayOfStrings = {"Tree", "Forest"};
```

Example

accessing

```
dataType[] array = new dataType[]{el1, el2, ... , eln};
dataType[] array = {el1, el2, ... , eln};
```

```
int[] arrayOfInts = new int[]{10, 15, 20, 25, 30};
int[] arrayOfInts = {10, 15, 20, 25, 30};

String[] arrayOfStrings = new String[]{"Tree", "Forest"};
String[] arrayOfStrings = {"Tree", "Forest"};
```

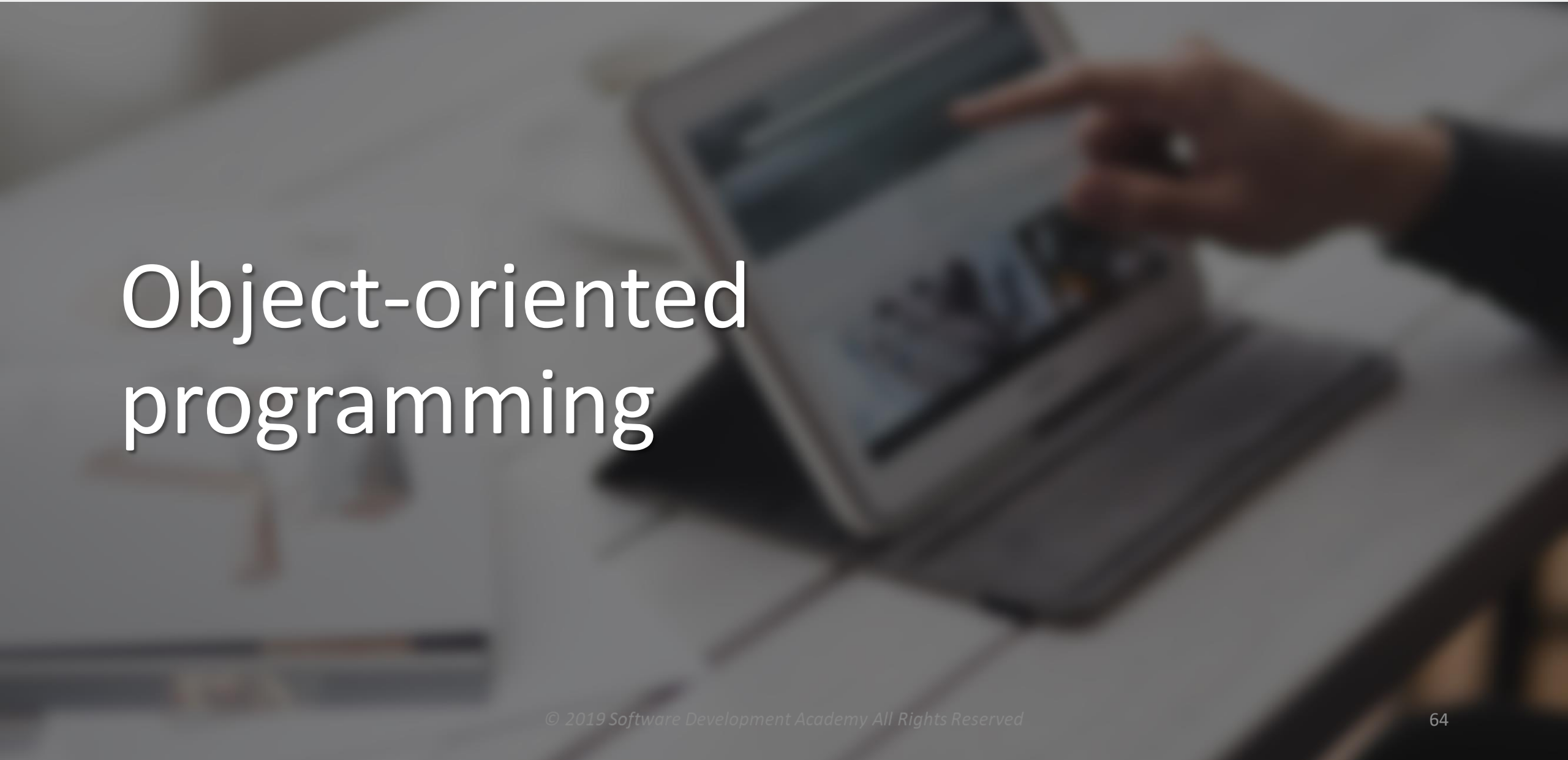
Example

accessing

```
int[] arrayOfInts = {10, 15, 20, 25, 30};
System.out.println(arrayOfInts[0]); // prints 10
System.out.println(arrayOfInts[2]); // prints 20
System.out.println(arrayOfInts[4]); // prints 30

// prints 10 15 20 25 30
for (int i = 0; i < arrayOfInts.length; i++) {
    System.out.print(arrayOfInts[i] + " ");
}

System.out.println(); // go to the next line

// prints 10 15 20 25 30
for(int i : arrayOfInts) {
    System.out.print(i + " ");
}
```

Exercises – arrays

1. Create int array with the specified size. Fill it with different values. Print all values to the console using enhanced for loop.
2. The same as above, but array size should come from user.
3. The same as above, but values should also come from user.
4. Print sum of all of the values from your array.
5. *Create a multiplication table. Your application should write all values to the multidimentional array at first and then print its values.



Object-oriented programming

Class

A class is **the template** or **blueprint** from which **objects are made.**

Describes the **behavior/state** that **the object** of its **type** support.

Class declaration

Class declarations can include these components, in order:

1. **Modifiers** such as **public**, **private** (if any), and a number of others that you will encounter later.
2. The **class name**, with the initial letter capitalized by convention.
3. **The class body**, surrounded by braces, {}.

Class declaration

Example

```
class Bicycle {
    // class body
}
```

```
public class Bicycle {
    // class body
}
```

```
private class Bicycle {
    // class body
}
```

Object

Objects have **states** and **behaviors**.

Example: A dog has **states** - color, name, breed as well as behaviors – wagging the tail, barking, eating.

An object is an instance of a class.

Fields, methods, constructors packages and imports

Access modifiers

There are four **access controls** (levels of **access**) but only three **access modifiers**:

- **public** - visible to the world
- **protected** - visible to the package and all subclasses
- **default** - visible to the package
- **private** - visible to the class only

Variable scope

The scope of a variable is the **part of the code** in which you can **access it**.

There are **four basic** variable **scopes**:

- **static** – they are created when the class is loaded and they survive as long as the class stays loaded in the JVM
- **instance** – they are created when a new instance is created, and they live until the instance is removed
- **local** – they live as long as their method remains on the stack
- **block** – live only as long as the code block is executing

Fields

The Bicycle class uses the following lines of code to define its fields:

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
}
```

Fields

Field declarations are composed of **three components**, in order:

1. Zero or more **modifiers**.
2. The field's **type**.
3. The field's **name**.

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
}
```

Constructors

Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds **a default constructor for that class**.

Each time a **new object is created**, at least **one constructor** will be invoked. The main rule of constructors is that they should have **the same name as the class**. A class can have **more than one constructor**.

Example

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int cadence, int gear, int speed) {
        this.cadence = cadence;
        this.gear = gear;
        this.speed = speed;
    }

}
```

Instantiating a class

The **new** operator **instantiates a class** by allocating memory for a **new object** and returning **a reference to that memory**.

The new operator also invokes the object **constructor**.

„instantiating a class” means the same thing as **„creating an object”**.

Instantiating a class

Example

```
Bicycle bike = new Bicycle(75, 2, 20);
```

```
Cat garfield = new Cat("Gerfield");
```

```
Integer age = new Integer(34);
```

Methods

Methods are fundamental building **blocks** of Java programs.

Each Java method is **a collection of statements** that are **grouped together** to perform **an operation**.

Methods declaration

Method declarations have five **components**, in order:

1. **Modifier** – it defines the access type of the method and it is optional to use.
2. **Return type** − method may return a value.
3. **Method name**.
4. **Parameter list** in parenthesis – it is the type, order and number of parameters of a method.
5. **Method body** – defines what the method does with the statements.

Methods declaration

Example

```
public int sum(int a, int b) {
    // return a + b;
}
```

```
void draw(String s) {
    // perform some draw functions
}
```

```
private boolean isNew() {
    // return true or false according to some rules
}
```

Methods calling

Example

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int cadence, int gear, int speed) {
        this.cadence = cadence;
        this.gear = gear;
        this.speed = speed;
    }

    public int getCadence() {
        return cadence;
    }

}
```

Methods calling

Example

```
Bicycle bike = new Bicycle(75, 2, 20);
bike.getCadence(); // should return 75
System.out.println(bike.getCadence()); // should print 75

int cadence = bike.getCadence();
System.out.println("Cadence is: " + cadence);
```

Exercises – object-oriented programming

1. Create a class called „Dog”.
    a) Create a declaration for all required fields, like „name”, „age”, ...
    b) Create default constructor that will be able to set instantiate those values.
    c) Create a method, that will print default sound that dog can emit.
    d) *Sound should depend on the dogs age.
    e) Print that sound to the console. Change dogs age, print it once more.
2. *Create a class called „Room”.
    a) Create a declaration for all required fields, like „width”, „heigth”, ...
    b) Create default constructor that will be able to set instantiate those values.
    c) Create methods, that will calculate rooms surface area and volume.
    d) Print calulated values to the console.

Packages

Packages are used in Java in order:

- to **prevent** naming **conflicts,**
- **to control access,**
- to make **searching/locating and usage of classes**, **interfaces**, **enumerations** and **annotations easier**, etc.

Packages creating

Example

```
package vehicle;

public class Bicycle {
    // class body
}
```

```
package ro.sdacademy.animals.mammals;

public class Cat {
    // class body
}
```

Imports

If you want to use **a class from a package**, you can refer to it by its **full name** (package name plus class name). Classes from *java.lang* package are imported automatically.

For example, *java.util.Scanner* refers to the *Scanner* class in the *java.util* package:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Example

You can import a name with an *import* statement:

```
import java.util.Scanner;
```

or import all classes from the *java.util* package

```
import java.util.*;
```

and then you can write:

```
Scanner in = new Scanner(System.in);
```

Static field, methods and imports

Static fields and methods

The keyword **static** indicates that the **particular member** belongs to **a type itself**, rather than **to an instance of that type**.

This means that **only one instance of that static** member **is created** which is shared **across all instances of the class**.

Static fields - declaration

Example

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;
    static int count = 0;

    public Bicycle(int cadence, int gear, int speed) {
        this.cadence = cadence;
        this.gear = gear;
        this.speed = speed;
        this.count++;
    }

}
```

Static fields - access

Example

```
Bicycle bike = new Bicycle(75, 2, 20);
System.out.println(Bicycle.count); // should print 1

Bicycle anotherBike = new Bicycle(80, 4, 25);
System.out.println(Bicycle.count); // should print 2

// should prints true in both cases
System.out.println(Bicycle.count == bike.count);
System.out.println(bike.count == anotherBike.count);
```

Static methods - declaration

Example

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;
    private static int count = 0;

    public Bicycle(int cadence, int gear, int speed) {
        // fields assignment omitted for brevity
        this.count++;
    }

    public static int getCount() {
        return count;
    }

}
```

Static methods - access

Example

```
Bicycle bike = new Bicycle(75, 2, 20);
System.out.println(Bicycle.getCount()); // should print 1

Bicycle anotherBike = new Bicycle(80, 4, 25);
System.out.println(Bicycle.getCount()); // should print 2

// should prints true in both cases
System.out.println(Bicycle.getCount() == bike.getCount());
System.out.println(bike.getCount() == anotherBike.getCount());
```

Command-line parameters

It is possible to **pass some information** into **a program** when run it. This is accomplished by passing command-line arguments to **main() method**.

A command-line **argument** is the information that directly **follows** the **program's name** on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are **stored as strings** in the **String array** passed to **main() method**.

Command-line parameters

It is possible to **pass some information** into **a program** when run it. This is accomplished by passing command-line arguments to **main() method**.

A command-line **argument** is the information that directly **follows** the **program's name** on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are **stored as strings** in the **String array** passed to **main() method**.

Command-line parameters

Example

```
public class CommandLine {

    public static void main(String[] args) {
        for (int i=0; i<args.length; i++) {
            System.out.println("args[" + i + "]: " + args[i]);
        }
    }

}
```

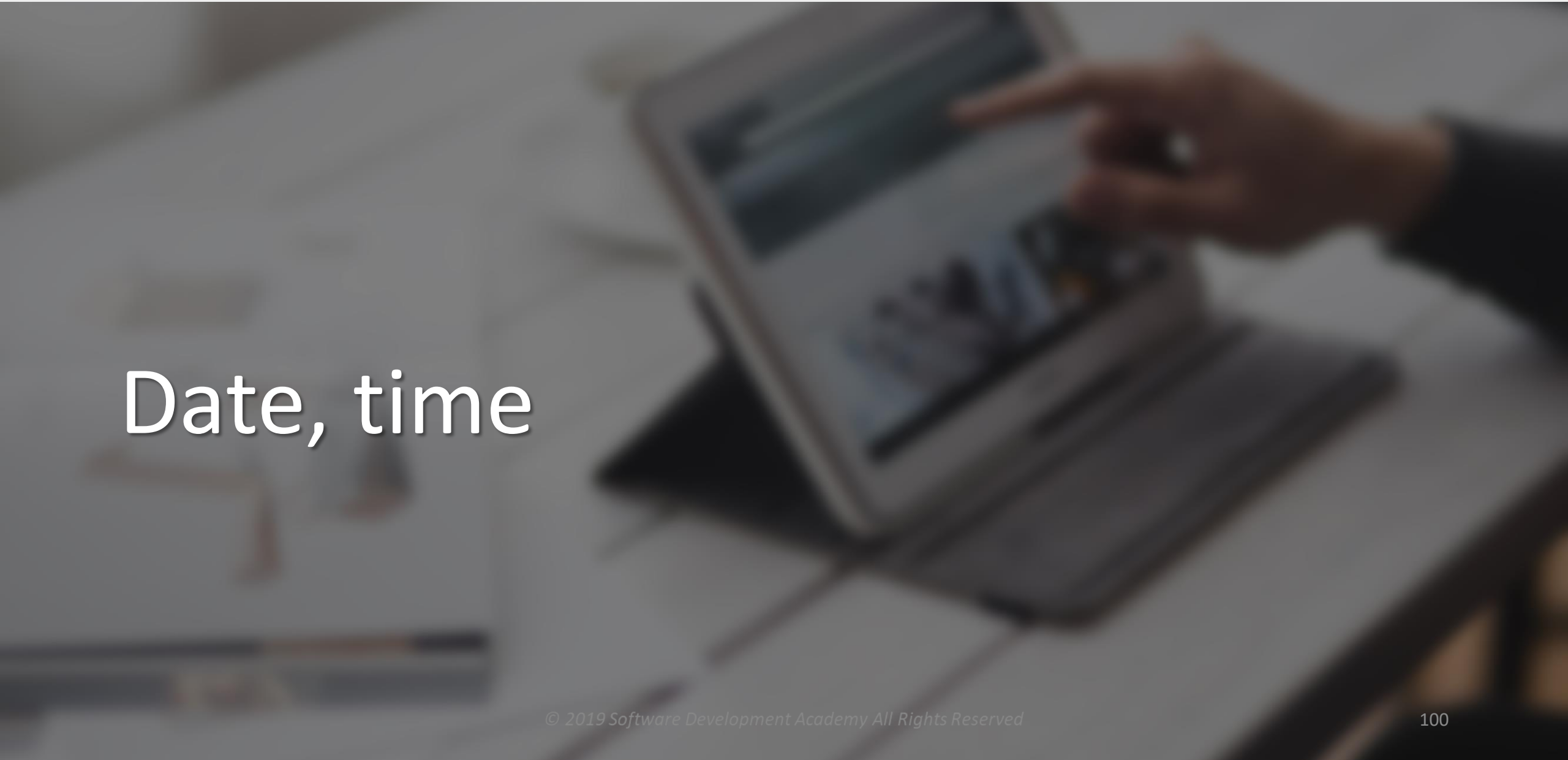
Varargs

Varargs

- **Varargs** allows the method to accept **zero or muliple arguments**.
- There can be **only one** variable **argument in a method**.
- **Variable** argument (**varargs**) must be **the last argument**.

Example

```
int sum(int... elements) {
    int result = 0;
    for (int i: elements) {
        result += i;
    }
    return result;
}

System.out.println(sum(1, 2, 3, 4)); // 10
System.out.println(sum(1));          // 1
System.out.println(sum());           // 0
```



Date, time

Date, time

There are **two basic ways** to represent **time**:

- represents time in **human terms/human time**, such as year, month, day, hour, minute and second,
- **machine time**, measures time continuously **along a timeline** from **an origin**, called **the epoch**, in **nanosecond** resolution.

Some classes in the Date-Time API are intended to represent **machine time**, and others are more suited to representing **human time.**

Date, time – legacy way

- *java.util.Date* - represents a specific **instance in time**, with **millisecond** precision.
- *java.util.Calendar* - is an **abstract class** that provides methods for **converting** between a specific **instant in time** and for **manipulating the calendar fields**. An instant in time can be represented by a **millisecond value** that is an offset **from the Epoch**, January 1, 1970 00:00:00.000 GMT (Gregorian).

Date, Time – legacy way

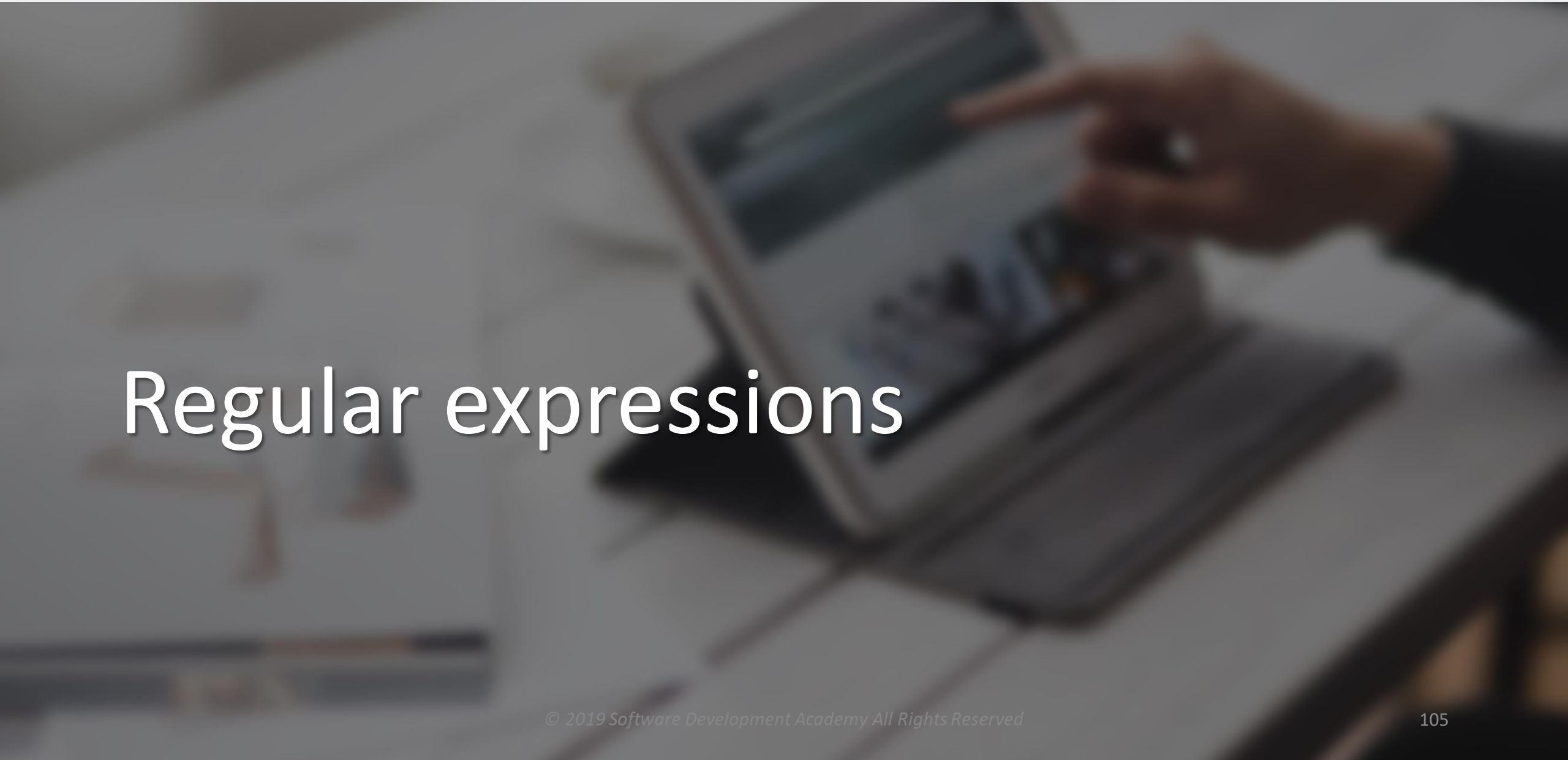
Example

```
Date now = new Date();
// or
long millis = System.currentTimeMillis();
Date now = new Date(millis);
System.out.println(now); // Wed Mar 13 21:38:09 CET 2019

Calendar cal = Calendar.getInstance();
Date date = cal.getTime(); // convert Calendar to Date
System.out.println(date);  // Wed Mar 13 21:38:09 CET 2019

cal.setTime(now);  // convert Date to Calendar
System.out.println(cal.get(Calendar.YEAR));          // 2019
System.out.println(cal.get(Calendar.DAY_OF_YEAR));   // 72
System.out.println(cal.get(Calendar.WEEK_OF_YEAR));  // 11
```

Java.time.LocalDateTime

Example

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30);
LocalDateTime.parse("2015-02-20T06:30:00");
now.plusDays(1);
now.minusHours(2);
now.getMonth();
```



Regular expressions

Regular expressions

- A **regular expression** defines a **search pattern** for **strings**.
- The search pattern can be anything from **a simple character**, a **fixed string** or a **complex expression** containing special characters describing the pattern.
- **The pattern** defined by the regex **may match one** or **several** times or **not at all** for **a given string**.
- Regular expressions can be used to **search**, **edit** and **manipulate text**.

Java Regex API

Java Regex API provides an **interface** and **three classes** in *java.util.regex* package:

- ***MatchResult*** interface
- ***Matcher*** class
- ***Pattern*** class
- ***PatternSyntaxException*** class

Matcher

It implements *MatchResult* interface. Is **the engine** that **interprets the pattern** and performs **match operations** against **an input string**.

Pattern

Is a **compiled representation** of
a regular expression.

Pattern and matcher

Example

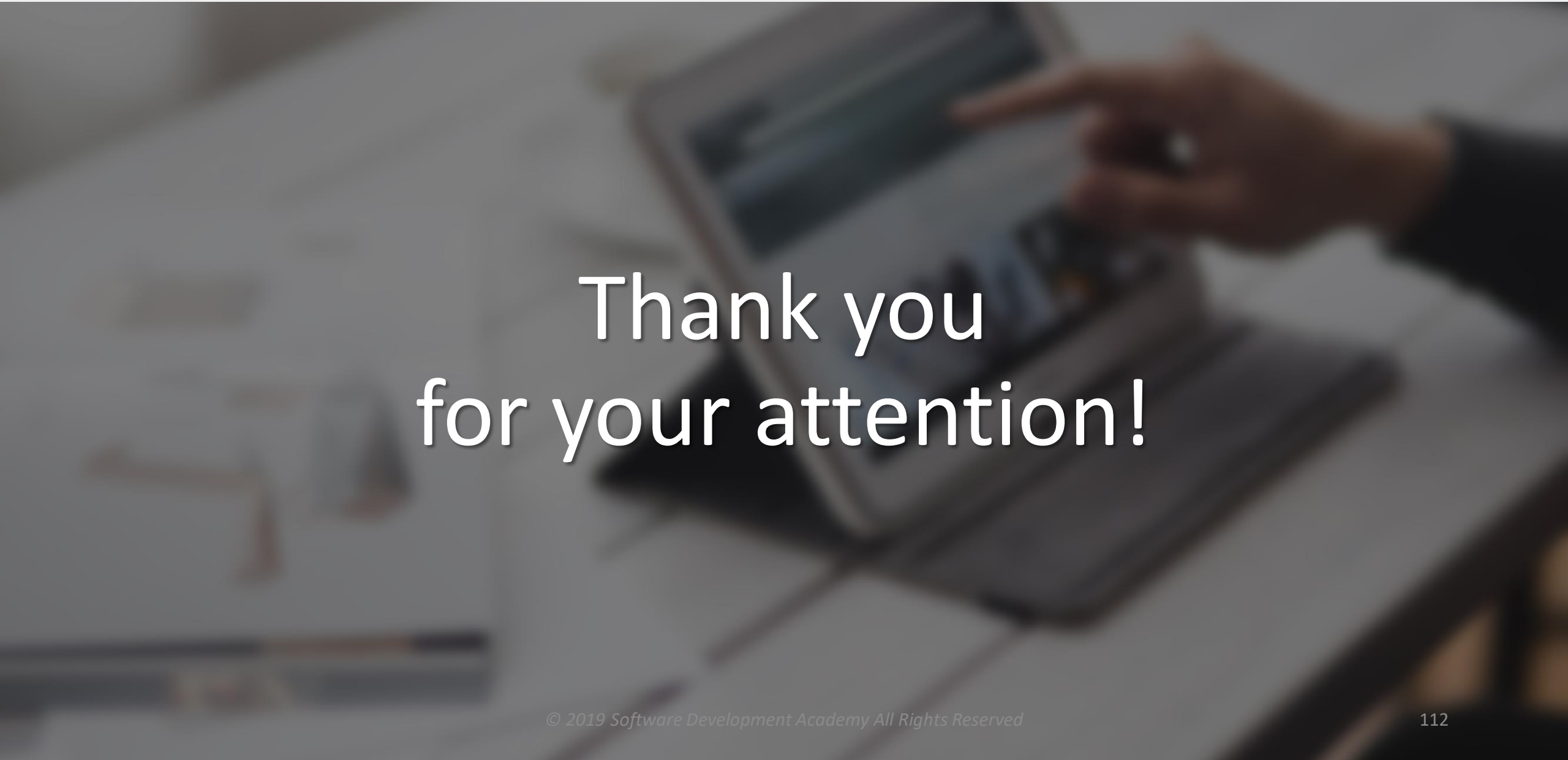
```
System.out.println(Pattern.matches(".s", "as"));     // true
System.out.println(Pattern.matches(".t", "dt"));     // true
System.out.println(Pattern.matches(".d", "odt"));    // false
System.out.println(Pattern.matches(".d", "oodt"));   // false
System.out.println(Pattern.matches("..t", "odt"));   // true

Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
System.out.println(m.matches()); // true
```

Exercises – regular expressions

Open www.regex101.com site. Supply „Test String” with text defined below (or generate it randomly by yourself here):

Anthony B. Carpenter, mob. 618-439-3833, AnthonyBCarpenter@rhyta.com, 821 Butternut Lane Benton, IL 62812

Fill Regular Expression to catch every single data into different group (e.g. group(1) should consist of name, group(2) – surname, group(3) phone number, etc.). Use Quick Reference to find out how to catch individual chars.



Thank you for your attention!