

# Java Introduction

Basic Syntax , I/O, Conditions, Loops and Debugging



SoftUni Team  
Technical Trainers



Software University  
<https://softuni.bg>

Have a Question?



**sli.do**

**#fund-java**

# Table of Contents

1. Introduction and Basic Syntax
2. Comparison Operators
3. The if-else / switch-case Statement
4. Logical Operators
5. Loops
6. Debugging and Troubleshooting





# Introduction and Basic Syntax

# Java – Introduction

- **Java** is modern, flexible, general-purpose programming language
- **Object-oriented** by nature, statically-typed, compiled



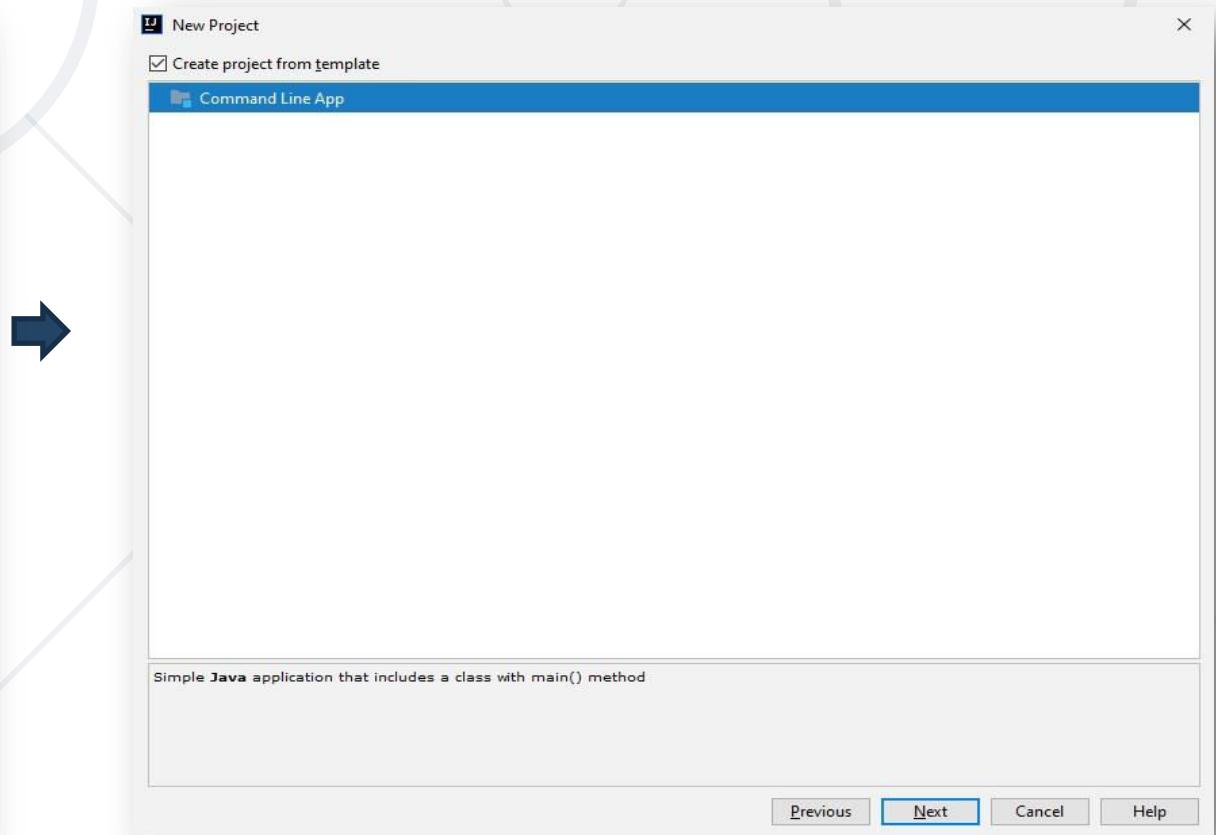
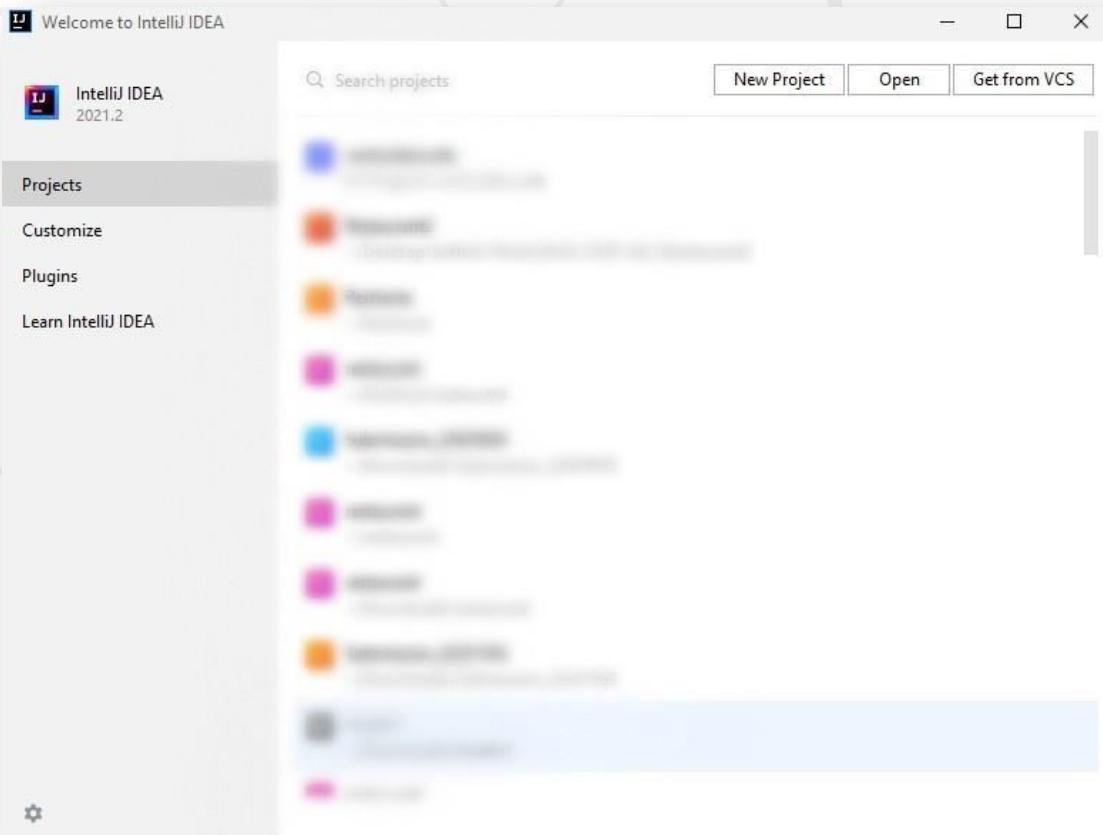
```
static void main(String[] args) {  
    //Source Code  
}
```

Program  
starting  
point

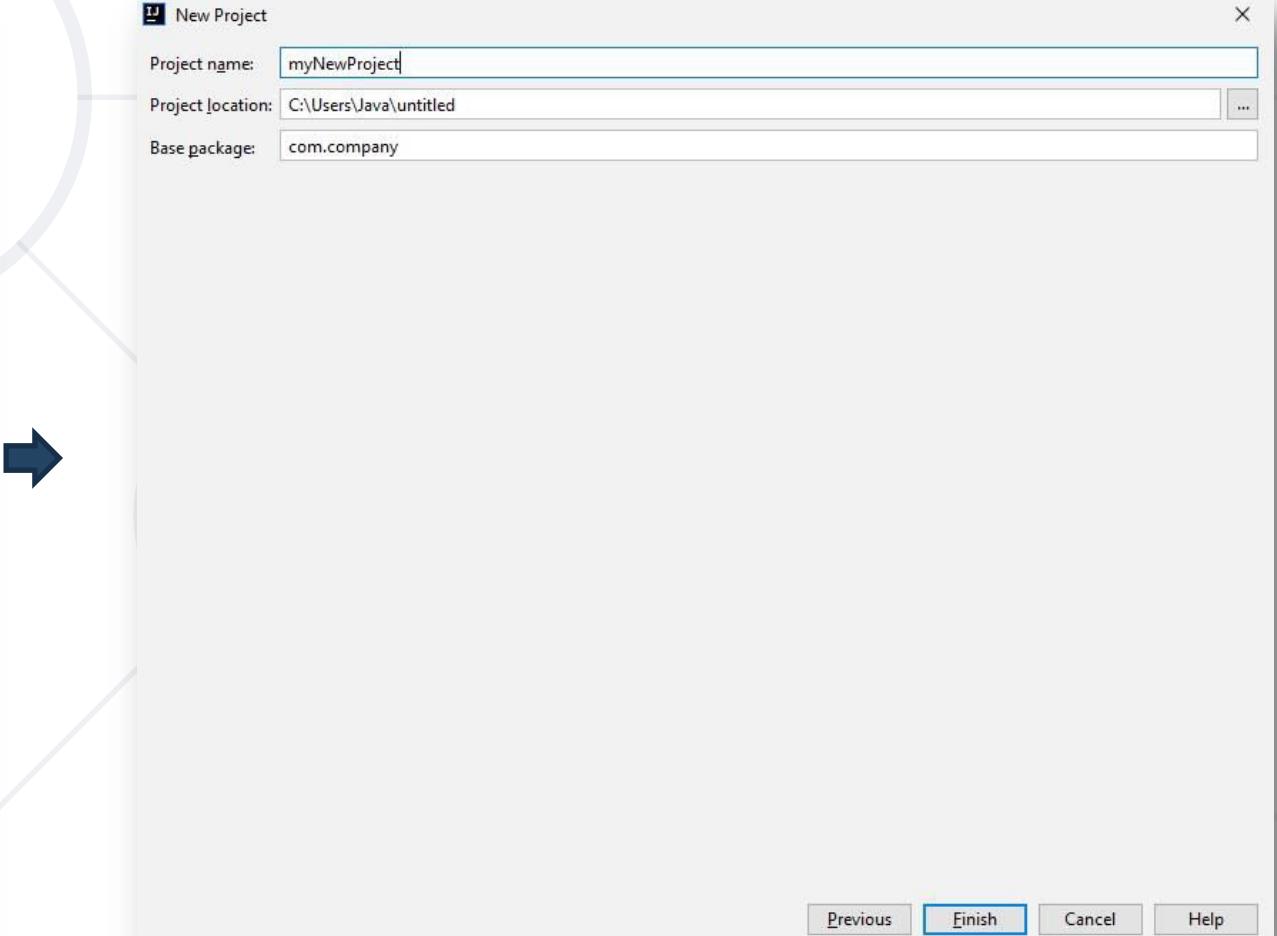
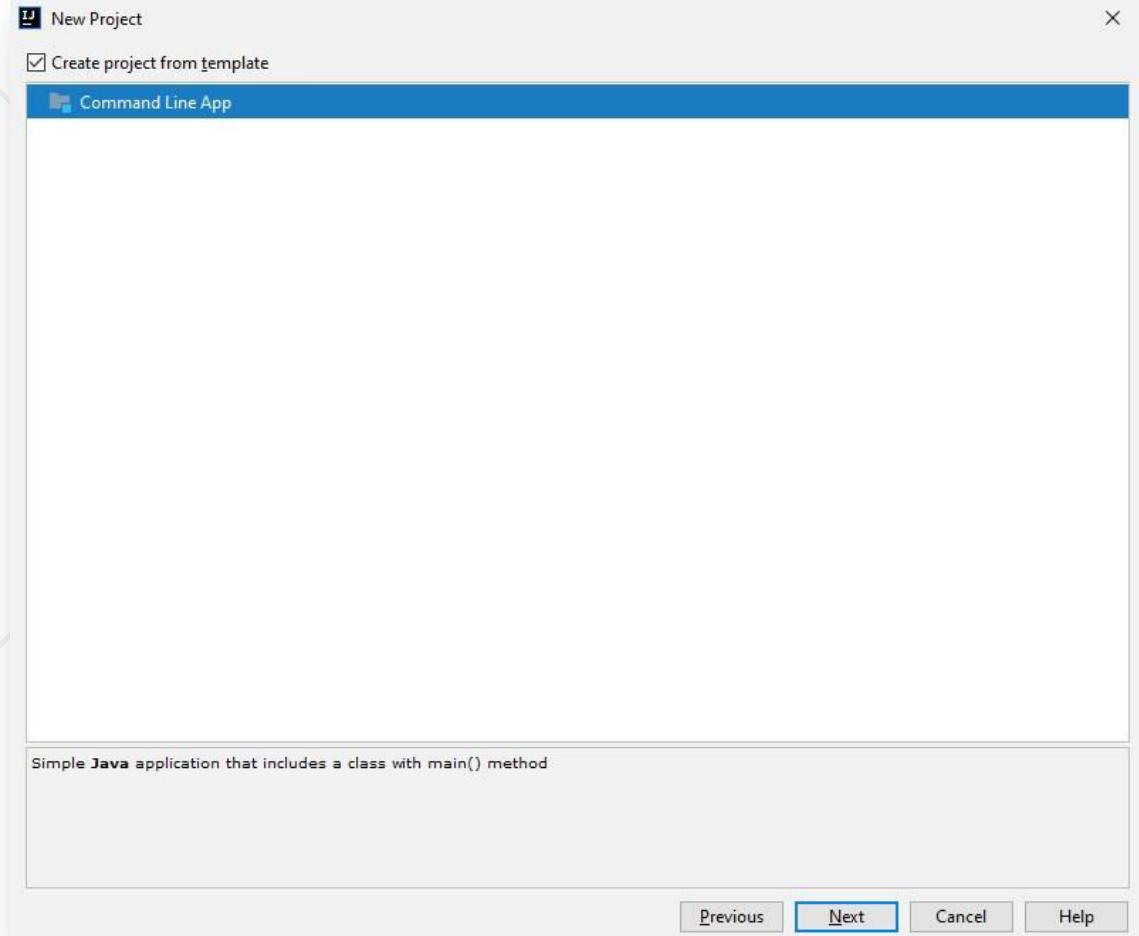
- In this course will use Java Development Kit (JDK) 13

# Using IntelliJ Idea

- **IntelliJ Idea** is powerful IDE for Java and other languages
- Create a project



# Using IntelliJ Idea



# Declaring Variables

- Defining and Initializing variables

```
{data type / var} {variable name} = {value};
```

- Example:

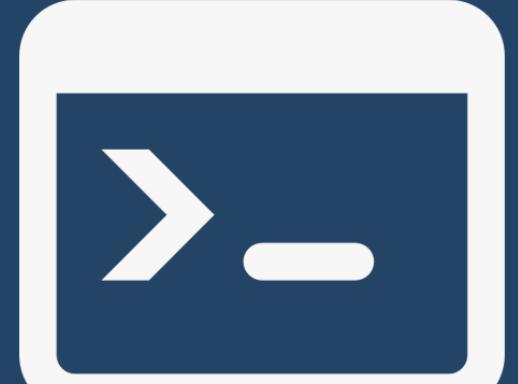
```
int number = 5;
```

Variable name

Data type

Variable value





# Console I/O

Reading from and Writing to the Console

# Reading from the Console

- We can **read/write** to the console, using the **Scanner** class
- Import the **java.util.Scanner** class

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



- Reading input from the console using

```
String name = sc.nextLine();
```

Returns **string**

# Converting Input from the Console

- `scanner.nextLine()` returns a **String**
- Convert the string to number by **parsing**:

```
import java.util.Scanner;  
...  
Scanner sc = new Scanner(System.in);  
String name = sc.nextLine();  
int age = Integer.parseInt(sc.nextLine());  
double salary = Double.parseDouble(sc.nextLine());
```



# Printing to the Console

- We can **print** to the console, using the **System** class
- Writing output to the console:
  - **System.out.print()**
  - **System.out.println()**

```
System.out.print("Name: ");

String name = scanner.nextLine();

System.out.println("Hi, " + name);

// Name: George
// Hi, George
```

# Using Print Format

- Using **format** to print at the console
- Examples:

```
String name = "George";  
int age = 5;  
System.out.printf("Name: %s, Age: %d", name, age);  
// Name: George, Age: 5
```

Placeholder **%s** stands  
for string and  
corresponds to **name**

Placeholder **%d**  
stands for integer  
number and  
corresponds to **age**

# Formatting Numbers in Placeholders

- **D** – format number to certain digits with leading zeros
- **F** – format floating point number with certain digits after the decimal point
- Examples:

```
int percentage = 55;  
  
double grade = 5.5334;  
  
System.out.printf("%03d", percentage); // 055  
  
System.out.printf("%.2f", grade); // 5.53
```

# Using String.format

- Using **String.format** to create a string by pattern
- Examples:

```
String name = "George";  
  
int age = 5;  
  
String result = String.format("Name: %s,  
                             Age: %d", name, age);  
  
System.out.println(result);  
  
//Name: George, Age 5
```

# Problem: Student Information

- You will be given 3 input lines:
  - Student Name, Age and Average Grade
- Print the input in the following format:
  - "Name: {name}, Age: {age}, Grade {grade}"
  - Format the grade to 2 decimal places

John  
15  
5.40



Name: John, Age: 15, Grade: 5.40

Check your solution here: <https://judge.softuni.org/Contests/1190/>

# Solution: Student Information

```
import java.util.Scanner;  
...  
Scanner sc = new Scanner(System.in);  
String name = sc.nextLine();  
int age = Integer.parseInt(sc.nextLine());  
double grade = Double.parseDouble(sc.nextLine());  
  
System.out.printf("Name: %s, Age: %d, Grade: %.2f",  
                           name, age, grade);
```



# Comparison Operators

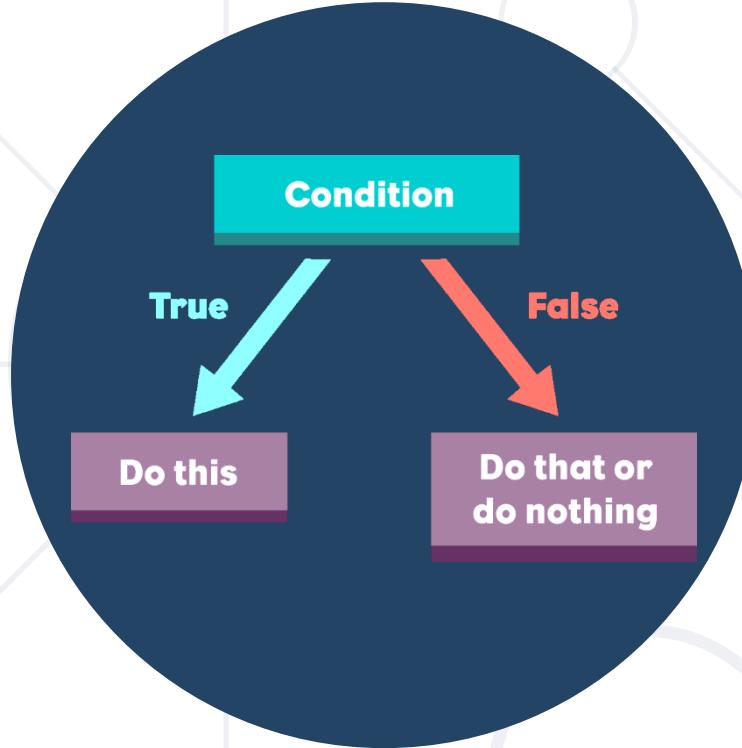
# Comparison Operators

Operator	Notation in Java
Equals	<code>==</code>
Not Equals	<code>!=</code>
Greater Than	<code>&gt;</code>
Greater Than or Equals	<code>&gt;=</code>
Less Than	<code>&lt;</code>
Less Than or Equals	<code>&lt;=</code>

# Comparing Numbers

- Values can be compared:

```
int a = 5;  
int b = 10;  
  
System.out.println(a < b);           // true  
System.out.println(a > 0);           // true  
System.out.println(a > 100);          // false  
System.out.println(a < a);            // false  
System.out.println(a <= 5);           // true  
System.out.println(b == 2 * a);        // true
```



# The If-else Statement

## Implementing Control-Flow Logic

# The If Statement

- The simplest conditional statement
  - Test for a condition
- Example: Take as an input a grade and check if the student has passed the exam ( $\text{grade} \geq 3.00$ )

```
double grade = Double.parseDouble(sc.nextLine());
if (grade >= 3.00) {
    System.out.println("Passed!");
}
```

In Java the opening bracket stays on the same line

# The If-else Statement

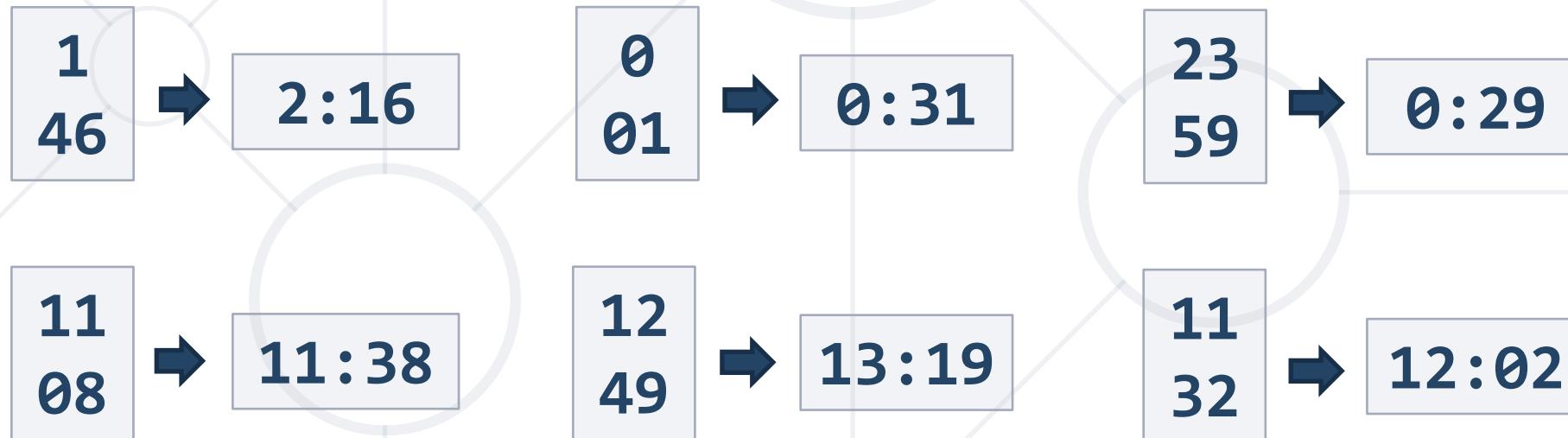
- Executes **one branch** if the condition is **true** and **another**, if it is **false**
- Example: **Upgrade** the last example, so it prints "**Failed!**", if the mark is lower than 3.00:

The **else** keyword stays on a new line

```
if (grade >= 3.00) {  
    System.out.println("Passed!");  
} else {  
    // TODO: Print the message  
}
```

# Problem: Back in 30 Minutes

- Write a program that reads hours and minutes from the console and calculates the time after 30 minutes
  - The hours and the minutes come on separate lines
- Example:



Check your solution here: <https://judge.softuni.org/Contests/1190/>

# Solution: Back in 30 Minutes (1)

```
int hours = Integer.parseInt(sc.nextLine());
int minutes = Integer.parseInt(sc.nextLine()) + 30;

if (minutes > 59) {
    hours += 1;
    minutes -= 60;
}

// Continue on the next slide
```

Check your solution here: <https://judge.softuni.org/Contests/1190/>

# Solution: Back in 30 Minutes (2)

```
if (hours > 23) {  
    hours = 0;  
}  
if (minutes < 10) {  
    System.out.printf("%d:%02d%n", hours, minutes);  
} else {  
    System.out.printf("%d:%d", hours, minutes);  
}
```

%n goes on  
the next line



# The Switch-Case Statement

Simplified If-else-if-else

# The Switch-case Statement

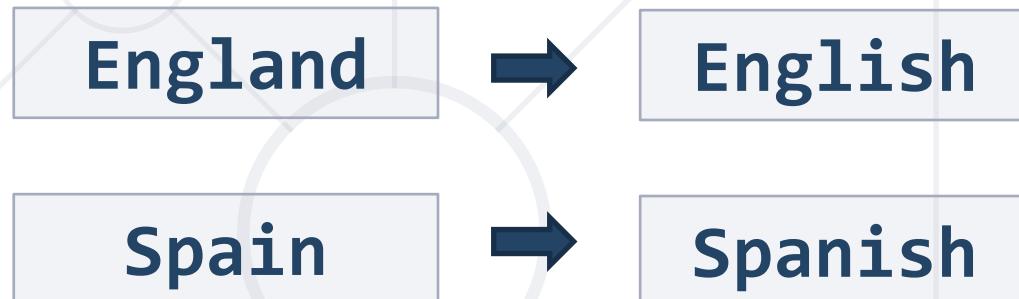
- Works as sequence of **if-else** statements
- Example: read input a number and print its corresponding month:

```
int month = Integer.parseInt(sc.nextLine());  
  
switch (month) {  
  
    case 1: System.out.println("January"); break;  
    case 2: System.out.println("February"); break;  
    // TODO: Add the other cases  
    default: System.out.println("Error!"); break;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/1190/>

# Problem: Foreign Languages

- By given country print its typical language:
  - English -> England, USA
  - Spanish -> Spain, Argentina, Mexico
  - other -> unknown



Check your solution here: <https://judge.softuni.org/Contests/1190/>

# Solution: Foreign Languages

```
//TODO: Read the input  
  
switch (country) {  
    case "USA":  
    case "England": System.out.println("English"); break;  
    case "Spain":  
    case "Argentina":  
    case "Mexico": System.out.println("Spanish"); break;  
    default: System.out.println("unknown"); break;  
}
```



**&&**

## Logical Operators

Writing More Complex Conditions

# Logical Operators

- Logical operators give us the ability to write multiple conditions in one **if** statement
- They return a boolean value and compare boolean values

Operator	Notation in Java	Example
Logical NOT	!	<code>!false -&gt; true</code>
Logical AND	<code>&amp;&amp;</code>	<code>true &amp;&amp; false -&gt; false</code>
Logical OR	<code>  </code>	<code>true    false -&gt; true</code>

# Problem: Theatre Promotions

- A theatre has the following ticket prices according to the age of the visitor and the type of day. If the age is  $< 0$  or  $> 122$ , print "Error!":

Day / Age	$0 \leqslant \text{age} \leqslant 18$	$18 < \text{age} \leqslant 64$	$64 < \text{age} \leqslant 122$
Weekday	12\$	18\$	12\$
Weekend	15\$	20\$	15\$
Holiday	5\$	12\$	10\$

**Weekday 42** → **18\$**

**Holiday -12** → **Error!**

Check your solution here: <https://judge.softuni.org/Contests/1190/>

# Solution: Theatre Promotions (1)

```
String day = sc.nextLine().toLowerCase();
int age = Integer.parseInt(sc.nextLine());
int price = 0;
if (day.equals("weekday")) {
    if ((age >= 0 && age <= 18) || (age > 64 && age <= 122)) {
        price = 12;
    }
    // TODO: Add else statement for the other group
}
// Continue...
```

# Solution: Theatre Promotions (2)

```
else if (day.equals("weekend")) {  
    if ((age >= 0 && age <= 18) || (age > 64 && age <= 122)) {  
        price = 15;  
    } else if (age > 18 && age <= 64) {  
        price = 20;  
    }  
} // Continue...
```

# Solution: Theatre Promotions (3)

```
else if (day.equals("holiday")){
    if (age >= 0 && age <= 18)
        price = 5;
    // TODO: Add the statements for the other cases
}
if (price != 0)
    System.out.println(price + "$");
else
    System.out.println("Error!");
```



# Loops

## Code Block Repetition

# Loop: Definition

- A **loop** is a control statement that repeats the execution of a block of statements. The loop can:
  - **for** loop
    - Execute a code block a fixed number of times
  - **while** and **do...while**
    - Execute a code block while a given condition returns true





## For-Loops

Managing the Count of the Iteration

# For-Loops

- The for loop executes statements a fixed number of times:

Initial value

End value

Increment

Loop body

```
for (int i = 1; i <= 10; i++) {  
    System.out.println("i = " + i);  
}
```

Executed  
at each  
iteration

The bracket is  
again on the  
same line

# Example: Divisible by 3

- Print the numbers from 1 to 100, that are divisible by 3

```
for (int i = 3; i <= 100; i += 3) {  
    System.out.println(i);  
}
```



- You can use "fori" live template in IntelliJ



```
for (int i = 0; i < ; i++) {  
}
```

# Problem: Sum of Odd Numbers

- Write a program to print the first **n** odd numbers and their sum

Check your solution here: <https://judge.softuni.org/Contests/1190/>

# Solution: Sum of Odd Numbers

```
int n = Integer.parseInt(sc.nextLine());  
  
int sum = 0;  
  
for (int i = 1; i <= n; i++) {  
    System.out.println(2 * i - 1);  
    sum += 2 * i - 1;  
}  
  
System.out.printf("Sum: %d", sum);
```

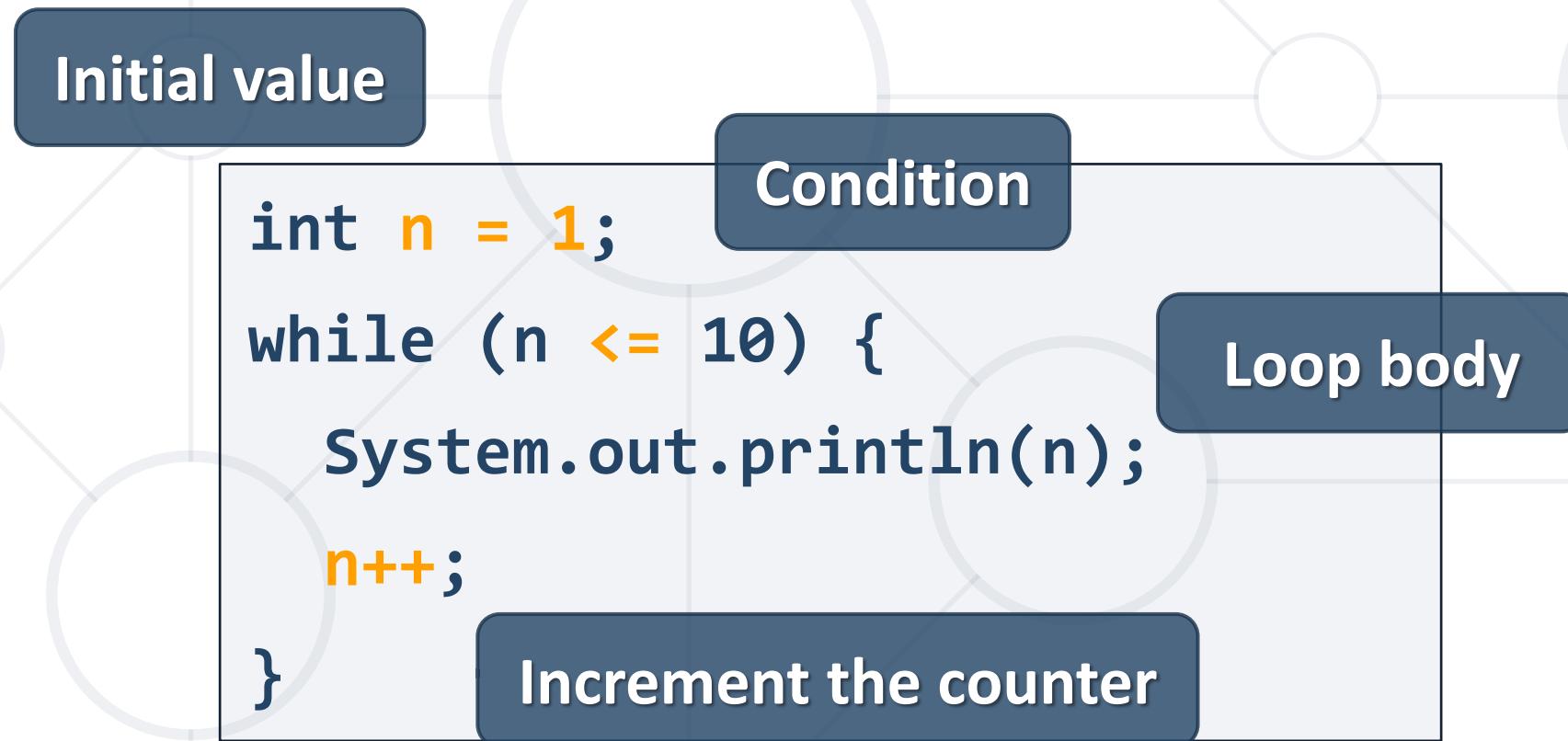


# While Loops

Iterations While a Condition is True

# While Loops

- Executes commands while the condition is true:

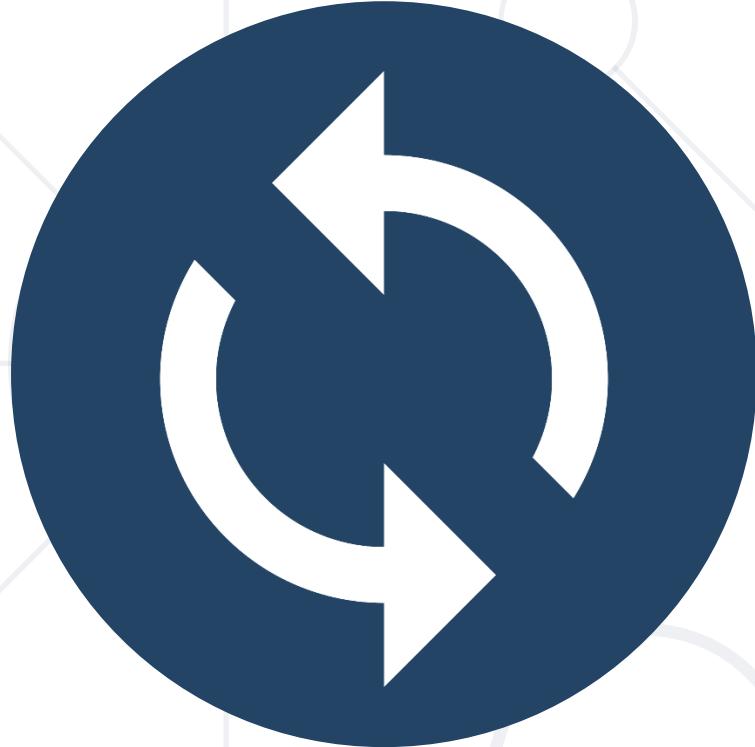


# Problem: Multiplication Table

- Print a table holding number\*1, number\*2, ..., number\*10

```
int number = Integer.parseInt(sc.nextLine());
int times = 1;
while (times <= 10) {
    System.out.printf("%d x %d = %d%n",
                      number, times, number * times);
    times++;
}
```

Check your solution here: <https://judge.softuni.org/Contests/1190/>

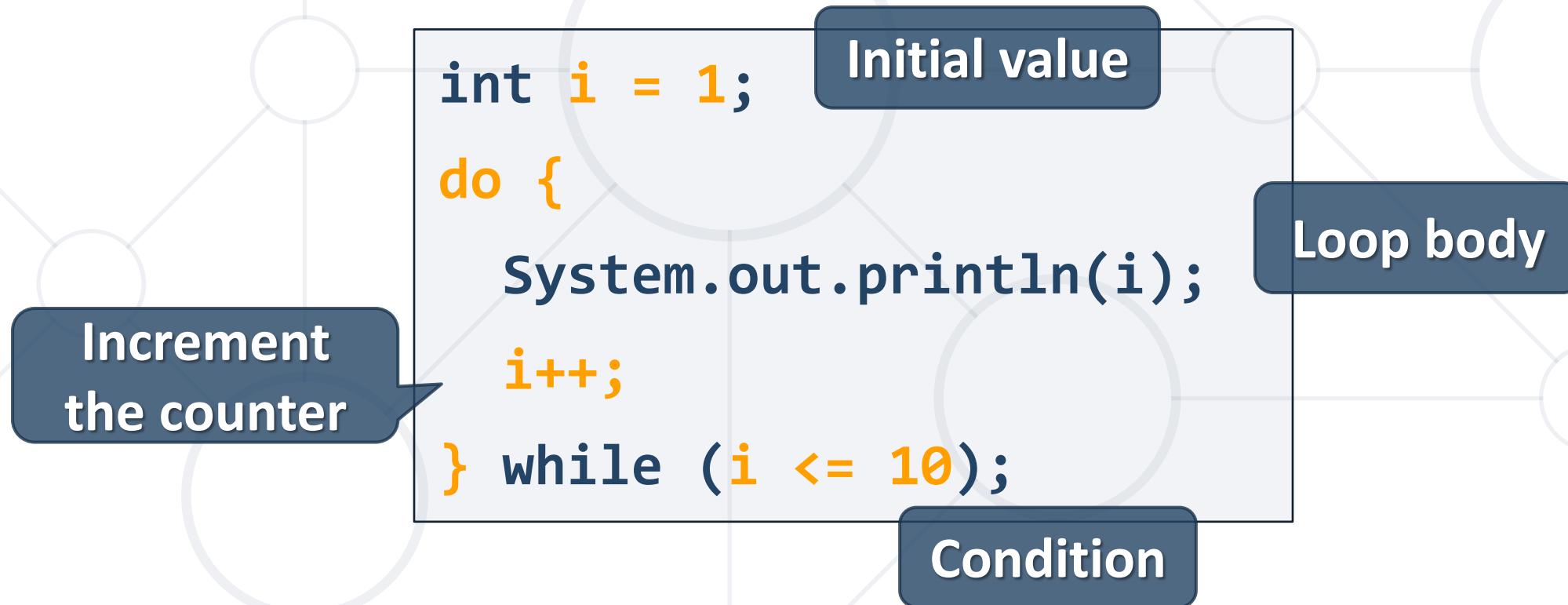


## Do...While Loop

Execute a Piece of Code One or More Times

# Do ... While Loop

- Similar to the **while** loop, but always executes at least once:



# Problem: Multiplication Table 2.0

- Upgrade your program and take the initial times from the console

```
int number = Integer.parseInt(sc.nextLine());
int times = Integer.parseInt(sc.nextLine());
do {
    System.out.printf("%d X %d = %d%n",
                      number, times, number * times);
    times++;
} while (times <= 10);
```



# Debugging the Code

## Using the IntelliJ Debugger

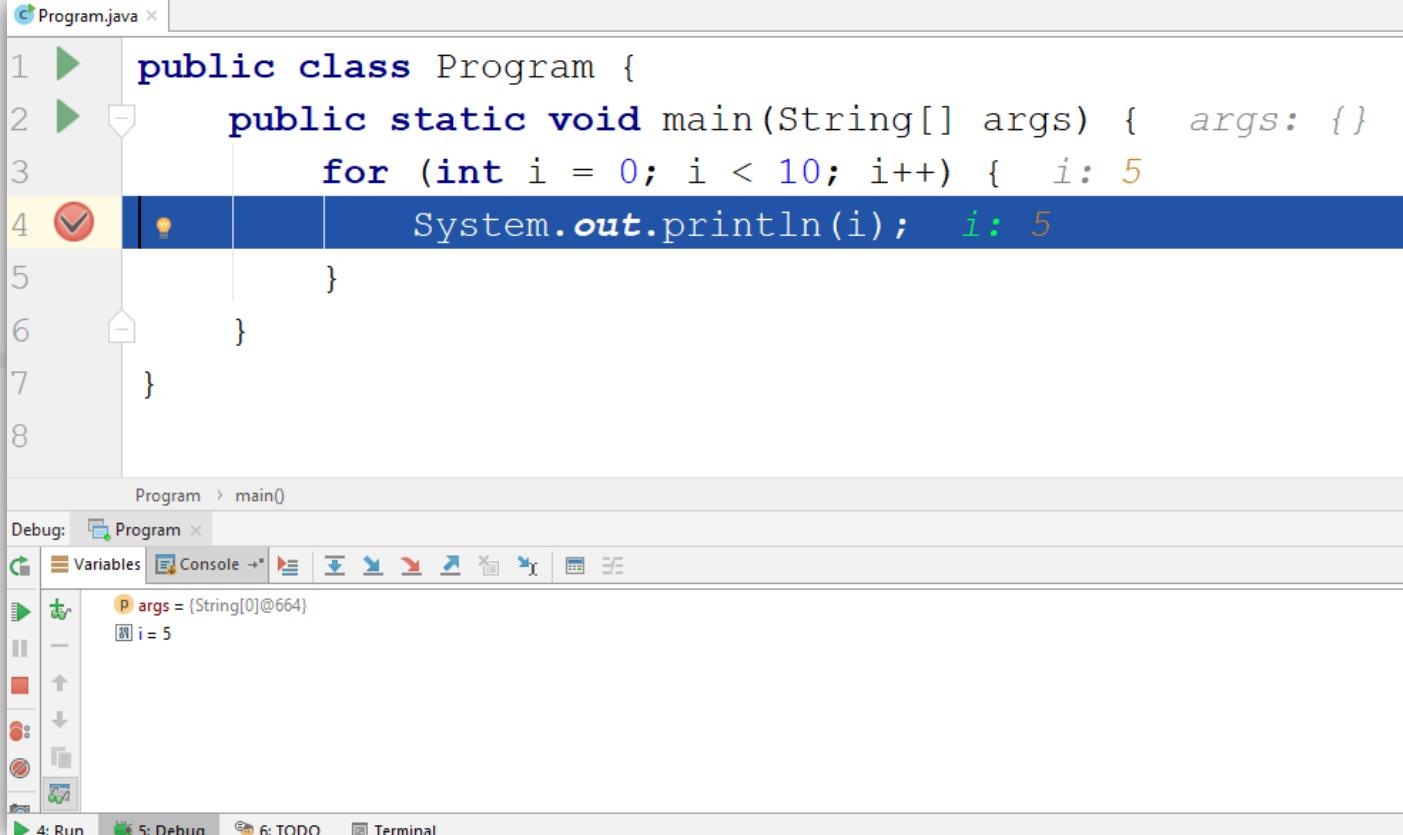
# Debugging the Code

- The process of **debugging application** includes:
  - Spotting an error
  - Finding the lines of code that cause the error
  - Fixing the error in the code
  - Testing to check if the error is gone and no new errors are introduced
- Iterative and continuous process



# Debugging in IntelliJ

- IntelliJ has a built-in **debugger**
- It provides:
  - **Breakpoints**
  - Ability to **trace** the code execution
  - Ability to **inspect** variables at runtime



```
1 public class Program {  
2     public static void main(String[] args) { args: {}  
3         for (int i = 0; i < 10; i++) { i: 5  
4             System.out.println(i); i: 5  
5         }  
6     }  
7 }  
8
```

Program > main()

Debug: Program

Variables

P args = [String[0]@664]  
I i = 5

Run, Debug, TODO, Terminal

# Using the Debugger in IntelliJ

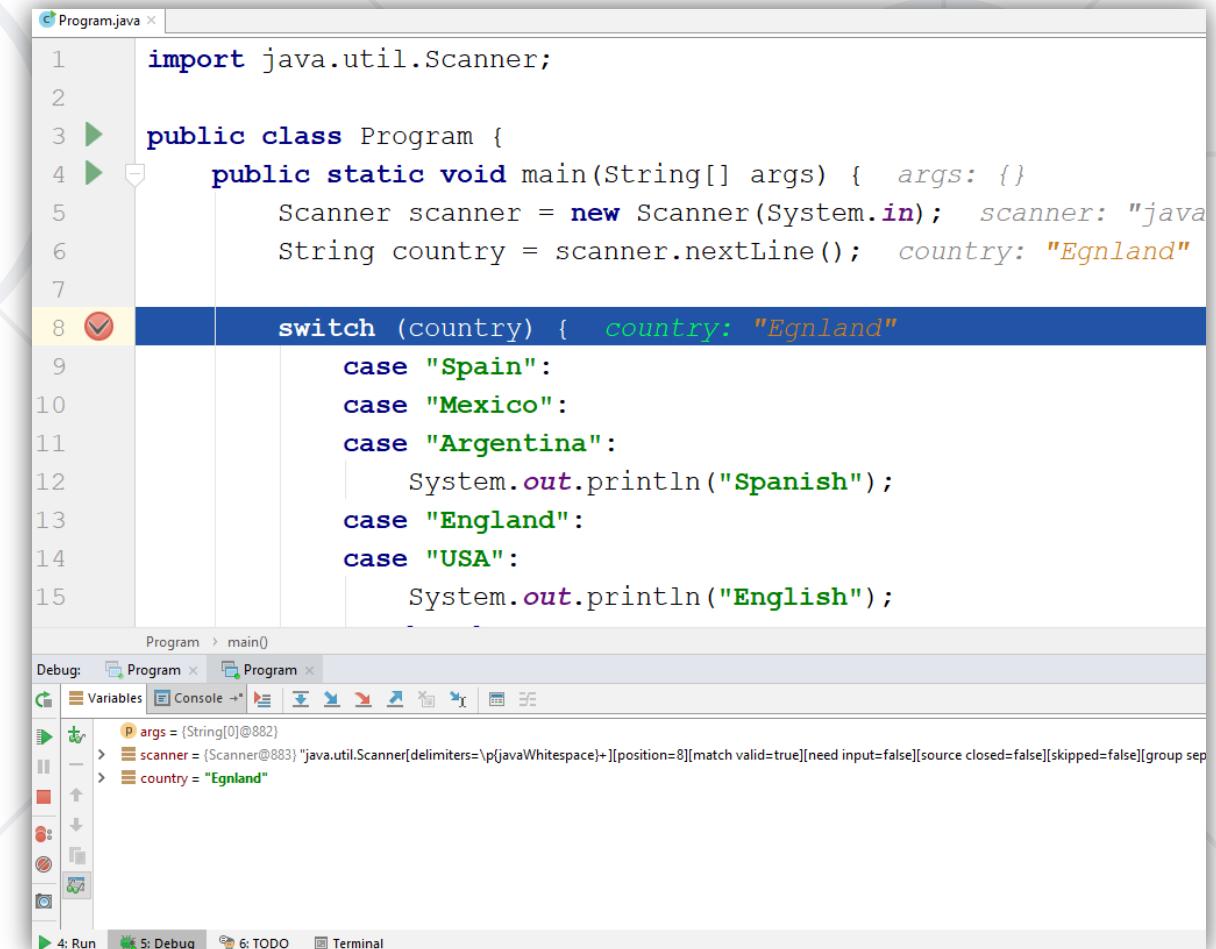
- Start without Debugger: **[Ctrl+Shift+F10]**

- Toggle a breakpoint: **[Ctrl+F8]**

- Start with the Debugger:  
**[Alt+Shift+F9]**

- Trace the program: **[F8]**

- Conditional breakpoints



The screenshot shows the IntelliJ IDEA interface with the following details:

- Code Editor:** The main window displays a Java file named "Program.java". The code defines a class "Program" with a main method that reads a country name from the user and prints it in Spanish or English based on the input.
- Breakpoints:** A conditional breakpoint is set on line 8, which is highlighted in blue. The condition for this breakpoint is "country: "Egnland"".
- Debug Tool Window:** Below the code editor, the "Debug" tool window is open, showing the current state of variables:
  - args = [String[0]@882]
  - scanner = {Scanner@883} "java.util.Scanner[delimiters=\p[javaWhitespace]+][position=8][match valid=true][need input=false][source closed=false][skipped=false][group sep=
  - country = "Egnland"
- Bottom Bar:** The navigation bar at the bottom includes tabs for Run, Debug, TODO, and Terminal.

# Problem: Find and Fix the Bugs in the Code

- A program aims to print the first **n** odd numbers and their sum

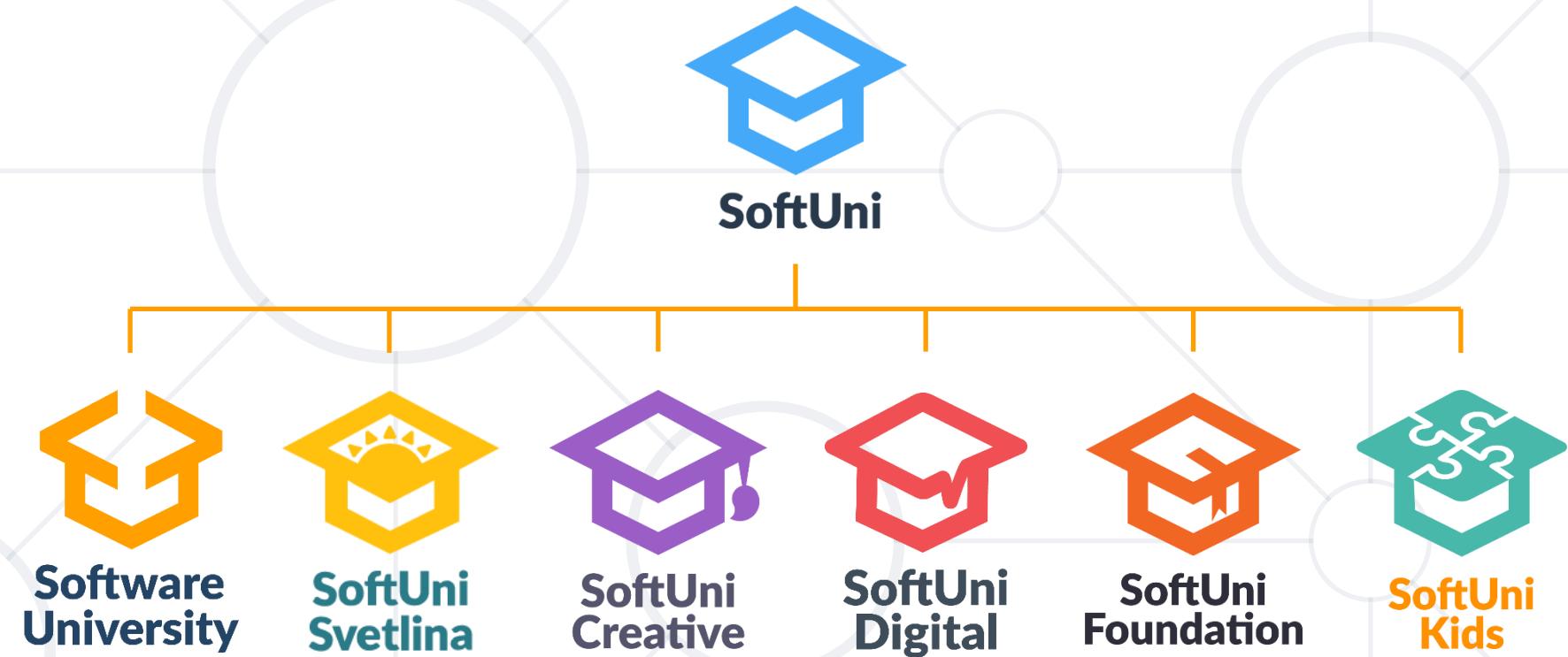
```
Scanner sc = new Scanner(System.in);
int n = Integer.parseInt(sc.nextLine());
int sum = 1;
for (int i = 0; i <= n; i++) {
    System.out.print(2 * i + 1);
    sum += 2 * i;
}
System.out.printf("Sum: %d%n", sum);
```

10

- Declaring **Variables**
- **Reading** from / **Printing** to the **Console**
- **Conditional Statements** allow implementing programming logic
- **Loops** repeat code block multiple times
- Using the debugger



# Questions?



# SoftUni Diamond Partners

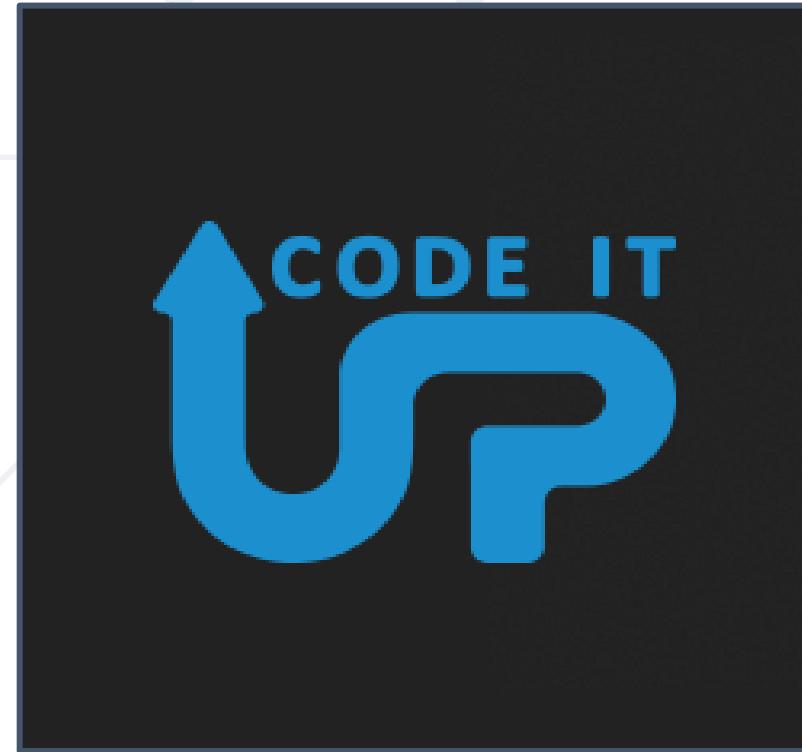


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



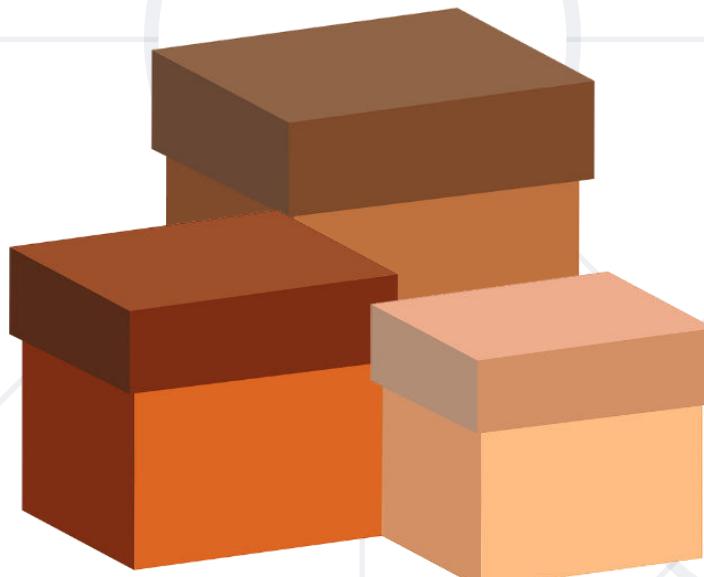
- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Data Types and Variables

Numerical Types, Text Types and Type Conversion

SoftUni Team  
Technical Trainers



SoftUni



Software University  
<https://softuni.bg>

Have a Question?



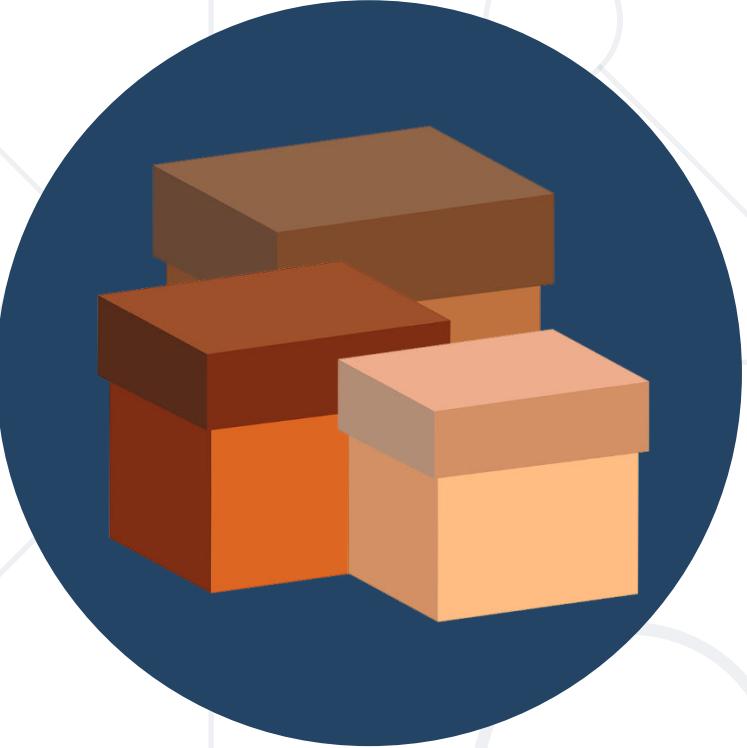
**sli.do**

**#fund-java**

# Table of Contents

1. Data Types and Variables
2. Integer and Real Number Type
3. Type Conversion
4. Boolean Type
5. Character and String Type

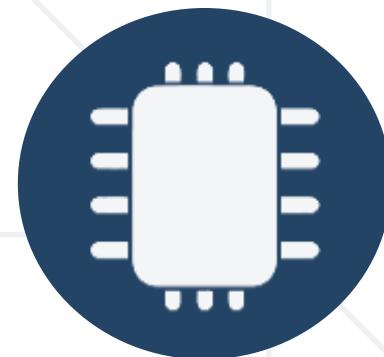
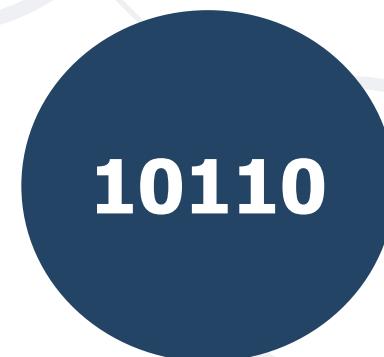




# Data Types

# How Computing Works?

- Computers are machines that process data
  - Instructions and data are stored in the computer memory



# Variables

- Variables have name, data type and value
  - Assignment is done by the operator "="
  - Example of variable definition and assignment



Variable name

Data type

```
int count = 5;
```

Variable value

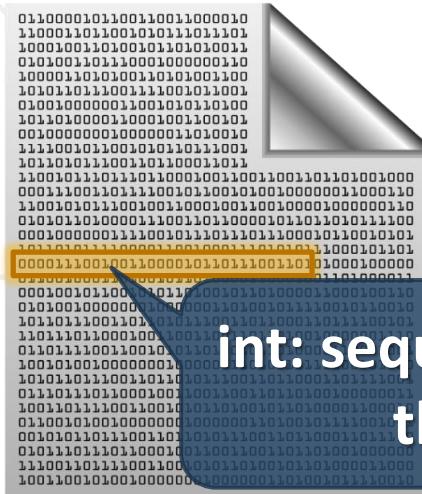
- When processed, data is stored back into variables

# What is a Data Type?

- A **data type**
  - Is a **domain of values** of similar characteristics
  - Defines the type of information stored in the computer memory (in a **variable**)
- Examples:
  - Positive integers: **1, 2, 3, ...**
  - Alphabetical characters: **a, b, c, ...**
  - Days of week: **Monday, Tuesday, ...**

# Data Type Characteristics

- A data type has:
    - Name (Java keyword)
    - Size (how much memory is used)
    - Default value
  - Example:
    - Name: int
    - Size: 32 bits (4 bytes)
    - Default value: 0



**int: sequence of 32 bits in  
the memory**

**int: 4 sequential bytes in  
the memory**



# Naming Variables

- Always refer to the naming **conventions** of a programming language
  - **camelCase** is used in Java
- Preferred form: **[Noun]** or **[Adjective] + [Noun]**
- Should explain the purpose of the variable  
(Always ask "**What does this variable contain?**")



`firstName, report, config, usersList, fontSize`



`foo, bar, p, p1, populate, LastName, last_name`

# Variable Scope and Lifetime

- **Scope** - where you can access a variable (global, local)
- **Lifetime** - how long a variable stays in memory

Accessible in the `main()`

```
String outer = "I'm inside the Main();  
for (int i = 0; i < 10; i++) {  
    String inner = "I'm inside the loop";  
}  
System.out.println(outer);  
// System.out.println(inner); Error
```

Accessible only in the loop

# Variable Span

- Variable span is how long before a variable is called
- Always declare a variable as late as possible (e.g. shorter span)

```
static void main(String[] args) {  
    String outer = "I'm inside the main()";  
    for (int i = 0; i < 10; i++)  
        String inner = "I'm inside the loop";  
        System.out.println(outer);  
        //System.out.println(inner); Error  
}
```

"outer"  
variable span

# Keep Variable Span Short

- Shorter span simplifies the code
  - Improves its **readability** and **maintainability**

```
for (int i = 0; i < 10; i++) {  
    String inner = "I'm inside the loop";  
}  
String outer = "I'm inside the main()"; }  
System.out.println(outer);  
// System.out.println(inner); Error
```

"outer" variable  
span – reduced



`int`

**Integer Types**

# Integer types

Type	Default Value	Min Value	Max Value	Size
<b>byte</b>	0	-128 ( $-2^7$ )	127 ( $2^7 - 1$ )	8 bit
<b>short</b>	0	-32768 ( $-2^{15}$ )	32767 ( $2^{15} - 1$ )	16 bit
<b>int</b>	0	-2147483648 ( $-2^{31}$ )	2147483647 ( $2^{31} - 1$ )	32 bit
<b>long</b>	0	-9223372036854775808 ( $-2^{63}$ )	9223372036854775807 ( $2^{63} - 1$ )	64 bit

# Centuries – Example

- Depending on the unit of measure we can use different data types

```
byte centuries = 20;  
short years = 2000;  
int days = 730484;  
long hours = 17531616;
```

```
System.out.printf("%d centuries = %d years = %d days = %d hours.",  
                  centuries, years, days, hours)  
//20 centuries = 2000 years = 730484 days = 17531616 hours.
```

# Beware of Integer Overflow!

- Integers have **range** (minimal and maximal value)
- Integers could overflow → this leads to incorrect values

```
byte counter = 0;  
for (int i = 0; i < 130; i++) {  
    counter++;  
    System.out.println(counter);  
}
```

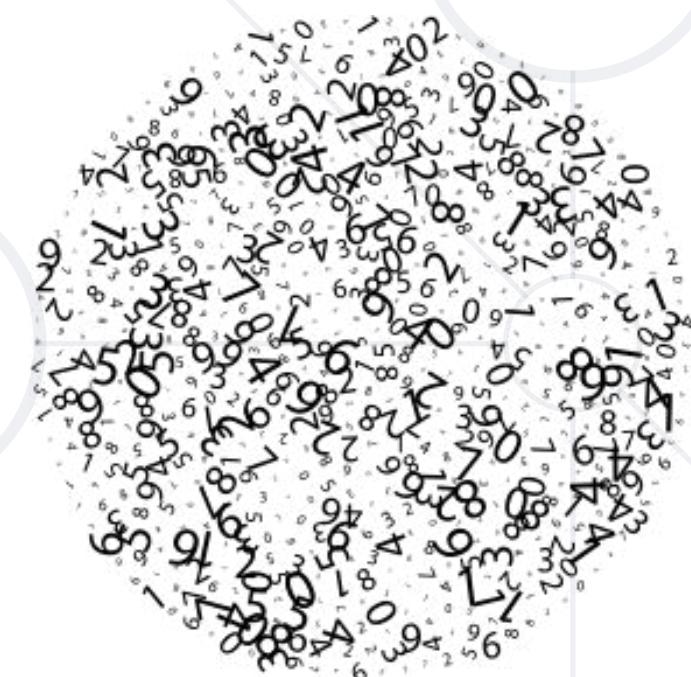


1
2
...
127
-128
-127

# Integer Literals

- Examples of integer literals:
  - The '**0x**' and '**0X**' prefixes mean a hexadecimal value
    - E.g. **0xFE**, **0xA8F1**, **0xFFFFFFFF**
  - The '**l**' and '**L**' suffixes mean a **long**
    - E.g. **9876543L**, **0L**

```
int hexa = 0xFFFFFFFF; // -1
long number = 1L;      // 1
```



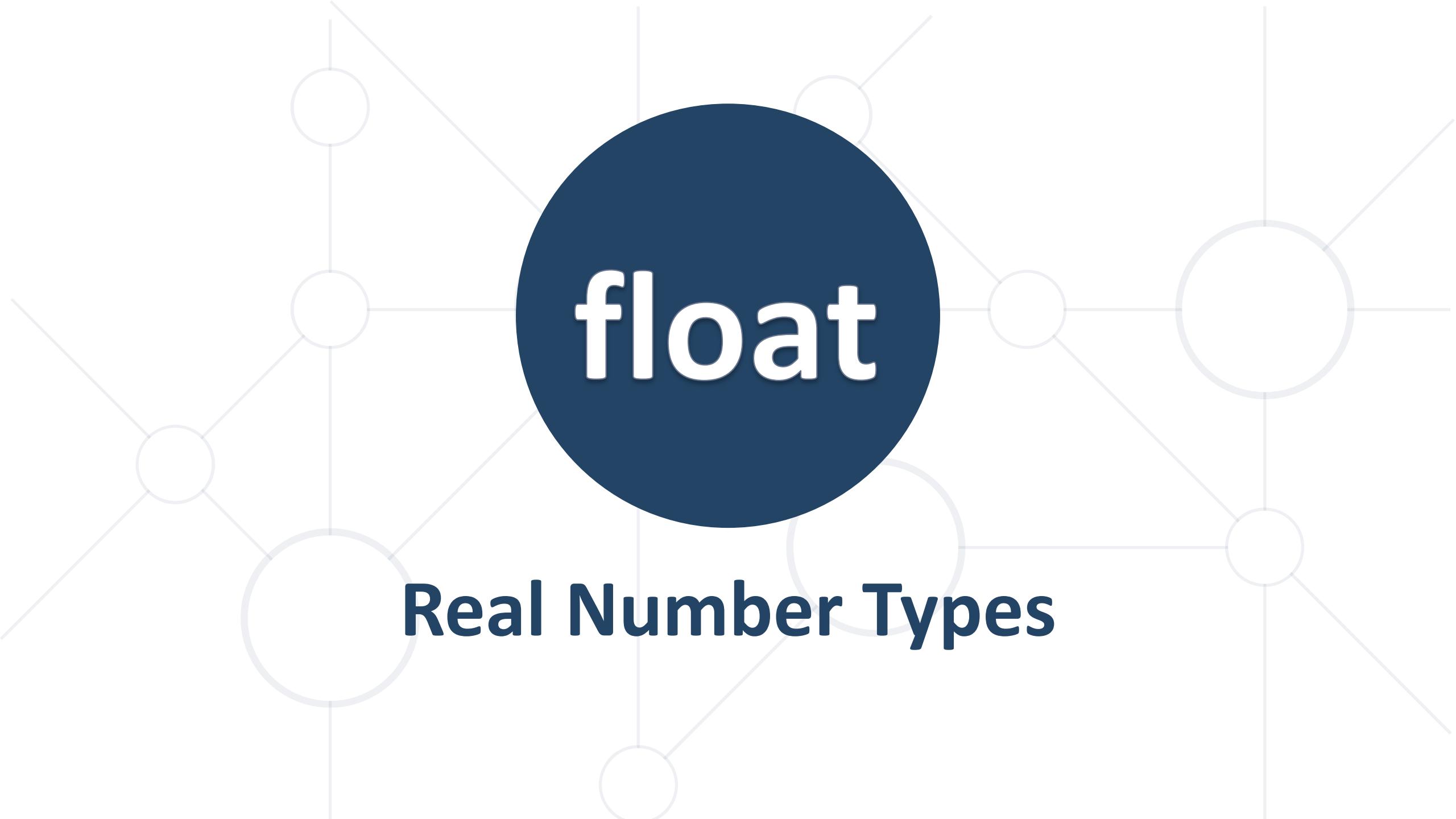
# Problem: Convert Meters to Kilometres

- Write a program that converts meters to kilometers formatted to the second decimal point
- Examples:  

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

int meters = Integer.parseInt(scanner.nextLine());
double kilometers = meters / 1000.0;
System.out.printf("%.2f", kilometers);
```

Check your solution here: <https://judge.softuni.org/Contests/1227/>



**float**

**Real Number Types**

# What Are Floating-Point Types?

- **Floating-point** types:
  - Represent real numbers, e.g. **1.25**, **-0.38**
  - Have range and precision depending on the memory used
  - Sometimes behave abnormally in the calculations
  - May hold very small and very big values like **0.0000000000001** and **10000000000000000000000000000000000.0**



# Floating-Point Numbers

- Floating-point types are:
  - **float** ( $\pm 1.5 \times 10^{-45}$  to  $\pm 3.4 \times 10^{38}$ )
    - 32-bits, the precision of 7 digits
  - **double** ( $\pm 5.0 \times 10^{-324}$  to  $\pm 1.7 \times 10^{308}$ )
    - 64-bits, the precision of 15-16 digits
- The default value of floating-point types:
  - Is **0.0F** for the **float** type
  - Is **0.0D** for the **double** type



# PI Precision – Example

- Difference in precision when using **float** and **double**:

```
float floatPI = 3.141592653589793238f;
```

3. 1415927

```
double doublePI = 3.141592653589793238;
```

```
System.out.println("Float PI is: " + floatPI);
```

```
System.out.println("Double PI is: " + doublePI);
```

3. 141592653589793

- NOTE: The "**f**" suffix in the first statement!

- Real numbers are by default interpreted as **double**
- One should explicitly convert them to **float**

# Problem: Pound to Dollars

- Write a program that converts British pounds to US dollars formatted to 3rd decimal point
- 1 British Pound = 1.36 Dollars

80



108.800

39



53.040

```
double num = Double.parseDouble(scanner.nextLine());
double result = num * 1.36;
System.out.printf("%.3f", result);
```

Check your solution here: <https://judge.softuni.org/Contests/1227/>

# Scientific Notation

- Floating-point numbers can use scientific notation, e.g.
    - **1e+34, 1E34, 20e-3, 1e-12, -6.02e28**

```
double d = 1000000000000000000000000000000000000000000000000000000000000000.0;
System.out.println(d); // 1.0E34
double d2 = 20e-3;
System.out.println(d2); // 0.02
double d3 = Double.MAX_VALUE;
System.out.println(d3); //1.7976931348623157E308
```

# Floating-Point Division

- Integral division and floating-point division are different:

```
System.out.println(10 / 4);      // 2 (integral division)
System.out.println(10 / 4.0);    // 2.5 (real division)
System.out.println(10 / 0.0);    // Infinity
System.out.println(-10 / 0.0);   // -Infinity
System.out.println(0 / 0.0);     // NaN (not a number)
System.out.println(8 % 2.5);    // 0.5 (3 * 2.5 + 0.5 = 8)
System.out.println(10 / 0);      // ArithmeticException
```

# Floating-Point Calculations – Abnormalities

- Sometimes floating-point numbers work incorrectly!
- Read more about **IEEE 754**

```
double a = 1.0f;
double b = 0.33f;
double sum = 1.33d;
System.out.printf("a+b=%f sum=%f equal=%b",
                  a+b, sum, (a + b == sum));
// a+b=1.33000001311302 sum=1.33 equal = false
double num = 0;
for (int i = 0; i < 10000; i++) num += 0.0001;
System.out.println(num); // 0.999999999999062
```

# BigDecimal

- Built-in Java Class
- Provides arithmetic operations
- Allows calculations with very **high precision**
- Used for financial calculations



```
BigDecimal number = new BigDecimal(0);
number = number.add(BigDecimal.valueOf(2.5));
number = number.subtract(BigDecimal.valueOf(1.5));
number = number.multiply(BigDecimal.valueOf(2));
number = number.divide(BigDecimal.valueOf(2));
```

# Problem: Exact Sum of Real Numbers

- Write program to enter **n** numbers and print their exact sum:

```
2  
10000000000000000000  
5
```



```
10000000000000000005
```

```
2  
0.0000000003  
33333333333.3
```



```
33333333333.3000000003
```

# Solution: Exact Sum of Real Numbers

```
int n = Integer.parseInt(sc.nextLine());
BigDecimal sum = new BigDecimal(0);
for (int i = 0; i < n; i++) {
    BigDecimal number = new BigDecimal(sc.nextLine());
    sum = sum.add(number);
}
System.out.println(sum);
```

Check your solution here: <https://judge.softuni.org/Contests/1227/>



# Live Exercises

## Integer and Real Numbers



# Type Conversion

# Type Conversion

- Variables hold values of a certain type
- Type can be **changed (converted)** to another type
  - **Implicit** type conversion (**lossless**): variable of the bigger type (e.g. **Double**) takes a smaller value (e.g. **float**)

```
float heightInMeters = 1.74f;  
double maxHeight = heightInMeters;
```

Implicit conversion

- **Explicit** type conversion (**lossy**) – when precision can be lost:

```
double size = 3.14;  
int intSize = (int) size;
```

Explicit conversion

# Problem: Centuries to Minutes

- Write a program to enter an integer number of centuries and convert it to years, days, hours, and minutes

1



1 centuries = 100 years = 36524 days  
= 876581 hours = 52594877 minutes

5



5 centuries = 500 years = 182621 days  
= 4382906 hours = 262974384 minutes

The output is  
on one row

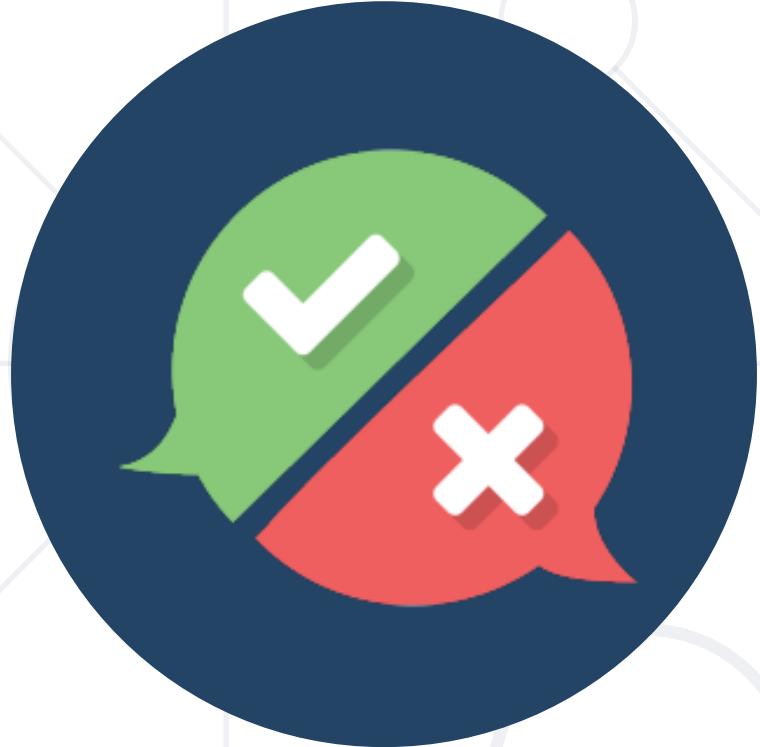
Check your solution here: <https://judge.softuni.org/Contests/1227/>

# Solution: Centuries to Minutes

```
int centuries = Integer.parseInt(sc.nextLine());
double years = centuries * 100;
double days = years * 365.2422;
double hours = 24 * days;
double minutes = 60 * hours;
System.out.printf(
    "%d centuries = %.0f years = %.0f days = %.0f hours = %.0f minutes",
    centuries, years, days, hours, minutes);
```

Tropical year has  
**365.2422** days

Check your solution here: <https://judge.softuni.org/Contests/1227/>



# Boolean Type

# Boolean Type

- Boolean variables (**boolean**) hold **true** or **false**:

```
int a = 1;  
int b = 2;  
boolean greaterAB = (a > b);  
System.out.println(greaterAB); // False  
boolean equalA1 = (a == 1);  
System.out.println(equalA1); // True
```

# Problem: Special Numbers

- A number is special when its sum of digits is 5, 7 or 11
  - For all numbers **1...n** print the number and if it is special



1 -> false	8 -> false	15 -> false
2 -> false	9 -> false	16 -> true
3 -> false	10 -> false	17 -> false
4 -> false	11 -> false	18 -> false
5 -> true	12 -> false	19 -> false
6 -> false	13 -> false	20 -> false
7 -> true	14 -> true	

Check your solution here: <https://judge.softuni.org/Contests/1227/>

# Solution: Special Numbers

```
int n = Integer.parseInt(sc.nextLine());
for (int num = 1; num <= n; num++) {
    int sumOfDigits = 0;
    int digits = num;
    while (digits > 0) {
        sumOfDigits += digits % 10;
        digits = digits / 10;
    }
    // TODO: check whether the sum is special
}
```

Check your solution here: <https://judge.softuni.org/Contests/1227/>



'A'

**Character Type**

# The Character Data Type

- The character data type
  - Represents symbolic information
  - Is declared by the **char** keyword
  - Gives each symbol a corresponding integer code
  - Has a '**\0**' default value
  - Takes 16 bits of memory (from **U+0000** to **U+FFFF**)
  - Holds a single Unicode character (or part of character)

# Characters and Codes

- Each **character** has an unique **Unicode** value (**int**):

```
char ch = 'a';
System.out.printf("The code of '%c' is: %d%n", ch, (int) ch);
ch = 'b';
System.out.printf("The code of '%c' is: %d%n", ch, (int) ch);
ch = 'A';
System.out.printf("The code of '%c' is: %d%n", ch, (int) ch);
ch = 'щ'; // Cyrillic letter 'sht'
System.out.printf("The code of '%c' is: %d%n", ch, (int) ch);
```

# Problem: Reversed Chars

- Write a program that takes 3 lines of characters and prints them in reversed order with a space between them
- Examples:



# Solution: Reversed Chars

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

char firstChar = scanner.nextLine().charAt(0);
char secondChar = scanner.nextLine().charAt(0);
char thirdChar = scanner.nextLine().charAt(0);

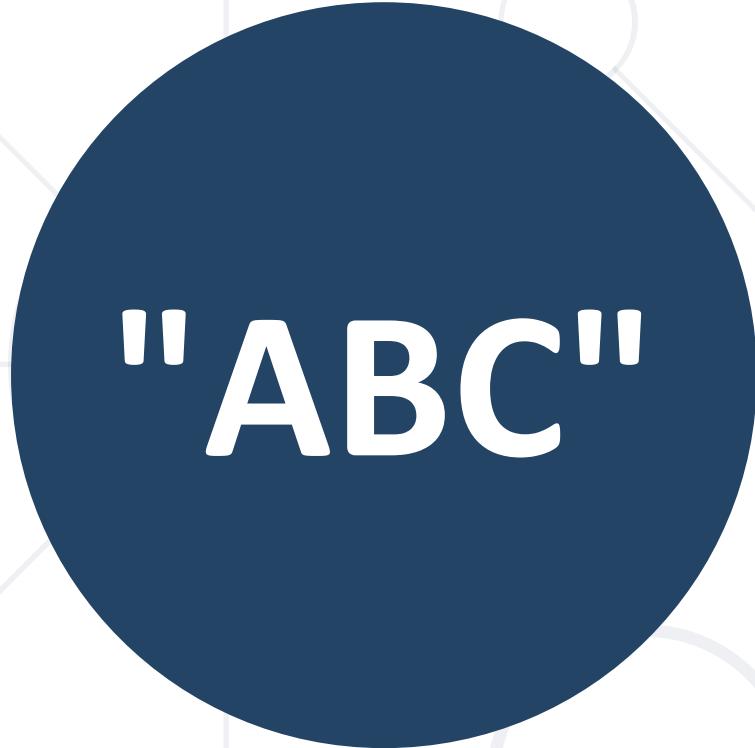
System.out.printf("%c %c %c",
                  thirdChar, secondChar, firstChar);
```

# Escaping Characters

- Escaping sequences are:
  - Represent a special character like ', " or \n (new line)
  - Represent system characters (like the [TAB] character \t)
- Commonly used escaping sequences are:
  - \' → for single quote      \" → for double quote
  - \\ → for backslash      \n → for a new line
  - \uXXXX → for denoting any other Unicode symbol

# Character Literals – Example

```
char symbol = 'a'; // An ordinary character
symbol = '\u006F'; // Unicode character code in a
                  // hexadecimal format (letter 'o')
symbol = '\u8449'; // 葉 (Leaf in Traditional Chinese)
symbol = '\''; // Assigning the single quote character
symbol = '\\'; // Assigning the backslash character
symbol = '\n'; // Assigning new Line character
symbol = '\t'; // Assigning TAB character
symbol = "a"; // Incorrect: use single quotes!
```



"ABC"

**String**  
Sequence of Letters

# The String Data Type

- The string data type
  - Represents a sequence of characters
  - Is declared by the **String** keyword
  - Has a default value **null** (no value)
- Strings are enclosed in quotes:

```
String s = "Hello, JAVA";
```
- Strings can be concatenated
  - Using the **+** operator



# Formatting Strings

- Strings are enclosed in quotes "":

```
String file = "C:\\Windows\\\\win.ini";
```

The backslash \ is  
escaped by \\

- Format strings insert variable values by pattern:

```
String firstName = "Svetlin";
String lastName = "Nakov";
String fullName = String.format(
    "%s %s", firstName, lastName);
```

# Saying Hello – Examples

- Combining the names of a person to obtain the full name:

```
String firstName = "Ivan";
String lastName = "Ivanov";
String fullName = String.format(
    "%s %s", firstName, lastName);
System.out.printf("Your full name is %s.", fullName);
```

- We can concatenate strings and numbers by the + operator:

```
int age = 21;
System.out.println("Hello, I am " + age + " years old");
```

# Problem: Concat Names

- Read first and last name and delimiter
- Print the first and last name joined by the delimiter

John  
Smith  
->

John->Smith

Jan  
White  
<->

Jan<->White

Linda  
Terry  
=>

Linda=>Terry

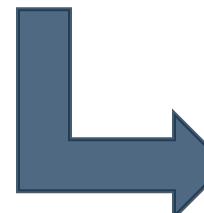
Lee  
Lewis  
---

Lee---Lewis

# Solution: Concat Names

```
String firstName = sc.nextLine();
String lastName = sc.nextLine();
String delimiter = sc.nextLine();

String result = firstName + delimiter + lastName;
System.out.println(result);
```



Jan<->White



# Live Exercises

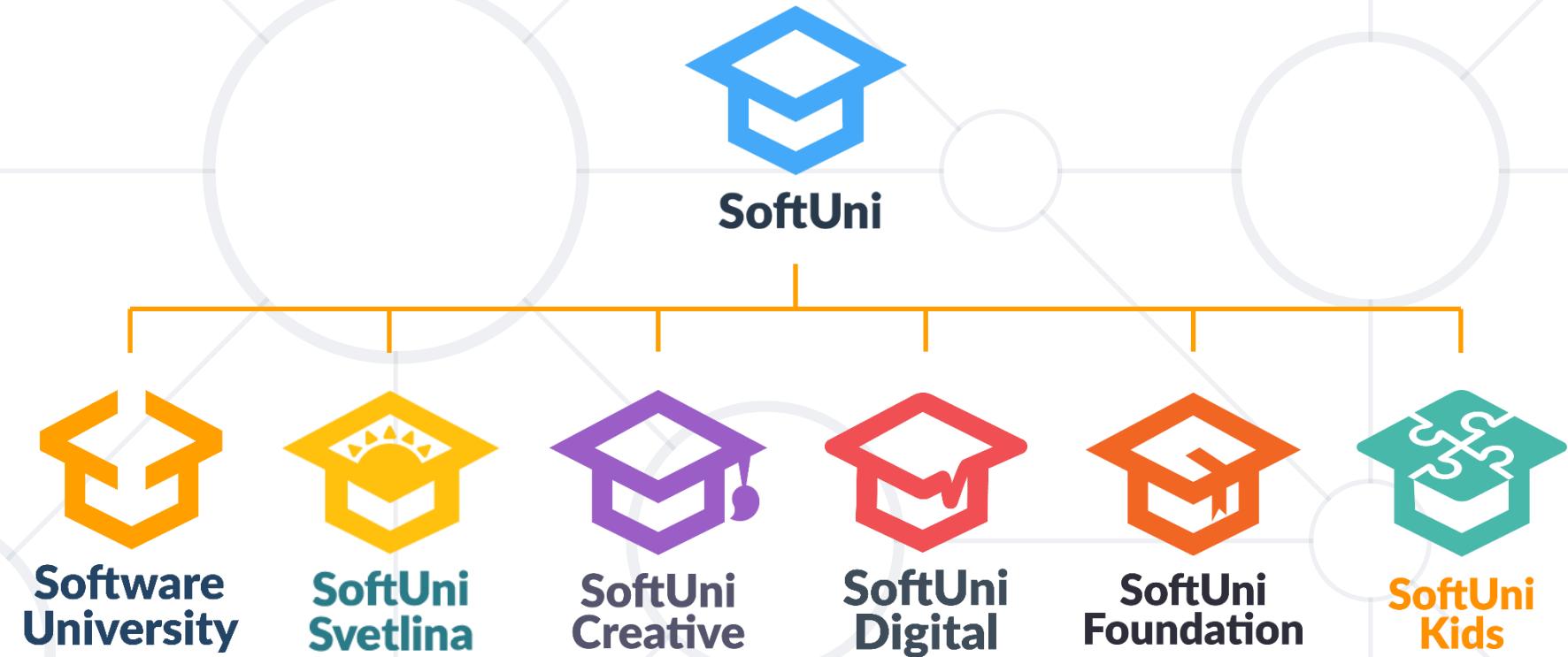
## Data Types

# Summary

- **Variables** – store data
- Numeral types:
  - Represent **numbers**
  - Have **specific ranges** for every type
- String and text types:
  - Represent **text**
  - **Sequences of Unicode characters**
- Type conversion: **implicit** and **explicit**



# Questions?



# SoftUni Diamond Partners

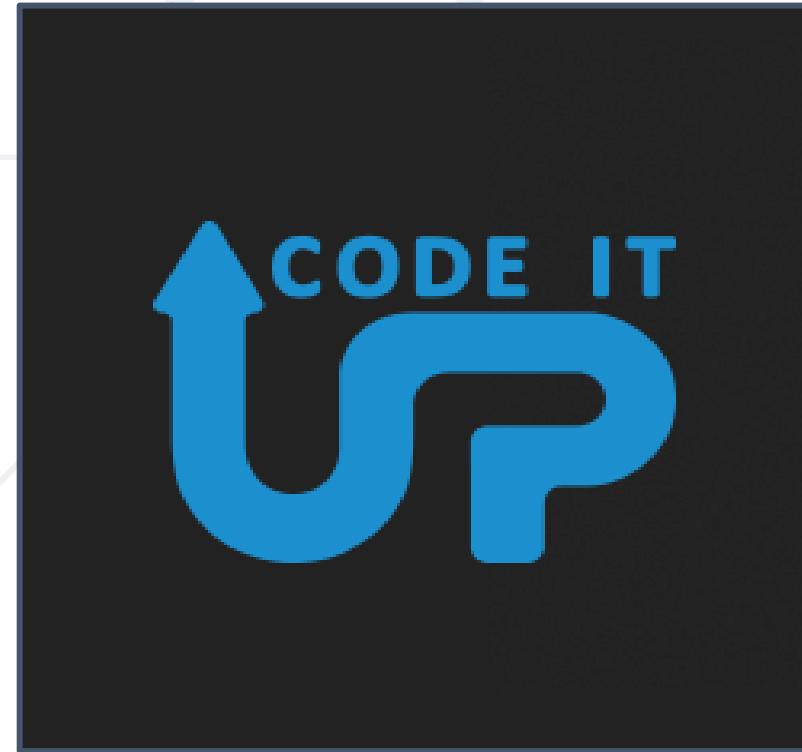


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

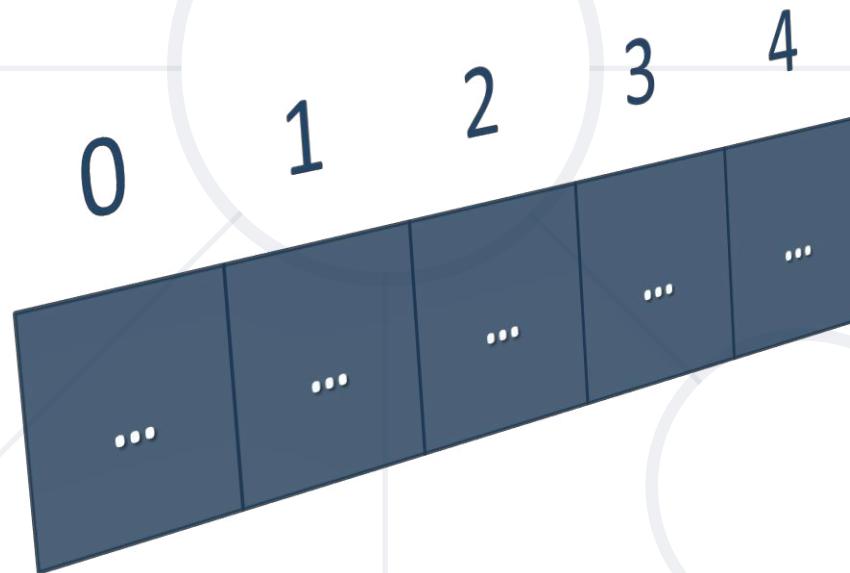


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Arrays

Fixed-Size Sequences of Elements



SoftUni Team

Technical Trainers



SoftUni



Software University  
<https://softuni.bg>

Have a Question?



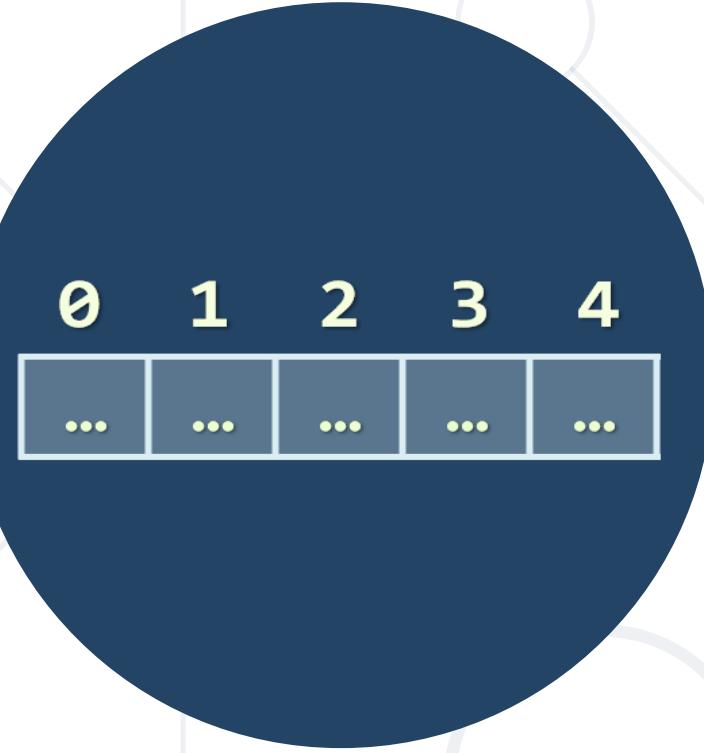
**sli.do**

**#fund-java**

# Table of Contents

1. Arrays
2. Reading Arrays from the Console
3. Foreach Loop





# Arrays

Working with Arrays of Elements

# What Are Arrays?

- In programming, an **array** is a **sequence of elements**



Array of 5 elements



Element's index

Element of an array

- Arrays have **fixed size (`array.length`)**  
cannot be resized
- Elements are of the **same type** (e.g. integers)
- Elements are numbered from **0** to **length-1**

# Working with Arrays

- Allocating an array of 10 integers:

```
int[] numbers = new int[10];
```

All elements are initially == 0

- Assigning values to the array elements:

```
for (int i = 0; i < numbers.length; i++)  
    numbers[i] = 1;
```

The length holds the number of array elements

- Accessing array elements by index:

```
numbers[5] = numbers[2] + numbers[7];
```

```
numbers[10] = 1; // ArrayIndexOutOfBoundsException
```

The [] operator accesses elements by index

# Days of Week – Example

- The days of a week can be stored in an array of strings:

```
String[] days = {  
    "Monday",  
    "Tuesday",  
    "Wednesday",  
    "Thursday",  
    "Friday",  
    "Saturday",  
    "Sunday"  
};
```



Operator	Value
days[0]	Monday
days[1]	Tuesday
days[2]	Wednesday
days[3]	Thursday
days[4]	Friday
days[5]	Saturday
days[6]	Sunday

# Problem: Day of Week

- Enter a day number [1...7] and print the day name (in English) or "Invalid day!"

```
String[] days = { "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "Sunday" };
int day = Integer.parseInt(sc.nextLine());
if (day >= 1 && day <= 7)
    System.out.println(days[day - 1]);
else
    System.out.println("Invalid day!");
```

The first day in our array is on index 0, not 1.



# Reading Array

Using a for Loop or String.split()

# Reading Arrays from the Console

- First, read the array **length** from the console :

```
int n = Integer.parseInt(sc.nextLine());
```

- Next, create an array of given size **n** and read its **elements**:

```
int[] arr = new int[n];
for (int i = 0; i < n; i++) {
    arr[i] = Integer.parseInt(sc.nextLine());
}
```

# Reading Array Values from a Single Line

- Arrays can be read from a **single line** of **separated values**

```
2 8 30 25 40 72 -2 44 56
```

```
String values = sc.nextLine();
String[] items = values.split(" ");
int[] arr = new int[items.length];

for (int i = 0; i < items.length; i++)
    arr[i] = Integer.parseInt(items[i]);
```

# Shorter: Reading Array from a Single Line

- Read an array of integers using functional programming:

```
String inputLine = sc.nextLine();
String[] items = inputLine.split(" ");
int[] arr = Arrays.stream(items)
    .mapToInt(e -> Integer.parseInt(e))
    .toArray();
```

import  
java.util.Arrays;

```
int[] arr = Arrays
    .stream(sc.nextLine().split(" "))
    .mapToInt(e -> Integer.parseInt(e))
    .toArray();
```

You can chain  
methods

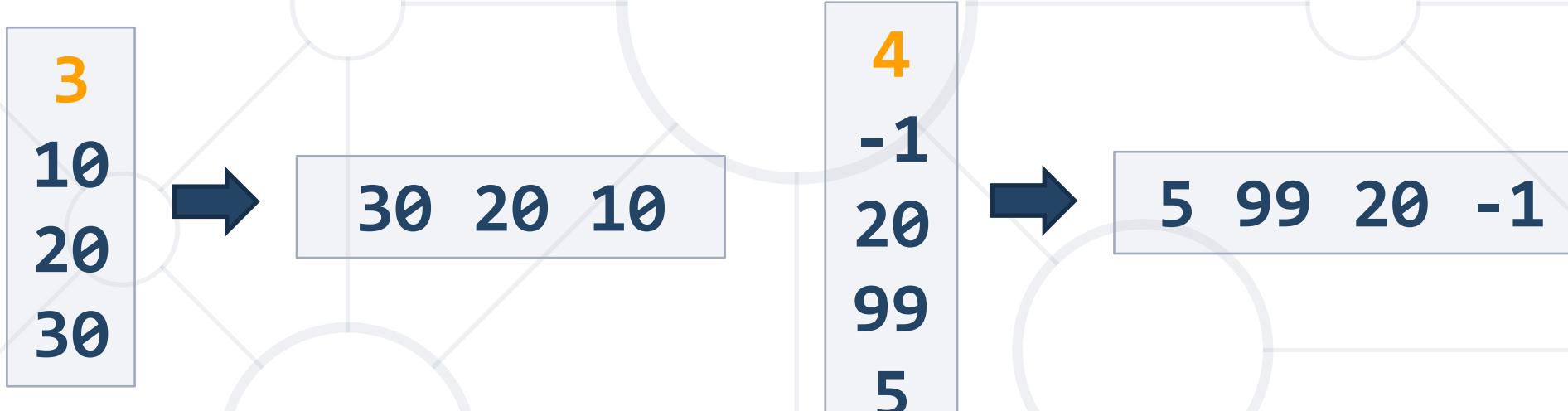
# Printing Arrays On the Console

- To print all array elements, a for-loop can be used
  - Separate elements with white space or a new line

```
String[] arr = {"one", "two"};
// == new String [] {"one", "two"};
// Process all array elements
for (int i = 0; i < arr.length; i++) {
    System.out.printf("arr[%d] = %s%n", i, arr[i]);
}
```

# Problem: Reverse an Array of Integers

- Read an array of integers (**n** lines of integers), **reverse** it and print its elements on a single line, space-separated:



Check your solution here: <https://judge.softuni.org/Contests/1248/>

# Solution: Reverse an Array of Integers

```
// Read the array (n Lines of integers)
int n = Integer.parseInt(sc.nextLine());
int[] arr = new int[n];
for (int i = 0; i < n; i++)
    arr[i] = Integer.parseInt(sc.nextLine());
// Print the elements from the last to the first
for (int i = n - 1; i >= 0; i--)
    System.out.print(arr[i] + " ");
System.out.println();
```

Check your solution here: <https://judge.softuni.org/Contests/1248/>

# Printing Arrays with for / String.join(...)

- Use for-loop:

```
String[] arr = {"one", "two"};
for (int i = 0; i < arr.length; i++)
    System.out.println(arr[i]);
```

- Use **String.join(separator, array)**:

```
String[] strings = { "one", "two" };
System.out.println(String.join(" ", strings)); // one two
int[] arr = { 1, 2, 3 };
System.out.println(String.join(" ", arr)); // Compile error
```

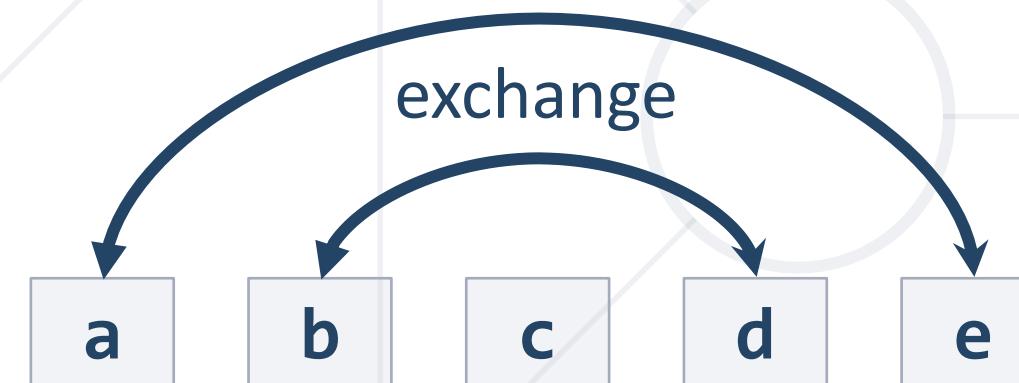
Works only  
with strings

# Problem: Reverse Array of Strings

- Read an array of strings (space separated values), reverse it and print its elements:



- Reversing array elements:



Check your solution here: <https://judge.softuni.org/Contests/1248/>

# Solution: Reverse Array of Strings

```
String[] elements = sc.nextLine().split(" ");
for (int i = 0; i < elements.length / 2; i++) {
    String oldElement = elements[i];
    elements[i] = elements[elements.length - 1 - i];
    elements[elements.length - 1 - i] = oldElement;
}
System.out.println(String.join(" ", elements));
```

Check your solution here: <https://judge.softuni.org/Contests/1248/>



# Foreach Loop

Iterate Through Collections

# Foreach Loop

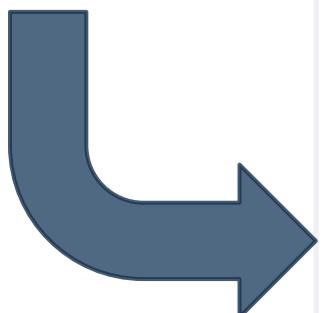
- Iterates through all elements in a collection
- Cannot access the current index
- **Read-only**

```
for (var item : collection) {  
    // Process the value here  
}
```



# Print an Array with Foreach

```
int[] numbers = { 1, 2, 3, 4, 5 };
for (int number : numbers) {
    System.out.println(number + " ");
}
```



```
1 2 3 4 5
```

# Problem: Even and Odd Subtraction

- Read an array of integers
- Sum all even and odd numbers
- Find the difference
- Examples:



Check your solution here: <https://judge.softuni.org/Contests/1248/>

# Solution: Even and Odd Subtraction

```
int[] arr = Arrays.stream(sc.nextLine().split(" ")).  
                  .mapToInt(e -> Integer.parseInt(e)).toArray();  
  
int evenSum = 0;  
  
int oddSum = 0;  
  
for (int num : arr) {  
    if (num % 2 == 0) evenSum += num;  
    else oddSum += num;  
}  
  
//TODO: Find the difference and print it
```



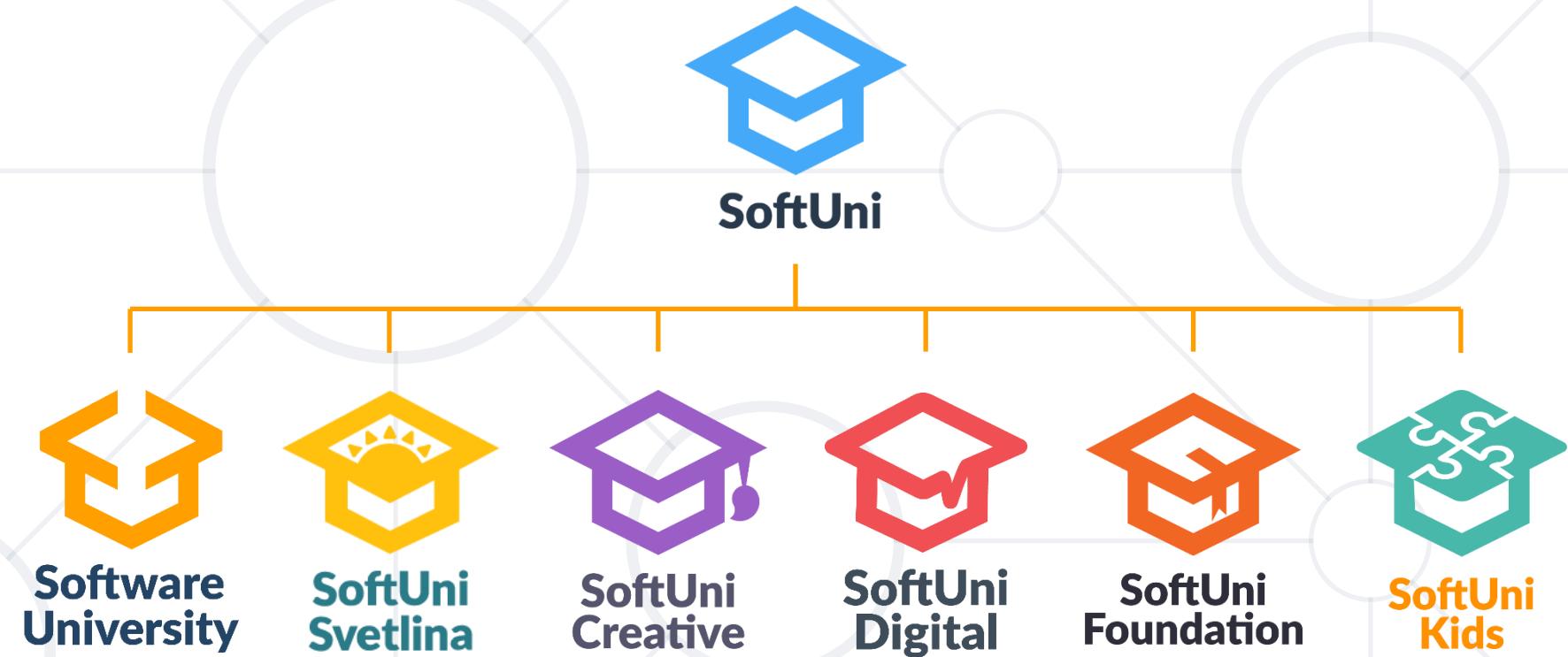
# Live Exercises

# Summary

- Arrays hold a **sequence** of elements
  - Elements are numbered from **0** to **length - 1**
- Creating (allocating) an array
- Accessing array elements by **index**
- Printing array elements



# Questions?



# SoftUni Diamond Partners

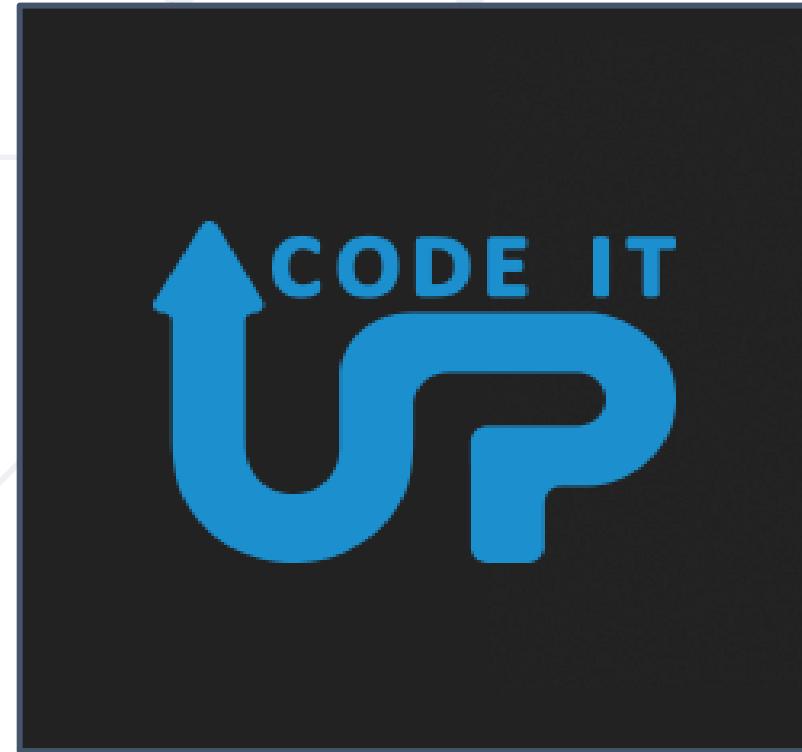


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

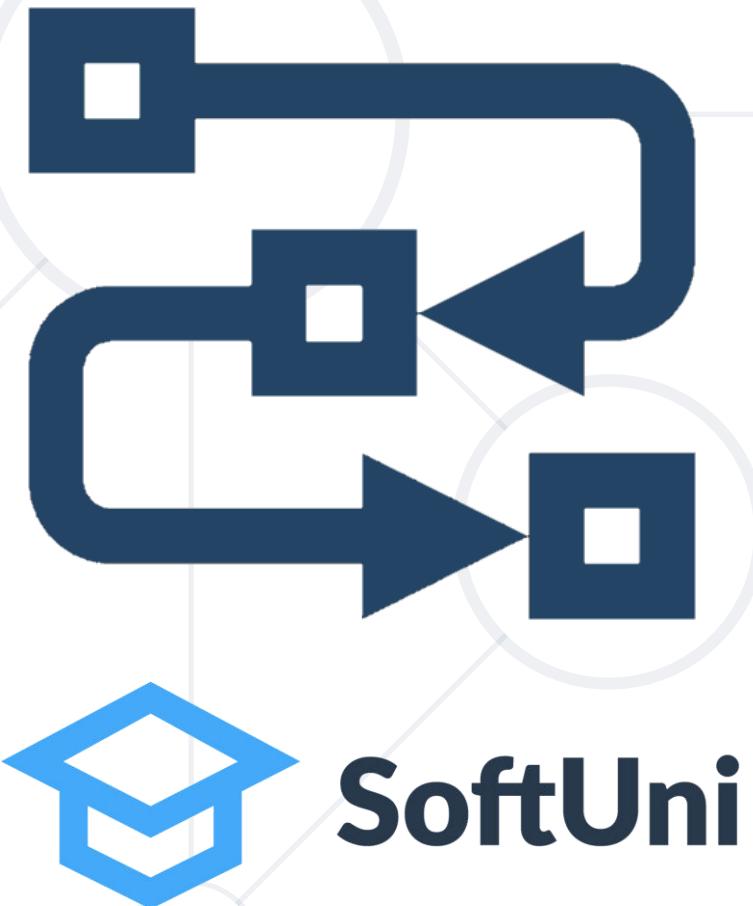


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Methods

Defining and Using Methods, Overloads



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Have a Question?



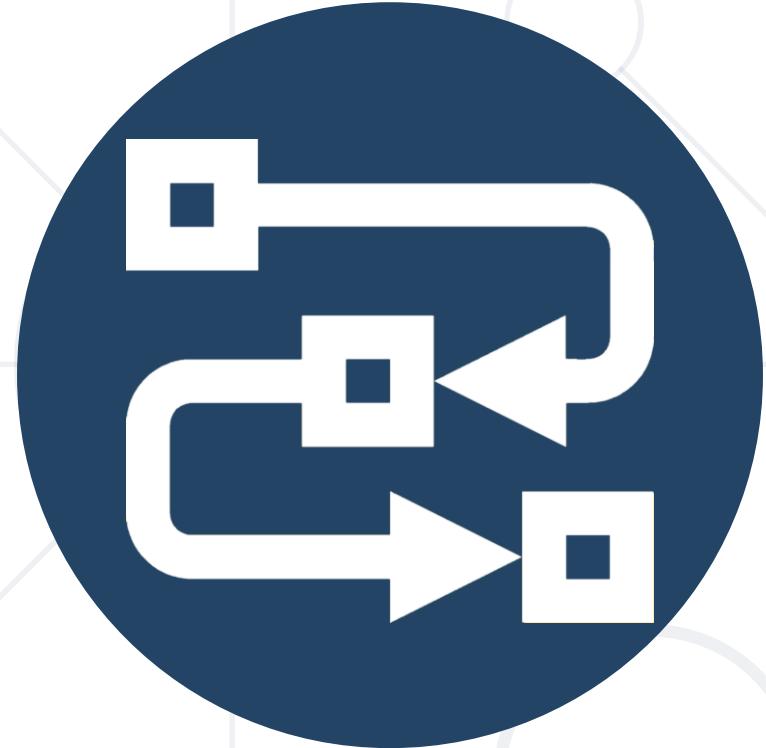
**sli.do**

**#fund-java**

# Table of Contents

1. What Is a Method?
2. Naming and Best Practices
3. Declaring and Invoking Methods
  - Void and Return Type Methods
4. Methods with Parameters
5. Value vs. Reference Types
6. Overloading Methods
7. Program Execution Flow





**What is a Method?**

**Void Method**

# Simple Methods

- Named **block of code**, that can be invoked later
- Sample method **definition**:

Method named  
**printHello**

```
public static void printHello () {  
    System.out.println("Hello!");  
}
```

Method **body**  
always  
surrounded  
by **{ }**

- **Invoking** (calling) the method several times:

```
printHello();  
printHello();
```



# Why Use Methods?

- 
- More **manageable programming**
    - Splits large problems into small pieces
    - Better organization of the program
    - Improves code readability
    - Improves code understandability
  - Avoiding **repeating code**
    - Improves code maintainability
  - **Code reusability**
    - Using existing methods several times

# Void Type Method

- Executes the code between the brackets
- Does **not** return result

```
public static void printHello() {  
    System.out.println("Hello");  
}
```

Prints  
"Hello" on  
the console

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

**main()** is  
also a  
method



# Naming and Best Practices

# Naming Methods

- Methods naming guidelines
  - Use **meaningful** method names
  - Method names should answer the question:
    - **What does this method do?**
  - If you cannot find a good name for a method, think about whether it has a **clear intent**



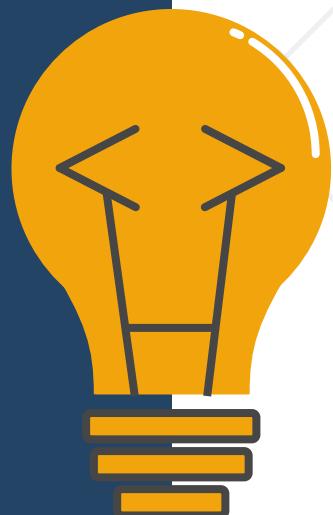
`findStudent, loadReport, sine`



`Method1, DoSomething, HandleStuff, SampleMethod`

# Naming Method Parameters

- Method parameters names
  - Preferred form: [Noun] or [Adjective] + [Noun]
  - Should be in **camelCase**
  - Should be **meaningful**  
`firstName, report, speedKmH,  
usersList, fontSizeInPixels, font`
- Unit of measure should be obvious  
`p, p1, p2, populate, LastName, last_name, convertImage`



# Methods – Best Practices

- Each method should perform a **single**, well-defined task
  - A Method's name should **describe that task** in a clear and non-ambiguous way
- **Avoid** methods **longer than one screen**
  - **Split them** to several shorter methods

```
private static void printReceipt() {  
    printHeader();  
    printBody();  
    printFooter();  
}
```

**Self documenting  
and easy to test**

# Code Structure and Code Formatting

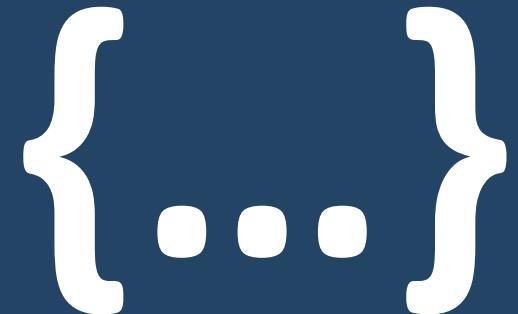
- Make sure to use correct **indentation**

```
static void main(args) {  
    // some code...  
    // some more code...  
}
```

```
static void main(args)  
    {  
        // some code...  
        // some more code...  
    }
```



- Leave a **blank line** between **methods**, after **loops** and after **if statements**
- Always use **curly brackets** for loops and if statements bodies
- **Avoid long lines and complex expressions**



# Declaring and Invoking Methods

# Declaring Methods

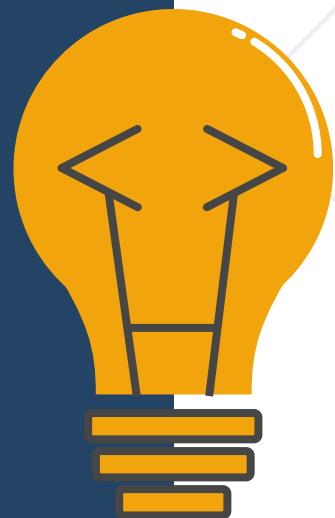
Type

Method Name

Parameters

```
public static void printText(String text) {  
    System.out.println(text);  
}
```

Method Body



- Methods are declared **inside a class**
- **main()** is also a method
- Variables inside a method are **local**

# Invoking a Method

- Methods are first **declared**, then **invoked** (many times)

```
public static void printHeader() {  
    System.out.println("-----");  
}
```

Method  
Declaration

- Methods can be **invoked (called)** by their name + ():

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

Method  
Invocation

# Invoking a Method (2)

- A method can be invoked from:

- The main method – **main()**

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

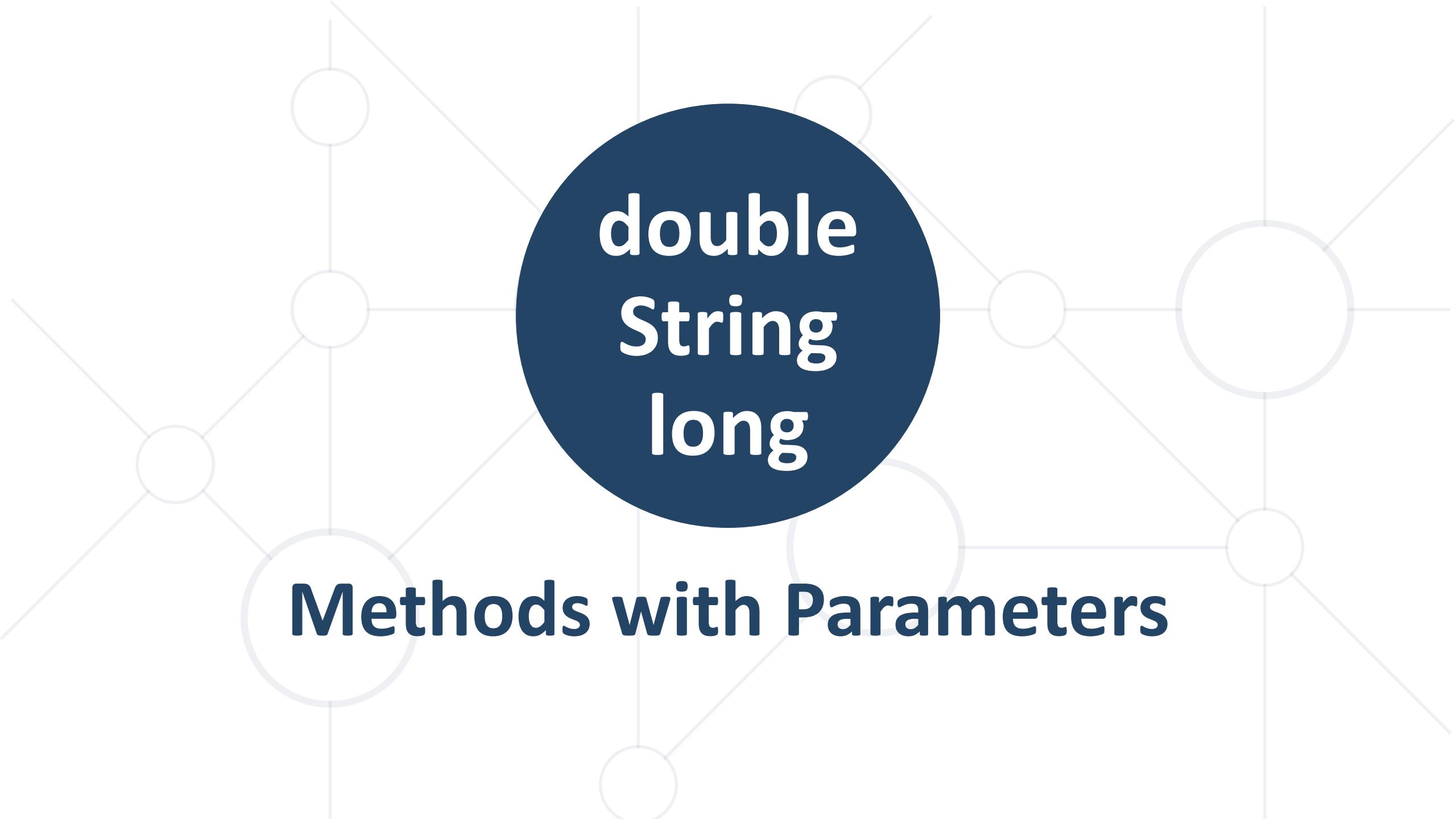
- Its own body – **recursion**

```
static void crash() {  
    crash();  
}
```

# Invoking a Method (3)

- Some **other method**

```
public static void printHeader() {  
    printHeaderTop();  
    printHeaderBottom();  
}
```



**double  
String  
long**

**Methods with Parameters**

# Method Parameters

- Method **parameters** can be of **any data type**

```
static void printNumbers(int start, int end) {  
    for (int i = start; i <= end; i++) {  
        System.out.printf("%d ", i);  
    }  
}
```

Multiple parameters  
separated by comma

- Call the method with certain values (**arguments**)

```
public static void main(String[] args) {  
    printNumbers(5, 10);  
}
```

Passing arguments  
at invocation

# Method Parameters (2)

- You can pass **zero or several** parameters
- You can pass parameters of **different types**
- Each parameter has **name** and **type**

Multiple parameters  
of different types

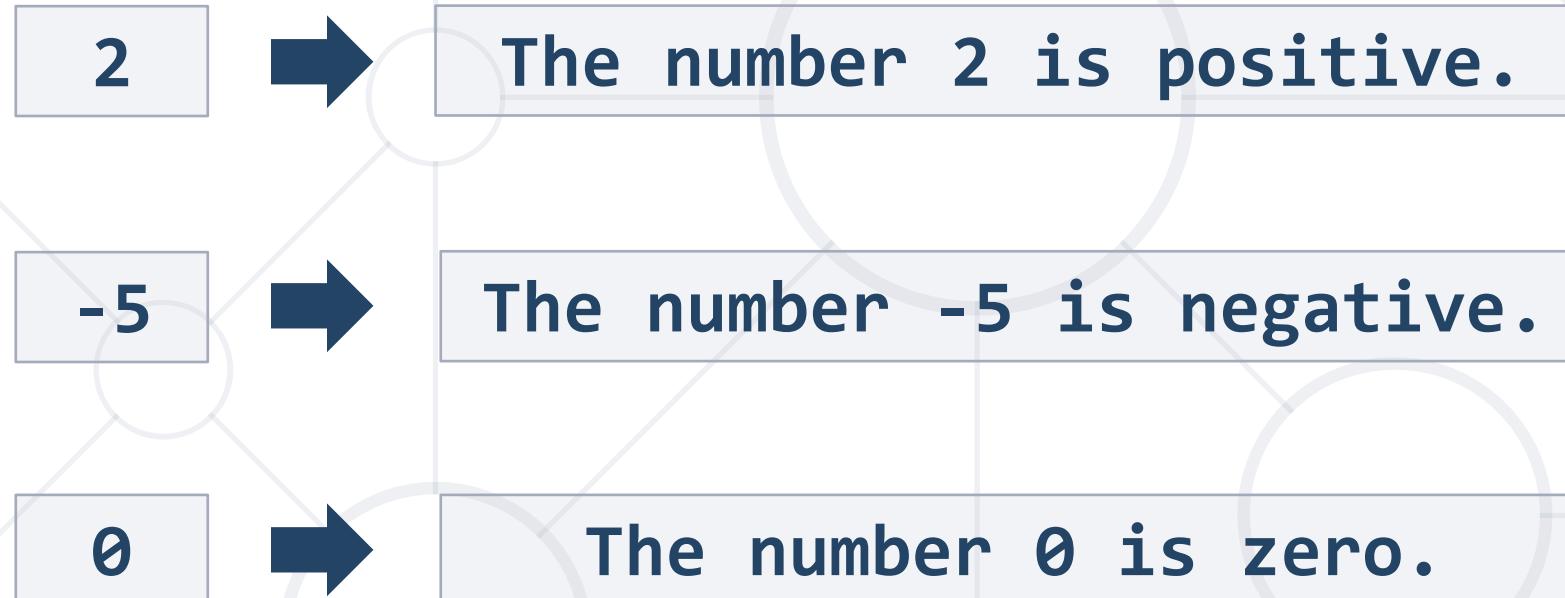
Parameter  
type

Parameter  
name

```
public static void printStudent(String name, int age, double grade) {  
    System.out.printf("Student: %s; Age: %d, Grade: %.2f\n",  
        name, age, grade);  
}
```

# Problem: Sign of Integer Number

- Create a method that prints the **sign** of an integer number **n**:



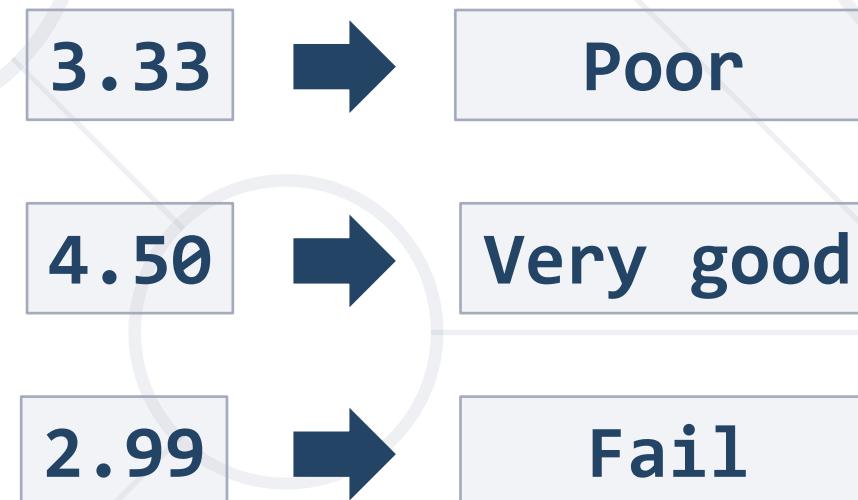
Check your solution here: <https://judge.softuni.org/Contests/1260/>

# Solution: Sign of Integer Number

```
public static void main(String[] args) {  
    printSign(Integer.parseInt(sc.nextLine()));  
}  
  
public static void printSign(int number) {  
    if (number > 0)  
        System.out.printf("The number %d is positive.", number);  
    else if (number < 0)  
        System.out.printf("The number %d is negative.", number);  
    else  
        System.out.printf("The number %d is zero.", number);  
}
```

# Problem Grades

- Write a method that receives a grade between 2.00 and 6.00 and prints the corresponding grade in words
  - 2.00 - 2.99 - "Fail"
  - 3.00 - 3.49 - "Poor"
  - 3.50 - 4.49 - "Good"
  - 4.50 - 5.49 - "Very good"
  - 5.50 - 6.00 - "Excellent"



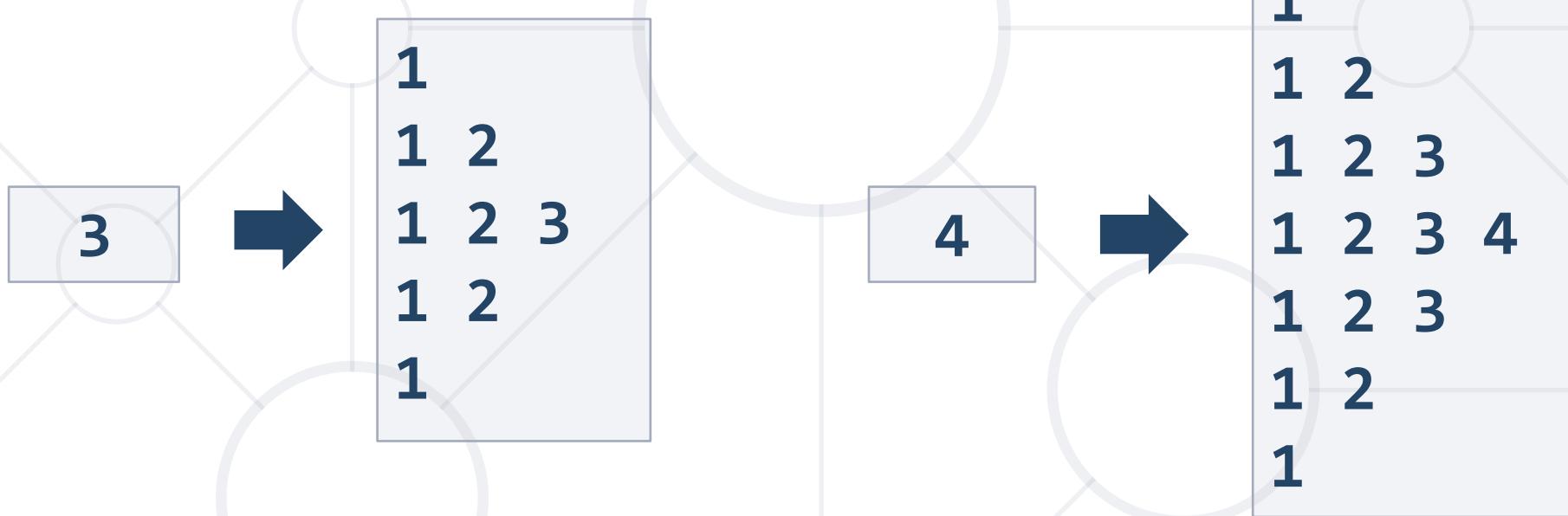
Check your solution here: <https://judge.softuni.org/Contests/1260/>

# Solution Grades

```
public static void main(String[] args) {  
    printInWords(Double.parseDouble(sc.nextLine()));  
}  
  
public static void printInWords(double grade) {  
    String gradeInWords = "";  
    if (grade >= 2 && grade <= 2.99)  
        gradeInWords = "Fail";  
    //TODO: make the rest  
    System.out.println(gradeInWords);  
}
```

# Problem: Printing Triangle

- Create a method for printing triangles as shown below:



Check your solution here: <https://judge.softuni.org/Contests/1260/>

# Solution: Printing Triangle (1)

- Create a method that **prints a single line**, consisting of numbers from a **given start** to a **given end**:

```
public static void printLine(int start, int end) {  
    for (int i = start; i <= end; i++) {  
        System.out.print(i + " ");  
    }  
    System.out.println();  
}
```

Check your solution here: <https://judge.softuni.org/Contests/1260/>

# Solution: Printing Triangle (2)

- Create a method that prints the **first half (1..n)** and then the **second half (n-1...1)** of the triangle:

```
public static void printTriangle(int n) {  
    for (int line = 1; line <= n; line++)  
        printLine(1, line);  
  
    for (int line = n - 1; line >= 1; line--)  
        printLine(1, line);  
}
```

Method with  
parameter n

Lines 1...n

Lines n-1...1



# Live Exercises



**Returning Values from Methods**

# The Return Statement

- The **return** keyword immediately stops the method's execution
- Returns the specified value

```
public static String readFullName(Scanner sc) {  
    String firstName = sc.nextLine();  
    String lastName = sc.nextLine();  
    return firstName + " " + lastName;  
}
```

Returns a **String**

- Void methods can be **terminated** by just using **return**



# Using the Return Values

- Return value can be:
  - **Assigned** to a variable

```
int max = getMax(5, 10);
```
  - **Used** in expression

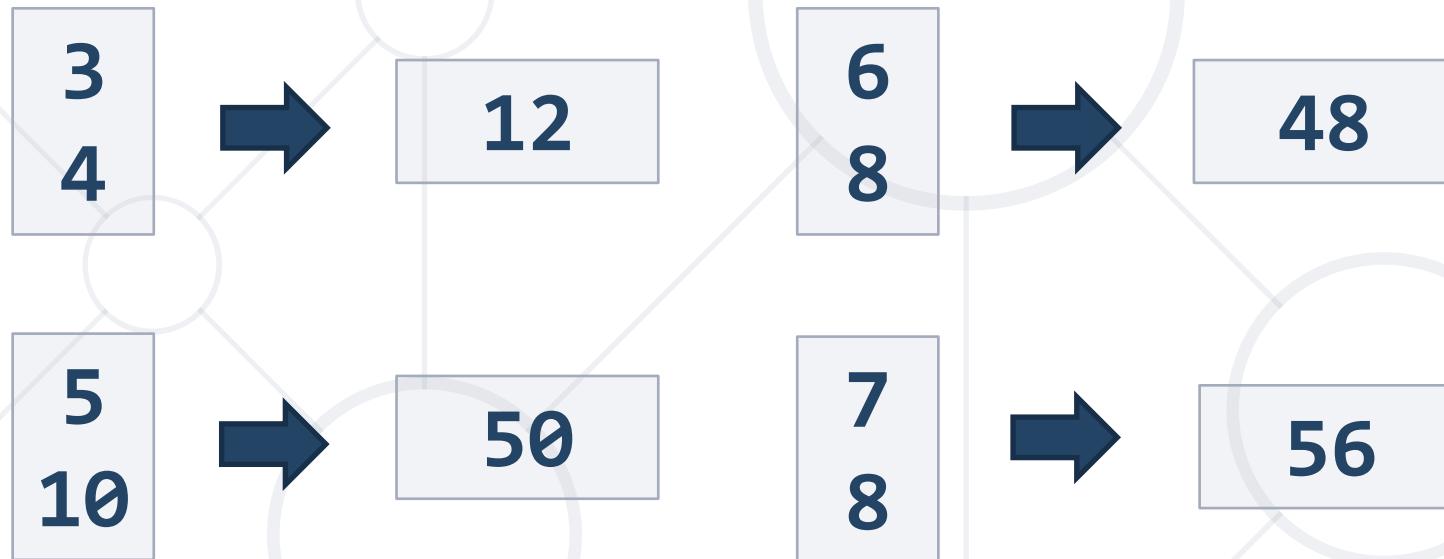
```
double total = getPrice() * quantity * 1.20;
```
  - **Passed** to another method



```
int age = Integer.parseInt(sc.nextLine());
```

# Problem: Calculate Rectangle Area

- Create a method which returns rectangle area with given width and height



Check your solution here: <https://judge.softuni.org/Contests/1260/>

# Solution: Calculate Rectangle Area

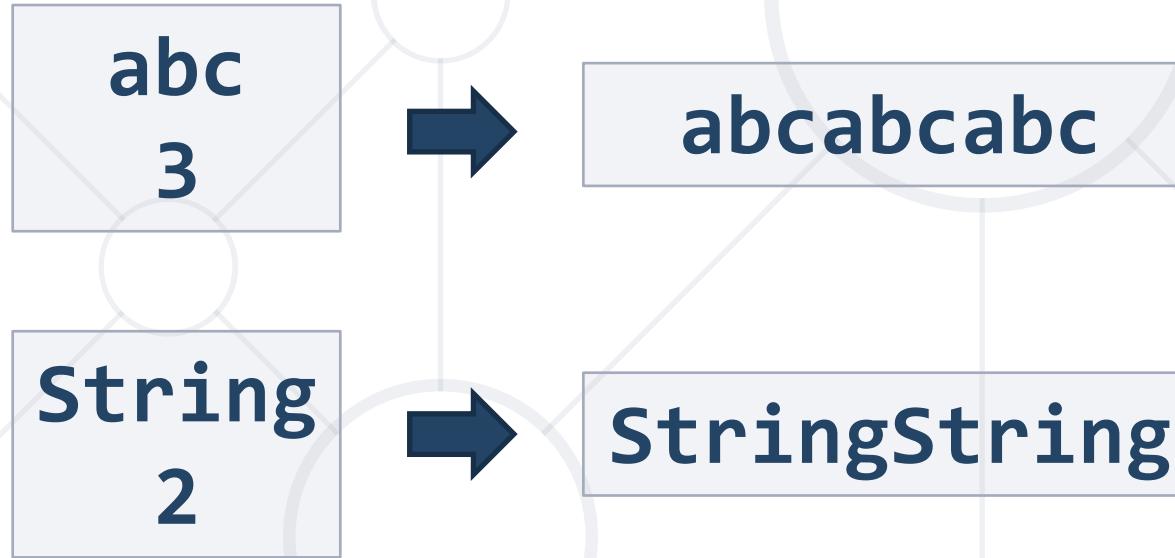
```
public static void main(String[] args) {  
    double width = Double.parseDouble(sc.nextLine());  
    double height = Double.parseDouble(sc.nextLine());  
    double area = calcRectangleArea(width, height);  
    System.out.printf("%.0f%n", area);  
}
```

```
public static double calcRectangleArea(double width,  
                                      double height) {  
    return width * height;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/1260/>

# Problem: Repeat String

- Write a method that receives a string and a repeat count n
- The method should return a new string



Check your solution here: <https://judge.softuni.org/Contests/1260/>

# Solution: Repeat String

```
public static void main(String[] args) {  
    String inputStr = sc.nextLine();  
    int count = Integer.parseInt(sc.nextLine());  
    System.out.println(repeatString(inputStr, count));  
}  
  
private static String repeatString(String str, int count) {  
    String result = "";  
    for (int i = 0; i < count; i++) result += str;  
    return result;  
}
```

# Problem: Math Power

- Create a method that calculates and returns the value of a **number raised to a given power**

 $2^8$ 

256

 $5.5^3$ 

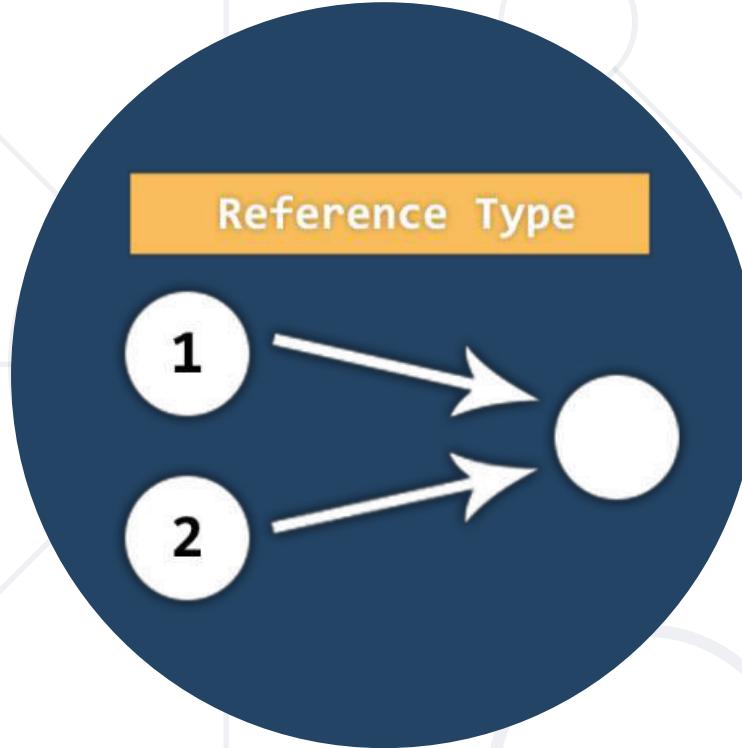
166.375

```
public static double mathPower(double number, int power) {  
    double result = 1;  
    for (int i = 0; i < power; i++)  
        result *= number;  
    return result;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/1260/>



# Live Exercises



# Value vs. Reference Types

Memory Stack and Heap

# Value vs. Reference Types

*pass by reference*

cup = 

fillCup( )

*pass by value*

cup = 

fillCup( )

# Value Types

- **Value type** variables hold directly their value
  - **int, float, double, boolean, char, ...**
  - Each variable has its own **copy** of the **value**

```
int i = 42;  
char ch = 'A';  
boolean result = true;
```



# Reference Types

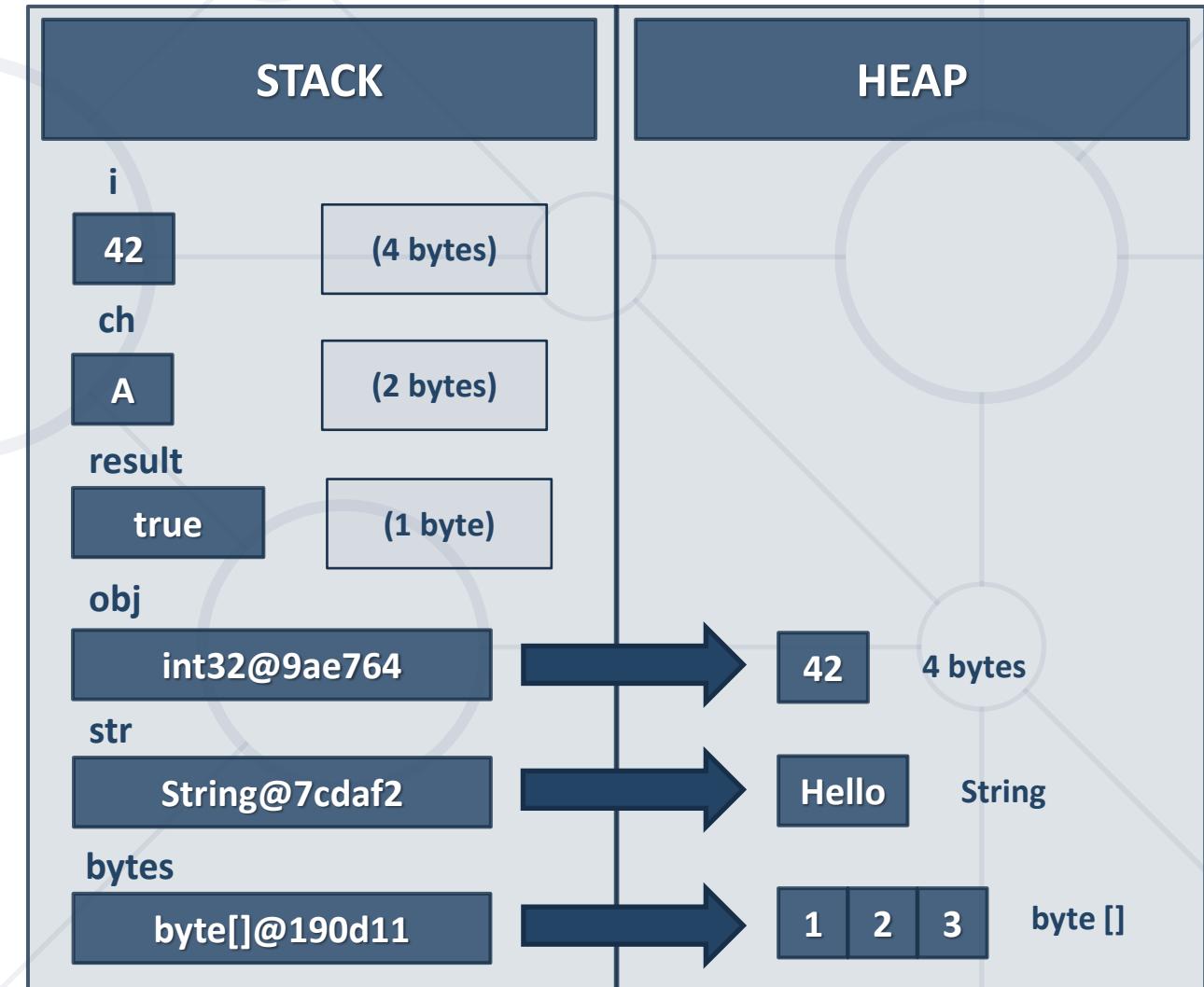
- Reference type variables hold a reference (pointer / memory address) of the value itself
  - String, int[], char[], String[]
- Two reference type variables can reference the same object
  - Operations on both variables access / modify the same data



# Value Types vs. Reference Types

```

int i = 42;
char ch = 'A';
boolean result = true;
Object obj = 42;
String str = "Hello";
byte[] bytes = { 1, 2, 3 };
    
```



# Example: Value Types

```
public static void main(String[] args) {  
    int num = 5;  
    increment(num, 15);  
    System.out.println(num);  
}
```

num == 5

```
public static void increment(int num, int value) {  
    num += value;  
}
```

num == 20

# Example: Reference Types

```
public static void main(String[] args) {  
    int[] nums = { 5 };  
    increment(nums, 15);  
    System.out.println(nums[0]);  
}
```

nums[0] == 20

```
public static void increment(int[] nums, int value) {  
    nums[0] += value;  
}
```

nums[0] == 20



# Live Exercises



# Overloading Methods

# Method Signature

- The combination of method's **name** and **parameters** is called **signature**

```
public static void print(String text) {  
    System.out.println(text);  
}
```

Method's  
signature

- Signature **differentiates** between methods with same names
- When methods with the **same name** have **different signature**, this is called method "**overloading**"

# Overloading Methods

- Using the same name for multiple methods with different **signatures** (method **name** and **parameters**)

```
static void print(int number) {  
    System.out.println(number);  
}
```

```
static void print(String text) {  
    System.out.println(text);  
}
```

```
static void print(String text, int number) {  
    System.out.println(text + ' ' + number);  
}
```

Different  
method  
signatures

# Signature and Return Type

- Method's return type **is not part** of its signature

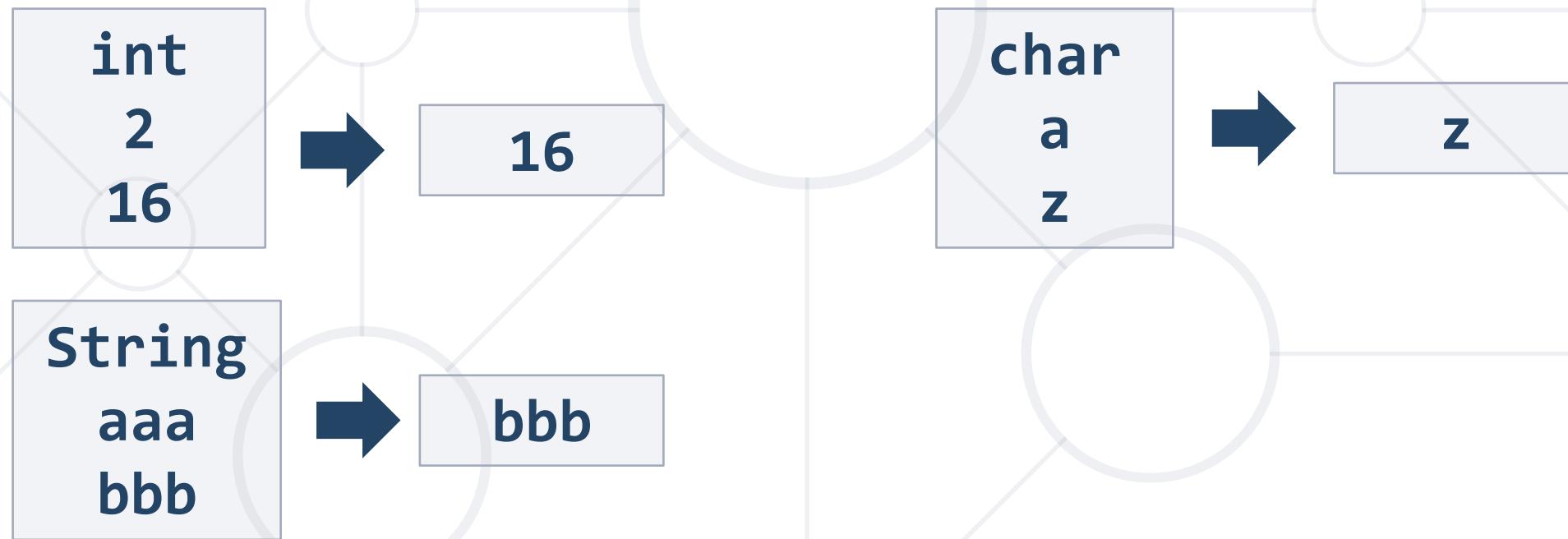
```
public static void print(String text) {  
    System.out.println(text);  
}  
  
public static String print(String text) {  
    return text;  
}
```

Compile-time  
error!

- How would the compiler know **which method to call?**

# Problem: Greater of Two Values

- Create a method **getMax()** that **returns the greater** of two values (the values can be of type **int**, **char** or **String**)

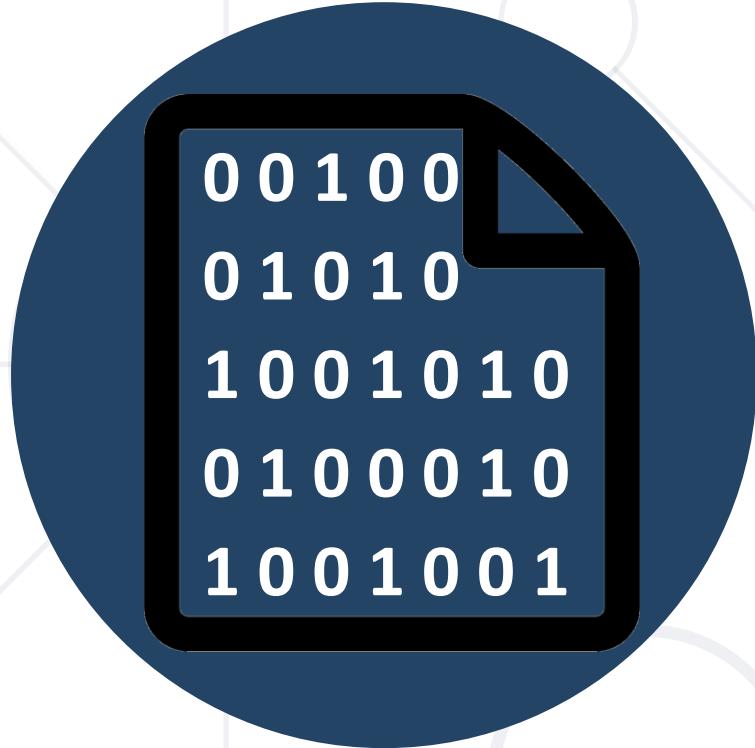


Check your solution here: <https://judge.softuni.org/Contests/1260/>



# Live Exercises

# Program Execution Flow



```
00100  
01010  
1001010  
0100010  
1001001
```

# Program Execution

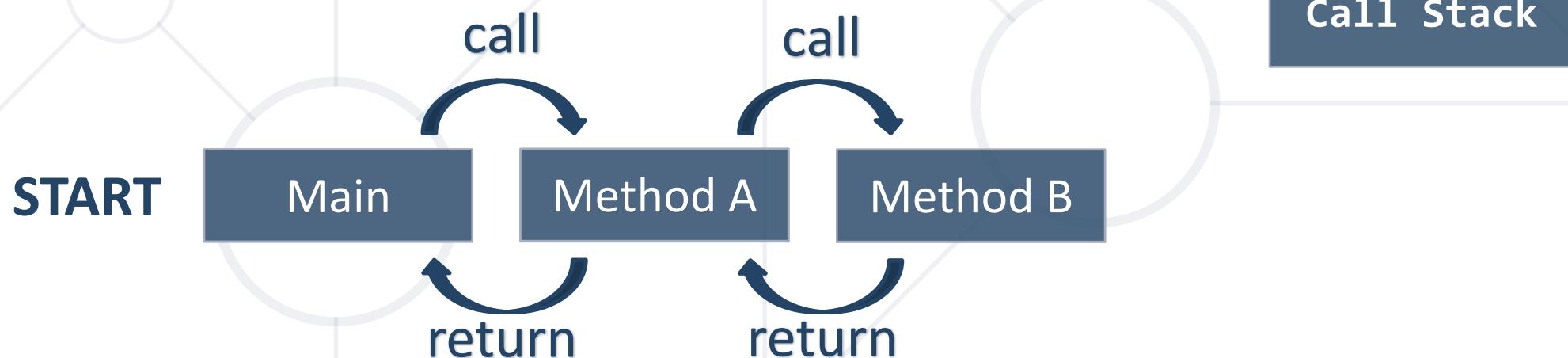
- The program continues, after a method execution completes:

```
public static void main(String[] args) {  
    System.out.println("before method executes");  
    printLogo();  
    System.out.println("after method executes");  
}
```

```
public static void printLogo() {  
    System.out.println("Company Logo");  
    System.out.println("http://www.companywebsite.com");  
}
```

# Program Execution – Call Stack

- "The stack" **stores information** about the **active subroutines** (methods) of a computer program
- Keeps track of **the point** to which each active subroutine should **return control** when it **finishes executing**



# Problem: Multiply Evens by Odds

- Create a program that **multiplies the sum of all even digits** of a number **by the sum of all odd digits** of the same number:
  - Create a method called **getMultipleOfEvensAndOdds()**
  - Create a method **getSumOfEvenDigits()**
  - Create **getSumOfOddDigits()**
  - You may need to use **Math.abs()** for negative numbers

-12345



Evens: 2 4  
Odds: 1 3 5



Even sum: 6  
Odd sum: 9



54



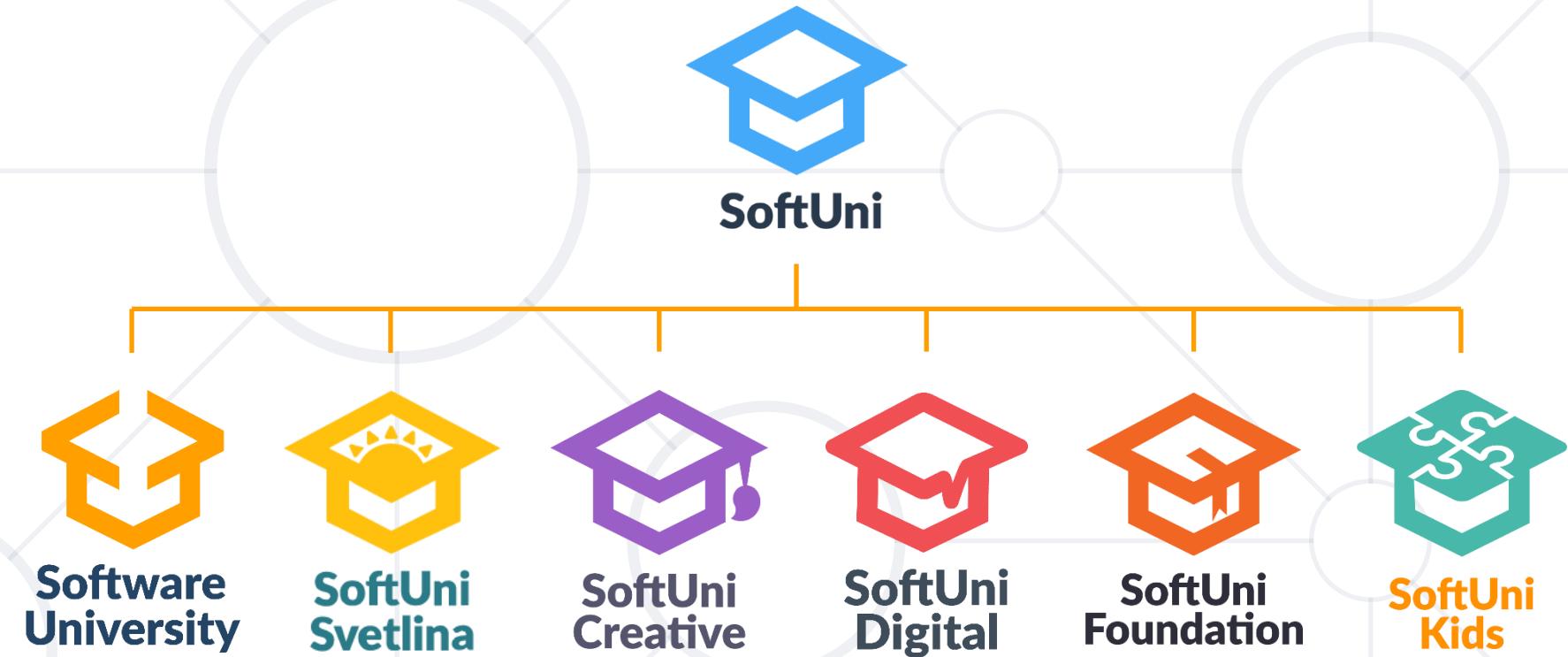
# Live Exercises

# Summary

- Break large programs into simple **methods** that solve small sub-problems
- Methods consist of **declaration** and **body**
- Methods are invoked by their **name + ()**
- Methods can accept **parameters**
- Methods can **return** a value or nothing (**void**)



# Questions?



# SoftUni Diamond Partners

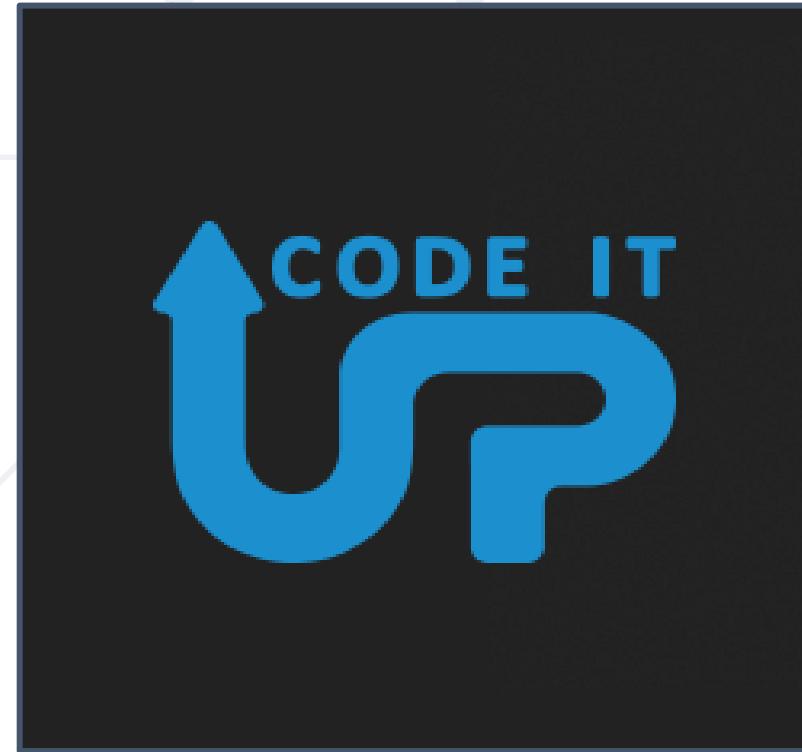


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

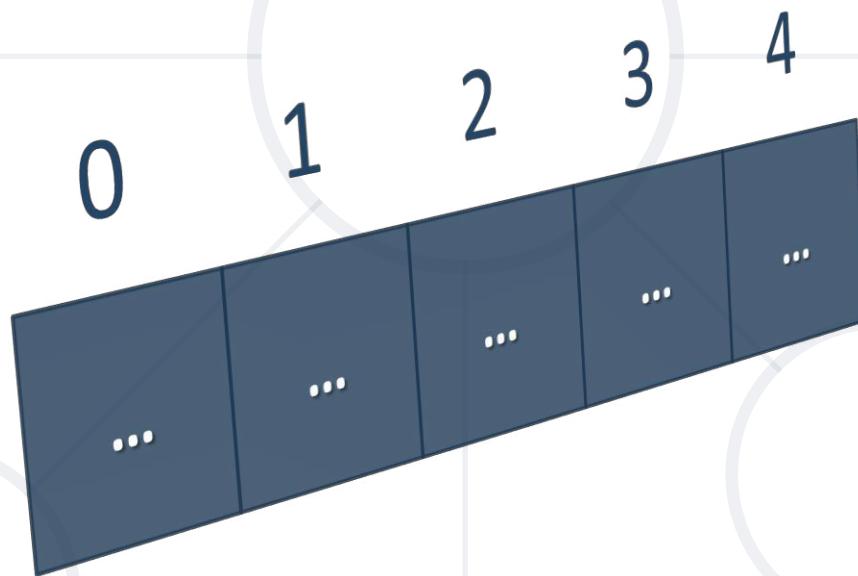


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Lists

Processing Variable-Length Sequences of Elements



SoftUni Team

Technical Trainers

 Software  
University

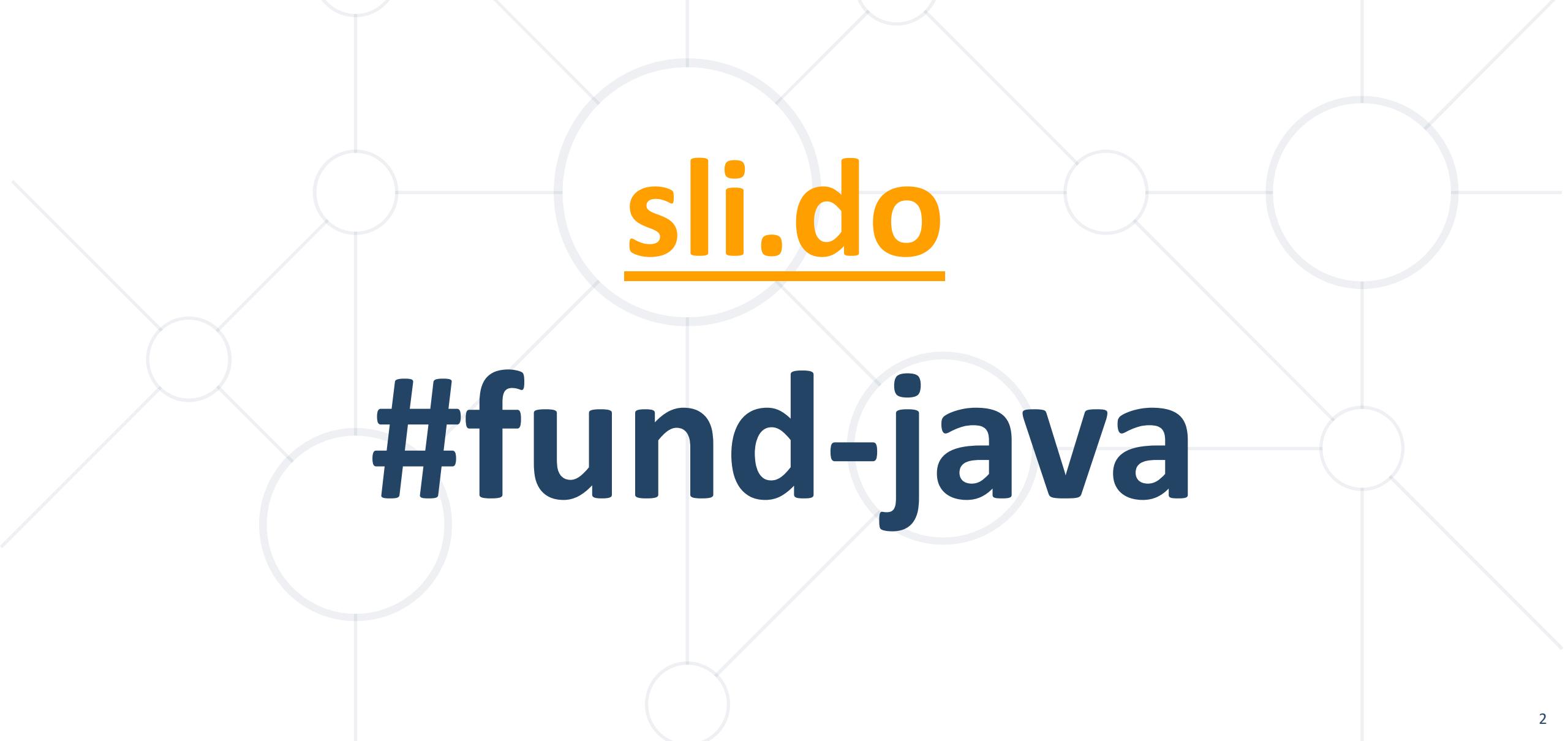


SoftUni

Software University

<https://softuni.bg>

Questions?



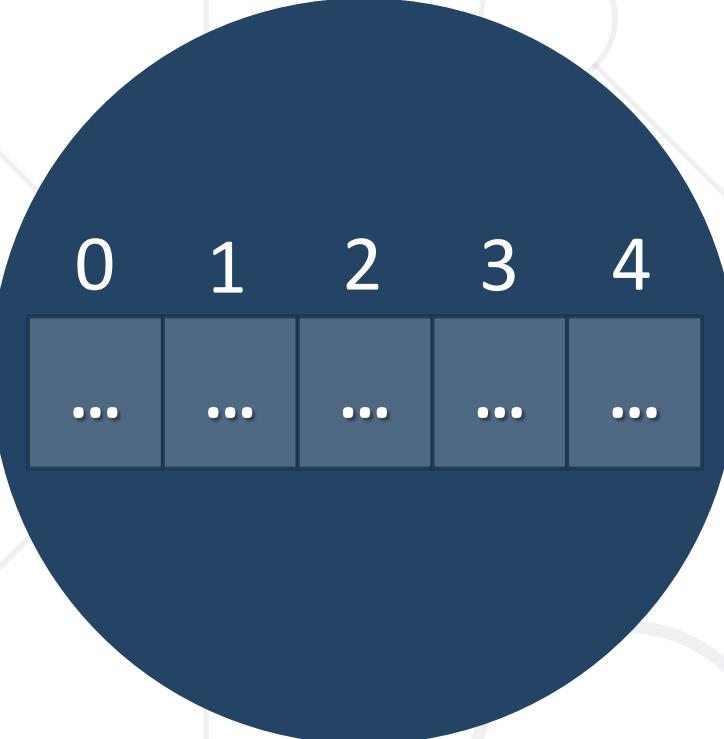
**sli.do**

**#fund-java**

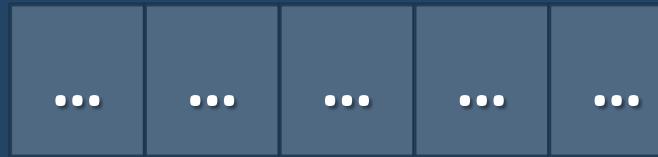
# Table of Contents

1. Lists Overview
2. List Manipulating
3. Reading Lists from the Console
4. Sorting Lists and Arrays





0 1 2 3 4



# Lists

# List<E> – Overview

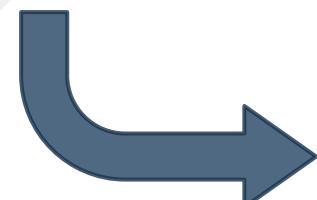
- List<E> holds a list of elements of any type

```
List<String> names = new ArrayList<>();  
//Create a list of strings  
names.add("Peter");  
names.add("Maria");  
names.add("George");  
names.remove("Maria");  
for (String name : names)  
    System.out.println(name);  
//Peter, George
```



# List<E> – Overview (2)

```
List<Integer> nums = new ArrayList<>(  
    Arrays.asList(10, 20, 30, 40, 50, 60));  
  
nums.remove(2); Remove by index  
  
nums.remove(Integer.valueOf(40)); Remove by value  
  
nums.add(100); Inserts an element to index  
  
nums.add(0, -100); Items count  
  
for (int i = 0; i < nums.size(); i++)  
  
    System.out.print(nums.get(i) + " ");
```

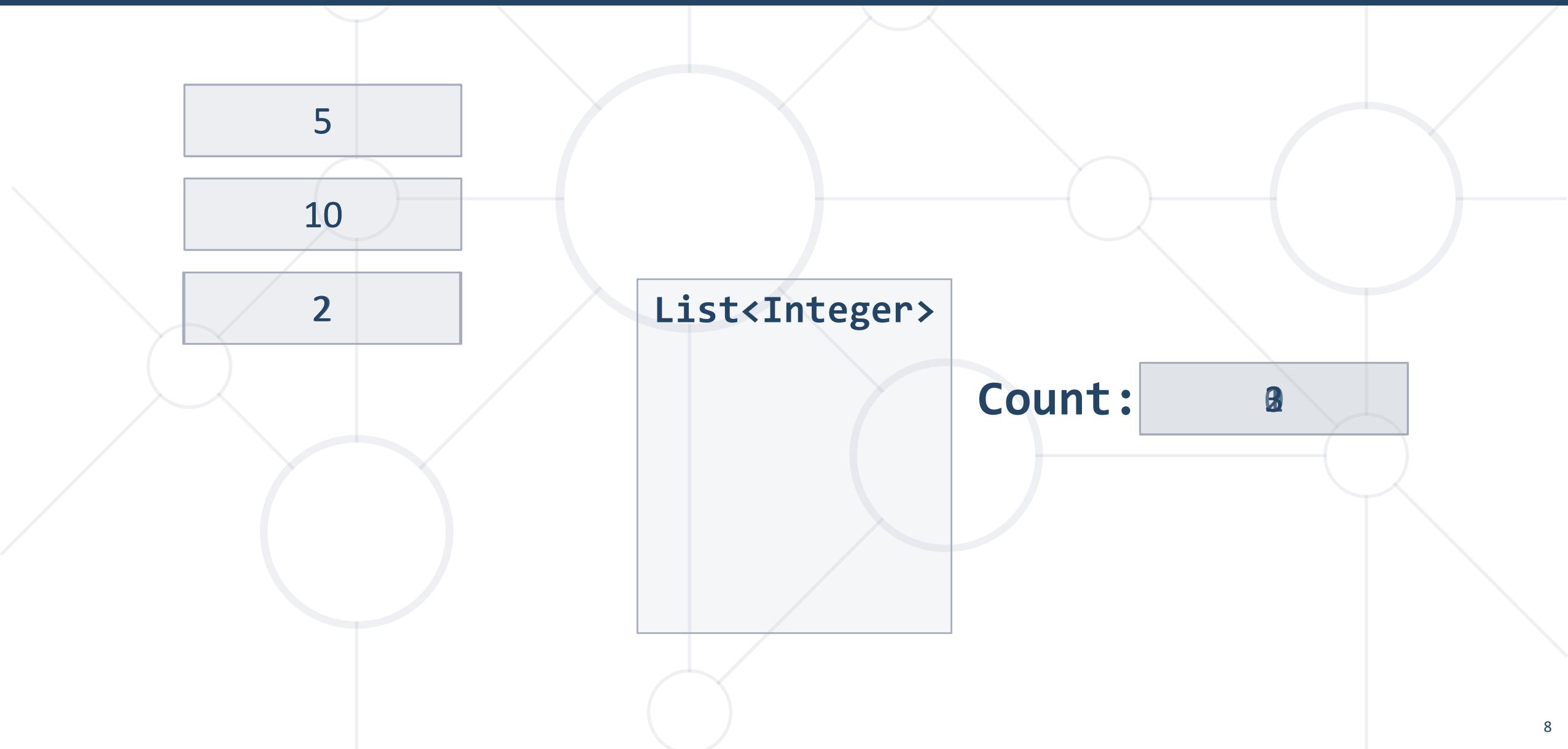


-100 10 20 50 60 100

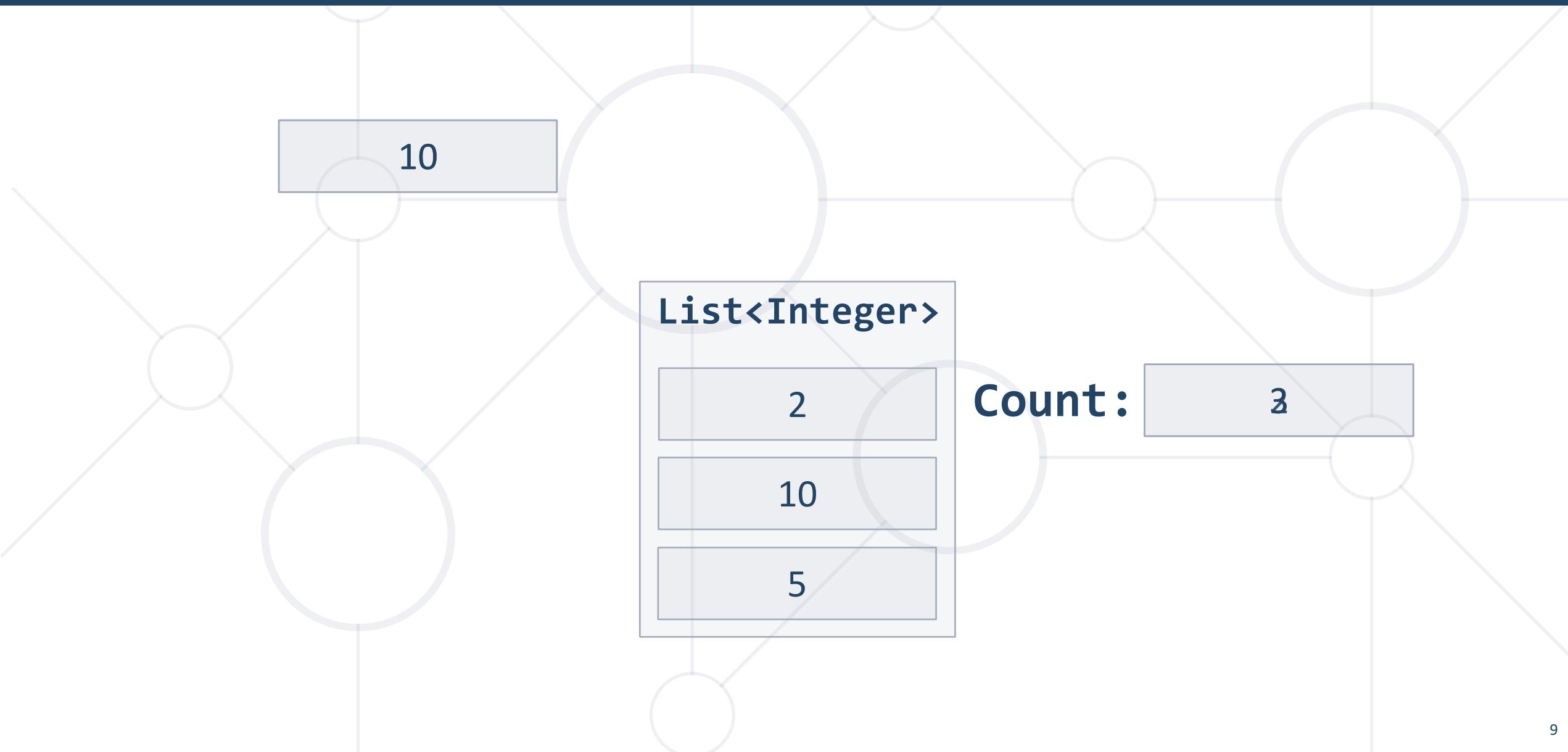
# List<E> – Data Structure

- **List<E>** holds a list of elements (like array, but extendable)
- Provides operations to **add** / **insert** / **remove** / **find** elements:
  - **size()** – number of elements in the List<E>
  - **add(element)** – adds an element to the List<E>
  - **add(index, element)** – inserts an element to given position
  - **remove(element)** – removes an element (returns true / false)
  - **remove(index)** – removes element at index
  - **contains(element)** – determines whether an element is in the list
  - **set(index, item)** – replaces the element at the given index

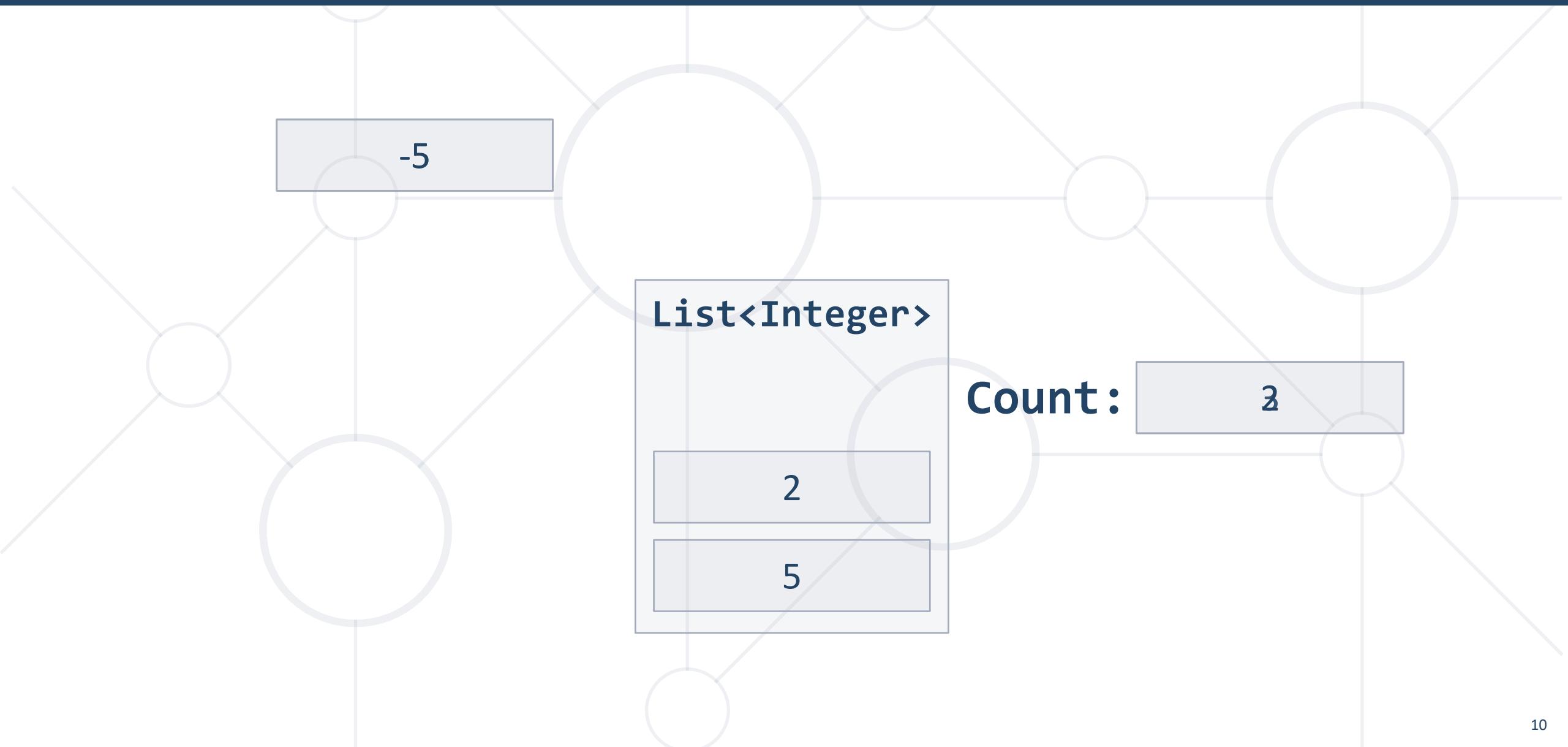
# Add – Appends an Element

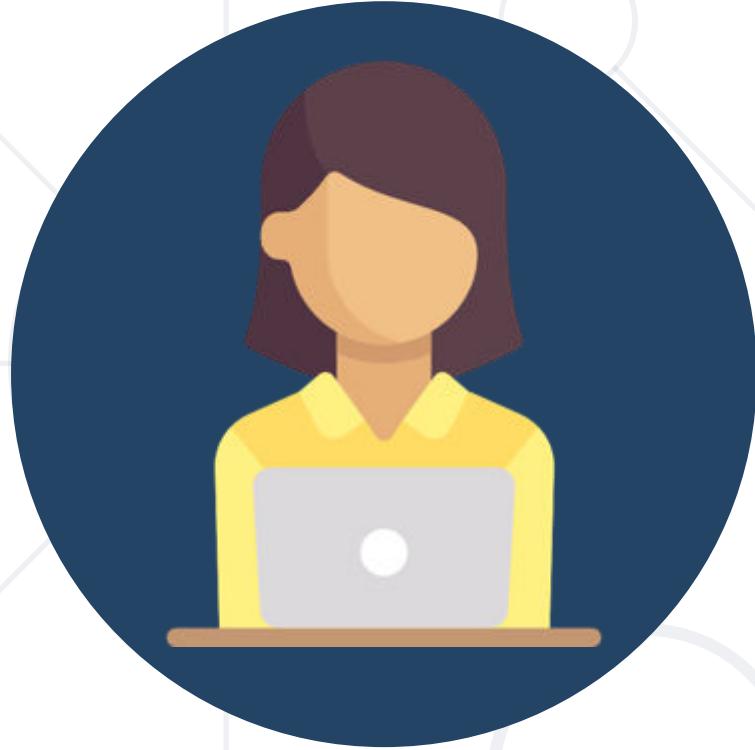


# Remove – Deletes an Element



# Add (Index, El) – Inserts an Element at Position





# Reading Lists from the Console

Using for Loop or String.split()

# Reading Lists from the Console

- First, read from the console the array **length**:

```
Scanner sc = new Scanner(System.in);
int n = Integer.parseInt(sc.nextLine());
```

- Next, create a list of given size **n** and read its **elements**:

```
List<Integer> list = new ArrayList<>();
for (int i = 0; i < n; i++) {
    int number = Integer.parseInt(sc.nextLine());
    list.add(number);
}
```

# Reading List Values from a Single Line

- Lists can be read from a **single line of space separated values**:

```
2 8 30 25 40 72 -2 44 56
```

```
String values = sc.nextLine();
List<String> items = Arrays.stream(values.split(" "))
    .collect(Collectors.toList());
List<Integer> nums = new ArrayList<>();
for (int i = 0; i < items.size(); i++)
    nums.add(Integer.parseInt(items.get(i)));
```

Convert a collection  
into **List**

```
List<Integer> items = Arrays.stream(values.split(" "))
    .map(Integer::parseInt).collect(Collectors.toList());
```

# Printing Lists On the Console

- Printing a list using a **for**-loop:

```
List<String> list = new ArrayList<>(Arrays.asList(  
    "one", "two", "three", "four", "five", "six"));  
for (int index = 0; index < list.size(); index++)  
    System.out.printf  
        ("arr[%d] = %s%n", index, list.get(index));
```

- Printing a list using a **String.join()**:

```
List<String> list = new ArrayList<>(Arrays.asList(  
    "one", "two", "three", "four", "five", "six"));  
System.out.println(String.join(";", list));
```

Gets an element  
at given index

# Problem: Sum Adjacent Equal Numbers

- Write a program to sum all adjacent equal numbers in a list of decimal numbers, starting from left to right
- Examples:

3 3 6 1



12 1

8 2 2 4 8 16



16 8 16

5 4 2 1 1 4



5 8 4

# Solution: Sum Adjacent Equal Numbers (1)

```
Scanner sc = new Scanner(System.in);
List<Double> numbers = Arrays.stream(sc.nextLine().split(" "))
    .map(Double::parseDouble).collect(Collectors.toList());
for (int i = 0; i < numbers.size() - 1; i++)
    if (numbers.get(i).equals(numbers.get(i + 1))) {
        numbers.set(i, numbers.get(i) + numbers.get(i + 1));
        numbers.remove(i + 1);
        i = -1;
    }
//Continue on the next slide
```

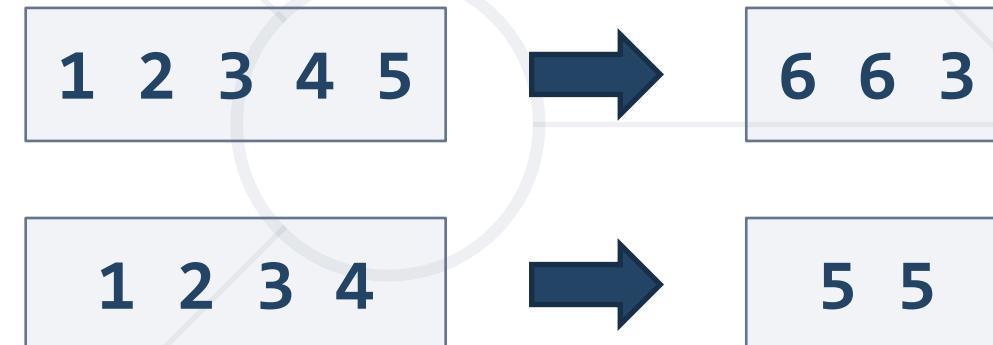
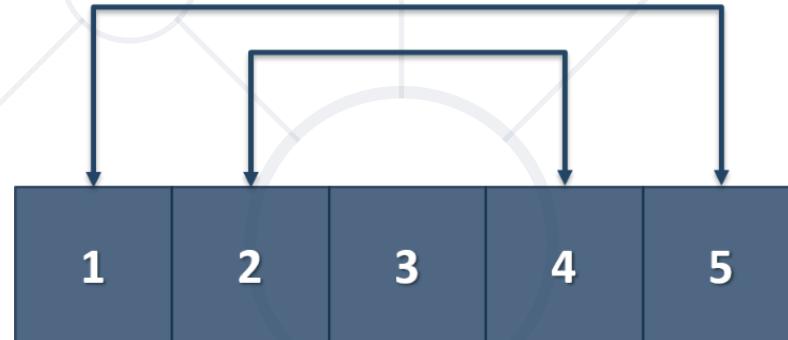
# Solution: Sum Adjacent Equal Numbers (2)

```
String output = joinElementsByDelimiter(numbers, " ");
System.out.println(output);
```

```
static String joinElementsByDelimiter
    (List<Double> items, String delimiter) {
    String output = "";
    for (Double item : items)
        output += (new DecimalFormat("0.#").format(item) + delimiter);
    return output;
}
```

# Problem: Gauss' Trick

- Write a program that sum all numbers in a list in the following order:
  - $\text{first} + \text{last}, \text{first} + 1 + \text{last} - 1, \text{first} + 2 + \text{last} - 2, \dots, \text{first} + n, \text{last} - n$
- Examples:



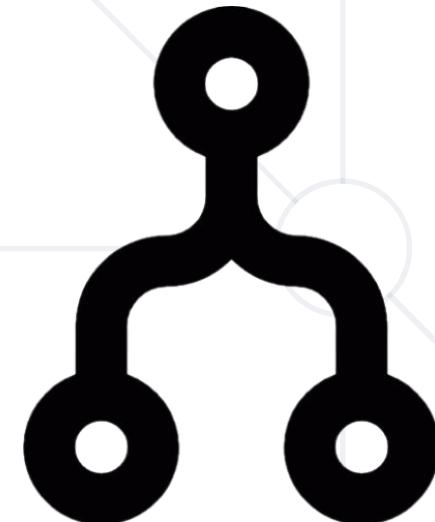
Check your solution here: <https://judge.softuni.org/Contests/1295/>

# Solution: Gauss' Trick

```
Scanner sc = new Scanner(System.in);
List<Integer> numbers = Arrays.stream(sc.nextLine().split(" "))
    .map(Integer::parseInt).collect(Collectors.toList());
int size = numbers.size();
for (int i = 0; i < size / 2; i++) {
    numbers.set(i, numbers.get(i) + numbers.get(numbers.size() - 1));
    numbers.remove(numbers.size() - 1);
}
System.out.println(numbers.toString().replaceAll("[\\\[\\],]", ""));
```

# Problem: Merging Lists

- You receive two lists with numbers. Print a result list which contains the numbers from both of the lists
  - If the length of the two lists is not equal, just add the remaining elements at the end of the list
  - list1[0], list2[0], list1[1], list2[1], ...



Check your solution here: <https://judge.softuni.org/Contests/1295/>

# Solution: Merging Lists (1)

```
//TODO: Read the input

List<Integer> resultNums = new ArrayList<>();

for (int i = 0; i < Math.min(nums1.size(), nums2.size()); i++) {
    //TODO: Add numbers in resultNums
}

if (nums1.size() > nums2.size())
    resultNums.addAll(getRemainingElements(nums1, nums2));
else if (nums2.size() > nums1.size())
    resultNums.addAll(getRemainingElements(nums2, nums1));

System.out.println(resultNums.toString().replaceAll("[\\\[\\],]", ""));
```

# Solution: Merging Lists (2)

```
public static List<Integer> getRemainingElements  
    (List<Integer> longerList, List<Integer> shorterList) {  
    List<Integer> nums = new ArrayList<>();  
    for (int i = shorterList.size(); i < longerList.size(); i++)  
        nums.add(longerList.get(i));  
    return nums;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/1295/>



# Reading and Manipulating Lists

Live Exercises



# Sorting Lists and Arrays

# Sorting Lists

- Sorting a list == reorder its elements incrementally: **Sort()**
  - List items should be **comparable**, e.g. numbers, strings, dates, ...

```
List<String> names = new ArrayList<>(Arrays.asList(  
    "Peter", "Michael", "George", "Victor", "John"));  
Collections.sort(names);  
System.out.println(String.join(", ", names));  
// George, John, Michael, Peter, Victor  
Collections.sort(names);  
Collections.reverse(names);  
System.out.println(String.join(", ", names));  
// Victor, Peter, Michael, John, George
```

Sort in natural  
(ascending) order

Reverse the sorted result

# Problem: List of Products

- Read a number **n** and **n** lines of products. Print a numbered list of all the products ordered by name
- Examples:

```
4
Potatoes
Tomatoes
Onions
Apples
```



```
1.Apples
2.Onions
3.Potatoes
4.Tomatoes
```

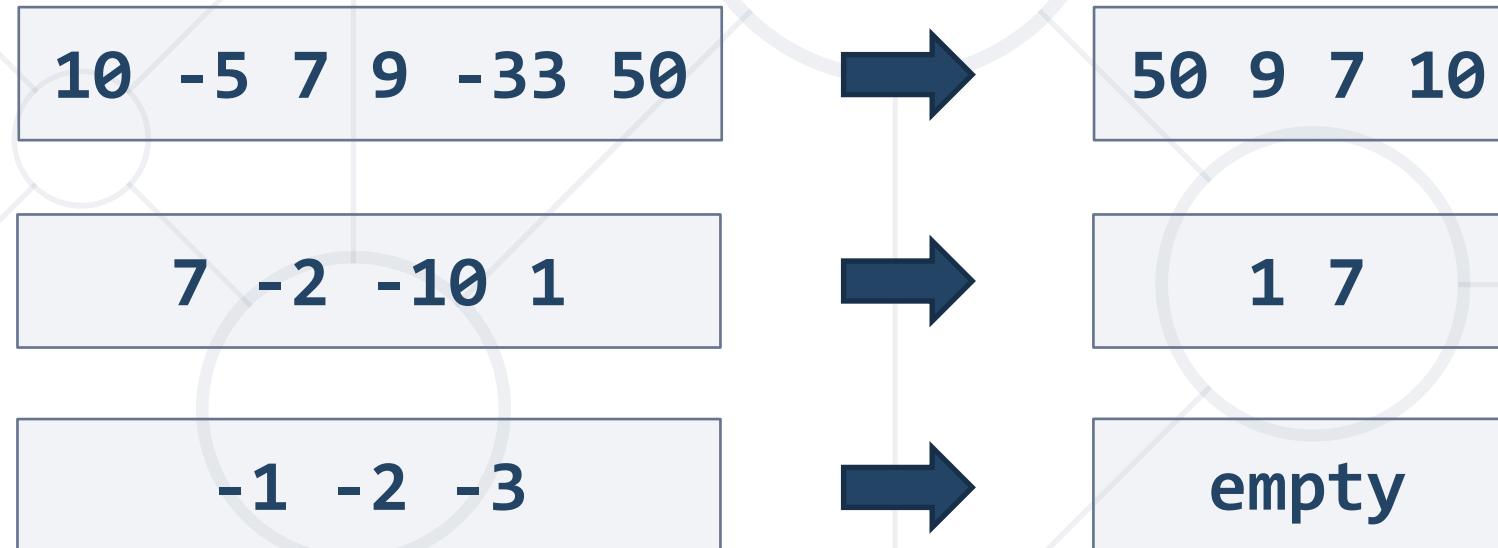
A  
Z

# Solution: List of Products

```
int n = Integer.parseInt(sc.nextLine());  
  
List<String> products = new ArrayList<>();  
  
for (int i = 0; i < n; i++) {  
  
    String currentProduct = sc.nextLine();  
  
    products.add(currentProduct);  
}  
  
Collections.sort(products);  
  
for (int i = 0; i < products.size(); i++)  
  
    System.out.printf("%d.%s%n", i + 1, products.get(i));
```

# Problem: Remove Negatives and Reverse

- Read a list of integers, remove all negative numbers from it
  - Print the remaining elements in reversed order
  - In case of no elements left in the list, print "empty"



Check your solution here: <https://judge.softuni.org/Contests/1295/>

# Solution: Remove Negatives and Reverse

```
List<Integer> nums = Arrays.stream(sc.nextLine().split(" ")).  
    .map(Integer::parseInt).collect(Collectors.toList());  
for (int i = 0; i < nums.size(); i++)  
    if (nums.get(i) < 0)  
        nums.remove(i--);  
Collections.reverse(nums);  
if (nums.size() == 0)  
    System.out.println("empty");  
else  
    System.out.println(nums.toString().replaceAll("[\\\[\\],]",  
""));
```



# Sorting Lists

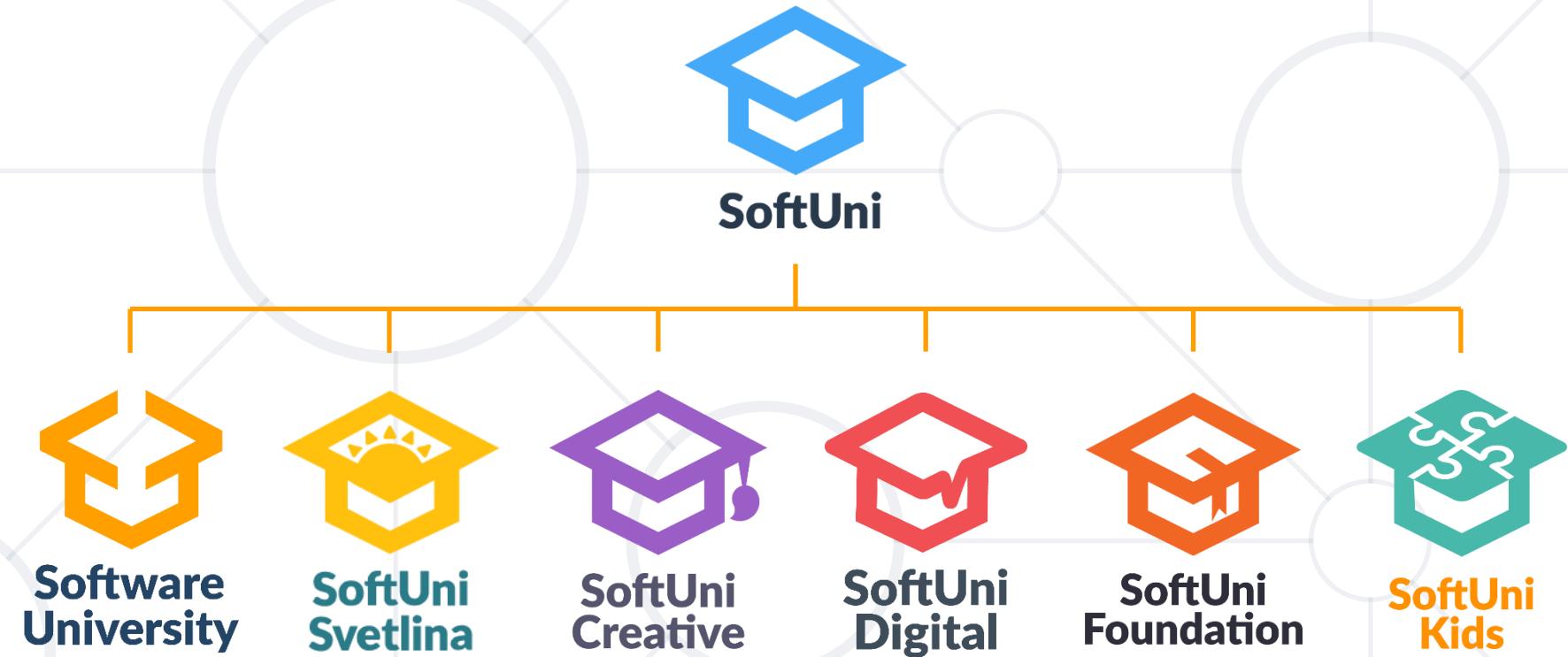
## Live Exercises

# Summary

- Lists hold a sequence of elements (variable-length)
- Can **add** / **remove** / **insert** elements at runtime
- Creating (allocating) a list:  
**`new ArrayList<E>()`**
- Accessing list elements by index
- Printing list elements: **`String.join(...)`**



# Questions?



# SoftUni Diamond Partners

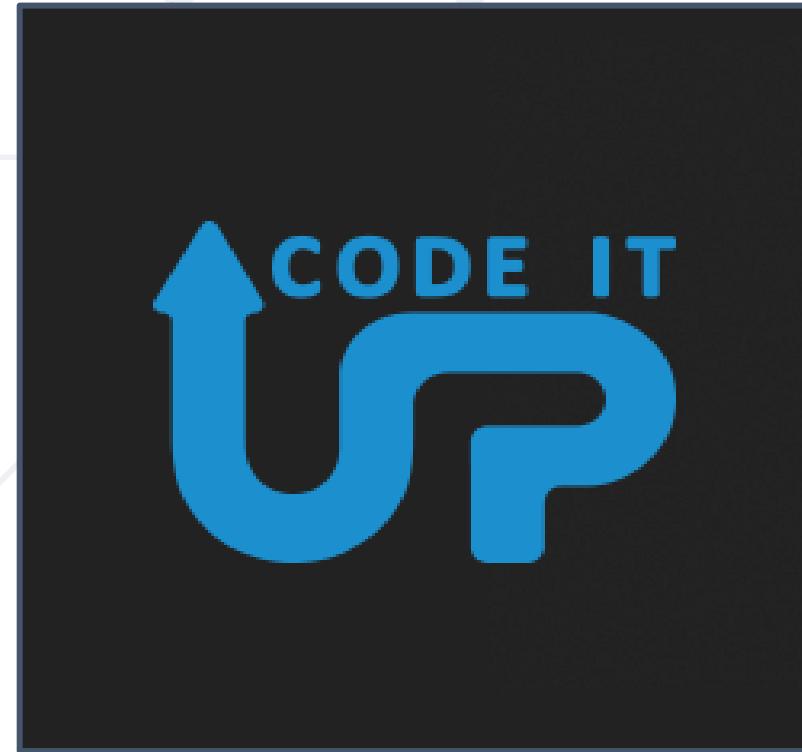


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

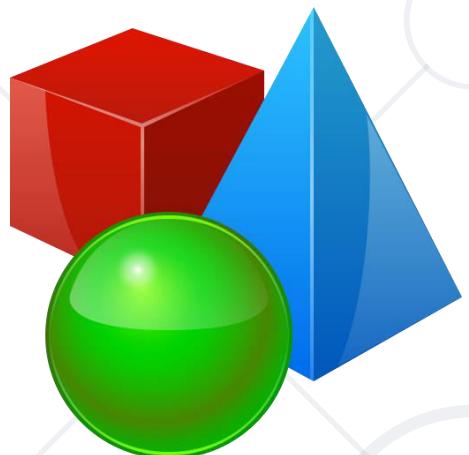


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Objects and Classes

Using Objects and Classes  
Defining Simple Classes



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Have a Question?



**sli.do**

**#fund-java**

# Table of Contents

1. Objects
2. Classes
3. Built in Classes
4. Defining Simple Classes
  - Fields
  - Constructors
  - Methods





# Objects and Classes

# Objects

- An **object** holds a set of named values
  - E.g. **birthday** object holds the day, month, and year
  - Creating a birthday object:



Birthday

day = 27

month = 11

year = 1996

Object  
name

Object  
fields

```
LocalDate birthday =  
    LocalDate.of(2018, 5, 5);  
  
System.out.println(birthday);
```

Create a new object of  
type LocalDate

# Classes

- In programming **classes** provide the structure for **objects**
  - Act as a **blueprint** for **objects** of the same type
- Classes define:
  - **Fields (private variables)**, e.g. day, month, year
  - **Getters/Setters**, e.g. getDay, setMonth, getYear
  - Actions (**behavior**), e.g. plusDays(count), subtract(date)
- Typically, a class has multiple **instances** (objects)
  - Sample class: **LocalDate**
  - Sample objects: **birthdayPeter**, **birthdayMaria**



# Objects – Instances of Classes

- Creating the object of a defined class is called **instantiation**
- The **instance** is the object itself, which is created runtime
- All instances have common **behavior**

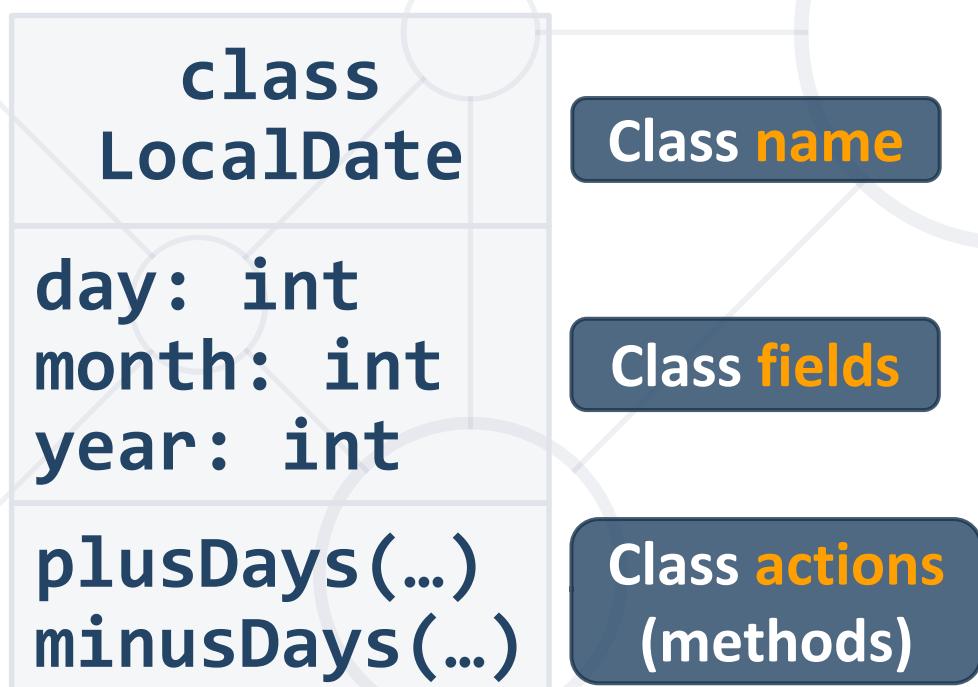
```
LocalDate date1 = LocalDate.of(2018, 5, 5);  
LocalDate date2 = LocalDate.of(2016, 3, 5);  
LocalDate date3 = LocalDate.of(2013, 3, 2);
```



# Classes vs. Objects

- Classes provide structure for creating objects

```
class LocalDate
day: int
month: int
year: int
plusDays(...)
minusDays(...)
```



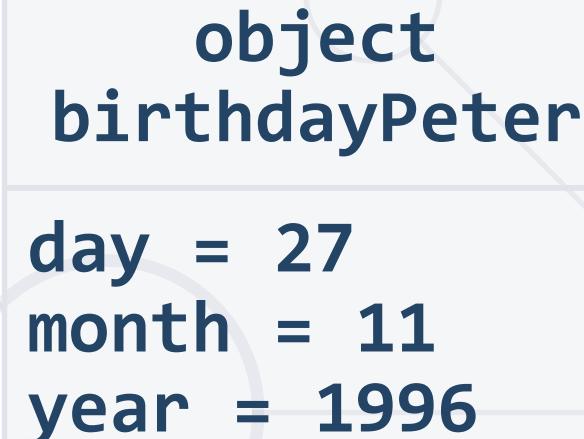
Class name

Class fields

Class actions (methods)

- An object is a single instance of a class

```
object birthdayPeter
day = 27
month = 11
year = 1996
```

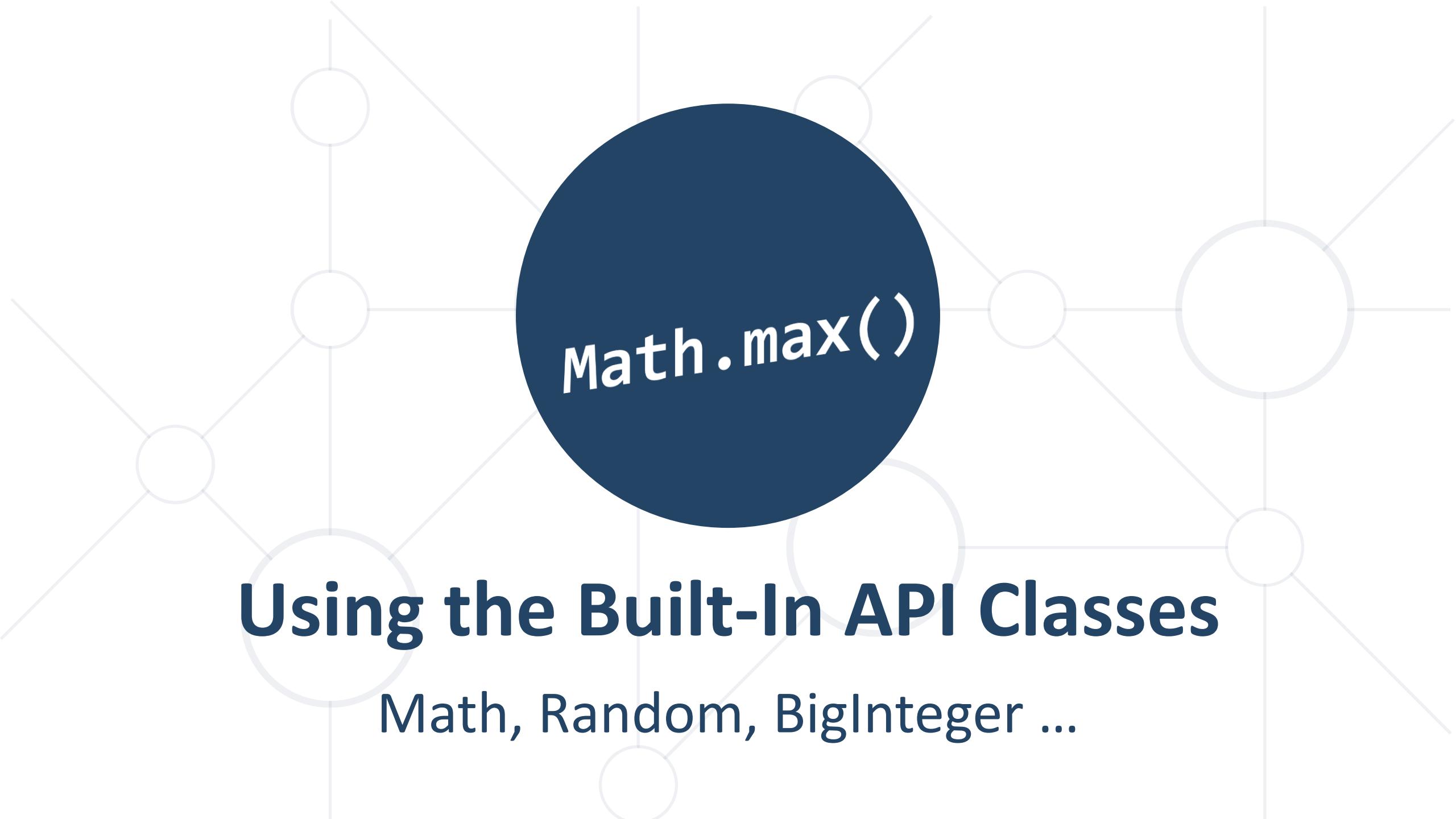


Object name

Object data

Object actions (methods)





`Math.max()`

# Using the Built-In API Classes

Math, Random, BigInteger ...

# Built-In API Classes in Java

- Java provides **ready-to-use** classes:
  - Organized inside Packages like:  
**java.util.Scanner, java.utils.List**, etc.

- Using static class members:

```
LocalDateTime today = LocalDateTime.now();
double cosine = Math.cos(Math.PI);
```

- Using non-static Java classes:

```
Random rnd = new Random();
int randomNumber = rnd.nextInt(99);
```

# Problem: Randomize Words

- You are given a list of words
  - Randomize their order and print each word on a separate line



Note: the output is a sample.  
It should always be different!

# Solution: Randomize Words

```
Scanner sc = new Scanner(System.in);
String[] words = sc.nextLine().split(" ");
Random rnd = new Random();
for (int pos1 = 0; pos1 < words.length; pos1++) {
    int pos2 = rnd.nextInt(words.length);
    //TODO: Swap words[pos1] with words[pos2]
}
System.out.println(String.join(
    System.lineSeparator(), words));
```

# Problem: Big Factorial

- Calculate  $n!$  ( $n$  factorial) for very big  $n$  (e.g. 1000)

$$\begin{array}{r} 5 \\ \times 10 \\ \hline 120 \end{array}$$
$$\begin{array}{r} 10 \\ \times 12 \\ \hline 3628800 \end{array}$$
$$\begin{array}{r} 12 \\ \times 12 \\ \hline 479001600 \end{array}$$

50 → 3041409320171337804361260816606476884437764156  
89605120000000000000

88 → 1854826422573984391147968456455462843802209689  
4939934668442158098688956218402819931910014124  
480450182841663351685120000000000000000000000000000

Check your solution here: <https://judge.softuni.org/Contests/1319/>

# Solution: Big Factorial

```
import java.math.BigInteger;  
...  
int n = Integer.parseInt(sc.nextLine());  
BigInteger f = new BigInteger(String.valueOf(1));  
for (int i = 1; i <= n; i++) {  
    f = f.multiply(BigInteger  
        .valueOf(Integer.parseInt(String.valueOf(i))));  
}  
System.out.println(f);
```

Use the  
java.math.BigInteger

N!



# Defining Classes

Creating Custom Classes

# Defining Simple Classes

- Specification of a given type of objects from the real-world
- **Classes** provide structure for describing and creating objects



Keyword

```
class Dice {  
    ...  
}
```

Class name

Class body

# Naming Classes

- Use **PascalCase** naming
- Use **descriptive** nouns
- Avoid abbreviations (except widely known, e.g. URL, HTTP, etc.)



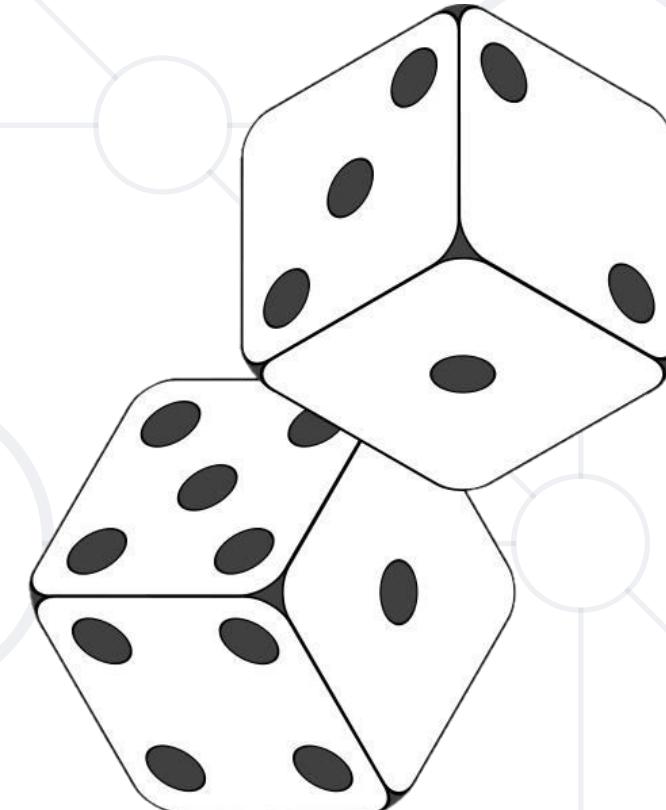
```
class Dice { ... }  
class BankAccount { ... }  
class IntegerCalculator { ... }
```

```
class TPMF { ... }  
class bankaccount { ... }  
class intcalc { ... }
```

# Class Members

- Class is made up of **state** and **behavior**
- Fields **store values**
- Methods **describe behavior**

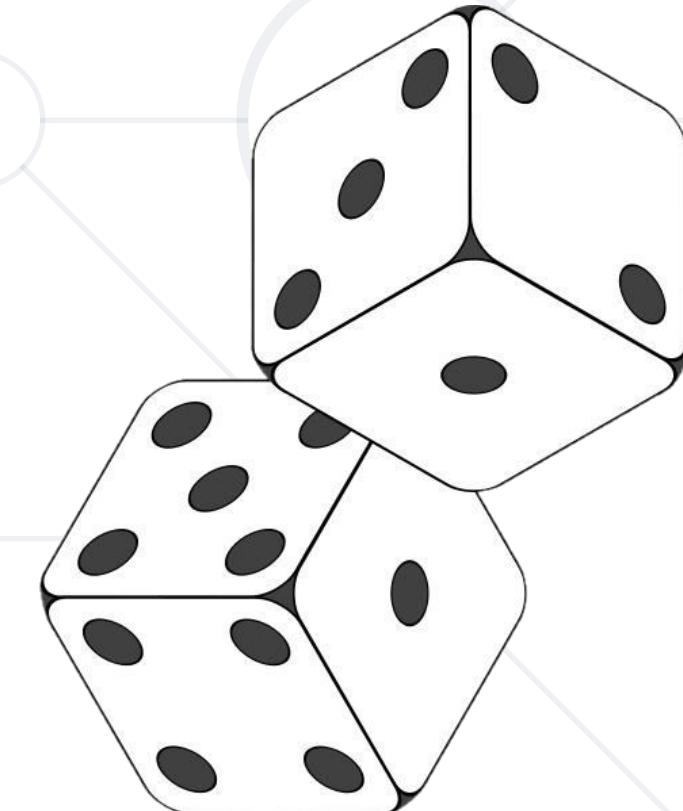
```
class Dice {  
    private int sides; Field  
    public void roll() { ... } Method  
}
```



# Methods

- Store executable code (algorithm)

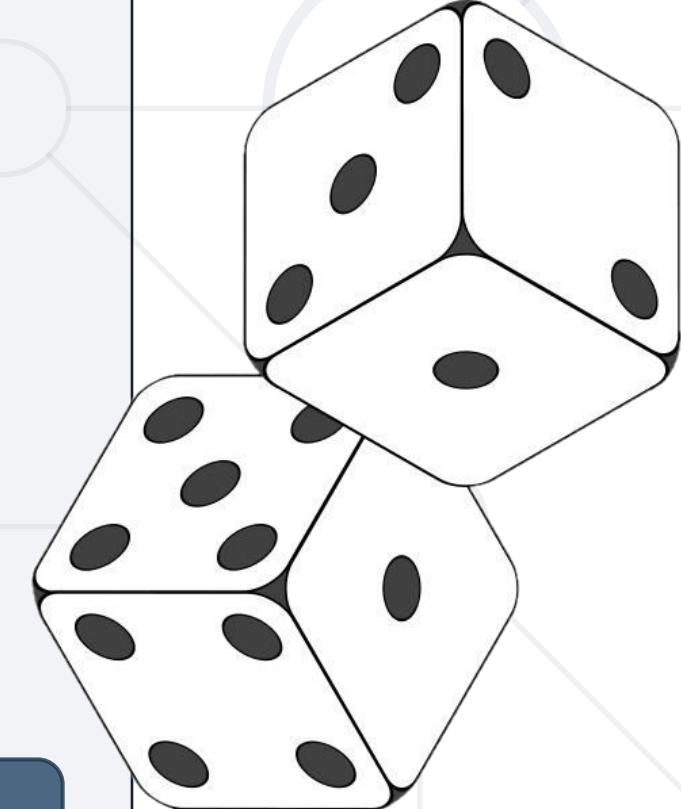
```
class Dice {  
    public int sides;  
    public int roll() {  
        Random rnd = new Random();  
        int sides = rnd.nextInt(this.sides + 1);  
        return sides;  
    }  
}
```



# Getters and Setters

```
class Dice {  
    . . .  
    public int getSides() { return this.sides; }  
    public void setSides(int sides) {  
        this.sides = sides;  
    }  
    public String getType() { return this.type; }  
    public void setType(String type) {  
        this.type = type;  
    }  
}
```

Getters & Setters



# Creating an Object

- A class can have many **instances** (objects)

```
class Program {  
    public static void main(String[] args) {  
        Dice diceD6 = new Dice();  
        Dice diceD8 = new Dice();  
    }  
}
```

Use the **new** keyword

Variable stores a  
**reference**



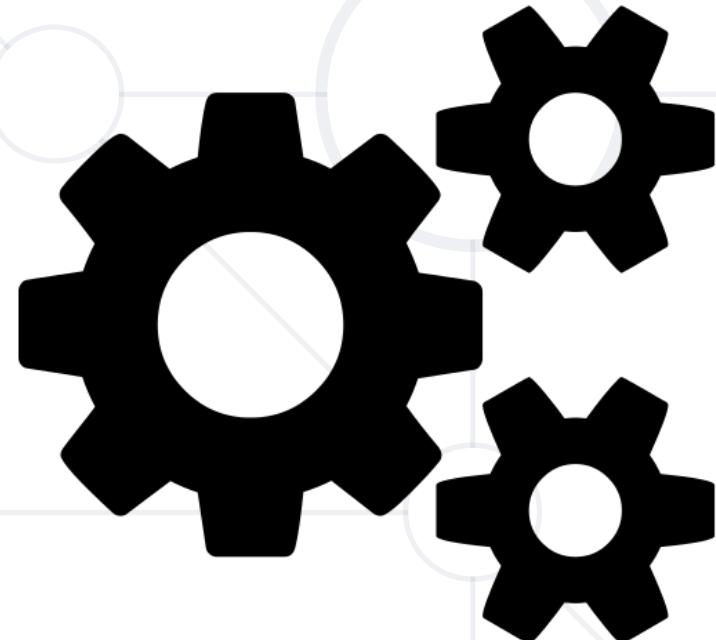
# Constructors

- Special methods, executed during object creation

```
class Dice {  
    public int sides;  
    public Dice() {  
        this.sides = 6;  
    }  
}
```

Overloading default constructor

Constructor name is  
the same as the name  
of the class



# Constructors (2)

- You can have multiple constructors in the same class

```
class Dice {  
    public int sides;  
    public Dice() { }  
    public Dice(int sides) {  
        this.sides = sides;  
    }  
}
```

```
class StartUp {  
    public static void main(String[] args) {  
        Dice dice1 = new Dice();  
        Dice dice2 = new Dice(7);  
    }  
}
```

# Problem: Students

- Read students until you receive "end" in the following format:
  - "{firstName} {lastName} {age} {hometown}"
  - Define a class **Student**, which holds the needed information
  - If you receive a student which already exists (matching **firstName** and **lastName**), overwrite the information
- After the end command, you will receive a city name
- Print students which are from the given city in the format:  
" {firstName} {lastName} is {age} years old."

# Solution: Students (1)

```
public Student(String firstName, String lastName,  
              int age, String city){  
  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
    this.city = city;  
  
    // TODO: Implement Getters and Setters  
}
```

# Solution: Students (2)

```
List<Student> students = new ArrayList<>();
String line;
while (!line.equals("end")) {
    // TODO: Extract firstName, lastName, age, city from the input
    Student existingStudent = getStudent(students, firstName, lastName);
    if(existingStudent != null) {
        existingStudent.setAge(age);
        existingStudent.setCity(city);
    } else {
        Student student = new Student(firstName, lastName, age, city);
        students.add(student);
    }
    line = sc.nextLine();
}
```

# Solution: Students (3)

```
static Student getStudent(List<Student> students, String firstName,  
                        String lastName) {  
  
    for (Student student : students){  
        if(student.getFirstName().equals(firstName)  
            && student.getLastName().equals(lastName))  
            return student;  
    }  
  
    return null;  
}
```



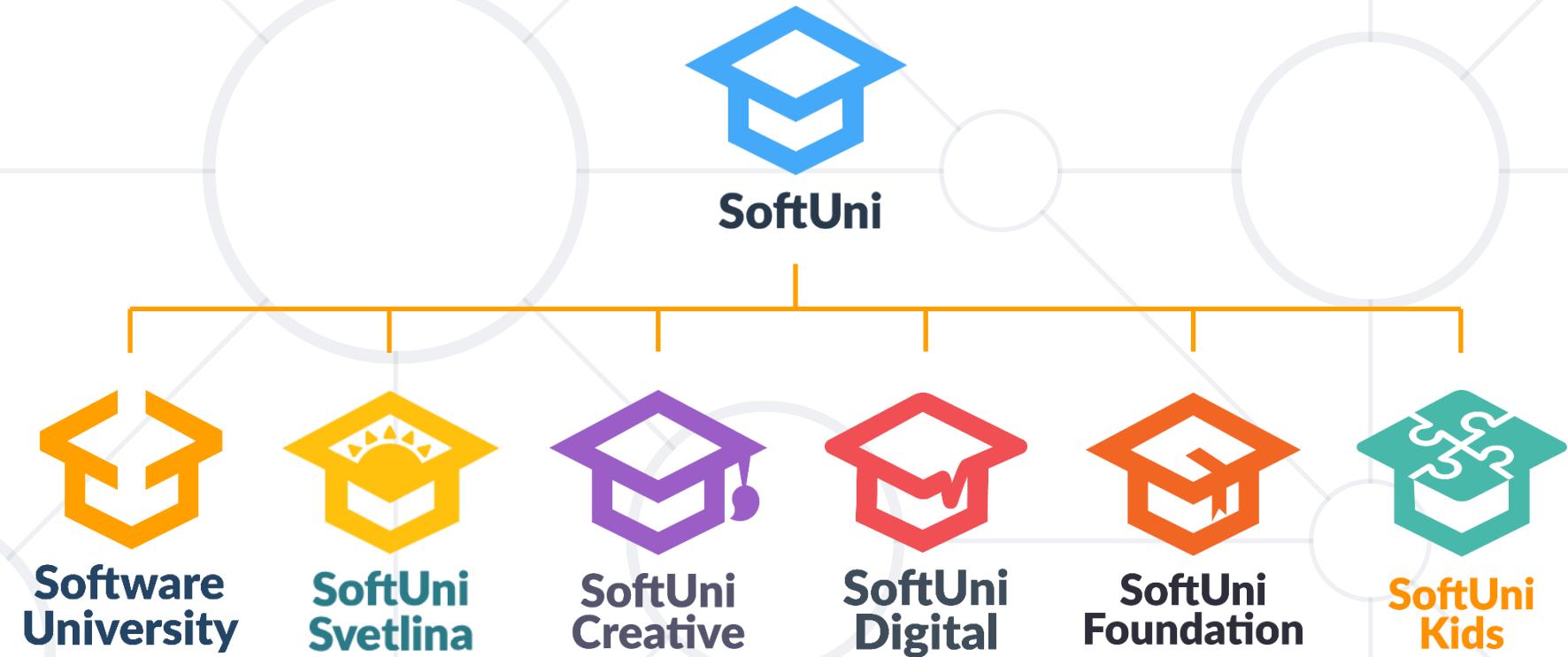
# Live Exercises

# Summary

- Classes define templates for objects
  - Fields
  - Constructors
  - Methods
- Objects
  - Hold a set of **named values**
  - **Instance** of a class



# Questions?



# SoftUni Diamond Partners



**SCHWARZ**



Coca-Cola HBC  
Bulgaria



**Postbank**

Решения за твоето упре



**Bosch.IO**



**SmartIT**



**POKERSTARS**



**CAREERS**



**AMBITIONED**



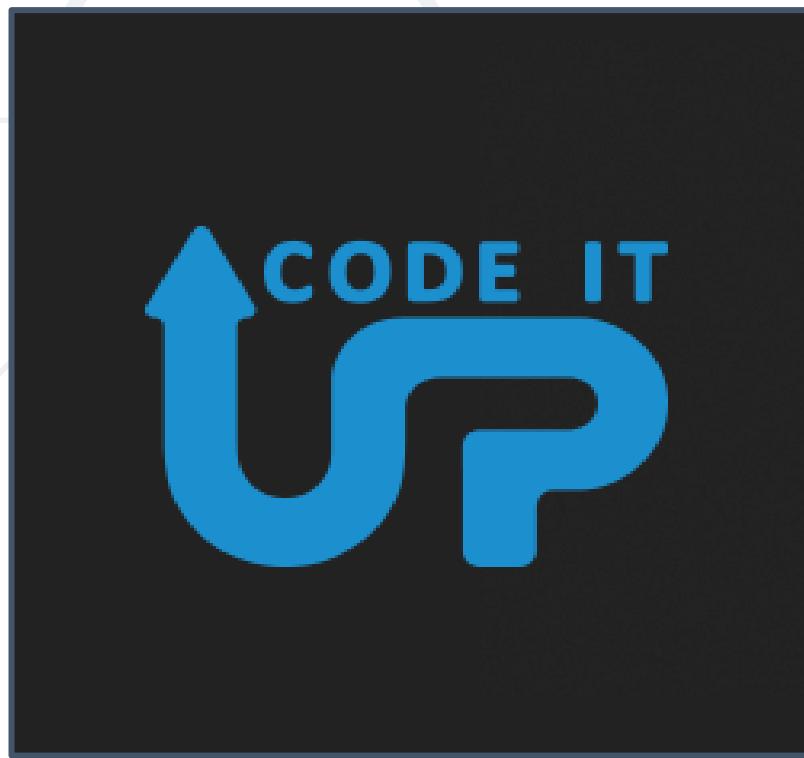
**INDEAVR**  
Serving the high achievers



**createX**

**SUPER  
HOSTING  
.BG**

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Associative Arrays, Lambda and Stream API

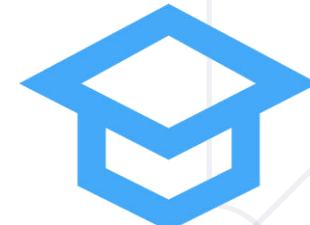
## Collections and Queries



SoftUni Team

Technical Trainers

 Software  
University



SoftUni



Software University

<https://softuni.bg>

Questions?

sli.do

#fund-java

# Table of Contents

## 1. Associative Arrays

- `HashMap <key, value>`
- `LinkedHashMap <key, value>`
- `TreeMap <key, value>`

## 2. Lambda

## 3. Stream API

- Filtering
- Mapping
- Ordering





# Associative Arrays

Collection of Key and Value Pairs

# Associative Arrays (Maps)

- Associative arrays are arrays indexed by **keys**
  - Not by the numbers 0, 1, 2, ... (like arrays)
- Hold a set of pairs **{key → value}**



Key	Value
John Smith	+1-555-8976
Lisa Smith	+1-555-1234
Sam Doe	+1-555-5030

# Collections of Key and Value Pairs

- **HashMap<K, V>**
  - Keys are **unique**
  - Uses a **hash-table + list**
- **LinkedHashMap<K, V>**
  - Keys are **unique**
  - Keeps the keys in **order of addition**
- **TreeMap<K, V>**
  - Keys are **unique**
  - Keeps its **keys always sorted**
  - Uses a **balanced search tree**



# Built-In Methods

- **put(key, value)** method

```
HashMap<String, Integer> airplanes = new HashMap<>();  
airplanes.put("Boeing 737", 130);  
airplanes.put("Airbus A320", 150);
```

- **remove(key)** method

```
HashMap<String, Integer> airplanes = new HashMap<>();  
airplanes.put("Boeing 737", 130);  
airplanes.remove("Boeing 737");
```

# Built-In Methods (2)

- **containsKey(key)**

```
HashMap<String, Integer> map = new HashMap<>();  
map.put("Airbus A320", 150);  
if (map.containsKey("Airbus A320"))  
    System.out.println("Airbus A320 key exists");
```

- **containsValue(value)**

```
HashMap<String, Integer> map = new HashMap<>();  
map.put("Airbus A320", 150);  
System.out.println(map.containsValue(150)); //true  
System.out.println(map.containsValue(100)); //false
```

# HashMap: Put()

Peter	0881-123-987
George	0881-123-789
Alice	0881-123-978

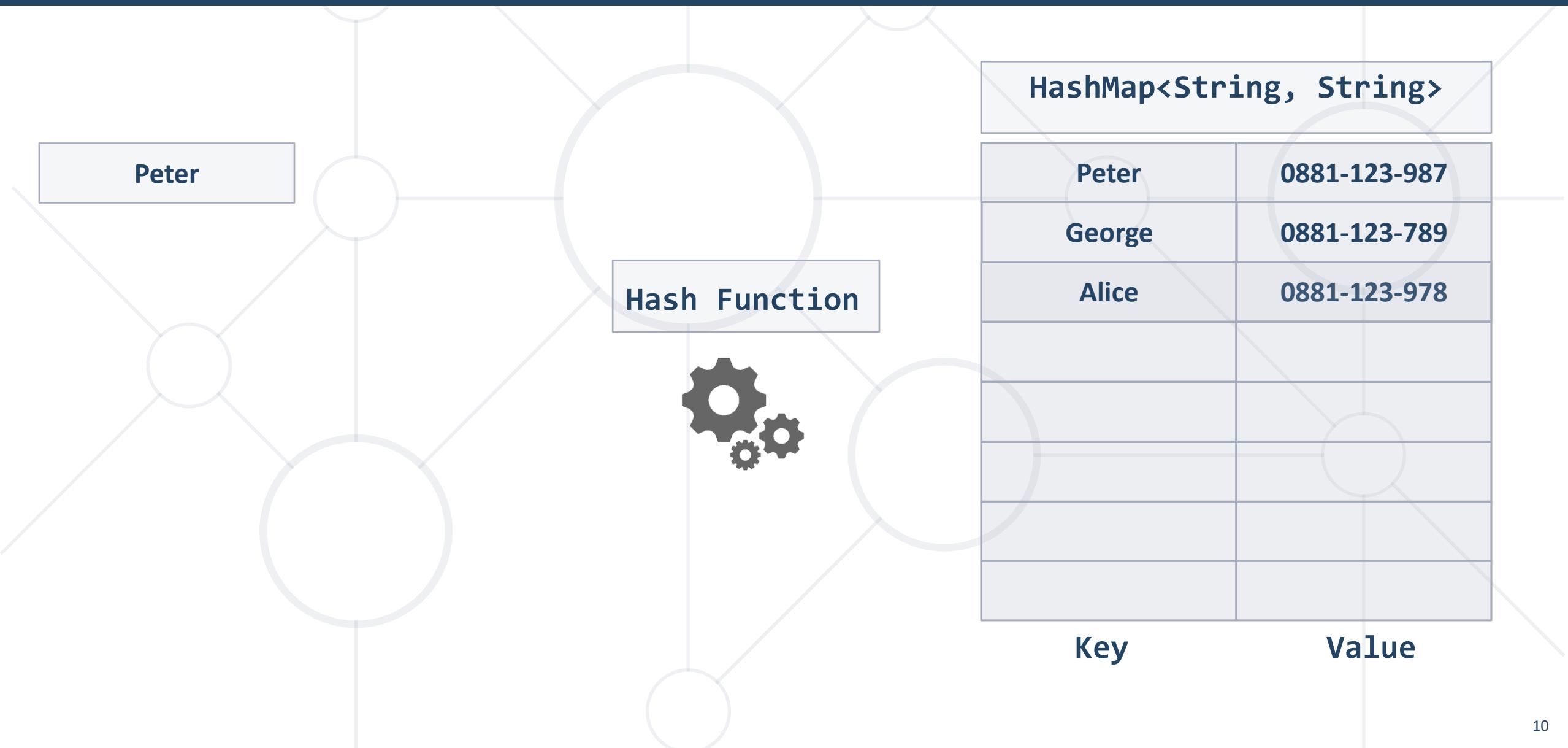
Hash Function



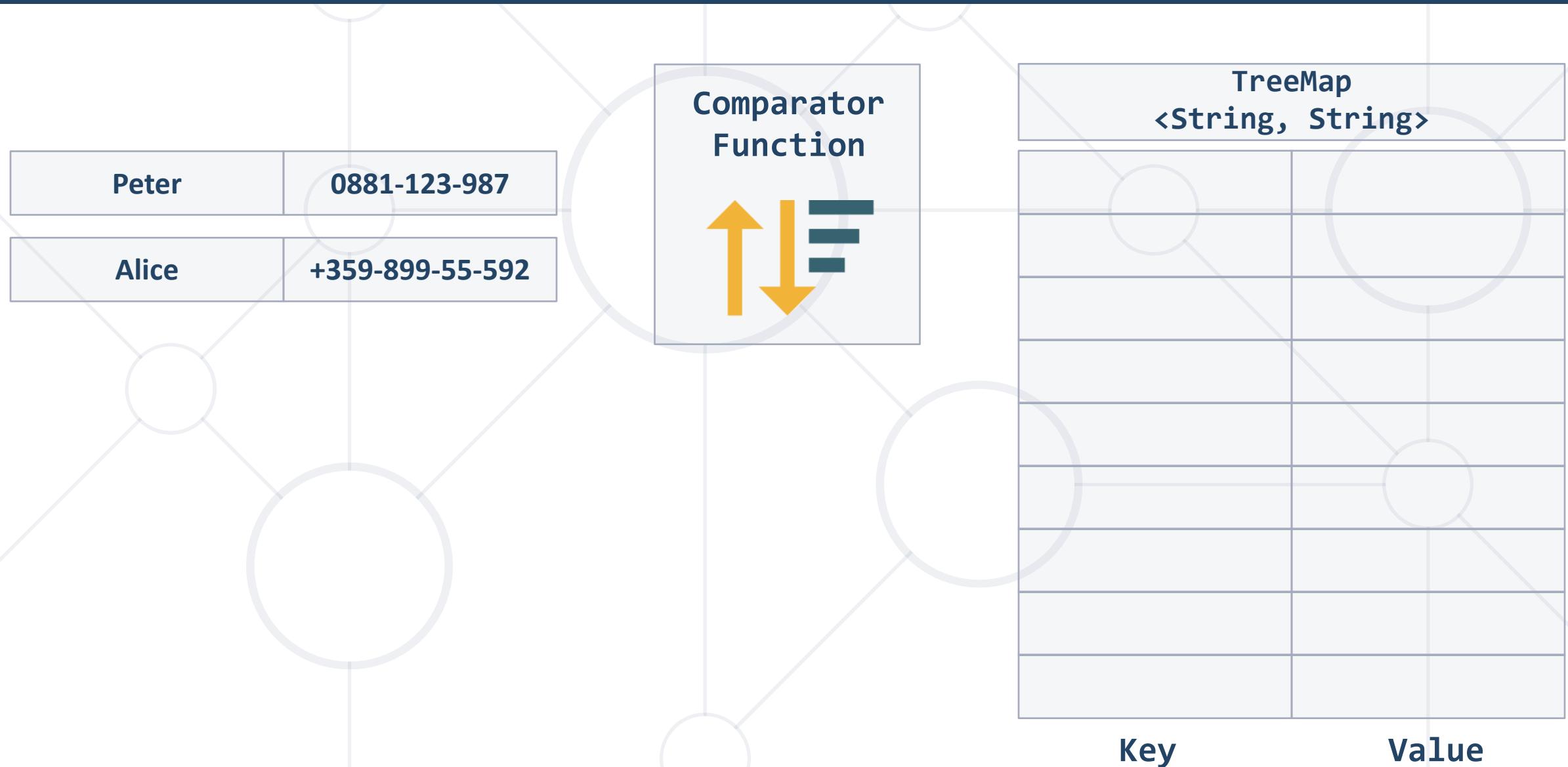
Key

Value

# HashMap: Remove()



# TreeMap<K, V> – Example



# Iterating Through Map

- Iterate through objects of type **Map.Entry<K, V>**
- Cannot modify the collection (**read-only**)

```
Map<String, Double> fruits = new LinkedHashMap<>();  
fruits.put("banana", 2.20);  
fruits.put("kiwi", 4.50);  
for (Map.Entry<K, V> entry : fruits.entrySet()) {  
    System.out.printf("%s -> %.2f%n",  
                      entry.getKey(), entry.getValue());  
}
```

entry.getKey() -> fruit name  
entry.getValue() -> fruit price

# Problem: Count Real Numbers

- Read a list of real numbers and print them in ascending order along with their number of occurrences

8 2 2 8 2



2 -> 3  
8 -> 2

1 5 1 3



1 -> 2  
3 -> 1  
5 -> 1

Check your solution here: <https://judge.softuni.org/Contests/1311/>

# Solution: Count Real Numbers

```
double[] nums = Arrays.stream(sc.nextLine().split(" ")).  
                      .mapToDouble(Double::parseDouble).toArray();  
  
Map<Double, Integer> counts = new TreeMap<>();  
  
for (double num : nums) {  
    if (!counts.containsKey(num))  
        counts.put(num, 0);  
    counts.put(num, counts.get(num) + 1);  
}  
  
for (Map.Entry<Double, Integer> entry : counts.entrySet()) {  
    DecimalFormat df = new DecimalFormat("#.#####");  
    System.out.printf("%s -> %d%n", df.format(entry.getKey()), entry.getValue());  
}
```

Overwrite  
the value

Check your solution here: <https://judge.softuni.org/Contests/1311/>

# Problem: Words Synonyms

- Read **2 \* N** lines of pairs **word** and **synonym**
- Each word may have **many** synonyms

3
cute
adorable
cute
charming
smart
clever

cute - adorable, charming  
smart - clever

# Solution: Word Synonyms

```
int n = Integer.parseInt(sc.nextLine());  
Map<String, ArrayList<String>> words = new LinkedHashMap<>();  
for (int i = 0; i < n; i++) {  
    String word = sc.nextLine();  
    String synonym = sc.nextLine();  
    words.putIfAbsent(word, new ArrayList<>());  
    words.get(word).add(synonym);  
}  
  
//TODO: Print each word and synonyms
```

Adding the key if  
it does not exist



# Lambda Expressions

Anonymous Functions

# Lambda Functions

- A lambda expression is an anonymous function containing expressions and statements

```
(a -> a > 5)
```

- Lambda expressions
- Use the lambda operator `->`
  - Read as "**goes to**"
- The **left** side specifies the **input** parameters
- The **right** side holds the **expression or statement**



# Lambda Functions

- Lambda functions are **inline methods** (functions) that take input parameters and return values:

```
x -> x / 2
```



```
static int func(int x) { return x / 2; }
```

```
x -> x != 0
```



```
static boolean func(int x) { return x != 0; }
```

```
() -> 42
```



```
static int func() { return 42; }
```



# Stream API

Traversing and Querying Collections

# Processing Arrays with Stream API (1)

- **min()** - finds the **smallest** element in a collection:

```
int min = Arrays.stream(new int[]{15, 25, 35}).min().getAsInt();
```

15

```
int min = Arrays.stream(new int[]{15, 25, 35}).min().orElse(2);
```

```
int min = Arrays.stream(new int[]{}).min().orElse(2); // 2
```

- **max()** - finds the **largest** element in a collection:

```
int max = Arrays.stream(new int[]{15, 25, 35}).max().getAsInt();
```

35

# Processing Arrays with Stream API (2)

- **sum()** - finds the **sum** of all elements in a collection:

```
int sum = Arrays.stream(new int[]{15, 25, 35}).sum();
```

75

- **average()** - finds the **average** of all elements:

```
double avg = Arrays.stream(new int[]{15, 25, 35})  
    .average().getAsDouble();
```

25.0

# Processing Collections with Stream API (1)

```
ArrayList<Integer> nums = new ArrayList<>() {  
    add(15); add(25); add(35);  
};
```

- **min()**

```
int min = nums.stream().mapToInt(Integer::intValue)  
    .min().getAsInt();
```

15

```
int min = nums.stream()  
    .min(Integer::compareTo).get();
```

# Processing Collections with Stream API (2)

## ■ max()

```
int max = nums.stream().mapToInt(Integer::intValue)  
    .max().getAsInt();
```

35

```
int max = nums.stream()  
    .max(Integer::compareTo).get();
```

## ■ sum()

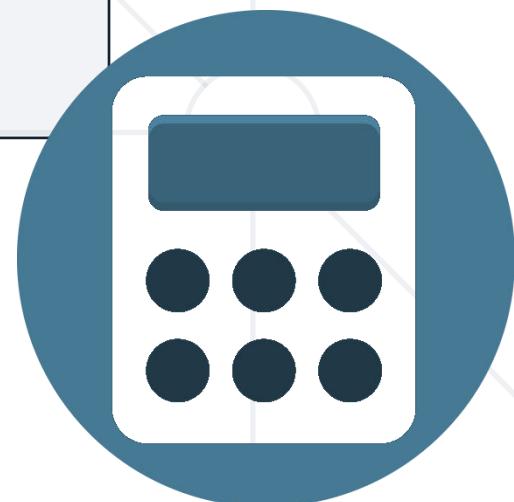
```
int sum = nums.stream()  
    .mapToInt(Integer::intValue).sum();
```

75

# Processing Collections with Stream API (3)

## ■ **average()**

```
double avg = nums.stream()  
    .mapToInt(Integer::intValue)  
    .average()  
    .getAsDouble(); 25.0
```



# Manipulating Collections

- **map()** - manipulates elements in a collection:

```
int[] nums = Arrays.stream(sc.nextLine().split(" "))  
    .mapToInt(e -> Integer.parseInt(e))  
    .toArray();
```

Parse each  
element to  
Integer

```
String[] words = {"abc", "def", "geh", "yyy"};  
  
words = Arrays.stream(words)  
    .map(w -> w + "yyy")  
    .toArray(String[]::new);  
  
//abcyyy, defyyy, gehyyy, yyyyyy
```

# Converting Collections

- Using **toArray()**, **toList()** to convert collections:

```
int[] nums = Arrays.stream(sc.nextLine().split(" "))

    .mapToInt(e -> Integer.parseInt(e))

    .toArray();
```

```
List<Integer> nums = Arrays.stream(sc.nextLine())

    .split(" "))

    .map(e -> Integer.parseInt(e))

    .collect(Collectors.toList());
```

# Filtering Collections

- Using **filter()**

```
int[] nums = Arrays.stream(sc.nextLine().split(" "))  
    .mapToInt(e -> Integer.parseInt(e))  
    .filter(n -> n > 0)  
    .toArray();
```



# Problem: Word Filter

- Read a string array
- Print only words which length is even

kiwi orange banana apple



kiwi  
orange  
banana

pizza cake pasta chips



cake

Check your solution here: <https://judge.softuni.org/Contests/1311/>

# Solution: Word Filter

```
String[] words = Arrays.stream(sc.nextLine().split(" "))

    .filter(w -> w.length() % 2 == 0)

    .toArray(String[]::new);

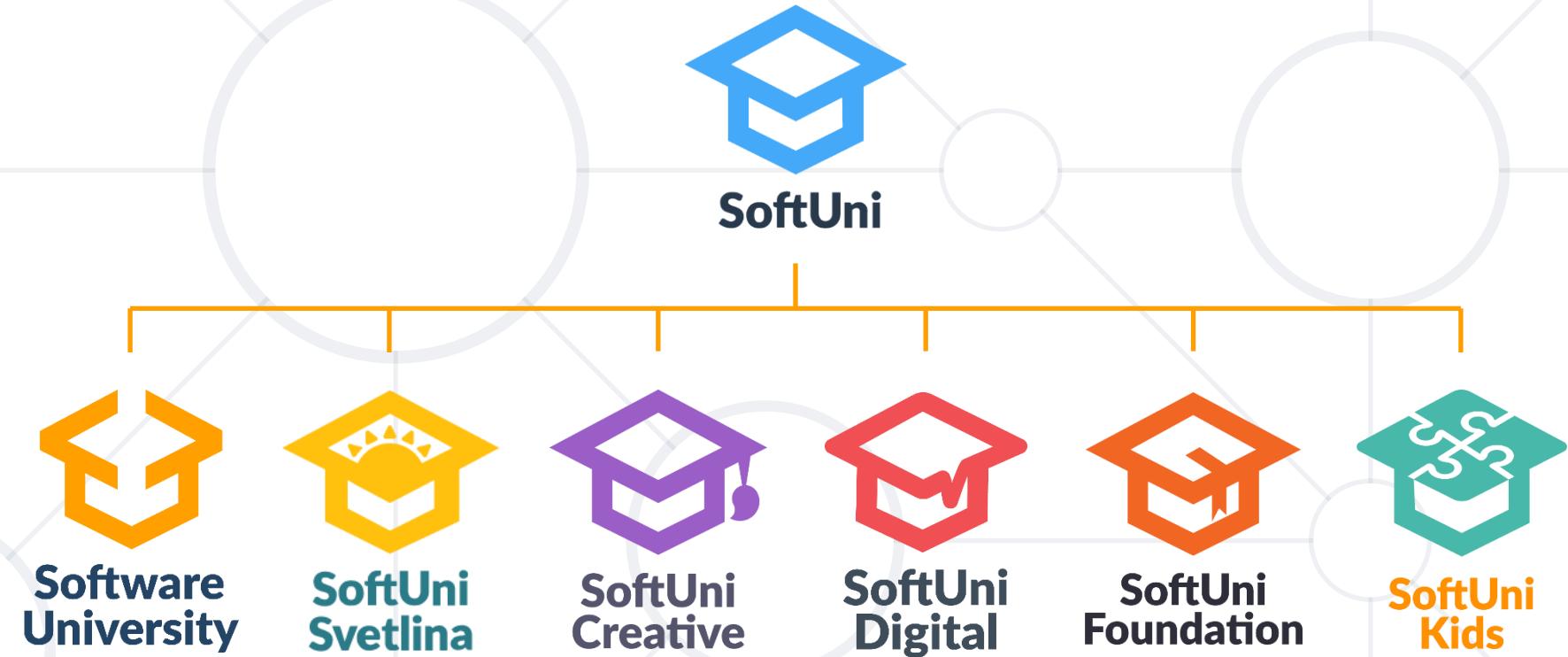
for (String word : words) {
    System.out.println(word);
}
```

Check your solution here: <https://judge.softuni.org/Contests/1311/>

- Maps hold **{key → value}** pairs
  - **Keyset** holds a set of **unique keys**
  - **Values** hold a collection of values
  - Iterating over a map takes the entries as **Map.Entry<K, V>**
- **Lambda** and **Stream API** help collection processing



# Questions?



# SoftUni Diamond Partners



**SCHWARZ**



Coca-Cola HBC  
Bulgaria



**Postbank**

Решения за твоето упре



**Bosch.IO**



**SmartIT**



**POKERSTARS**



**CAREERS**



**AMBITIONED**



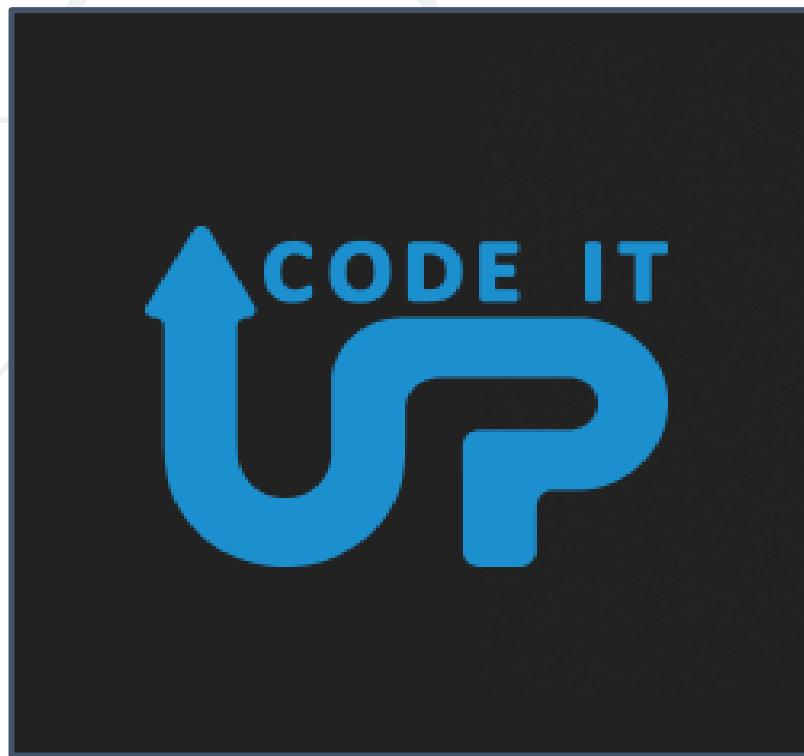
**INDEAVR**  
Serving the high achievers



**createX**

**SUPER  
HOSTING  
.BG**

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

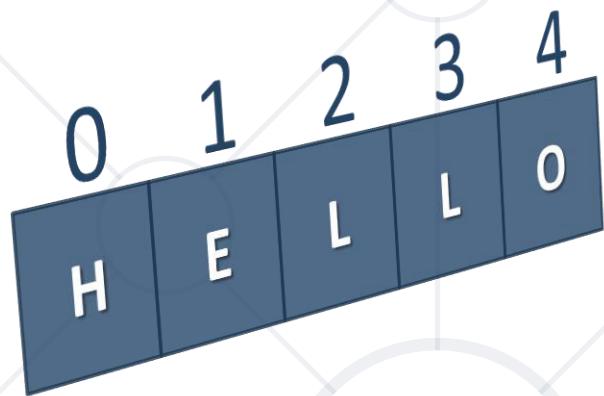


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Text Processing

## Manipulating Text



SoftUni Team

Technical Trainers



SoftUni



Software University  
<https://softuni.bg>

# Table of Contents

1. What Is a String?
2. Manipulating Strings
3. Building and Modifying Strings
  - Using StringBuilder Class
  - Why Concatenation Is a Slow Operation?

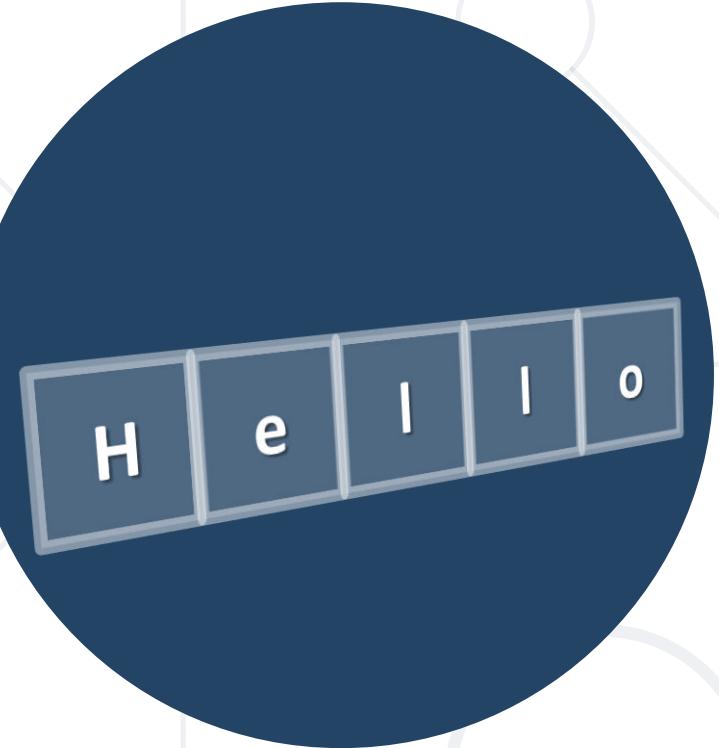


Questions?



**sli.do**

**#fund-java**



# What is a String?

Strings

# What is a String?

- Strings are **sequences** of characters (texts)
- The string data type in Java
  - Declared by the **String**
- Strings are enclosed in double quotes:

```
String text = "Hello, Java";
```



# Strings Are Immutable

- Strings are **immutable** (read-only) sequences of characters

- Accessible by **index** (read-only)

```
String str = "Hello, Java";  
char ch = str.charAt(2); // l
```

- Strings use **Unicode**  
(can use most alphabets, e.g. Arabic)

```
String greeting = "你好"; // (Lí-hó) Taiwanese
```



# Initializing a String

- Initializing from a string literal:

```
String str = "Hello, Java";
```

- Reading a **string** from the console:

```
String name = sc.nextLine();
System.out.println("Hi, " + name);
```

- Converting a **string** from and to a **char array**:

```
String str = new String(new char[] {'s', 't', 'r'});
char[] charArr = str.toCharArray();
// ['s', 't', 'r']
```





# Manipulating Strings

# Concatenating

- Use the **+** or the ****+=**** operators

```
String text = "Hello" + ", " + "world!";  
// "Hello, world!"
```

```
String text = "Hello, ";  
text += "John"; // "Hello, John"
```

- Use the **concat()** method

```
String greet = "Hello, ";  
String name = "John";  
String result = greet.concat(name);  
System.out.println(result); // "Hello, John"
```



# Joining Strings

- `String.join("", ...)` concatenates strings

```
String t = String.join("", "con", "ca", "ten", "ate");
// "concatenate"
```

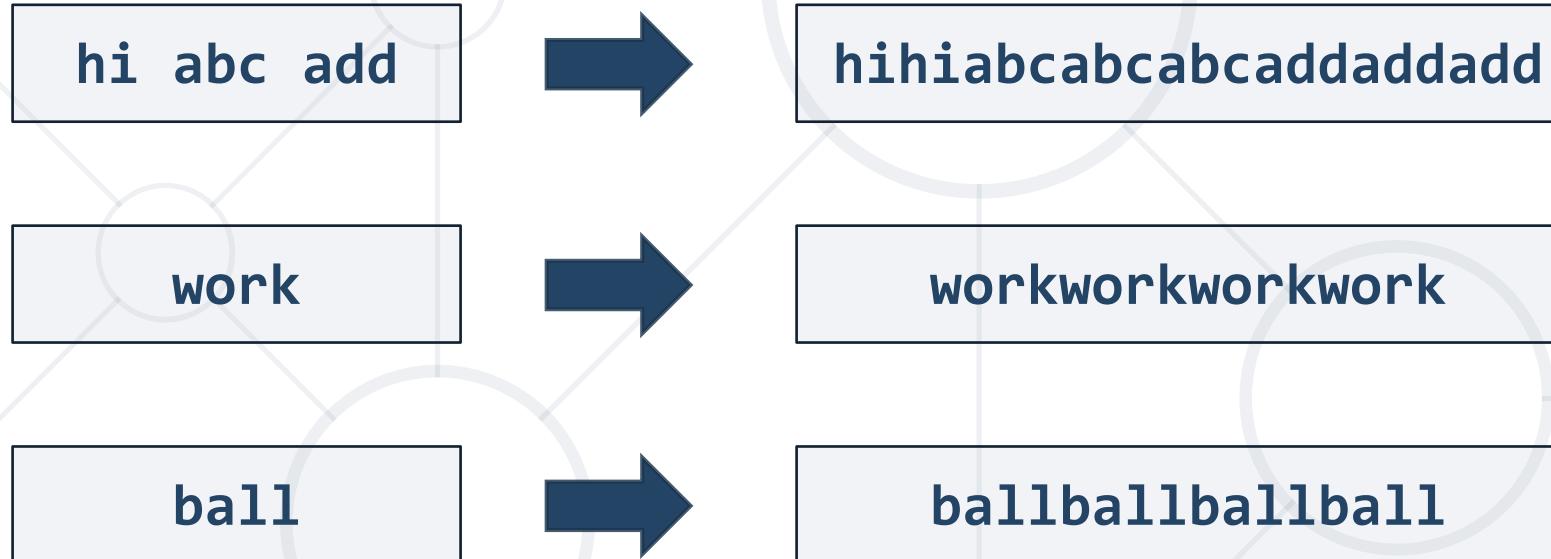
- Or an array/list of strings
  - Useful for repeating a string

```
String s = "abc";
String[] arr = new String[3];
for (int i = 0; i < arr.length; i++) { arr[i] = s; }
String repeated = String.join("", arr); // "abcabcabc"
```



# Problem: Repeat Strings

- Read an array from strings
- Repeat each word **n** times, where **n** is the length of the word



Check your solution here: <https://judge.softuni.org/Contests/1669/>

# Solution: Repeat Strings (1)

```
String[] words = sc.nextLine().split(" ");
List<String> result = new ArrayList<>();
for (String word : words) {
    result.add(repeat(word, word.length()));
}
System.out.println(String.join("", result));
```

Check your solution here: <https://judge.softuni.org/Contests/1669/>

# Solution: Repeat Strings (2)

```
static String repeat(String s, int repeatCount) {  
    String[] repeatArr = new String[repeatCount];  
    for (int i = 0; i < repeatCount; i++) {  
        repeatArr[i] = s;  
    }  
    return String.join("", repeatArr);  
}
```

# Substring

- **substring(int startIndex, int endIndex)**

```
String card = "10C";  
String power = card.substring(0, 2);  
System.out.println(power); // 10
```

- **substring(int startIndex)**

```
String text = "My name is John";  
String extractWord = text.substring(11);  
System.out.println(extractWord); // John
```



# Searching (1)

- **indexOf()** - returns the first match index or -1

```
String fruits = "banana, apple, kiwi, banana, apple";
System.out.println(fruits.indexOf("banana"));      // 0
System.out.println(fruits.indexOf("orange"));       // -1
```



- **lastIndexOf()** - finds the last occurrence

```
String fruits = "banana, apple, kiwi, banana, apple";
System.out.println(fruits.lastIndexOf("banana")); // 21
System.out.println(fruits.lastIndexOf("orange")); // -1
```

# Searching (2)

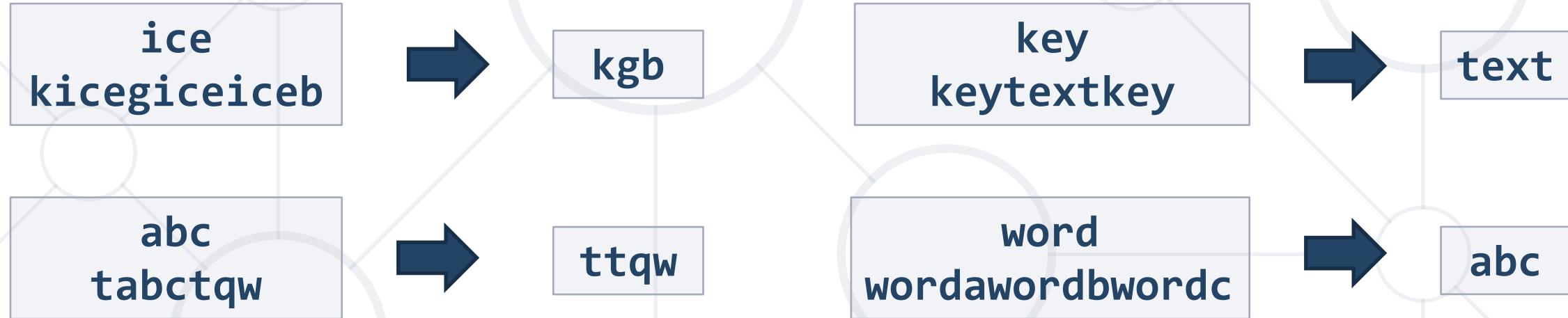
- **contains()** - checks whether one string contains another

```
String text = "I love fruits.";  
System.out.println(text.contains("fruits"));  
// true  
  
System.out.println(text.contains("banana"));  
// false
```



# Problem: Substring

- You are given a **remove word** and a **text**
- Remove all substrings that are equal to the remove word



Check your solution here: <https://judge.softuni.org/Contests/1669/>

# Solution: Substring

```
String key = sc.nextLine();
String text = sc.nextLine();

int index = text.indexOf(key);
while (index != -1) {
    text = text.replace(key, "");
    index = text.indexOf(key);
}

System.out.println(text);
```

Check your solution here: <https://judge.softuni.org/Contests/1669/>

# Splitting

- Split a string by a given pattern

```
String text = "Hello, john@softuni.bg, you have been  
using john@softuni.bg in your registration";
```

```
String[] words = text.split(", ");
```

```
// words[]: "Hello", "john@softuni.bg", "you have been..."
```

- Split by multiple separators

```
String text = "Hello, I am John.>";
```

```
String[] words = text.split("[, .]+");
```

```
// "Hello", "I", "am", "John"
```



# Replacing

- **replace(match, replacement)** - replaces all occurrences
  - The result is a new **string** (strings are **immutable**)



```
String text = "Hello, john@softuni.bg, you have been  
using john@softuni.bg in your registration.";  
  
String replacedText = text  
    .replace("john@softuni.bg", "john@softuni.com");  
  
System.out.println(replacedText);  
  
// Hello, john@softuni.com, you have been using  
john@softuni.com in your registration.
```

# Problem: Text Filter

- You are given a string of banned words and a text
  - Replace all banned words in the text with asterisks (\*)

Linux, Windows

It is not Linux, it is GNU/Linux. Linux is merely the kernel, while GNU adds the functionality...



It is not \*\*\*\*, it is GNU/\*\*\*\*. \*\*\*\* is merely the kernel, while GNU adds the functionality...

# Solution: Text Filter (1)

```
String[] banWords = sc.nextLine().split(", ");  
String text = sc.nextLine();  
for (String banWord : banWords) {  
    if (text.contains(banWord)) {  
        String replacement = repeatStr("*",  
            banWord.length());  
        text = text.replace(banWord, replacement);  
    }  
}  
System.out.println(text);
```

contains(...) checks if string contains another string

replace() a word with a sequence of asterisks of the same length

# Solution: Text Filter (2)

```
private static String repeatStr(String str, int length) {  
    String replacement = "";  
    for (int i = 0; i < length; i++) {  
        replacement += str;  
    }  
    return replacement;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/1669/>



# Live Exercises



# Building and Modifying Strings

## Using the StringBuilder Class

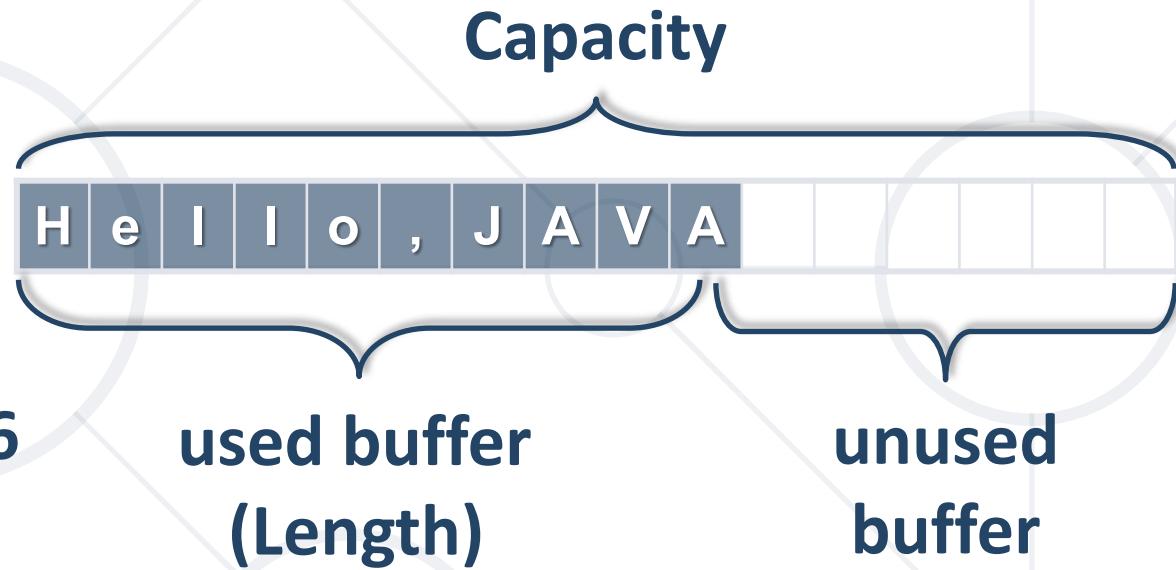
# StringBuilder: How It Works?



**StringBuilder:**

`length() = 10`

`capacity() = 16`



- **StringBuilder** keeps a buffer space, allocated in advance
  - Do not allocate memory for most operations → performance

# Using StringBuilder Class

- Use the **StringBuilder** to build/modify strings

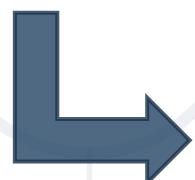
```
StringBuilder sb = new StringBuilder();
sb.append("Hello, ");
sb.append("John! ");
sb.append("I sent you an email.");
System.out.println(sb.toString());
// Hello, John! I sent you an email.
```



# Concatenation vs. StringBuilder (1)

- Concatenating strings is a slow operation because each iteration creates a new string

```
System.out.println(new Date());  
  
String text = "";  
  
for (int i = 0; i < 1000000; i++)  
    text += "a";  
  
System.out.println(new Date());
```



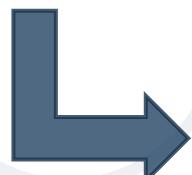
Tue Jul 10 13:57:20 EEST 2018  
Tue Jul 10 13:58:07 EEST 2018



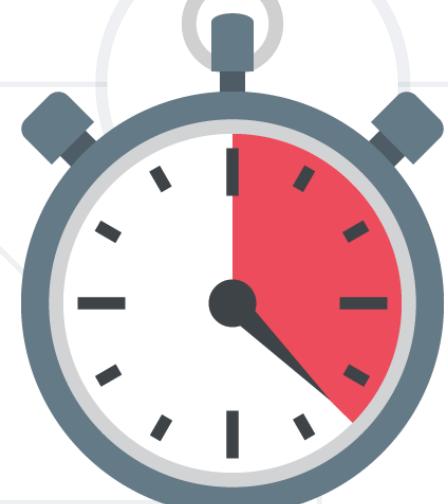
# Concatenation vs. StringBuilder (2)

- Using **StringBuilder**

```
System.out.println(new Date());  
  
StringBuilder text = new  
StringBuilder();  
  
for (int i = 0; i < 1000000; i++)  
    text.append("a");  
  
System.out.println(new Date());
```



```
Tue Jul 10 14:51:31 EEST 2018  
Tue Jul 10 14:51:31 EEST 2018
```



# StringBuilder Methods (1)

- **append()** - appends the string representation of the argument

```
StringBuilder sb = new StringBuilder();
sb.append("Hello Peter, how are you?");
```

- **length()** - holds the length of the string in the buffer

```
sb.append("Hello Peter, how are you?");
System.out.println(sb.length()); // 25
```

- **setLength(0)** - removes all characters



# StringBuilder Methods (2)

- **charAt(int index)** - returns char on index

```
StringBuilder sb = new StringBuilder();
sb.append("Hello Peter, how are you?");
System.out.println(sb.charAt(1)); // e
```

- **insert(int index, String str)** –  
inserts a string at the specified character position

```
sb.insert(11, " Ivanov");
System.out.println(sb);
// Hello Peter Ivanov, how are you?
```



# StringBuilder Methods (3)

- **replace(int startIndex, int endIndex, String str)** - replaces the chars in a substring

```
sb.append("Hello Peter, how are you?");  
sb.replace(6, 11, "George");
```

- **toString()** - converts the value of this instance to a String

```
String text = sb.toString();  
System.out.println(text);  
// Hello George, how are you?
```



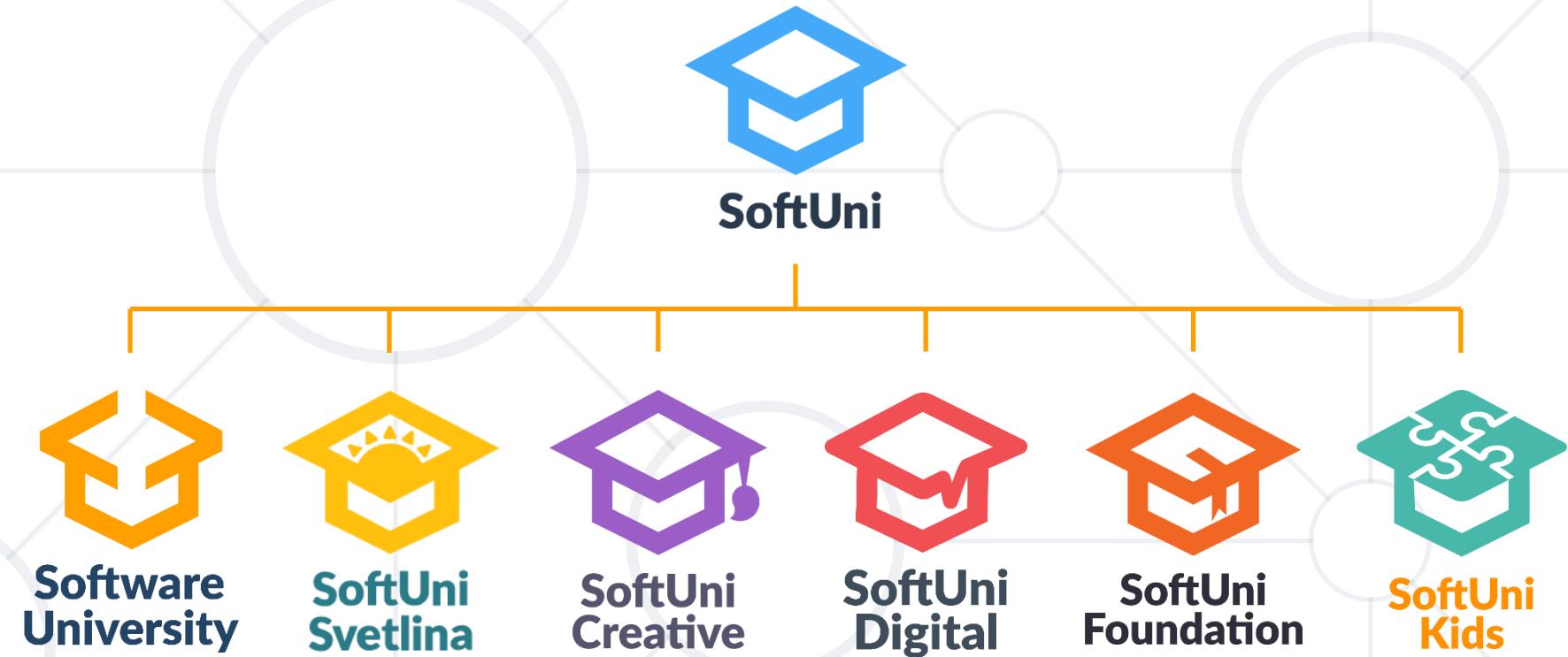


# Live Exercises

- Strings are **immutable** sequences of Unicode characters
- String processing methods
  - **concat()**, **indexOf()**, **contains()**, **substring()**, **split()**, **replace()**, ...
- **StringBuilder** efficiently builds/modifies strings



# Questions?



# SoftUni Diamond Partners



**SCHWARZ**



Coca-Cola HBC  
Bulgaria

Postbank  
*Решения за твоето упре*

Bosch.**IO**

SmartIT

POKERSTARS

CAREERS

AMBITIONED

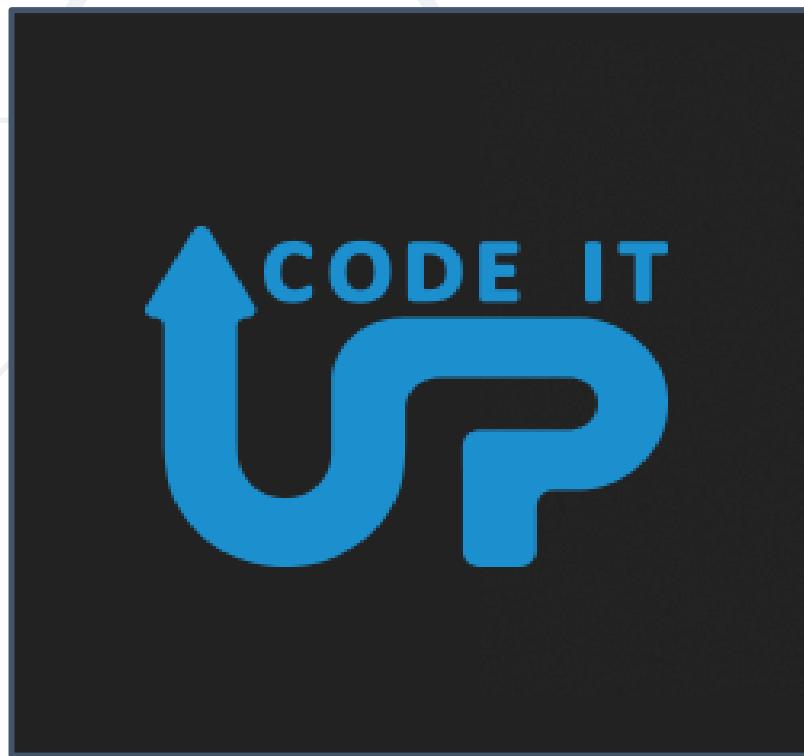
INDEAVR  
Serving the high achievers

create**X**

DRAFT  
KINGS

SUPER  
HOSTING  
.BG

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

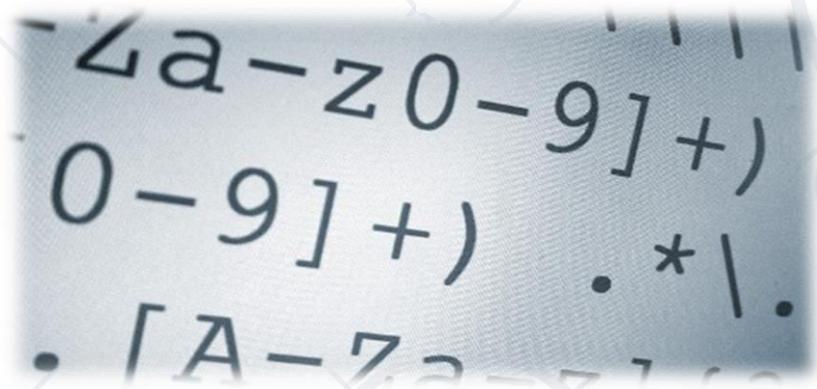


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Regular Expressions (RegEx)

## Regular Expressions Language Syntax



[a-z0-9]+)  
0-9]+) .\*|. . [A-Za-zA-Z]+

SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Have a Question?



**sli.do**

**#fund-java**

# Table of Contents

## 1. Regular Expressions Syntax

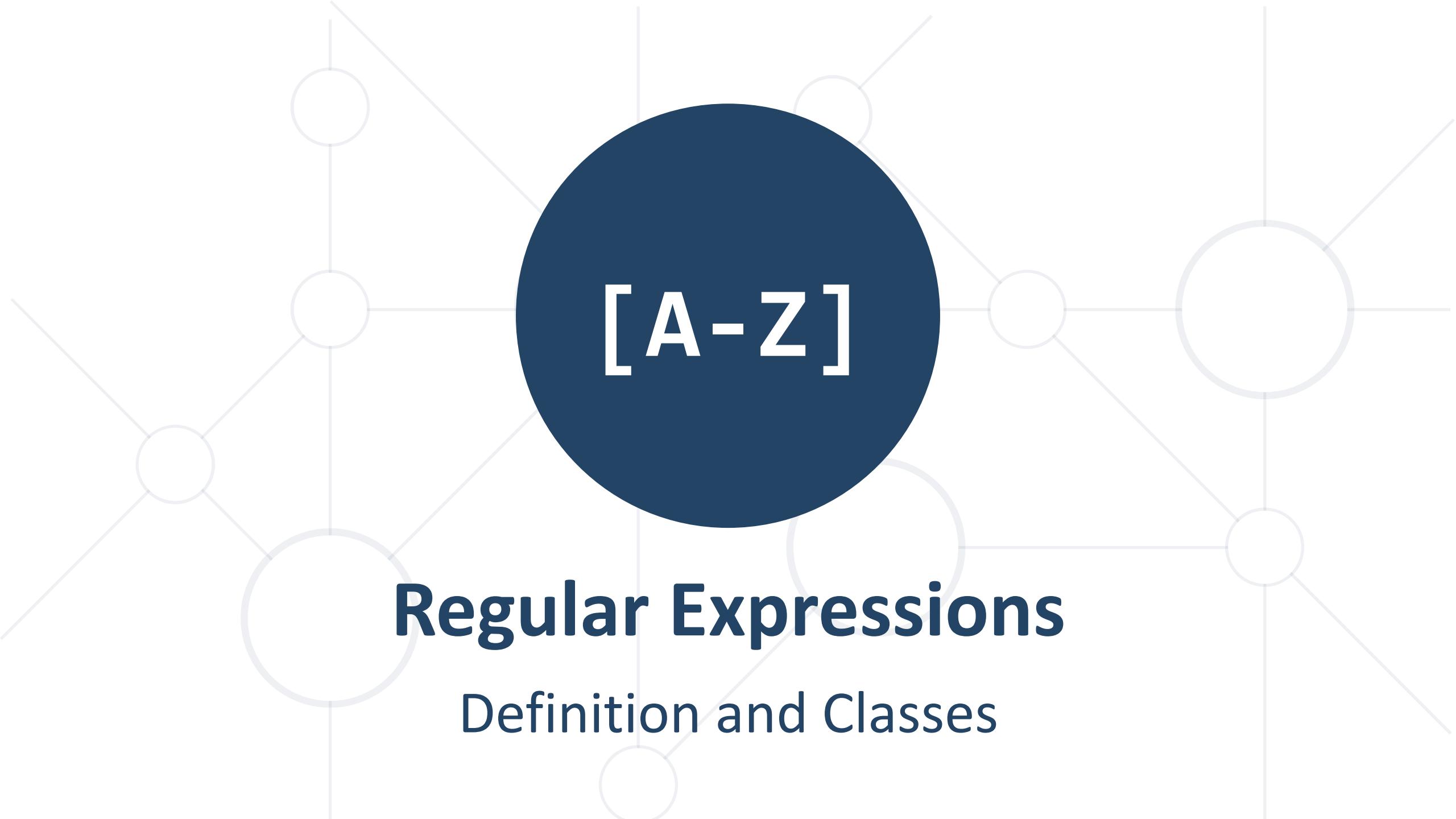
- Definition and Pattern
- Predefined Character Classes

## 2. Quantifiers and Grouping

## 3. Backreferences

## 4. Regular Expressions in Java





[A-Z]

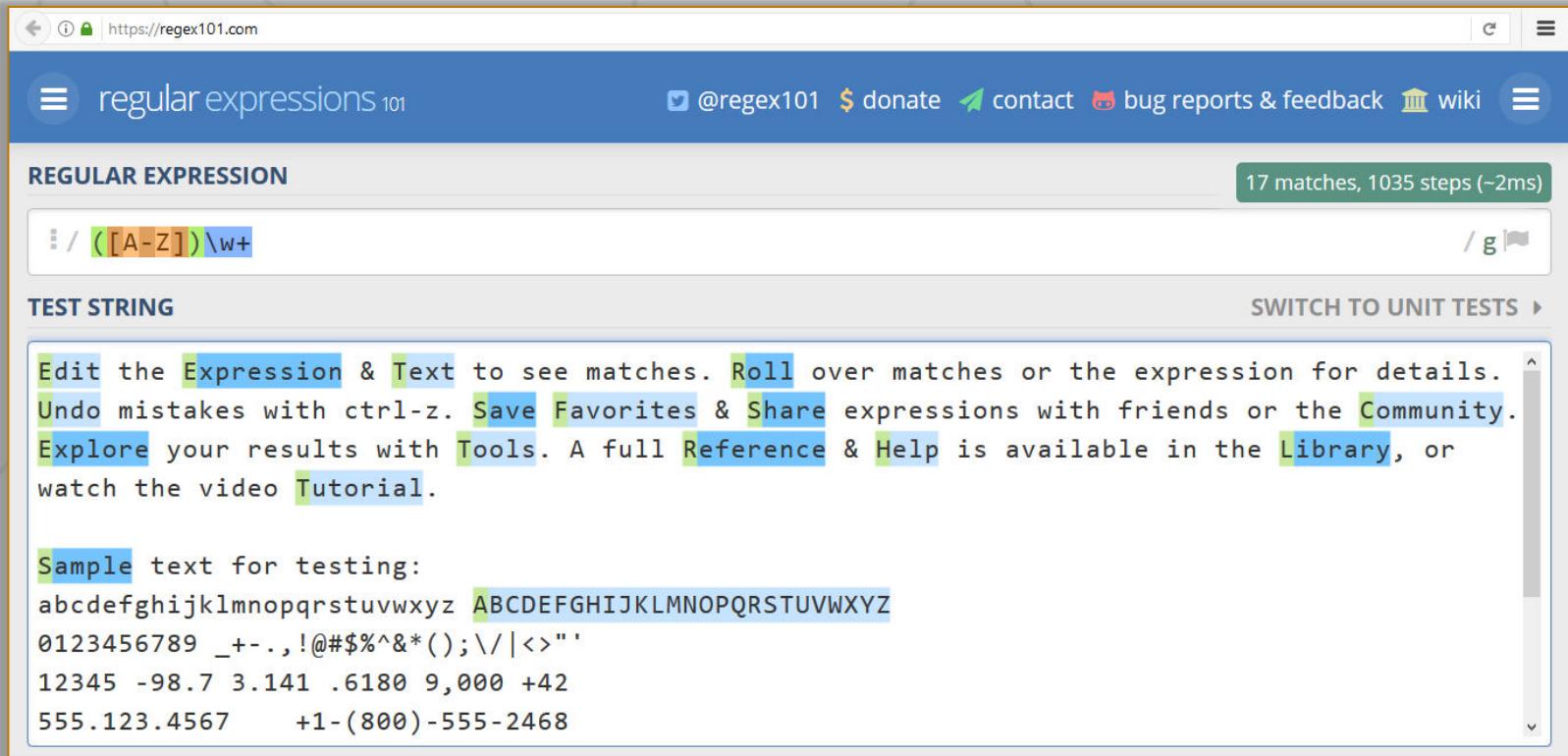
# Regular Expressions

## Definition and Classes

# What Are Regular Expressions?

- Regular expressions (regex)
  - Match text by pattern
- Patterns are defined by special syntax, e.g.
  - **[0-9]+** matches non-empty sequence of digits
  - **[A-Z][a-z]\*** matches a capital + small letters
- Play with regex live at: [regextester.com](https://regextester.com), [regex101.com](https://www.regex101.com)





# Live Demo

Www.regex101.com

# Regular Expression Pattern – Example

- Regular expressions (regex) describe a search pattern
- Used to find / extract / replace / split data from text by pattern

[A-Z][a-z]+ [A-Z][a-z]+

John Smith

Linda Davis

Contact: Alex Scott

# Character Classes: Ranges

- **[nvj]** matches any character that is either **n**, **v** or **j**

node.js v0.12.2

- **[^abc]** - matches any character that is **not** a, b or c

Abraham

- **[0-9]** - character range matches any digit from **0** to **9**

John is 8 years old.

# Predefined Classes

- **\w** - matches any **word character** (a-z, A-Z, 0-9, \_)
- **\W** - matches any **non-word character** (the opposite of \w)
- **\s** - matches any **white-space character**
- **\S** - matches any **non-white-space** character (opposite of \s)
- **\d** - matches any **decimal digit** (0-9)
- **\D** - matches any **non-decimal character** (the opposite of \d)



( \w+ )

Quantifiers

Grouping

# Quantifiers

- **\*** - matches the previous element zero or more times

```
\+\d*
```



```
+359885976002 a+b
```

- **+** - matches the previous element one or more times

```
\+\d+
```



```
+359885976002 a+b
```

- **?** - matches the previous element zero or one time

```
\+\d?
```



```
+359885976002 a+b
```

- **{3}** - matches the previous element exactly 3 times

```
\+\d{3}
```

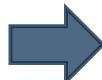


```
+359885976002 a+b
```

# Grouping Constructs

- **(subexpression)** - captures the matched subexpression as numbered group

```
\d{2}-(\w{3})-\d{4}
```



```
22-Jan-2015
```

- **(?:subexpression)** - defines a non-capturing group

```
^(?:Hi|hello), \s*(\w+)$
```



```
Hi, Peter
```

- **(?<name>subexpression)** - defines a named capturing group

```
(?<day>\d{2})-(?<month>\w{3})-  
(?<year>\d{4})
```



```
22-Jan-2015
```

# Problem: Match All Words

- Write a regular expression in [www.regex101.com](http://www.regex101.com) that extracts all word char sequences from given text

\_ (Underscores) are also word characters!



\_|Underscores|are|also|  
word|characters

# Problem: Match Dates

- Write a regular expression that extracts **dates** from text
  - Valid date format: **dd-MMM-yyyy**
  - Examples: **12-Jun-1999, 3-Nov-1999**

I am born on **30-Dec-1994**.  
My father is born on the **9-Jul-1955**.  
**01-July-2000** is not a valid date.

# Problem: Email Validation

- Write a regular expression that performs simple **email validation**
  - An email consists of **username @ domain name**
  - **Usernames** are alphanumeric
  - **Domain names** consist of **two strings**, separated by a **period**
  - **Domain names** may contain only **English letters**

Valid:

valid123@email.bg

Invalid:

invalid\*name@email1.bg



# Backreferences

## Numbered Capturing Group

# Backreferences Match Previous Groups

- **\number** - matches the value of a numbered capture group

```
<(\w+)[^>]*>.*?<\/\1>
```

**<b>Regular Expressions</b>** are cool!

**<p>I am a paragraph</p>** ... some text after

Hello, **<div>I am a<code>DIV</code></div>**!

**<span>Hello, I am Span</span>**

**<a href="https://softuni.bg/">SoftUni</a>**



# Regular Expressions

## Using Built-In Regex Classes

- Regex in Java library
  - `java.util.regex.Pattern`
  - `java.util.regex.Matcher`

```
Pattern pattern = Pattern.compile("a*b");
Matcher matcher = pattern.matcher("aaaab");

boolean match = matcher.find();
String matchText = matcher.group();
```

Searches for the  
next match

Gets the matched text

# Checking for a Single Match

- **find()** - gets the first pattern match

```
String text = "Andy: 123";
String pattern = "([A-Z][a-z]+): (?<number>\\d+)";
```

```
Pattern regex = Pattern.compile(pattern);
Matcher matcher = regex.matcher(text);
```

```
System.out.println(matcher.find());           // true
System.out.println(matcher.group());          // Andy: 123
System.out.println(matcher.group(0));          // Andy: 123
System.out.println(matcher.group(1));          // Andy
System.out.println(matcher.group(2));          // 123
System.out.println(matcher.group("number"));    // 123
```

+ - Matches the element one or more times

# Replacing with Regex

- To replace **every/first** subsequence of the input sequence that matches the pattern with the given replacement string
  - **replaceAll(String replacement)**
  - **replaceFirst(String replacement)**

```
String regex = "[A-Za-z]+";
String string = "Hello Java";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(string);
String res = matcher.replaceAll("hi");      // hi hi
String res2 = matcher.replaceFirst("hi"); // hi Java
```

# Splitting with Regex

- **split(String pattern)** - splits the text by the pattern
  - Returns **String[]**

```
String text = "1 2 3 4";
```

```
String pattern = "\\s+";
```

Matches whitespaces

```
String[] tokens = text.split(pattern);
```

tokens = {"1", "2", "3", "4"}

# Problem: Match Full Name

- You are given a list of names
  - Match all full names

Ivan Ivanov, Ivan ivanov, ivan Ivanov, IVan Ivanov, Georgi Georgiev, Ivan Ivanov



Ivan Ivanov Georgi Georgiev

Check your solution here: <https://judge.softuni.org/Contests/1672/>

# Solution: Match Full Names

```
String listOfNames = reader.readLine();

String regex = "\\b[A-Z][a-z]+ [A-Z][a-z]+";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(listOfNames);

while (matcher.find()) {
    System.out.print(matcher.group() + " ");
}
```

Check your solution here: <https://judge.softuni.org/Contests/1672/>

# Problem: Match Dates

- You are given a string
  - Match all dates in the format  
**"dd{separator}MMM{separator}yyyy"** and print them space-separated

13/Jul/1928, 01/Jan-1951



Day: 13, Month: Jul, Year: 1928

Check your solution here: <https://judge.softuni.org/Contests/1672/>

# Solution: Match Dates

```
String input = reader.readLine();

String regex =
"\b(?<day>\d{2})(\.|\/|\\-)(?<month>[A-Z][a-
z]{2})\d{2}(?<year>\d{4})\b";

Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(dates);

while (matcher.find()) {
    System.out.println(String.format("Day: %s, Month: %s, Year: %s",
    matcher.group("day"), matcher.group("month"),
    matcher.group("year")));
}
```

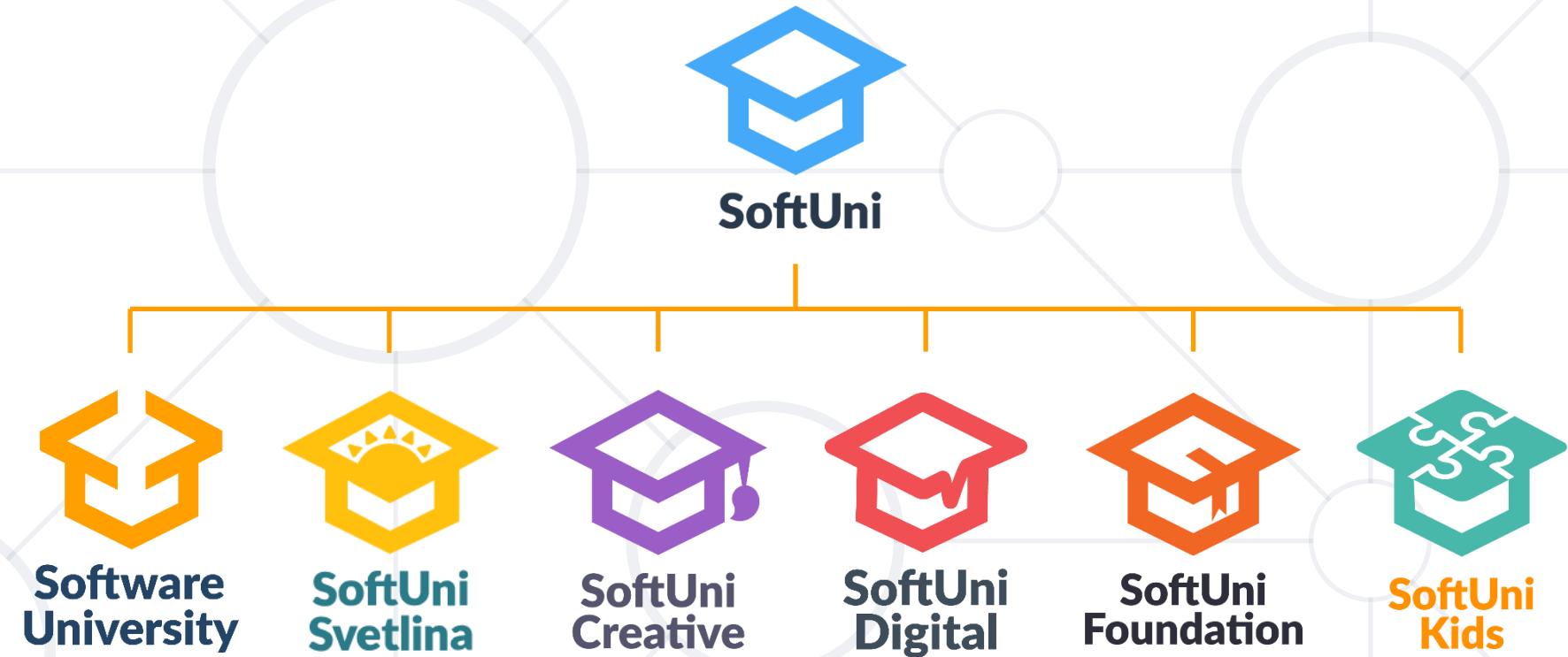
# Helpful Resources

- <https://regex101.com> and <http://regexpr.com> - websites to test Regex using different programming languages
- <https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html> - a quick reference for Regex from Oracle
- <http://regexone.com> - interactive tutorials for Regex
- <http://www.regular-expressions.info/tutorial.html> - a comprehensive tutorial on regular expressions

- Regular expressions describe patterns for searching through text
- Define special characters, operators and constructs for building complex pattern
- Can utilize character classes, groups, quantifiers and more



# Questions?



# SoftUni Diamond Partners



**SCHWARZ**



Coca-Cola HBC  
Bulgaria



**Postbank**

Решения за твоето упре



**Bosch.IO**



**SmartIT**



**POKERSTARS**



**CAREERS**



**AMBITIONED**



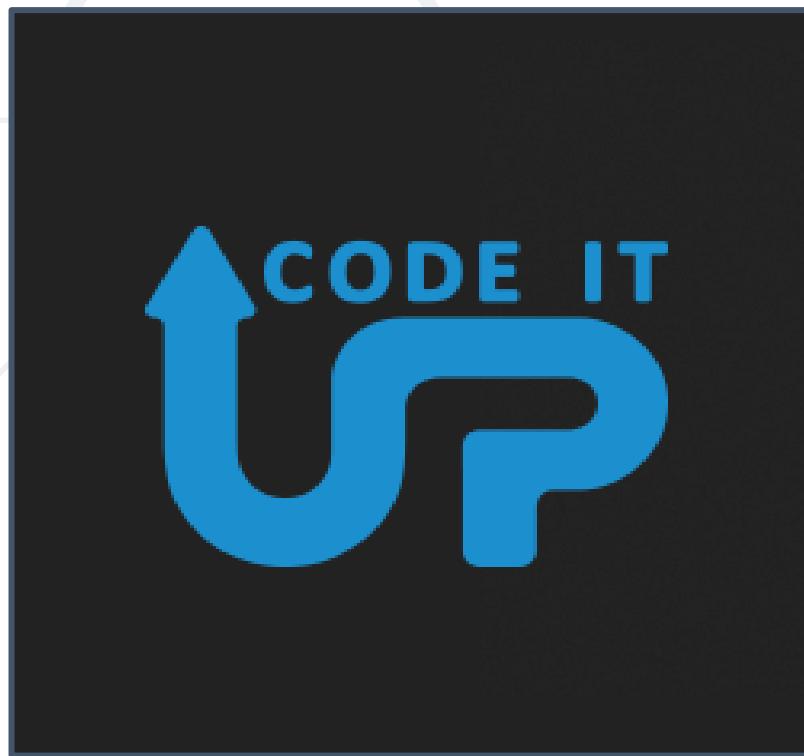
**INDEAVR**  
Serving the high achievers



**createX**

**SUPER  
HOSTING  
.BG**

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

