Introduction and Basic Syntax

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

Formatting Numbers in Placeholders

- System.out.printf("%03d", percentage);   // 055
- System.out.printf("%.2f", grade);   // 5.53

Using String.format to create a string by pattern

String result = String.format("Name: %s, Age: %d", name, age);

The If Statement

The simplest conditional statement. Executes one branch if the condition is true and another, if it is false

The Switch-case Statement

Works as sequence of if-else statements

Logical Operators - They return a boolean value and compare boolean values

Operator	**Notation in Java**	Example
Logical NOT	**!**	!false -> true
Logical AND	**&&**	true && false -> false
Logical OR	**||**	true || false -> true

Loops - Code Block Repetition

- 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. Managing the Count of the Iteration
  - **while and do…while** Execute a code block while a given condition returns true. Iterations While a Condition is True

Data Types

Variables

Variables have name, data type and value. Assignment is done by the operator "=". When processed, data is stored back into variables.

What is a data type

- Is a domain of values of similar characteristics
- Defines the type of information stored in the computer memory (in a variable)

Data Type Characteristics

A data type has: **Name** (Java keyword) **Size** (how much memory is used) **Default value**

Naming Variables

- Always refer to the naming conventions of a programming language
- camelCase is used in Java
- Preferred form: [Noun] or [Adjective] + [Noun]

Variable Scope and Lifetime

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

Variable Span

- Variable span is how long before a variable is called
- Always declare a variable as late as possible (e.g. shorter span)
- Shorter span simplifies the code
    - Improves its **readability** and **maintainability**
- **Integer types**

Type	Default Value	Min Value	Max Value	Size
byte	0	-128 (-27)	127 (27-1)	8 bit
short	0	-32768 (-215)	32767 (215 - 1)	16 bit
int	0	-2147483648 (-231)	2147483647 (231 – 1)	32 bit
Long	0	-9223372036854770000 (-2^{63})	9223372036854770000 (2^{63}-1)	64 bit

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

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**

Float Real Number 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 - **IEEE 754**
- 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

Floating-Point Division

- 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*

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));**

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**)
    - **Explicit** type conversion (**lossy**) – when precision can be lost:
        - double size = 3.14;
        - int intSize = (int) size;

Boolean Type

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

Character 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)

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

String

- 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:
- Strings can be concatenated using the **+** operator

Arrays - Fixed-Size Sequences of Elements

In programming, an array is a sequence of elements

- 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

Allocating an array of 10 integers -> int[] numbers = new int[10];

Assigning values to the array elements: - > for (int i = 0; i < numbers.length; i++)

numbers[i] = 1;

Accessing array elements by index: - > numbers[5] = numbers[2] + numbers[7];

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

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

String values = sc.nextLine();

String[] items = values.split(" ");

Read an array of integers **using functional programming**:

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

Use **String.join(separator, array)**: **Works only with strings**

System.out.println(String.join(" ", strings));

Foreach Loop

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

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

Methods

Simple Methods

- **Named block of code**, that can be invoked later
- Sample method **definition**:
- **Invoking** (calling) the method several times:

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

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

Naming Methods

Methods naming guidelines. Use meaningful method names. Method names should answer the question: What does this method do?

- Method parameters names
  - Preferred form: [**Noun**] or [**Adjective**] + [**Noun**]
  - Should be in **camelCase**
  - Should be **meaningful**

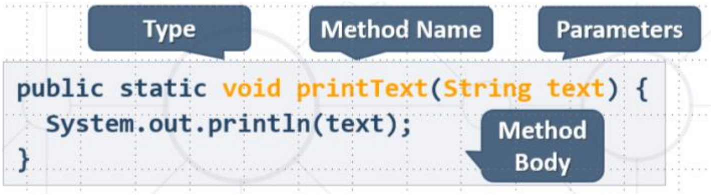
Best Practices

- 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

Code Structure and Code Formatting

- Make sure to use correct **indentation**
- 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 Methods



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

- Methods are declared **inside a class**
- **main()** is also a method
- Variables inside a method are **local**
- Methods are first **declared**, then **invoked** (many times)
- **Methods** can be **invoked (called)** by their name + **()**:
- A method can be invoked from:
  - The main method – **main()**
  - Its own body – **recursion**

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

Method Signature

- The combination of method's name and parameters is called signature
- Signature differentiates between methods with same names
- When methods with the same name have different signature, this is called method "**overloading**"
- Method's return type **is not part of** its signature

Void Type Method

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

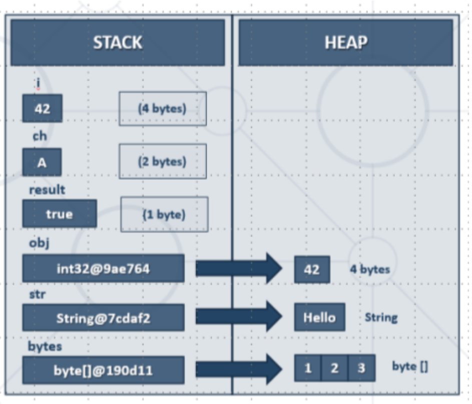
Methods with Parameters

- Method **parameters** can be of **any data type**
- Call the method with certain values (**arguments**)
- You can pass **zero** or **several** parameters
- You can pass parameters of **different types**
- Each parameter has **name** and **type**

Returning Values from Methods - The Return Statement

- The **return** keyword immediately stops
  the method's execution
- Returns the specified value
- Void methods can be **terminated** by just using **return**
- Return value can be:
    - **Assigned** to a variable
    - **Used** in expression
    - **Passed** to another method

Value Types vs. Reference Types



Value type variables hold directly their value

- **int, float, double, boolean, char, …**

- Each variable has its own copy of the value

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

Program Execution - Call Stack

- The program continues, after a method execution completes:
- "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**

Lists - Processing Variable-Length Sequences of Elements

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

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

Reading Lists from the Console

```
List<String> items = Arrays.stream(values.split(" "))
                .collect(Collectors.toList());
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:
- Printing a list using a **String.join()**: ->System.out.println(String.join("; ", list));

Sorting Lists

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

Objects and Classes

Objects -> An object is a single instance of a class

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

Classes -> In programming classes provide the structure for creating 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

Using the Built-In API Classes

- 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();

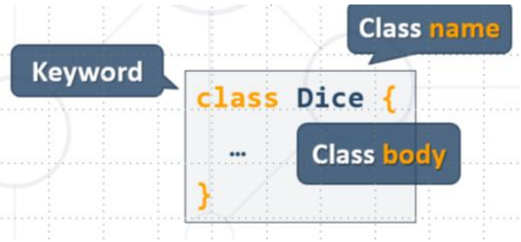
- Using **non-static Java classes**:

  Random rnd = new Random();

  int randomNumber = rnd.nextInt(99);

Defining Simple Classes

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



```
class Dice {
  …
}
```

Naming Classes

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

Class Members

- Class is made up of **state** and **behavior**
- Fields **store values**
- Methods **describe behaviour -** Store executable code (algorithm)
  - **Getters and Setters**

Creating an Object

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

Constructors -> Special methods, executed during object creation

- **Constructor name is the same as the name of the class**
- **Overloading default constructor**
- You can have multiple constructors in the same class

```
public Dice() { }
```

празен конструктор, за да се инициализира нов обект, без параметри за него

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

Classes define templates for object

- **Fields**
- **Constructors**
- **Methods**

Objects

- Hold a set of **named values**
- **Instance** of a class

Associative Arrays

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}**

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
- **remove(key)** method
- **containsKey(key)**
- **containsValue(value)**

Iterating Through Map

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

for (Map.Entry<K, V> entry : fruits.entrySet())

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>

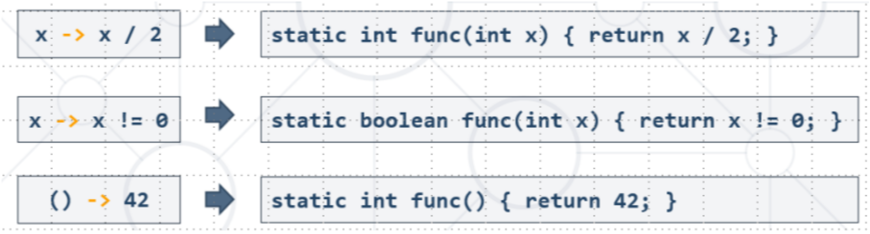
Lambda Expressions - Anonymous Functions

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 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

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

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

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

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

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

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

Processing Collections with Stream API

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

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

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

- **max()**

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

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

- **sum()**

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

- **average()**

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

Manipulating Collections

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

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

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

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()**

```
String[] words =
Arrays.stream(sc.nextLine().split(" "))
          .filter(w -> w.length() % 2 == 0)
          .toArray(String[]::new);
```

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

Text Processing

Strings Are Immutable

- Strings are sequences of characters (texts)
- Strings **are immutable (read-only)** sequences of characters
- Accessible by index (read-only)
- Strings use Unicode (can use most alphabets, e.g. Arabic)

Initializing a String

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

  Converting a **string** from and to a **char array**: -> char[] charArr = str.toCharArray();

Manipulating Strings

Concatenating

- Use the **+** or the **+=** operators
- Use the **concat()** method

Joining Strings

- **String.join("", …)** concatenates strings
  - Or an array/list of strings
  - Useful for repeating a string

Substring

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

Searching

- **indexOf()** - returns the first match index or -1
- **lastIndexOf()** - finds the last occurrence
- **contains()** - checks whether one string contains another

Splitting

- **Split** a string by a given **pattern ->** **text.split(", ");**
- **Split** by **multiple separators - >** **text.split("[, .]+");**

Replacing

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

StringBuilder Class

- **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();

Concatenation vs. StringBuilder

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

StringBuilder Methods

- **append()** - appends the string representation of the argument
- **length()** - holds the length of the string in the buffer
- **setLength(0)** - removes all characters
- **charAt(int index)** - returns char on index
- **insert(int index, String str)** – inserts a string at the specified character position
- **replace(int startIndex**, **int endIndex**, **String str)** - replaces the chars in a substring
- **toString()** - converts the value of this instance to a String

Regular Expressions (RegEx)

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

Character Classes: Ranges

- **[nvj]** matches any character that is either **n**, **v** or **j**
- **[^abc]** - matches any character that is **not a**, **b** or **c**
- **[0-9]** - character range matches any digit from **0** to **9**

Predefined Classes

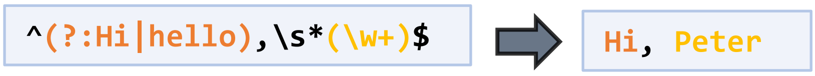
- **\w** - matches any **word character** (a-z, A-Z, 0-9, _)
- **\W** - matches any **non-word character** (the opposite of \w)
- **\s** - matches any **white-space** character
- **\S** - matches any **non-white-space** character (opposite of \s)
- **\d** - matches any **decimal digit** (0-9)
- **\D**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

Quantifiers

- - matches the previous element zero or more times
- **+** - matches the previous element one or more times
- **?** - matches the previous element zero or one time
- **{3}** - matches the previous element exactly 3 times

Grouping Constructs

- **(subexpression) -** captures the matched subexpression as numbered group
- **(?:subexpression) -** defines a non-capturing group



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

```
Hi, Peter
```

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

Backreferences - Numbered Capturing Group

- **\number -** matches the value of a numbered capture group **- >** <(\w+)[^>]*>.*?<\/**\1**>

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();
```

- **find() - gets the first pattern match**
- **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)**
- **split(String pattern) - splits the text by the pattern**
    - **Returns String[]**