Java SE & Spring

Module 1: Java SE

03.Data Types, Variables and Blocks

Keywords in Java

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs.

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

Added on version : **1.2, ***1.4,****1.5, *****9

Keywords in Java

- The keywords `const` and `goto` are reserved, even though they are not currently used.
- `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.
- `var` reserved word is added to the Java language since Java 10 (JDK 10), supporting a new feature called local variable type inference, in which Java compiler guesses the type of the variables based on the surround context – allowing programmers to not declare the type explicitly.
- You cannot use `var` to declare a variable without explicit initialization.
- We will talk about details for `var` reserved word in next slides.

Hello, World! - Our First Java Program

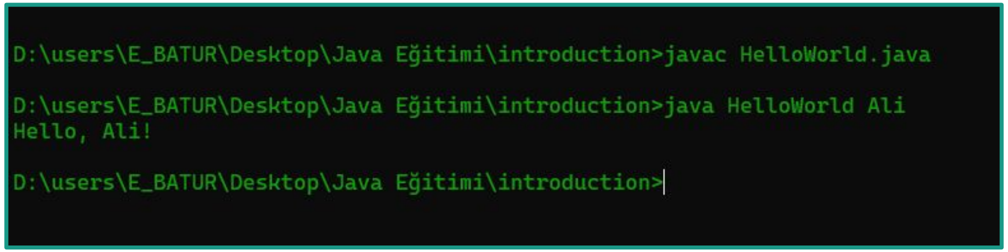
- Create a file with extension `java`, and name it `HelloWorld.java`.
- Open the file and write the code below:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println( "Hello, " + args[0] + "!");
    }
}
```

Hello, World! - Our First Java Program

- Open terminal and compile HelloWorld.java with, `javac HelloWorld.java`
- The command above generates **HelloWorld.class** which is compiled from **HelloWorld.java.**

  HelloWorld.class contains bytecode.
- Run the program with java HelloWorld Ali
- The output of the program is: Hello, Ali!

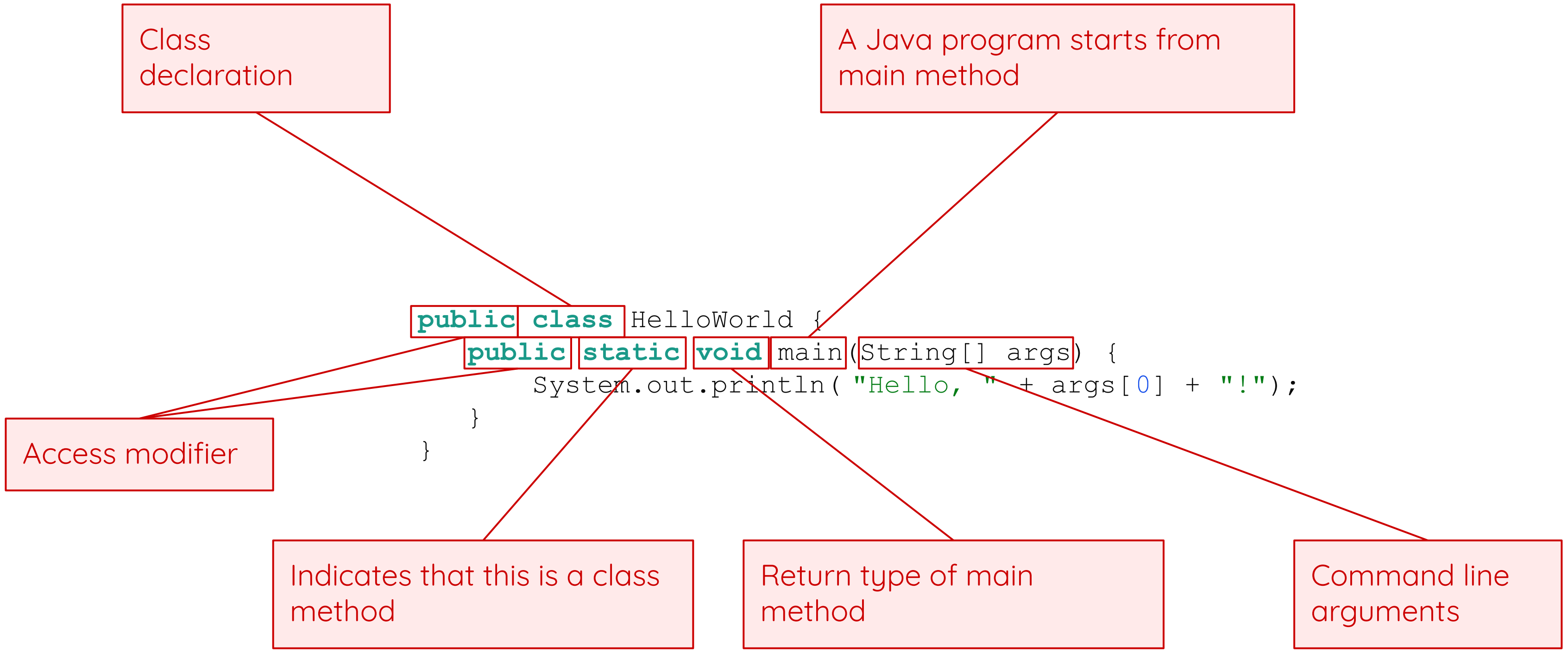


```
D:\users\E_BATUR\Desktop\Java Eğitimi\introduction>javac HelloWorld.java

D:\users\E_BATUR\Desktop\Java Eğitimi\introduction>java HelloWorld Ali
Hello, Ali!

D:\users\E_BATUR\Desktop\Java Eğitimi\introduction>
```

Structure of Our First Program


Class declaration
A Java program starts from main method
Access modifier
Indicates that this is a class method
Return type of main method
Command line arguments

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println( "Hello, " + args[0] + "!");
    }
}
```

Structure of Our First Program

➔ Main method

- Java main method is the entry point of any java program. Its syntax is always **`public static void main(String[] args)`.** You can only change the name of String array argument, for example you can change args to myArgs.
- The access modifier of the main method has to be **`public`** so that java runtime can execute this method.
- The main method has to be **`static`** so that JVM can load the class into memory and call the main method. If the main method won't be **`static`**, JVM would not be able to call it because there is no object of the class is present.

Primitive Data Types

➔ Primitive Data Types

- The Java programming language is statically-typed, which means that all variables must first be declared before they can be used.
- A primitive type is predefined by the language and is named by a reserved keyword.
- The Java language has 8 primitive types: **`boolean`**, **`byte`**, **`char`**, **`double`**, **`float`**, **`int`**, **`long`**, and **`short`**.
- 7 of these primitive data types are numerical, **`boolean`** type is a logical data type, contains only **`true`** or **`false`**.

Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive Data Types

- ➔ boolean
  - ◆ A **boolean** type represents either true or false value.
- ➔ char
  - ◆ The **char** data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).
  - ◆ It represents a single character, such as 'a', 'B', 'c' (between single quotation marks). Actually char type is 16-bit integer number (unsigned).

Further reading: https://home.unicode.org/

Primitive Data Types

- **byte**
  - The `byte` data type is an 8-bit signed integer. It has a minimum value of -128(-2^7) and a maximum value of 127(2^7-1).
- **short**
  - The `short` data type is a 16-bit signed integer. It has a minimum value of -32,768(-2^{15}) and a maximum value of 32,767(2^{15}-1).

Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive Data Types

- int
  - The `int` data type is a 32-bit signed integer, which has a minimum value of -2^{31} and a maximum value of 2^{31}-1. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 2^{32}-1.
- long
  - The `long` data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of 2^{63}-1. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2^{64}-1.

Primitive Data Types

➔ float

- The `float` data type is a single-precision 32-bit IEEE 754 floating point which has a minimum value of 1.40e-45f, and maximum value of 3.4028235e38f. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification.
- This data type should never be used for precise values, such as currency. For that, you will need to use the `java.math.BigDecimal` class instead.

Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive Data Types

➔ double

- The **`double`** data type is a single-precision 64-bit IEEE 754 floating point which has a minimum value of 4.9e-324d, and maximum value of 1.7976931348623157e308d. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification.
- This data type should never be used for precise values, such as currency. For that, you will need to use the **`java.math.BigDecimal`** class instead.
- For decimal values, this data type is generally the default choice.

Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive Data Types

➔ Default values

- Fields that are declared but not initialized will be set to a reasonable default by the compiler.
- Local variables are slightly different; You should assign a value to it. If don’t, accessing an uninitialized local variable will result in a compile-time error.

Data Type	Default Value (for fields)
`byte`	0
`short`	0
`int`	0
`long`	0L
`float`	0.0f
`double`	0.0d
`char`	'\u0000'
`boolean`	false

Text in Java (String)

- Strings are used for storing text. A String variable contains a collection of characters surrounded by double quotes(“”).

```
String greeting = "Hello";
```

- A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the `length()` method.

```
String txt = "Welcome to Java Course!";
System.out.println("The length of the txt string is: " +
txt.length());

// Output: The length of the txt string is: 23
```

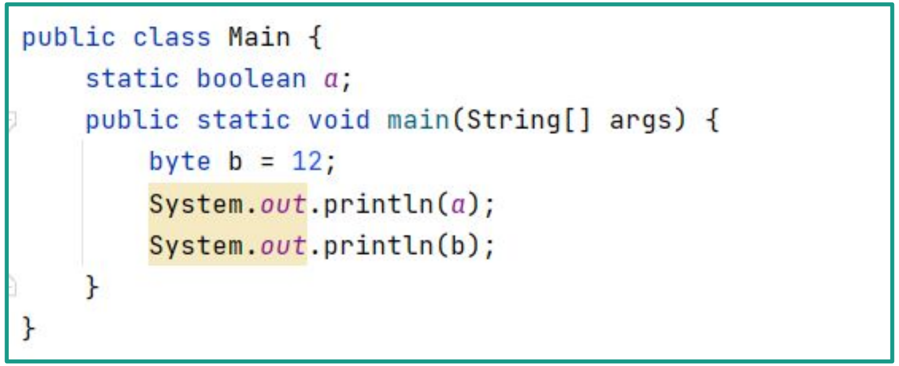
- There are many String methods, for example:

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

- We will discuss details about String in next sections.

Variables

In the Java programming language, the terms "field" and "variable" are both used; this is a common source of confusion among new developers, since both often seem to refer to the same thing.



```
public class Main {
    static boolean a;
    public static void main(String[] args) {
        byte b = 12;
        System.out.println(a);
        System.out.println(b);
    }
}
```

`a` and **`b`** are both variables in picture above. **`b`** is local variable, **`a`** is a class variable(static field)

Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

Variables

➔ Variable Types

- Instance Variables (Non-Static Fields): fields declared without the `static` keyword. Non-static fields are also known as instance variables because their values are unique to each instance of a class (to each object, in other words)
- Class Variables (Static Fields): A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.

Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

Variables

➔ <u>Variable Types</u>

- <u>Local Variables:</u> Similar to how an object stores its state in fields, a method will often store its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

---

*Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

Variables

➔ Variable Types

- Parameters: You've already seen examples of parameters in the main method of the "Hello World!" application. Recall that the signature for the main method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields".

*Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

Variables

- ➔ <u>Naming</u>
  - ◆ Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "_". The convention, however, is to always begin your variable names with a letter, not "$" or "_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "_", this practice is discouraged. White space is not permitted.

Variables

➔ <u>Naming</u>

- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations like a, b, x etc.
- A variable name must not be a [keyword or reserved word](#).
- If it consists of more than one word, capitalize the first letter of each subsequent word(camelCase). The names **`firstName`** and **`lastName`** are prime examples of this convention. If your variable stores a constant value, such as **`static final double CPU_COUNT = 8`**, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Local variables with `var` keyword

- ➔ `var` keyword is added to the Java language since Java 10 (JDK 10), supporting a new feature called local variable type inference.
- ➔ Java compiler guesses the type of the variables based on the surround context – allowing programmers to not declare the type explicitly.
- ➔ `var` keyword makes the code succinct, more readable and reduces boilerplate. However, there are some restrictions:
  - ◆ Can be used only for local variables (in methods). It cannot be used for instance variables (at class level).
  - ◆ Cannot be used in Lambda expressions (will be covered on next sections).
  - ◆ Cannot be used for method signatures (in return types and parameters).
  - ◆ Cannot be used to declare a variable without explicit initialization

Operators

Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

Category	Operators
Simple assignment	`=`
Arithmetic	`+ - * / %`
Unary	`+ - ++ -- !`
Relational	`== != > >= < <=`
Conditional	`&& || ? : (ternary)`
Type comparison	`instanceof`
Bitwise and Bit shift	`~ << >> >>> & ^ |`

Further reading: https://www.codejava.net/java-core/the-java-language/summary-of-operators-in-java-with-examples

Operators

➔ Simple Assignment Operator

- One of the most common operators that you'll encounter is the simple assignment operator "=".

  It assigns the value on its right to the operand on its left.

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

- This operator can also be used on objects to assign object references, as we will discuss about

  details in OOP section.

```
String greeting = "Hello, World!";
File pdf = new File("MyCV.pdf");
```

Further reading: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html

Operators

➔ <u>Arithmetic Operators</u>

The arithmetic operators are used to perform mathematical calculations. The following table lists all arithmetic operators in Java. here's a good chance you'll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is "**%**", which divides one operand by another and returns the remainder as its result.

Operator	Meaning
+	Addition (and strings concatenation) operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

Operators

➔ Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

Operator	Meaning
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

Operators

→ <u>The Equality and Relational Operators</u>

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "**==**", not "**=**", when testing if two primitive values are equal.

these operators generate a **`boolean`** result.

Operator	Meaning
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Operators

➔ Conditional Operators

The && and || operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

Operator	Meaning
&&	conditional -AND operator
||	conditional-OR operator
? :	ternary operator in form of: A ? B : C

The `?:` operator (ternary operator), can be thought of as shorthand for an `if-else` statement (will be discussed in the next session - Control Flow Statements)

Operators

➔ The Type Comparison Operator `instanceof`

The **`instanceof`** operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface. When using the instanceof operator, keep in mind that null is not an instance of anything.

```
public static void main(String[] args) {
        String name = "Java";

        if (name instanceof String) {
            System.out.println("an instance of String class");
        }
}
```

Operators

➔ Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

Operator	Meaning
~	unary bitwise complement; inverts a bit pattern
<<	signed left shift
>>	signed right shift
>>>	unsigned right shift
&	bitwise AND
^	bitwise exclusive OR
|	bitwise inclusive OR

Operators

➔ Bitwise and Bit Shift Operators

```
int a = 128;

// 0000 0000 0000 0000 0000 0000 1000 0000 -->
128

a = a >>> 5;

// 0000 0000 0000 0000 0000 0000 0000 0100 --> 4
```

Operator	Meaning
~	unary bitwise complement; inverts a bit pattern
<<	signed left shift
>>	signed right shift
>>>	unsigned right shift
&	bitwise AND
^	bitwise exclusive OR
|	bitwise inclusive OR

Expressions

- An expression is a construct made up of variables, operators, and method invocations that evaluates to a single value.

```
int count = 3;
count++;
```

- The data type of the value returned by an expression depends on the elements used in the expression.
- The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other.

```
(x + y) / 100
```

- When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

Statements

Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution, and can consists zero or more expressions. The following types of expressions can be made into a statement by terminating the expression with a semicolon (`;`). Sta

- Assignment expressions
- Any use of ++ or --
- Method invocations
- Object creation expressions

}→ Such statements are called expression statements.

```java
// assignment statement
resultValue = 8933.234;
// increment statement
resultValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Animal myPet = new Animal();
```

Statements

In addition to expression statements, there are two other kinds of statements: declaration statements and control flow statements. A declaration statement declares a variable. You've seen many examples of declaration statements already:

```
// declaration statement
double aValue = 8933.234;
```

Control flow statements regulate the order in which statements get executed. We'll talk about control flow statements in the next section.

Blocks

A block is a group of zero or more statements between balanced braces(or curly braces {}) and can be used anywhere a single statement is allowed.

```
class BlockDemo {
     public static void main(String[] args) {
          boolean condition =  true;
          if (condition) { // begin block 1
               System.out.println("Condition is true.");
          } // end block one
          else { // begin block 2
               System.out.println("Condition is false.");
          } // end block 2
     }
}
```

Java is a block structured language.

Scope

In Java, as in any programming language, each variable has a scope. This is the segment of the program where a variable can be used and is valid.

- Method Scope

  Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared.

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

    // Code here CANNOT use x

    int x = 100;

    // Code here can use x
    System.out.println(x);
  }
}
```

Scope

- <u>Block Scope</u>

A block of code refers to all of the code between curly braces {}. Variables declared inside blocks of code are only accessible by the code between the curly braces.

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

    // Code here CANNOT use x

    { // This is a block
      // Code here CANNOT use x
      int x = 100;

      // Code here CAN use x
      System.out.println(x);

   } // The block ends here
  // Code here CANNOT use x
  }
}
```

Practice Time: Let’s Code Together!

Questions ?

Next: Operators and Flow Control Statements