CHAPTER 1: INTRODUCTION TO JAVA

LEARNING OBJECTIVES

- Name the primitive data types in Java and explain their differences.
- Demonstrate and apply how output can be placed on the console
- Show how information can be read from the console
- Explain and use local variables
- Explain and use operators and know their precedence
- Explain and use conditional branching
- Explain and use different types of loops
- Define and call simple methods
- Describe and apply the process of Java execution

OVERVIEW VIDEO

1.1 JAVA IS ALSO A PROGRAMMING LANGUAGE

- Back in the 1970s Bill Joy (SUN Microsystems) had the desire to create a language that combines advantages of MESA and C
- First draft of a new object-oriented language in the 1990s in the article "Further"
- In parallel, James Gosling created the new language Oak (Object Application Kernel)
- First system *7 (Star Seven) was presented in 1992, consisting of the operating system Green-OS, Gosling's interpreter Oak and some hardware components

- After several changes of direction in the company and an orientation towards the World Wide Web, the Oak language was given the name Java in 1993, "since, as it later turned out, the name Oak could not be used for copyright reasons."
- First browser "HotJava" was introduced to the public at SunWorld '95 in 1995
- Breakthrough came with the licensing of Java to Netscape and with JDK 1.0 released on 23 January 1996 for the development of Java applications and applets (company JavaSoft)

A list of Java versions can be found here.

1.2 FIRST JAVA EXAMPLE

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

- Program name is MyProgram, which is also the name of the Java Class (see chapter 2) we define
- The main function is where program execution begins
- The first (and only) program statement outputs "Hello World!" to the console

For a start, the following can be stated:

- `public`, `class`, `static` and `void` are so called reserved keywords with a set interpretation in Java
- The name `MyProgram` may be choosen freely, but the file for class MyProgram must reflect the class name, in this case `MyProgramm.java`
- Inside the curly brackets after `public static void main(String[] args)` we find the statements that will be executed when this program is launched
- `System.out.println("Hello World!");` displays the text `Hello World!` on the screen.

EXERCISE: FIRST JAVA EXAMPLE

Which statements are correct?

```
public class MyProgram {
  public static void main(String[] args) {
    System.out.println("Java is cool!");
  }
}
```

- ☐ The name of the program or the class is MyProgram
- ☐ If the class is called Example, then the file containing the class must be Example.java
- ☐ The code does not produce any output

EXERCISE: HELLO, WORLD

Output the string `Hello, World!` to the console!!

1.3 RESERVED KEYWORDS

Java uses some words as so-called keywords with a special meaning.

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

The previous example already contains a few keywords:
`public`, `class`, `static` and `void`.

A keyword can not be used as identifier for the name of a class, of a variable or a method. The following page shows the reserved words in Java.

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

1.4 COMMENTS IN JAVA

Comments can be used to explain your Java code, thus making it more readable. It can also be used to prevent execution of parts of your code.

Java allows to specify single-line and multi-line comments.

SINGLE-LINE COMMENTS

```java
public class MyProgram {
  public static void main(String[] args) {
    // This is an example of a single-line comment
    System.out.println("Hallo Welt!"); // ... after a statement
  }
}
```

Single-line comments begin with two forward slashes.

MULTI-LINE COMMENTS

```
public class MyProgram {
  public static void main(String[] args) {
    /*
     * This is an example of a
     * multi-line comment
     */
    System.out.println("Hallo Welt!");
  }
}
```

Multi-line comments begin with `/*` and end with `*/`.

1.5 PRIMITIVE DATA TYPES

Java distinguishes between primitive and higher data types, such as Classes, Enumerations and Interfaces.

Primitive types include boolean values, integers and floating point numbers as well as single characters.

Values	Type	Size	Default	
Character	char	16 Bit	'\0'	a, b, c, … see UTF-16
Integers	byte	8 Bit	0	-128 to 127
	short	16 Bit	0	-32.768 to 32.767
	int	32 Bit	0	-2^{31} to 2^{31} -1
	long	64 Bit	0L	-2^{63} to 2^{63} - 1
Boolean	boolean	...	false	true or false
Fractional numbers	float	32 Bit	0.0f	+/-1,4E-45 to +/-3,4E+38
	double	64 Bit	0.0	+/-4,9E-324 to +/-1,7E+308

EXERCISE: PRIMITIVE DATA TYPES (1)

Which statements are correct?

- ☐ use `float` for fractional numbers
- ☐ use `double` für fractional numbers
- ☐ `char` can be used to store strings
- ☐ use `double` for integer numbers
- ☐ `float` and `double` store number with the same precision

EXERCISE: PRIMITIVE DATA TYPES (2)

Which statements are correct?

- ☐ use `int` for integer numbers
- ☐ the largest `long` number is twice as large as the largest `int` number
- ☐ the largest `long` number is more than 4 billion times as large as the largest `int` number
- ☐ the largest `long` number is more than 4 million times as large as the largest `int` number

1.6 VARIABLES

```
<data type> <name>;           // declare variable
<data type> <name> = <value>; // declase and initialize variable
```

```
int x;
int y = 42;
double pi = 3.14159265359;
char someChar = 'c';
```

- Create variable `x` of type `int`
- Create variable `y` and assign it the value 42
- Create variable `pi` of type `double`
- Create variable `someChar` of type `char` and assign it the value 'c'

Variables, which are typically used within methods (also called "function" in other programming languages) can be used to realise algorithms.

Java is a type-safe programming language, so a type - e.g. a primitive data type - is required for each variable when it is declared (created).

Using variables - first example:

```
public class MyProgram {
  public static void main(String[] args) {
    int number = 42;
    System.out.println(number);
  }
}
```

- Declaration and initialization of variable `number`
- Output of variable value using method `System.out.println`

EXERCISE: COMPLETE THE PROGRAM

This Java program is not complete. Supply the missing primitive data type for each variable.

1.7 STRINGS

You already know one string:

```
"Helle, World!"
```

In Java, a **string** is a sequence of characters. Strings are used very often, so there is a `String` data type for defining string variables. However, `String` is not a primitive data type - it is a Class (see chapter 2).

The use of the data type is analogous to the primitive data types:

```
String message = "Hell, World!";
```

Strings - a first example:

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

- Definition (=declaration and initialization) of variable `message`
- Output of variable value using `System.out.println`

The **addition operator** can be used to combine **strings** as well as values of other data types.
Attention: The result is always a string.

Simple example: producing a greeting

```java
public class MeinProgramm {
  public static void main(String[] args) {
    String name = "Sebastian";
    System.out.println("Hello " + name); // Output: Hello Sebastian
  }
}
```

1.8 CONSOLE OUTPUT

Simple outputs can be realised in Java via `System.out`. It contains methods for outputting values of different data types.

Something like this should be familiar by now:

```
System.out.println("Hello World!");
```

The suffix `ln` in `println` indicates that a **linefeed** will be output after the actual value.

Using `print` (without `ln`), will omit the linefeed:

```java
public class MyProgram {
  public static void main(String[] args) {
    System.out.print("Hello ");
    String name = "Sebastian";
    System.out.println(name);
  }
}
```

- Prints "Hello " to the console **without** a linefeed
- Prints "Sebastian" to the console, continuing on the same line **with** a linefeed at the end.

The `print` and `println` methods can output many different types of data. The values are automatically converted to strings.

```java
public class MyProgram {
  public static void main(String[] args) {
    System.out.println("String");
    System.out.println(42);
    System.out.println(3.14159265359);
  }
}
```

- Prints a string to the console
- Prints an integer number to the console
- Prints a fractional number to the console

`printf` is a more flexible way to output variable values combined with static text. The method is used to print formatted strings using various format specifiers

Example:

```
String s = "World";
System.out.printf("Hello %s!%n", s);  // output: Hello World!
System.out.printf("Hello %s!\n", s);  // output: Hello World!
```

with `%s` being the format specifier for a string value and `%n` being the format specifier for a linefeed. Please note that instead of `%n` you can also use `\n`.

More format specifiers:

- `%d` for int or long values
- `%f` for float or double values

EXERCISE: OUTPUT WITH PRINTLN

Use `System.out.println` and string concatenation with + to print the following sentence: **The car is from `<year>`, has the color `<color>`, consumes `<consumption>` l/100km and costs `<price>` Euros.**

EXERCISE: OUTPUT USING PRINTF

Use `System.out.printf` to print the following sentence: **The car is from `<year>`, has the color `<color>`, consumes `<consumption>` l/100km and costs `<price>` Euros.**

1.9 CONSOLE INPUT

Corresponding to `System.out` is `System.in`.

This allows inputs to be read in by the user on the console. However, its use is not entirely trivial.

The use can be simplified with the so-called 'scanner'. A scanner can read different types of data.

```java
java.util.Scanner scanner = new java.util.Scanner(System.in);
int x = scanner.nextInt();
System.out.println(x + 5);
```

- Create a `Scanner` for `System.in` (console input)
- Various methods like `nextInt` allow to read numbers and strings from the console

For example, we can ask the user for their name:

```
public class Myprogram {
  public static void main(String[] args) {
    java.util.Scanner scanner = new java.util.Scanner(System.in);
    System.out.print("Please enter your name: "); // produce user prompt
    String name = scanner.nextLine();
    System.out.print("Hello ");
    System.out.println(name);
  }
}
```

Read user input as string using `nextLine`.

EXERCISE: READ INPUTS FROM CONSOLE

Use `java.util.Scanner` to read user inputs from the console.

Change the existing statement `scanner.nextLine()` to save the result of the call in a variable of type `String`.

Use `System.out.println` to print the value of the variable.

Hints:

- A variable can be assigned a value using the *assignment operator*, e.g. `int x = 42;`
- The statement scanner.nextLine() yields a result of type String
- Results of statements like scanner.nextLine() can be assigned to a variable

1.10 OPERATORS

$$E = m * c^2$$

Java has several operators to perform calculations, comparisons or bit operations.

- Calculations are the classic addition, subtraction, multiplication and division.
- Comparisons are used to compare two values. The first value may be smaller than the second, larger than the second, or equal
- Bit operations can be used to perform bitwise operations on integer variables.

Each operator is assigned a precedence. The lower the precedence, the stronger the binding.

Here is a simple example for basic arithmetics: Multiplication is performed prior to addition. The precedence of the multiplication operator is therefore smaller than the precedence of the addition operator.

```java
int a = b + c * d;
```

Steps:

1. multiply c and d
2. add result of step 1 and b
3. save result of step 2 in a

When dealing with operators of the same precedence, their associativity is used to untie the knot. For example, addition and subtraction are left-associative operators. Accordingly, `x + y - z` will be processed as `(x + y) - z`.

Purpose	Operator	Precedence	Asso.	
Assignment	`=`, `+=`, `*=`, …	13	right	`x = 42;` `x +=1;`
Arithmetics	`+`	3	left	`x = 1 + 2;` also for Strings
	`-`	3	left	`x = 2 - 1;`
	`*`	2	left	`x = 3 * 4;`
	`/`	2	left	`x = 4 / 2;`
	`%`	2	left	`x = 4 % 3;`

Purpose	Operator	Precedence	Asso.	
Increment	`++`	1	right	`x++; or ++x;`
Decrement	`--`	1	right	`x--; or --x;`
Negation (arith.)	`-`	1	right	`int n1 = -n2;`
Negation (log.)	`!`	1	right	`boolean b1 = !b2;`

Purpose	Operator	Precedence	Asso.	
equality	`==`	6	left	`x == 42`
	`!=`	6	left	`x != 4`
	`<`	5	left	`x < 4`
	`<=`	5	left	`x <= 5`
	`>`	5	left	`x > 6`
	`>=`	5	left	`x >= 7`
Log. AND	`&&`	10	left	`x && y`
Log. OR	`||`	11	left	`x || y`

Purpose	Operator	Precedence	Asso.	
Shift left	`<<`	4	left	`x << 1`
Shift right	`>>`	4	left	`x >> 1` (signed)
	`>>>`	4	left	`x >>> 1` (unsigned)
Bitwise AND	`&`	7	left	`x & 1`
Bitwise OR	`|`	9	left	`x | 1`
Exclusive OR	`^`	8	left	`x ^ 1`

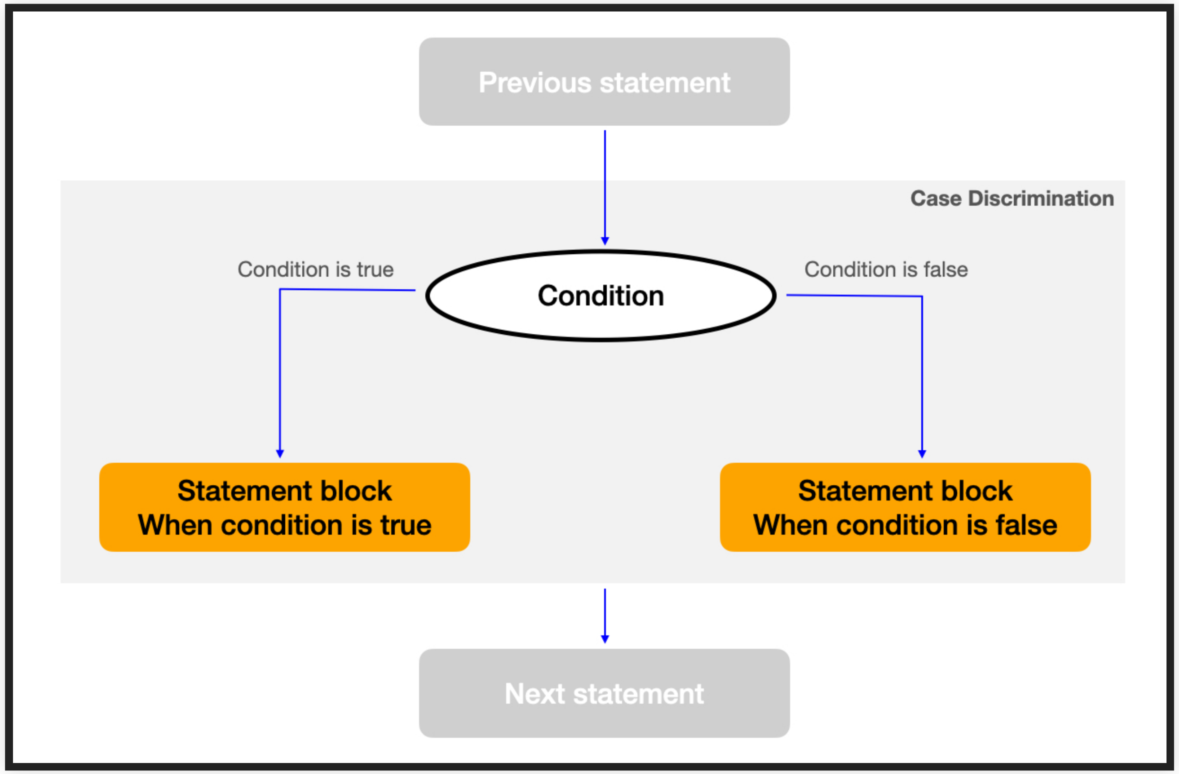
EXERCISE: OPERATORS

Given x = 5, y = 6 and z = 3.
Which calculations are correct in Java?

- ☐ x - y + z yields -4
- ☐ y - z * x yields 15
- ☐ x *= y + z sets x to 45
- ☐ z - x + y yields 4

1.11 BRANCHING STATEMENTS

Branches control the program flow depending on variable values in the condition.



Java allows to specify a branch using an **if statement**.

```
if (Condition>) {
    // statement block when condition is met
} else {
    // statement block when condition is not met
}
```

What happens in the example below?

```java
public class MyProgram {
  public static void main(String[] args) {
    java.util.Scanner scanner = new java.util.Scanner(System.in);
    System.out.print("Guess a number:  ");
    int x = scanner.nextInt();
    if (x == 42) {
        System.out.println("You gave the right answer!");
    }
  }
}
```

Branches are used to control certain parts of the program. Which branch is executed depends on the state of the application and is only determined at runtime.

In Java, we can distinguish between three branch statements:

If statement

If-Else statement

Switch statement

IF STATEMENT

Example:

```java
int x = 42;

if (x == 42) {
    System.out.println("condition is true");
}
```

Output:

```
condition is true
```

When executed, the condition is checked. If it evaluates to "true" the statement block will be executed.

IF-ELSE STATEMENT

Example:

```java
int x = 43;

if (x == 42) {
    System.out.println("condition is true");
} else {
    System.out.println("condition is false");
}
```

Output:

```
condition is false
```

When executed the condition is checked. If it evaluates to "true" the statement block will be executed. Otherwise the statements in the else-block will be executed. If-Else statemens can be nested as well as chained (see example on next page).

Example: Chained If-Else Statements

```java
// Character classifier: read character and print out its type
public class CharacterClassifier {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        System.out.print("Enter character: ");
        String line = scanner.nextLine();
        char c = line.charAt(0); // get first character in string
        if (c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z') {
            System.out.println("Letter");
        } else if (c >= '0' && c <= '9') {
            System.out.println("Digit");
        } else {
            System.out.println("Special character");
        }
    }
}
```

SWITCH STATEMENT

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths.

Example:

```
int x = 43;
switch(x) {
    case 42:
        System.out.println("It's 42!");
        break;
    case 43:
        System.out.println("It's 43!");
        break;
    default:
        System.out.println("Something else...");
}
```

Output:

```
It's 43!
```

BREAK IN SWITCH STATEMENTS

`break` must be used to end the execution of a case block. A missing `break` will result in the execution all the statements following the case block, no matter if the case conditions are met or not.

```java
int x = 42;
switch(x) {
    case 42:
        System.out.println("It's 42");
    case 43:
        System.out.println("It's 43");
        break;
    default:
        System.out.println("Something else...");
}
```

Output:

```
It's 42
It's 43
```

EXERCISE: ODD/EVEN NUMBERS

Write a Java program which reads a number from the console and prints information about the number being even or odd.
Hint: a number is even when it can be divided by 2 without a rest.

Example program runs:

```
Number: 13
The number 13 is odd

Number: 12
The number 12 is even
```

EXERCISE: COMPARE NUMBERS

In the following code fragment the numbers x and y are read in with a scanner. Check whether x is

- smaller,
- equal or
- greater than y

and output a message (see comments in the source code).

EXERCISE: NUMBER GUESSING

Develop a Java program with which the user can guess an imaginary number (here: 20). The user enters a number, then it is determined whether the number was guessed exactly or whether the number entered does not deviate from the number to guess by more than 30%.

3 sample run outputs illustrate the implementation:

```
Number: 20
Bullseye! You got it!
Number: 21
Super! The number to guess was 20. Your guess was within the range of (14, 2
Number: 10
What a pitty! The number to guess was 20.
You were outside of the range of (14, 26)
```

Attention: If necessary, use an (int) cast to convert a double value into an int value. Cast example:

```java
double amount = 3.15;
// the double value "amount" is converted into an int value
int amountAsInt = (int) amount;
```

EXERCISE: CONVERT DOG AGE TO HUMAN AGE

Implement the following algorithm in Java.

- A one-year-old dog corresponds to a 14-year-old child
- A two-year-old dog corresponds to a 22-year-old adult
- Each additional dog year corresponds to 5 human years

The code for reading the dog's age and the output of the result is already there.

EXERCISE: CHARACTER CLASSIFIER

Develop a character classifier.
A character is read into the variable "z", then an output is generated according to the classification of "z":

- '0' <= z <= '9': digit
- 'A' <= z <= 'Z': uppercase letter
- 'a' <= z <= 'z': lowercase letter
- otherwise: special character

EXERCISE: SALARY INCREASE

Develop a Java program that calculates the new annual salary.
The new annual salary should be computed as follows:

- For excellent employee performance, a bonus of 3000 EUR will be added
- For a good financial year for the company a bonus of EUR 2000 will be added
- Length of service for the company leads to the following increases in salary: 3-7 years → 0.5%, 8-15 years → 0.75%, from 16 years → 1%

Example output:

```
Annual salary: 60,000
Length of service (in years): 5
Excellent performance [Y/N]: J
Good financial year [Y/N]: N
New salary: 63300.00
```

1.12 LOOPS

How can we execute statements more than once?

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

Java offers three different loop types to execute certain parts of a program repeatedly.

While Loop

Do-While Loop

For Loop

WHILE LOOP

A while loop repeats a block of statements as long as a given condition evaluates to true. The condition is evaluated **before** the block is executed.

```java
int x = 0;

while (x < 3) {
    System.out.println("x = " + x);
    x++;
}
```

Output:

```
x = 0
x = 1
x = 2
```

DO-WHILE LOOP

A do-while loop repeats a block of statements as long as a given condition evaluates to true. The condition is evaluated **after** the block has been executed. Accordingly, the block will be executed at least once!

```java
int x = 0;
do {
    System.out.println("x = " + x);
    x ++;
} while (x < 3);
```

Output:

```
x = 0
x = 1
x = 2
```

FOR LOOP

The for statement provides a compact way to iterate over a range of values. In the loop head an **initializer expression**, a **test expression** and an **update expression** need to be specified.

```java
// for(<initializer>; <test>; <update>)
for (int x = 0; x < 3; x ++) {
    System.out.println("x = " + x);
}
```

Output:

```
x = 0
x = 1
x = 2
```

EXERCISE: SQUARE NUMBERS

Read an integer number from the console and output a table of square numbers ranging from 1 to the number your read.
Example program run:

```
Enter upper bound: 3
Number   Square
1        1
2        4
3        9
```

EXERCISE: EXPONENTIATE NUMBERS

In the following example, a scanner is used to read two integers x and y from the console.
Use a loop to calculate y^x. Use "System.out.println" to output the result to the console.

Caveat: Distinguish between three cases, see comments in the source code!

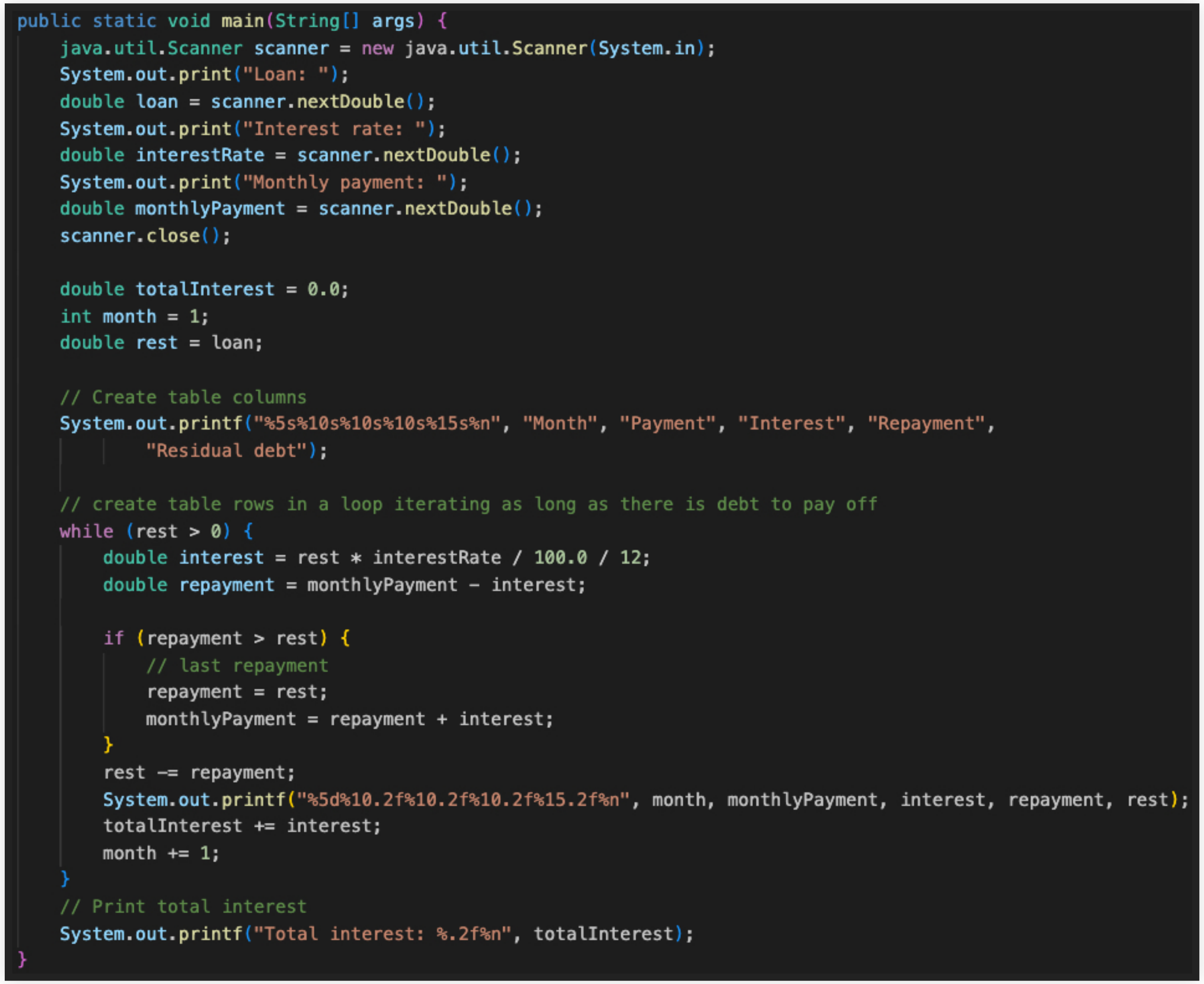
EXERCISE: INTEREST RATE CALCULATOR

Write a programme that creates an interest and repayment schedule for a loan.

Example program output:

```
Loan: 5500
Interest rate: 5,13
Monthly payment: 475
Month   Payment  Interest Repayment  Residual debt
    1    475,00     23,51    451,49        5048,51
    2    475,00     21,58    453,42        4595,09
    3    475,00     19,64    455,36        4139,74
    4    475,00     17,70    457,30        3682,44
    5    475,00     15,74    459,26        3223,18
    6    475,00     13,78    461,22        2761,96
    7    475,00     11,81    463,19        2298,77
    8    475,00      9,83    465,17        1833,59
    9    475,00      7,84    467,16        1366,43
   10    475,00      5,84    469,16         897,27
   11    475,00      3,84    471,16         426,11
   12    427,93      1,82    426,11           0,00
Total interest: 152,93
```

Unzip rate-calculator.zip and import the folder as "Existing Maven Project".



```java
public static void main(String[] args) {
    java.util.Scanner scanner = new java.util.Scanner(System.in);
    System.out.print("Loan: ");
    double loan = scanner.nextDouble();
    System.out.print("Interest rate: ");
    double interestRate = scanner.nextDouble();
    System.out.print("Monthly payment: ");
    double monthlyPayment = scanner.nextDouble();
    scanner.close();

    double totalInterest = 0.0;
    int month = 1;
    double rest = loan;

    // Create table columns
    System.out.printf("%5s%10s%10s%10s%15s%n", "Month", "Payment", "Interest", "Repayment",
            "Residual debt");

    // create table rows in a loop iterating as long as there is debt to pay off
    while (rest > 0) {
        double interest = rest * interestRate / 100.0 / 12;
        double repayment = monthlyPayment - interest;

        if (repayment > rest) {
            // last repayment
            repayment = rest;
            monthlyPayment = repayment + interest;
        }
        rest -= repayment;
        System.out.printf("%5d%10.2f%10.2f%10.2f%15.2f%n", month, monthlyPayment, interest, repayment, rest);
        totalInterest += interest;
        month += 1;
    }
    // Print total interest
    System.out.printf("Total interest: %.2f%n", totalInterest);
}
```

Solution Interest Rate Calculator

EXERCISE: OPERATORS AND LOOPS

How does the following loop work
and how many iterations will there be?

```
class Main {
  public static void main(String[] args) {
    int x = 0, y = 5;
    while(x++ < --y)
      System.out.println("Hello World");
  }
}
```

- ☐ 5 times
- ☐ 3 times
- ☐ 2 times

1.13 METHODS

So far one method has been used several times:

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

A `main`-Method defines a starting point for the execution of a Java program.

Just like many other programming languages, Java offers a function concept which allows to avoid repetitive instructions. A function is a block of statements that can be executed by calling the function, typically passing values as arguments (called parameters) to the function. A function may return a result value to the caller upon completion.

In Java, a function is called `method`.

`Static methods` can be called without "knowing" an object on which to call the method, which is the typical use of methods and objects. While the general concepts of methods will be covered in a later chapter, static methods and how they can be used will be explained here.

Methods have a name and can be called:

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

- The line `public static void sayHello()` defines a new method called `sayHello`.
- This line calls the new method.

Methods can return values and have a return type:

```java
public class MyProgram {
    public static String askForName() {
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        return scanner.nextLine();
    }

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

- Instead of `void` a return type can be specified, e.g. `String`
- Using the keyword `return` a value can be returned

Methods may have parameters:

The following example defines a method `arcMeasureToDegree`, which expects a parameter `x` of the type `double` and returns a value of type `double`.

```java
public class MyProgram {
    public static double arcMeasureToDegree(double x) {
        return x * 180.0 / 3.1415926536;
    }

    public static void main(String[] args) {
        System.out.println(arcMeasureToDegree(6.2828));
    }
}
```

Output:

```
359.977924
```

METHOD SIGNATURE

The signature of a method `m` is a tuple formed by the the parameter types and the name of `m`:

$$S_m := (\text{parType}_1, \text{parType}_2, \ldots, \text{parType}_n, \text{name})$$

EXERCISE: METHOD SIGNATURES

Which method signatures are correct?

- ☐ `public static String getNumber() { return "2.0"; }`
- ☐ `public static double getNumber { return 2.0f; }`
- ☐ `public static double getNumber() { System.out.println(2.0); }`
- ☐ `public static double getNumber() { return 2.0; }`
- ☐ `public static void getNumber() { return 2.0; }`
- ☐ `public static double getNumber() { return "2.0"; }`

EXERCISE: STATIC METHODS

This exercise is intended to create a simple static method that converts an angle from degrees to radians. In the following source code, add the method degreeToRadian, which expects a value of type double as a parameter and uses double as the return type.
The calculation can be done with the following formula: `x * 3.1415926536 / 180.0`

Then use this new method in the `main` method to convert the input to degrees. Save the result in a new variable called radians and output it with `System.out.println`.

1.14 DEBUGGING

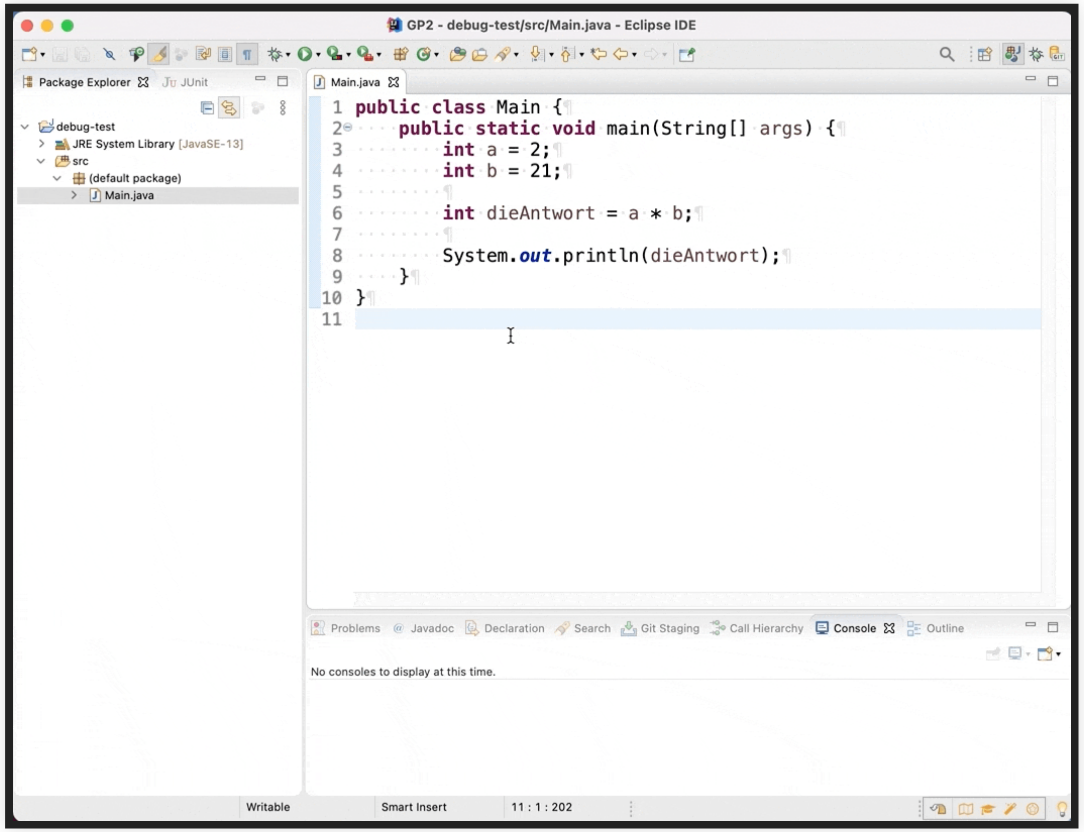
What is Debugging?

`Interactive execution of programs including the possibility to observe the change of state during execution.`

Modern IDEs like Eclipse offer powerful debugging tools that allow you to set breakpoints, step through code, inspect variables, and view the call stack.

- **Breakpoints**: A breakpoint is a marker that you place on a specific line of code. When the program execution reaches that line, it pauses, and you can examine variable values and step through the code line by line.
- **Step Through Code**: you can step through your code line by line. On "step over", the current line is executed while your focus stays in the current method, even if other methods are being called. On "step into", the current line is executed, with the focus moving to another method as it is called.
- **Inspect Variables**: While debugging, you can inspect the values of variables when a breakpoint is reached.
- **View Call Stack**: The call stack shows the sequence of method calls that led to the current point in the program. It can help you understand the flow of execution.

Running your application in debug mode

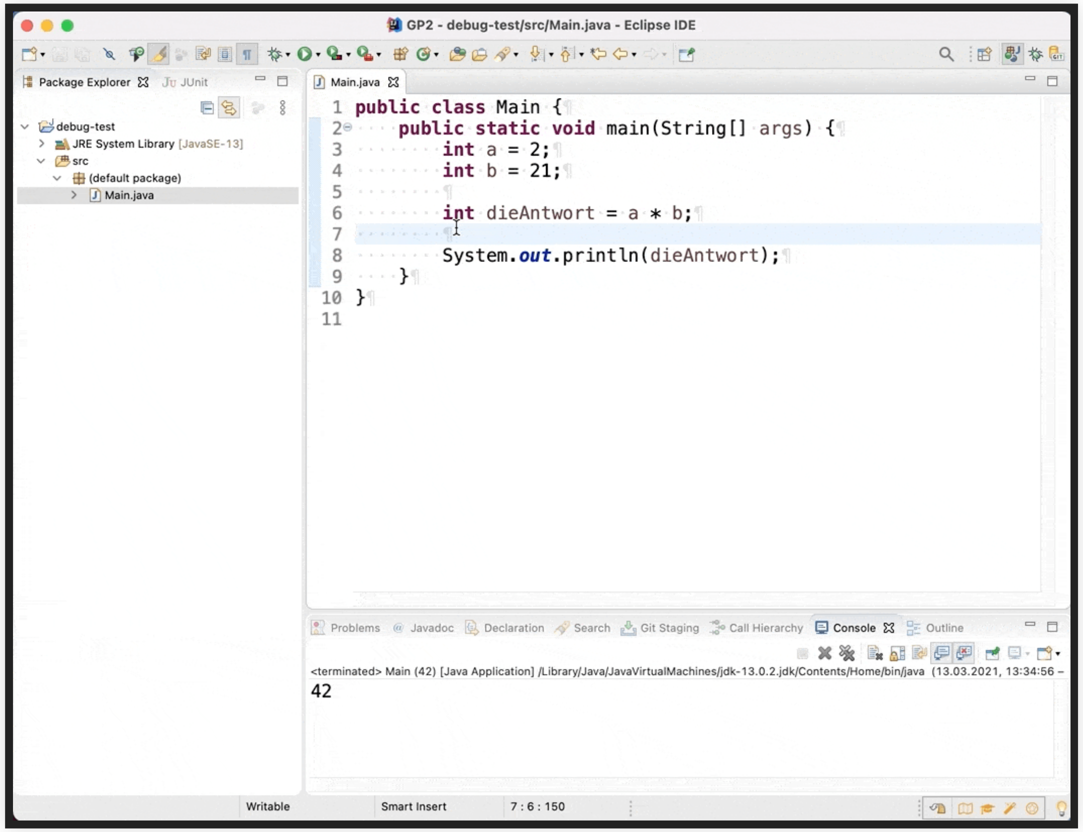


```java
public class Main {
    public static void main(String[] args) {
        int a = 2;
        int b = 21;

        int dieAntwort = a * b;

        System.out.println(dieAntwort);
    }
}
```

```
No consoles to display at this time.
```

Use the bug button to start debugging your application. With no breakpoint set, the application will run just normally.

Running your application in debug mode with breakpoint

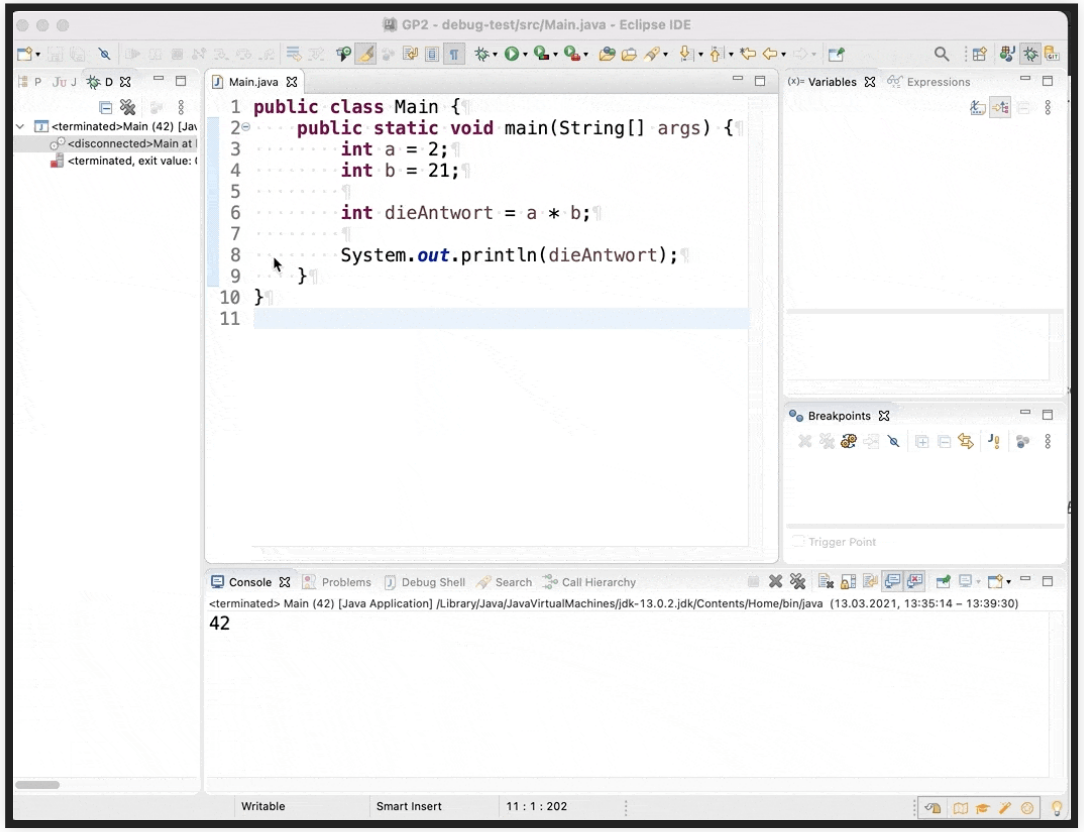


```java
public class Main {
    public static void main(String[] args) {
        int a = 2;
        int b = 21;

        int dieAntwort = a * b;

        System.out.println(dieAntwort);
    }
}
```

```
<terminated> Main (42) [Java Application] /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java  (13.03.2021, 13:34:56 –
42
```

When your set a breakpoint, Eclipse will automatically switch to the debug perspective and stop at the respective line. The perspective change is indicated by the the bug icon in the top-right corner of the eclipse window.

Stepping through your code

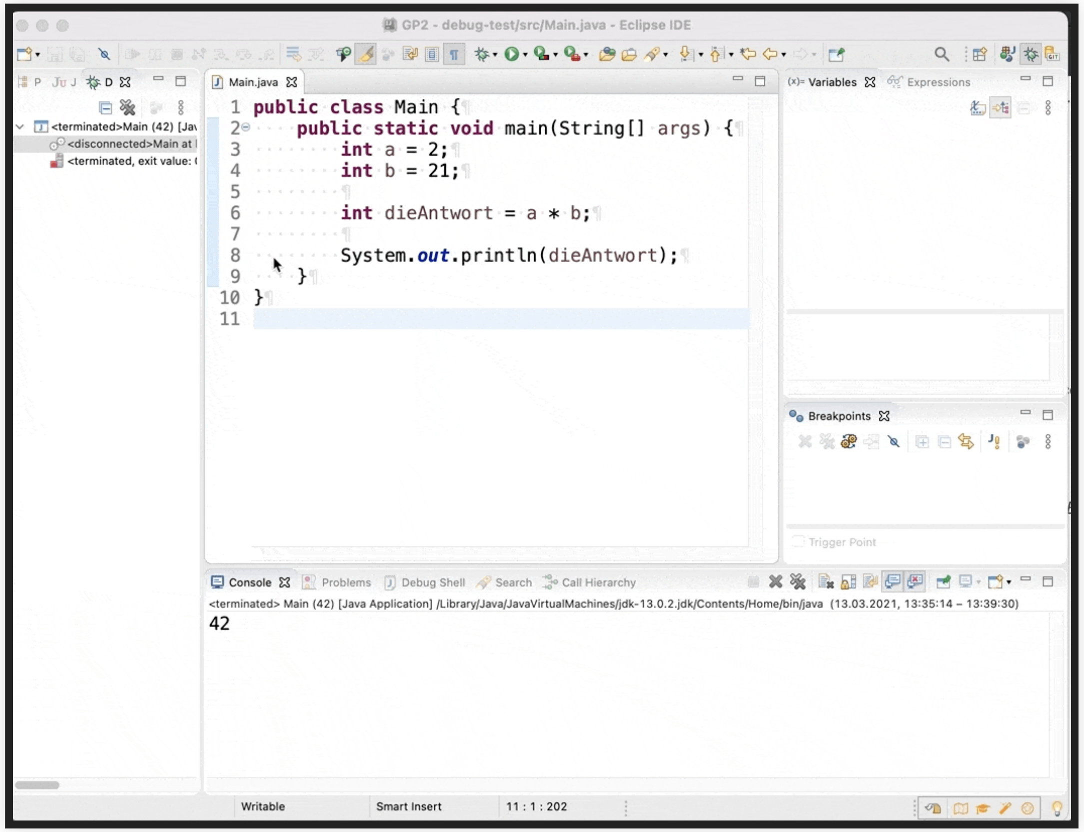


```java
public class Main {
    public static void main(String[] args) {
        int a = 2;
        int b = 21;

        int dieAntwort = a * b;

        System.out.println(dieAntwort);
    }
}
```

```
<terminated> Main (42) [Java Application] /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java  (13.03.2021, 13:35:14 – 13:39:30)
42
```

Once execution has stopped on a breakpoint you can step through your code line by line or resume execution until the next breakpoint is hit.

Observe variable values

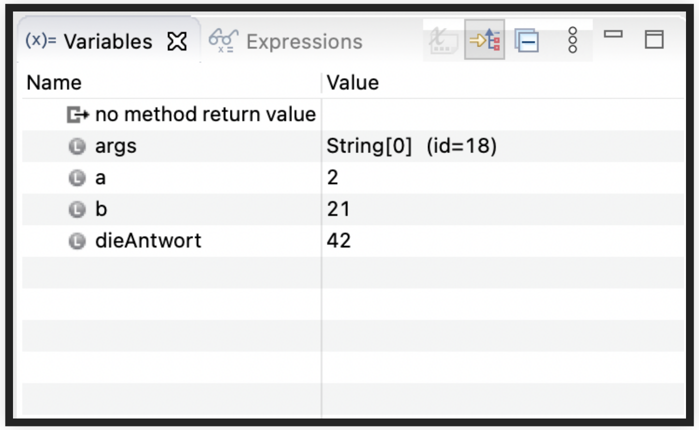


```
<terminated>Main (42) [Jav
<disconnected>Main at l
<terminated, exit value: (
```

```java
public class Main {
    public static void main(String[] args) {
        int a = 2;
        int b = 21;

        int dieAntwort = a * b;

        System.out.println(dieAntwort);
    }
}
```

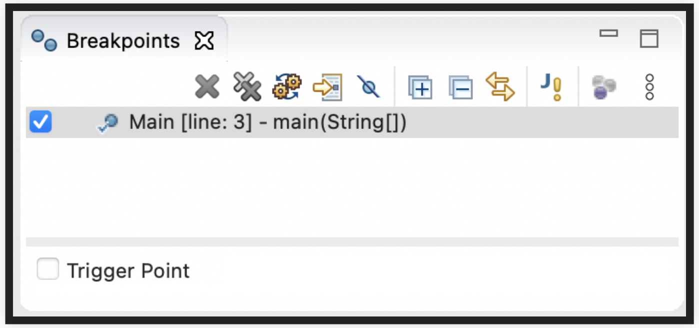
```
<terminated> Main (42) [Java Application] /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java  (13.03.2021, 13:35:14 – 13:39:30)
42
```

The variables view in top-right corner displays the variables (and attributes if objects are involved) which are visible in the current state of execution. For each variable or attribute the current value can be examined.



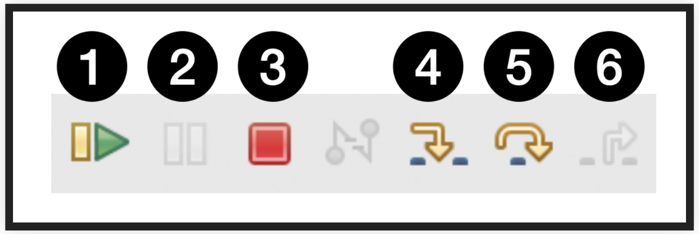
Name	Value
no method return value	
args	String[0] (id=18)
a	2
b	21
dieAntwort	42

For primitive data types, the values are output directly. For objects, the result of the `toString()` method is displayed (see chapter 2).



All breakpoints defined in the application can also be configured in the breakpoint view. This helps to stay oriented, especially when there are many Java classes.

NAVIGATING THROUGH YOUR CODE



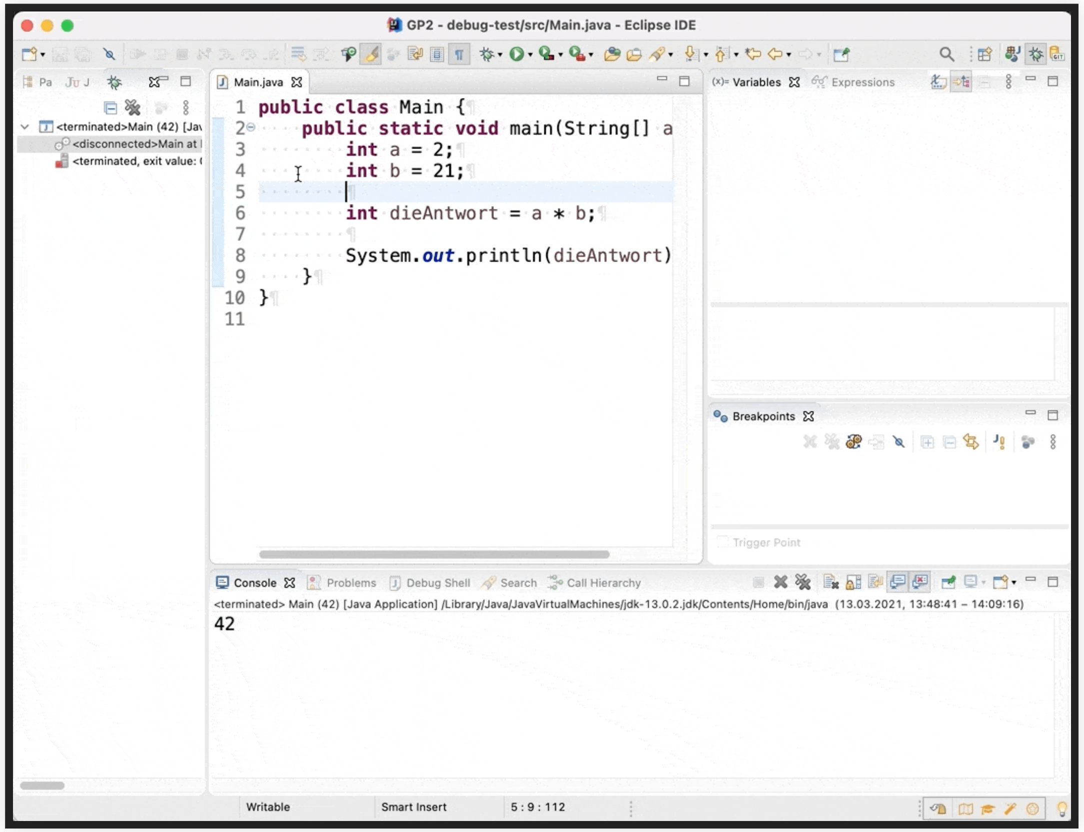
1. Play: resumes the execution of the application after it has been paused by a breakpoint.
2. Pause: suspends an application without the need for a breakpoint.
3. Stop: terminates the entire application.
4. Step into: Execute the current line and stop at the next line. If another method is called, this will stop at the first line of the method.
5. Step over: Execute the current line and stop at the next line in the current method.
6. Step return: Executes the remaining lines in the current method and stops after the return to the calling method.

CONDITIONAL BREAKPOINTS

Breakpoints may have conditions

- The breakpoint properties can be displayed by right-clicking in Eclipse.
- Here you can configure whether a breakpoint only pauses the current thread or the entire Java application.
- Boolean conditions for breakpoint activation can also be specified here.
- Caution: All forms of errors can also occur when evaluating conditions!

Configure a condition for a breakpoint



```java
public class Main {
    public static void main(String[] a
        int a = 2;
        int b = 21;

        int dieAntwort = a * b;

        System.out.println(dieAntwort)
    }
}
```

```
<terminated> Main (42) [Java Application] /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java  (13.03.2021, 13:48:41 – 14:09:16)
42
```

Once `a == 2` is satisfied, the application will pause.

1.15 COMPILING AND EXECUTING A JAVA PROGRAM

Java provides a way of modelling applications and formulating algorithms. A computer would not be able to execute it - it would be a simple text file on the hard drive. The task of compiling is to convert the Java program into an executable form.

Unlike "C", Java does not compile directly into machine-readable code. The result of the compilation process in Java is so-called **bytecode**. This is a machine-independent representation of the Java program, which can ultimately be executed by the `Java Virtual Machine` (JVM).

After installing the Java Development Kit (JDK) the application **javac** becomes available. **javac** converts a Java Class Definition (a text file!) into a binary bytecode file.
Attention: In case you installed a Java Runtime Environment (JRE) you will not find a Java compiler or other tools needed for development!

To translate file MyProgram.java into bytecode, just type the following line in a terminal. The result will be the file MyProgram.class.

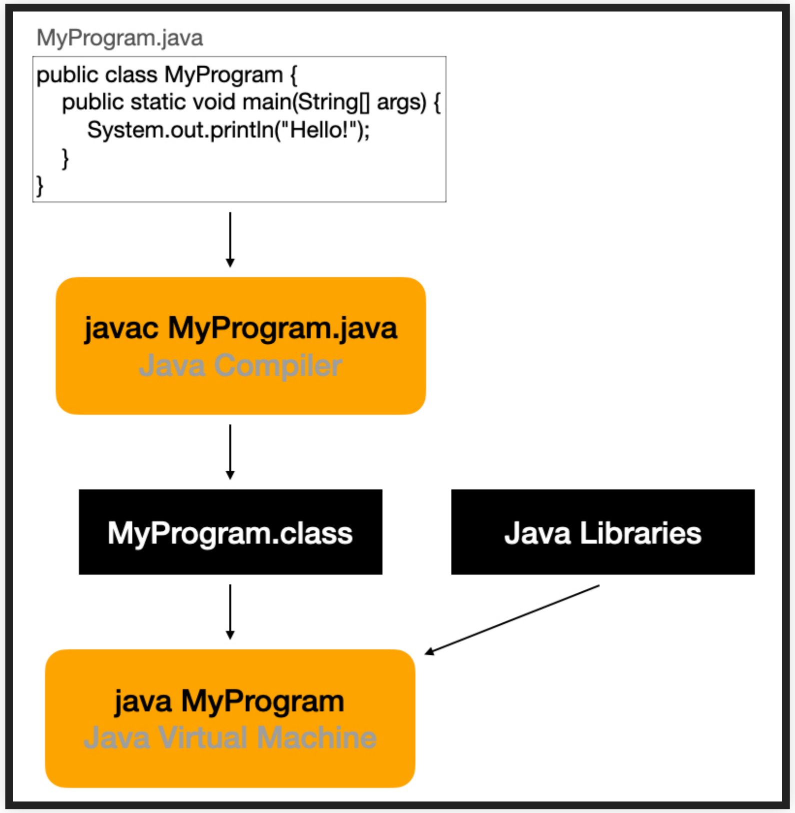
```
javac MyProgram.java
```

Even the bytecode in class files created by compiling Java files is not directly executable. This requires a special environment in which the application is executed, the Java Virtual Machine (JVM).

The JVM is supplied with both a JDK and a JRE. A class file is then executed via the command `java`, for example in the terminal:

```
java MyProgram
```

OVERVIEW OF THE COMPILE AND EXECUTE PROCEDURE



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

One way of distributing a set of class files in Java is to use so-called jar files (Java archives) - these are zip archives with the extension .jar which contain all the class files belonging to an application. In addition, a manifest file can be specified which determines the main class for the application (every class may define its own main method!).

In a Linux console one could do the following to create an executable jar file:

```
javac MyProgram.java
echo "Main-Class: MyProgram" > manifest.txt
jar cvfm MyJarName.jar manifest.txt MyProgram.class
```

To start the application:

```
java -jar MyJarName.jar
```

EXERCISE: COMPILER

Save the following code snippet in file `MyProgram.java`.

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

- Translate the Java file into a class file.
- Execute the class file, it should print "Hello World!" on the console
- Create a Jar file
- Execute the Jar file

1.16 EXERCISES

CALCULATING THE GREATEST COMMON DIVISOR (GCD) AND LEAST COMMON MULTIPLE (LCM)

Implement a simple method to calculate the GCD and calculate the LCM using integers.

Unzip gcd-lcm.zip and import the folder as "Existing Maven Project".

STEP 1: METHOD GCD

Create a simple method `GCD` in file Main.java that receives two integers as parameters and returns an integer. The method should calculate the greatest common divisor. Find out how to calculate the result from the inputs by searching the Internet and realise the calculation using a loop and local variables.

Test your implementation.

STEP 2: METHOD LCM

Create a simple method `LCM` in file Main.java that receives two integers as parameters and returns an integer containing the least common multiple of the inputs. Search the Internet to find out how to calculate the result. The calculation can be performed using the `GCD` method.

STEP 3: OUTPUT

Use the following Main method, calculate the greatest common divisor and the least common multiple using the two methods `GCD` and `LCM`, save the results in local variables and output them using the prepared output statements.

```java
public static void main(String[] args) {
  java.util.Scanner scanner = new java.util.Scanner(System.in);
  System.out.print("m = ");
  int m = scanner.nextInt();
  System.out.print("n = ");
  int n = scanner.nextInt();

  // ... call GCD and LCM

  // output results
  System.out.printf("GCD( %d, %d) = %d\n", m, n, gcd);
  System.out.printf("LCM( %d, %d) = %d\n", m, n, lcm);
}
```

1.17 JAVA BOOKS

For absolute beginners:

- Iuliana Cosmina: Java 17 for Absolute Beginners: Learn the Fundamentals of Java Programming; 2. Edition; Apress Berkeley, CA; 2021
- Mikael Olsson: Java 17 Quick Syntax Reference; 3. Edition; Apress Berkeley, CA; 2022