Lecture - Operators

Overview

In this module, you will learn:

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using the *Scanner* class

> "A computer terminal is not some clunky old television with a typewriter in front of it. It is an **interface** where the mind and body can connect with the universe and move bits of it about." ~ **Douglas Adams, *Mostly Harmless***

Question in lecture:

Q1 Java code can be compiled on any machine but can be run on the same machine/operating system only. → False

Q2 main is the entrypoint of a program: True

Q3: JVM and JIT are softwares that compiles the code from high level language to machine level language. True

Q4: bool is data type in Java: False (is boolean not bool)

Q5: System.out.println("Hello World %n Bye") should be printed on terminal

Hello World
Bye

→ False : Because we use println not printf so it won't go to the next line.

Q6. What will be the output?

```
System.out.printf("$%06.2f%n", 1.23)
```

$001,23

Operators and Operations

Remember the following primitive types in Java?

Type	Size (Bytes)	Contains	Values (Range)	Example	Default values (for fields)
boolean	not precisely definited, typically 1 bit but size is JVM dependent	boolean values true or false	-	boolean isStudent = true;	false
char	2 (16 bits)	unicode characters	\u0000' (or 0) to '\uffff' (or 65,535 inclusive)	char c = 'c';	\u0000'
byte	1 (8 bits)	signed integer	-128 to 127	bytes b = 100;	0
short	2 (16 bits)	signed integer	-32,768 to 32,767	short s = 1000;	0
int	4 (32 bits)	signed integer	-2,147,483,648 to 2,147,483,647	int i = 1000000;	0
long	8 (64 bits)	signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long l = 100000000L;	0
float	4 (32 bits)	IEEE 754 floating point	±3.40282347E+38F (6-7 significant decimal digits)	float f = 1.45f;	0.0f
double	8 (64 bits)	IEEE 754 floating point	±1.79769313486231570E+308 (15 significant decimal digits)	double d = 1.457891d;	0.0d

Number Types

Each type has certain operations that apply to it. Common ***operations (also called operators)*** for primitive number types are:

- Addition (+) and Subtraction (-)
- Multiplication (*) and Division (/) as well as Modulo (%), i.e., the remainder

The numbers we apply the operation to are called the ***operands***. The data type of the result is the same as the type of the operands.

```
int number1 = 10;
int number2 = 3;
int result = number1 + number2;
```

int result = number1 / number2 ⇒ 3 (not 3.33 because result variable is integer)

In the above example, *number1* and *number2* are the operands. The operation is addition (+).

Operations are used to construct *expressions*, which have values that can be assigned or used as operands:

```
int answer=(2+4)*7;
```

Here are these operations in action:

```
public class NumberPlay {
    public static void main(String[] args) {
        int number1 = 10;
        int number2 = 3;
        int addition = number1 + number2;
```

```
        int subtraction = number1 - number2;
        int multiplication = number1 * number2;
        int division = number1 / number2;
        int modulo = number1 % number2;

        System.out.println("Addition: " + number1 + " + " + number2 + " = " + addition);
        System.out.println("Subtraction: " + number1 + " - " + number2 + " = " + subtraction);
        System.out.println("Multiplication: " + number1 + " * " + number2 + " = " + multiplication);
        System.out.println("Division: " + number1 + " / " + number2 + " = " + division);
        System.out.println("Modulo: " + number1 + " % " + number2 + " = " + modulo);
    }
}
```

Run this, and experiment with different values of `number1` and `number2`.

Does the division seem strange to you?

3 fits into 10 exactly 3 times since both numbers are integers. If you want Java to properly calculate the quotient, you need to use floating point numbers, such as *float* or *double*.

```
public class Division {
    public static void main(String[] args) {
        float number1 = 10;
        float number2 = 3;
        float properDivision = number1 / number2;

        System.out.println("Division: " + number1 + " / " + number2 + " = " + properDivision);
    }
}
```

Comparison Operations

The following comparison operations also work for number types:

```
<     : less than
<=    : less than or equal to
>     : greater than
>=    : greater than or equal to
==    : equal to
!=    : not equal to
```

Comparisons always return a boolean (true / false) value:

```
public class Comparison {
    public static void main(String[] args) {
        boolean result = (5!=4);
        System.out.println(result);
    }
}
```

Operations for Booleans

You can construct logic statement in Java using:

```
&& (AND)    : is true if both operands are true
|| (OR)     : is true if either operand is true
```

Both of these are so-called 'short-circuit' operations, *i.e.*, the second operand is only evaluated if necessary. If the first argument of `&&` is `false`, or the first argument of `||` is `true`, then the second one is not necessary because it cannot change the value of the expression.

Then, there is also negation:

```
! (NOT)     : is true if its operand is false
```

Let's look at some examples:

```
public class Comparison {
    public static void main(String[] args) {
        // AND
        int x = 5;                    Both true
        boolean expression = (x!=4) && (x>3);=>True
        System.out.println(expression);

        // OR
        expression = (x!=4) || (x<3); // is true no matter what comes after ||
        System.out.println(expression);

        // NOT
        System.out.println(!(x==4));
    }
}
```

Increments and Decrements

There are two types of increments and decrements: pre and post. The **pre-increment** is a special expression that increments a number before returning the incremented value. The **pre-decrement** works similarly only that it decrements a number before returning it:

```
public class PreIncrement {
    public static void main(String[] args) {
        int x = 5;
        // pre-increment
        System.out.println(x);   5
        System.out.println(++x); 6

        // pre-decrement
        System.out.println(--x); 5
    }
}
```

Why lazy evaluation is tricky?
=>boolean expression = (--x!=4) && (x++>3);

The post-increment, on the other hand, returns the value before incrementing. The post-decrement works accordingly.

```
public class PostIncrement {
    public static void main(String[] args) {
        int x = 5;
        // pre-decrement
        System.out.println(x++);
        System.out.println(x);

        // pre-decrement
        System.out.println(x--);
        System.out.println(x);
    }
}
```

Pre/post increment/decrements can also be used as statements rather than expressions:

```
++x;
// or
x++;
```

When used as a statement, both versions just increment *x*.

> ⚠ If an expression uses a variable, say `x`, more than once, and one has an increment or decrement, the expression becomes confusing for the reader, even though it isn't to the compiler (https://stackoverflow.com/questions/23308228/how-are-java-increment-statements-evaluated-in-complex-expressions). It is generally best not to use increment or decrement in that way.

Type Conversions

Primitive operations generally work on operands of the **same** type. Java can, however, convert types automatically if the operands have different types. A **widening conversion** converts a number to a wider type so the value can always be converted successfully:

```
public class Conversion {
    public static void main(String[] args) {
        int x = 5;
        long y = 10;
        float z = 20;
        System.out.println(x+y);
        System.out.println(x+z);
    }
}
```

Java converts automatically between the following:

```
byte -> short -> int -> long -> float -> double
```

Of these, only `int -> float`, `long -> float` and `long -> double` can result in an inexact conversion.

Exercise: Find an example of a long value that is not converted exactly to a float. Use the above code block to experiment.

The *char* type is a special case. While it technically is not an *int* it is an integral type, *i.e.,* it is considered to be a whole number that can be converted to and from other integral types:

```
public class Conversion {
    public static void main(String[] args) {
        char c = 'J';
        long y = 10;
        System.out.println(c+y);

    }
}
```

A *char* converted to an *int* represents the corresponding ASCII code (or Unicode code point) of the character. **Typecasting** can be used to convert an *int* to a *char* type:

```
public class Conversion {
    public static void main(String[] args) {
        int j = 74;
        int a = 65;
        int v = 86;
        System.out.println((char)j + "" + (char)a + "" + (char)v + "" + (char)a);
    }
}
```

Such casting is a **narrowing conversion**. It needs to be specified by writing the name of the type to convert to in parentheses before the value that is to be converted. A cast can also be used to explicitly ask for a widening conversion:

```
public class Conversion {
    public static void main(String[] args) {
        //narrowing
        short x;
        int y = 50;
        x = (short) y;
        System.out.println(x);

        //narrowing
        int sum = 10;
        int count = 2;
        double average = (double)sum / count;
        System.out.println(average);
    }
}
```

List of Operators supported by Java

```
Simple Assignment Operator
=       Simple assignment operator

Arithmetic Operators
+       Additive operator (also used
        for String concatenation)
-       Subtraction operator
*       Multiplication operator
/       Division operator
%       Remainder operator

Unary Operators
+       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

Equality and Relational Operators
==      Equal to
!=      Not equal to
>       Greater than
>=      Greater than or equal to
<       Less than
<=      Less than or equal to

Conditional Operators
&&      Conditional-AND
||      Conditional-OR
?:      Ternary (shorthand for if-then-else statement)

Type Comparison Operator
instanceof      Compares an object to a specified type

Bitwise and Bit Shift Operators
~       Unary bitwise complement
<<      Signed left shift
>>      Signed right shift
>>>     Unsigned right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive OR
```

it's often recommended to avoid ***instanceof*** operator as it can sometimes indicate design issues that can hurt the maintainability and flexibility of your code. It is recommended to use Polymorphism instead, which we plan to cover in coming weeks.

example

```
int a= 20;
String message = (a>=0 ? "positive": "negative");
```

expression? true statement : false statement

Precedence and Associativity

Operator precedence determines the order in which the operators in an expression are evaluated.

$$\text{Precedence of two operators, say } \odot \text{ and } \oplus \text{, determines whether } a \odot b \oplus c \text{ is read as:}$$

$$(a \odot b) \oplus c \quad (\odot \text{ has higher precedence}) \text{ or}$$

$$a \odot (b \oplus c) \ (\odot \text{ has lower precedence})$$

For example, 2+3*4=14 as * has higher precedence.

$$\text{Associativity determines whether } a \odot b \odot c \text{ is read as:}$$

$$(a \odot b) \odot c \text{ (left associativy) or}$$

$$a \odot (b \odot c) \text{ (right associativity)}$$

For example, 3-2-1=0 (- associates left). *If* it were associated right (which it doesn't!), then 3-2-1 would be 3-(3-2) = 3-1 = 2.

The following table lists the precedence of operators in Java. The higher it appears in the table, the higher its precedence:

post-decrement

pre-increment

Operators	Precendence	Associativity
Postfix	++ - -	Left to right
Unary	+ - ! ~ ++ - -	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left

Here is the example of the operator precedence and associativity in action

```
public class OperatorPrecedenceDemo {
```

```
    public static void main(String[] args) {
        int x = 10, y = 5, z = 2;

        // Precedence: Multiplication (*) before addition (+)
        int result1 = x + y * z;  // Evaluates to 20, not 70
        System.out.println("result1: " + result1);  20

        // Precedence: Parenthesis () overrides default precedence
        int result2 = (x + y) * z;  // Evaluates to 30
        System.out.println("result2: " + result2);  30

        // Associativity: Arithmetic operators associate left-to-right
        int result3 = x - y + z;  // Evaluates to 7 (left-to-right: 5, then 7)
        System.out.println("result3: " + result3);  7
    }
}
```

I suggest extending the above code example to practice the associativity and precedence of other operators listed in the table above.

Additional Reading Resources

- WALTER, S. Absolute Java, Global Edition. [Harlow]: Pearson, 2016. (Chapter 1, 2 and 3)
- SCHILDT, H. Java: The Complete Reference, 12th Edition: McGraw-Hill, 2022 (Chapter 4)
- Language Basics (accessible on 14-02-2024) Oracle's Java Documentation. Available at: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html

Quick Quiz

The following quiz is designed to help you test your understanding of variable types and simple input/output behaviour in Java.

The quiz is not marked.

Question 1

Pre vs post increment

What is the output of the following code?

```
int x = 10;
int y = 5;
System.out.println(x++ - ++y);
```

10 - 6 : 4

- 3
- 4
- 5
- 6
- 7

Question 2

Precedence and Associativity

After running the following code, what will be the value of x, y, and z?

```
int x = 10, y = 5;
int z;

z = --x - y * 5 + x * (y++ - 4);
```

9 - 25 + 10

:- 7

- x=10, y=5, z=5
- x=9, y=5, z=-7
- x=9, y=5, z=5
- x=9, y=6, z=-7

Question 3

What will be the result if you put the following lines in your code?

```
x = 10;
y = (x++) * (++x)
```

10 11

y=120 X=12

- y = 121, x = 12
- y = 144; x = 12
- The programmer after you gets confused and wastes lots of time, then replaces it by something clearer.

non-primitive data type

The String Type

String is a class type, not a primitive type, so strings are objects. Strings are widely used in Java Programming and they are sequence of characters. A *string* constant is specified by enclosing it in double-quotes (""):

```
String greetings = "Hello World!";
```

is object

Using a backslash (\) allows you to include double-quotes and other special characters (including \ itself) in a *string*:

```
public class StringPlay {
    public static void main(String[] args) {
        System.out.println("He said a \"backslash (\\) is special!\"");
        System.out.println("Windows file names become C:\\users\\fred");
    }
}
```

escape

If you don't back slash, you need another blank slash to print it out.

one single slash

He said a "backslash (\) is special!"
Windows file names become C:\users\fred

Certain letters after a backslash are treated specially, for example, \n for a new line and \t for a tab character.

```
public class StringPlay {
    public static void main(String[] args) {
        System.out.println("I\nlike\nme\nsome\nnew\nlines!");
    }
}
```

It is a tab

Exercise: Edit the above to make it print two lines, with three tab-separated words on each.

String Operations

Two strings can be appended using +, an operation also called **concatenation**. If either operand is a *string*, the + operation will convert the other operand into a string:

```
public class StringPlay {
    public static void main(String[] args) {
        // simple concatenation
        System.out.println("Hello " + "String!");

        // conversion
        int x = 1;
        System.out.println("x = " + x);

        // can you figure out (and fix) what happens here?
        System.out.println("x + x = " + x + x);
```

X+X=11

ex: "X+X=" + (X+X); X+X=2

```
        // Explain what happens here.
        String strJ = "J";
        char chrJ = 'J';
        System.out.println ("With a string we get " + (strJ + 1));
        System.out.println ("With a char we get " + (chrJ + 1));
    }
}
```

Exercise: Explain what happens with the curious examples above.

The *String* class comes with a whole range of useful operations, which you can look up in the Java Doc. Try to find and add two more *string* operations to the following code:

```
public class StringPlay {
    public static void main(String[] args) {
        String s = "A piece of string walks into a bar...";

        // returns length of the string
        System.out.println("String length: " + s.length());

        // returns ALL UPPER CASE version of the string
        System.out.println("All upper case: " + s.toUpperCase());

        // s.substring(i, j) returns the substring of s from character i through j-1, counting the
        System.out.println("A substring: " + s.substring(0, 17));

        // string comparison: returns true two strings are identical
        System.out.println("String comparison: " + s.equals("something else"));
        // note: do not use ==, <, >, >=, or <= to compare strings

        // you can retrieve indices from strings
        System.out.println("Index of 'string': " + s.indexOf("string"));

        // TODO: find and try out two more string operations here:

    }
}
```

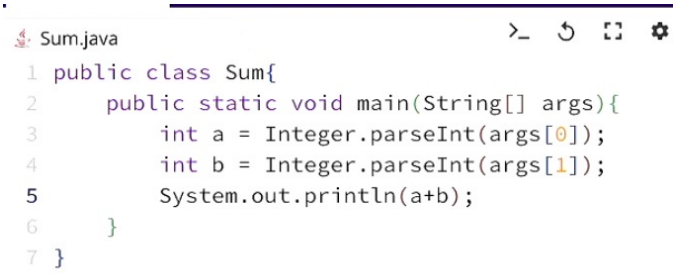
Exercise: Modify the code to find the index of the first "A" in the string. Does that surprise you?

Command Line Sum

Create a program (*Sum.java*) that takes *two numbers* as input, adds them, and returns the result to the console.

Here is an example invocation and output of your program:

```
$ java Sum 4 5
9
```



Interactive Sum

Just like before, create a program (*InteractiveSum.java*) that specifically asks the user for *two numbers*, adds them, and returns the result to the console.

Here is an example invocation and output of your program:

```
$ java InteractiveSum
Please enter the first number:
```

Let's say the user enters 4 and hits *return*:

```
Please enter the first number:
4
Please enter the second number:
```

Let's say the user enters 5 and hits *return*:

```
Please enter the second number:
5
The sum of 4 and 5 is 9
```

Is better to read the input by nextLine() not nextInt()

And then transfer the string to the integer by

int a = Integer.parseInt(keyboard.nextLine())

```
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Please enter the first number:")
    int a = Integer.parseInt(keyboard.nextLine());
    double b = Double.parseDouble(keyboard.nextInt());
    keyboard.nextLine();
    System.out.println("Please enter the second number:"
    int c = keyboard.nextInt();
    System.out.println("The sum of "+ a+ " and "+b+ " is
nteractiveSum.java 9:58  Spaces: 4 (Auto)        All changes saved
```

Quiz

The following quiz is designed to help you test your understanding of variable types and simple input/output behaviour in Java.

The quiz is not marked.

Question 1

What is the output of the following program:

```
int x = 10;
int y = 4;
System.out.println("Result is: " + x*y + (double)x/y + x%y);
```

40 2.52

- Result is: 44
- Result is: 44.5
- Result is: 402.52
- I don't know

Question 2

What is the output of the following program:

```
int x = 10;
int y = 4;
System.out.println((x > y) && (x % y ==2) || (++x == 10));
```

True True

- true
- false
- (40 > 4) && (10 % 4 == 2) || (++10 == 10)

- I don't know

Question 3

What is the output of the following program:

```
double pi = 3.14159265359;
System.out.printf("%.4f : %-10.4f : %d\n", pi, pi, (int)pi);
```

- 3.1416 : 3.1416 : 3.0
- 3.1416 : 3.1416 : 3
- 3.1416 : 3.1416 : 3
- I don't know

Question 4

What is the output of the following program if you run it using the following program execution:

```
java Quiz 10 4
```

Program:

```
public class Quiz {

    public static void main(String[] args) {
        System.out.println(args[0] + args[1]);
    }
}
```

- 104
- 14
- This code will produce an error/exception

- I don't know

Question 5

What are the values of x, s, and y after running the program below with the following parameters:

```
java IOTest
8
Test
9
```

Program:

```
import java.util.scanner;

class IOTest {
  public static void main (String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int x = keyboard.nextInt();
    String s = keyboard.nextLine();
    int y = keyboard.nextInt();
  }
}
```

- x = 8, s = "Test", y = "9"
- x = 8, s = "", y = "9"
- This code will produce an error/exception
- I don't know

X=8

S="\n"

y="Test"

Relevant Reading Resources

Additional Reading Resources

- WALTER, S. Absolute Java, Global Edition. [Harlow]: Pearson, 2016. (Chapter 1, 2 and 3)
- SCHILDT, H. Java: The Complete Reference, 12th Edition: McGraw-Hill, 2022 (Chapter 4)
- Language Basics (accessible on 14-02-2024) Oracle's Java Documentation. Available at: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html