Assignments

1. Write a program that will display a multiplication table to one hundred.

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

   - For formatting, you can use the
     - String.format("%4s", <value-to-display>)
2. Extend the multiplication table with horizontal and vertical lines and borders composed of characters |, – and +. For the table to 9 it could look like below.

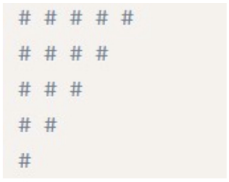
```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 2 | 4 | 6 |
+---+---+---+
| 3 | 6 | 9 |
+---+---+---+
```

3. Write a program that would draw the following patterns
   1. Triangle 1

```
#
# #
# # #
# # # #
# # # # #
# # # # # #
# # # # # # #
# # # # # # # #
```

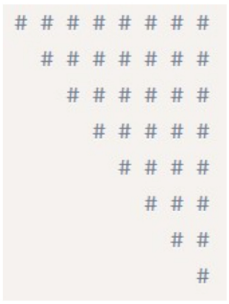
   2. Triangle 2

```
# # # # # # # # #
 # # # # # # # #
  # # # # # # #
```



3. Triangle 3

4. Triangle 4



5. Square

6. Letter *S*

7. Letter *Z*

8. Hourglass

9. Square with diagonals

10. Based on the above figures, propose your own ones; try also to fill in the selected parts of the figures.
11. Transfer the logic displaying the above figures to separate methods that take the size (e.g. a parameter named `size`) of the figure (the number of characters in the figure) as a parameter.

4. Write a program multiplies the digits for a given number (as `String`) until the result is a one-digit number. E.g.**Example for "123".**`123 123`
   - **Example for "123".**
     - 123
     - 123 → 1x2x3 = 6
   - **Example for "277777788888899".**
     - 277777788888899
     - 277777788888899 → 2x7x7x7x7x7x7x8x8x8x8x8x8x9x9 = 4996238671872
     - 4996238671872 → 4x9x9x6x2x3x8x6x7x1x8x7x2 = 438939648
     - 438939648 → 4x3x8x9x3x9x6x4x8 = 4478976
     - 4478976 → 4x4x7x8x9x7x6 = 338688
     - 338688 → 3x3x8x6x8x8 = 27648
     - 27648 → 2x7x6x4x8 = 2688
     - 2688 → 2x6x8x8 = 768
     - 768 → 7x6x8 = 336
     - 336 → 3x3x6 = 54
     - 54 → 5×4 = 20
     - 20 → 2×0 = 0
   - Use the code below to implement the `reduce(String number)` method.
     - public class Reducer {
     - public static void main(String[] args) {
     - Reducer reducer = new Reducer();
     - String numberToReduce = "<enter value here>";
     - int reducedValue = reducer.reduce(numberToReduce);
     - System.out.println("Number to reduce: " + numberToReduce);
     - System.out.println("After reduce: " + reducedValue);
     - }
     - public int reduce(String number) {

- public int reduce(String number) {
- return 0;
- }
- }

- Consider how to optimize this program. Hint: *digit zero*.

5. Write a program that checks who won a *Tic-Tac-Toe* game.
    1. The status of the board is presented as a `String`, e.g.

```
0X_00X0_X
```

+ where: * X means a *cross* * 0 means a *circle* * _ mean an empty field

1. Replace the above entry with a two-dimensional array:

```
0X_
00X
0_X
```

1. Determine who won or whether that was a tie.
2. Sample input data:

```
X0XXX000X
X0XX00XX0
X0XX000X0
0X_00X0_X
X00X0X0X0
0X0X00XXX
```

1. Adjust the program to handle uppercase and lowercase letters (x, o, X, 0) and `0` (zero) as a *circle*.
    1. Write a program that shortens the contents of text messages. The program should:
        - remove unnecessary blank characters at the beginning and end of the contents
        - remove empty characters between words and start each word with a capital letter Example of program operation

        Example of program operation

        ```
        Alice has a cat, and a cat has Alice!
        AliceHasACat,AndACatHasAlice! Hey, I
        will be back later tonight. Do not wait
        with dinner for me.
        Hey,IWillBebackLaterTonight.DoNotWaitWithDinnerForMe.
        ```

In addition, make the program display: * the number of characters of the original message * the number of characters after compression * the price for sending a text message, assuming that **1 text message** is **160 characters**, and each message costs 25 cents.

1. Write a program that validates the correctness of a given personal **ID number**

personal **ID number**.

The personal ID number has the following form:`RRMMDDXXXXK` where:

- `YY` – year
- `MM` – month
- `DD` – day
- `XXXX` – ordinal number
- `C` – control digitAssign the values to variables,
- e.g.`RRMMDDXXXXK abcdefghijk`
- `result` = “1”x“a” + “3”x“b” + “7”x“c” + “9”x“d” + “1”x“e” + “3”x“f” + “7”x“g” + “9”x“h” + “1”x“i” + “3”x“j” + “1”x“k”
- If `result % 10 = 0`, the personal ID number is correct!
    1. Suggest a signature for the method of validation of the personal ID number correctness

2. A `Car` class is provided
    - `public class Car {`
    - `private String model;`
    - `private int productionYear;`
    - `private String color;`
    - `private boolean used = false; }`
3. Generate *getter* and *setter* methods for all fields; use key shortcuts Alt+Ins
    1. Test the `Car` class using the `CarApplication` class
        - `public class CarApplication {`
        - `public static void main(String[] args) {`
        - `Car audi = new Car();`
        - `audi.setModel("A8");`
        - `audi.setColor("red");`
        - `audi.setProductionYear(2018);`
        - `audi.setUsed(true);`
        - `System.out.println(audi.getModel());`
        - `System.out.println(audi.getColor());`
        - `System.out.println(audi.getProductionYear());`
        - `System.out.println(audi.isUsed()); } }`
    2. Add a field to store the mileage (`int mileage`) to the `Car` program; set the default value to `0`
    3. Add (generate) a *getter* type method for this field (`getMileage()`)
    4. Add the `drive(int mileage)` method that increases the mileage
    5. Test the program operation
    6. Modify the `drive(int mileage)`) method to set the `used` field to the appropriate value; if the `mileage` is positive, the `used` field should have value `true`
    7. Test the operation
4. The `Calculator` and `CalculatorApplication` classes are provided
5. `public class Calculator { public int add(int a, int b) { return a + b; } } public class CalculatorApplication { public static void main(String[] args) { Calculator calc = new Calculator(); System.out.println(calc.add(5, 12));`

```
} }
```

1. Based on the `add(int a, int b)` method, implement the subsequent methods:
    1. `int subtract(int a, int b)` – subtraction: `a - b`
    2. `int multiply(int a, int b)` – multiplication: `a * b`
    3. `double divide(int a, int b)` – division: `a / b`
    4. `boolean isPositive(int a)` – checks whether the number is positive
    5. `boolean isNegative(int a)` – checks whether the number is negative
    6. `boolean isOdd(int a)` – checks whether the number is odd
    7. `int min(int a, int b)` – returns the smaller of the numbers
    8. `int max(int a, int b)` – returns the greater of the numbers
    9. `double average(int a, int b)` – returns the average for the numbers
    10. `int power(int a, int x)` – returns `a^m^` (`a` to the power `m`)
2. Based on the solution from point *a*, implement 3-argument versions of these methods (try to use

2 – argument versions of these methods):

```
1. `int add(int a, int b, int c)`

2. `int subtract(int a, int b, int c)`

3. `int multiply(int a, int b, int c)`

4. `double divide(int a, int b, int c)`

5. `int min(int a, int b, int c)`

6. `int max(int a, int b, int c)`

7. `double average(int a, int b, int c)`
```

4. Implement a stack to store `Integer` type numbers.`public class Stack { public Stack(int count) {} public void push(Integer e) {} public Integer pop() {} public boolean isEmpty() {} public boolean isFull() {} public String toString() {} }`
    1. the constructor creates a `count` element array for storing `Integer` type elements
    2. the `push()` method throws the element onto the stack (to the first free position). If the stack is empty, a

relevant message is displayed and nothing happens

```
3. the `pop()` method deletes (returns and removes from
the stack) the first element from the “top” of the
```

stack. If the stack is empty, the method displays an appropriate message and nothing is done (`null` to be returned)

```
4. the `isEmpty()` methods checks whether the stack is
empty, i.e. whether there is no element in the array

5. the `isFull()` method checks whether the stack is
full, i.e. for example at a 5 – element stack, all 5
array
```

elements have a value (other than `null`)

```
6. the `toString()` method displays the stack contents

7. hint: enter an additional variable that stores the
index of the current or next item in the array and
modify its
```

value when implementing the `push()` and `pop()` methods`

5. Based on the task above, write a program that will “mimic” an array of any size that can hold values of type `Integer`. There is a class:`public class IntArray { public IntArray() { } public void add(Integer value) { } public void add(Integer value, int idx) { } public Integer get(int idx) { return null; } public void remove(int idx) { } public void swap(int from, int to) { } public String toString() { return ""; } }` Functionalities:
    - `IntArray()` – constructor, it should create an initial array with an initial, default size
    - `void add(Integer value)` – adds an element to the next position; enlarges the array if necessary
        - creates a new, bigger one
        - rewrites the current values
        - adds a new item
    - `void add(Integer value, int idx)` – adds an element to the indicated position; if necessary, this is done by the method above
    - `Integer get(int idx)` – returns the item at position `idx`; if not there, it returns `null`
    - `void remove(int idx)` – removes the item at position `idx`
    - `void swap(int from, int to)` – swap items in the `from` and `to` positions; enlarges the array if necessary

- `String toString()` – displays the entire arrayAdditional functionalities:
  - have each method display additional messages, e.g..`I return item 5 from position 78 I replace item 23 from item 3 with item 55 from item 14 I put element 98 at position 56 (I enlarge the array)`
  - have the `toString ()` method display additional information about the number of elements and the size of the array

6. `Account` and `AccountApplication` classes are provided`public class Account { private String name; private int balance; } public class AccountApplication { public static void main(String[] args) { Account account = new Account(); account.setName("Premium Account"); System.out.println("Name: " + account.getName()); System.out.println("Account balance: " + account.getBalance()); } }`
   1. Set a default value `0` for the `balance` field
   2. Add (generate) *getter* and *setter* type methods for the `name` field
   3. Add (generate) a *getter* type method for the `balance` field
   4. Add a (private) `debit` field of the `boolean` type that determines whether the account balance is negative; set the default value to `false`
   5. Add the possibility to deposit and withdraw money`public void deposit(int amount) { } public void withdraw(int amount) { }`
      1. Implement the above methods
      2. The `withdraw` method is to set the `debit` field to `true` when the account balance is negative
      3. Check the operation of the method
   6. Add validation of the `amount` parameter in the `deposit` and `withdraw` methods;
      1. the methods are to perform the logics only when the `amount` value is positive
      2. otherwise they are to display the following message: `"The deposit/withdrawal amount must be positive!"`
   7. For the `withdraw` method, add the message display: `"Negative account balance"` if the `debit` field value is

`true`

```
8. For the `deposit` and `withdraw` methods, add a
summary display like the one below (e.g. for `deposit`
and
```

`withdraw`, respectively)

```` bash
"Account balance: 300 | Deposit: 250 | After operation:
550"
"Account balance: 200 | Withdrawal: 500 | After
operatio: -300"
````

9. Add a function to support the maximum debit, e.g.`
1000`. If the amount after the operation is lower, do
not

execute withdrawal but display the message: "You cannot perform an operation exceeding the debit"

10. Implement the `transfer` method for transfers from the current account to another one.

```` java
public void transfer(Account other, int amount) {

}
````

1. The `amount` is to be withdrawn from the current account

2. The `amount` should be paid to the `other` account

11. Add the `toString` method

```` java
public String toString() {
return " Account{name: " + name + ", balance: " +
balance + "}";
}
````

and invoke it as below:

```` java
public class AccountApplication {
public static void main(String[] args) {
Account account = new Account();
account.setName("Premium Account");
System.out.println(account);
}
}
````

CONGRATULATIONS!

You have completed learning the basics of Java programming.

Complete Lesson