Tutorial 1: Types, Methods and Arrays

1. Tutorial Objectives

This tutorial builds on the basic Java concepts introduced in the previous week and presents some other useful techniques. It is assumed you are now comfortable with last weeks ideas and able to perform basic operations in Java such as printing text to the console, accepting user input, looping and conditionals.

There are three levels of difficulty of the tutorial exercises marked with asterisks (*) as follows:

* Easy ** Medium *** Hard

It is important to do the tutorial exercises in order as they often rely on a previous question to be completed first. Hints will be marked in italics.

You are expected to be able to finish the easy and medium exercises without any help. However, never hesitate to ask the demonstrator if you have any questions.

2. Exercises

* Exercise 1 Types

In the previous tutorial you were using few variables to store data, such as counter and name. Each of these variables had a specific type: counter was of type *int* which stores integer numbers, while name was a String storing a sequence of characters. Java is a strongly typed language, which means that when you define a variable you have to decide which type it's going to take and you cannot change this type as long as the variable exists.

Hint: *Actually* ***String*** *is a class, which you can see as it starts with a capital letter, but don't worry about the difference too much yet.*

To recap: in order to de ne a variable of type int write:

```
int number;
```

You can also assign a value to a variable straight away – the following will initialize number with value 3:

```java
int number = 3;
```

It is important to get the types right, as you have found if you did the last exercise from Tutorial 0.2. There you had to take input from the user (which was a String) and interpret a part of it as a number in order to compare it to another number. Various methods exist for converting variables of different types into others, like the *Integer.parseInt()* mentioned before. Note however, that it is not always possible to do that. For instance, while converting a String "578" to an *int* will succeed, trying to do the same with "55a", or even "5 5" will result in an error.

1) Use the java documentation (link below) to find out what other primitive types exist in Java and discover differences between them. All primitive types have names beginning with lowercase letters, just like *int*.

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

* Exercise 2 Methods

Methods are pieces of code that have a defined, specific function in your program. Using methods is the most straightforward way to structure your Java program - you can think of them as “black boxes” that take input from you, do some processing, and return output. In other languages and programming paradigms you might refer to them as “functions” or “procedures”. In your previous exercises you only had one method called **main**(). This will change very soon.

Let's make it a bit more specific. Assume that you want to write a calculator program (which, in fact, you will by the end of this tutorial). Since this will be a very simple calculator, it will only have four functions: addition, substraction, multiplication and division. It is easy to see that such a calculator program can be logically broken down into five separate parts: one for each of the aforementioned operations and one method to rule them all and accept user input. Not surprisingly, this falls very nicely into 5 separate methods! (see Figure 1)

We will start by writing a method to add two numbers - you can find it below.

```java
public static int add(int a, int b){
    int result = a+b;
    return result;
}
```

Listing 1: Method to add two integers

Let's dissect the above method. The first two words on line 1, *public* and *static* are there so you can easily use this method in the tutorial - *public* makes it accessible from outside of the current class, while *static* lets you call the method without explicitly creating an object (instance) of that class. *int* specifies the return value of our method -in this case we want to return an integer number. Next we have a name - you can call the method whatever you

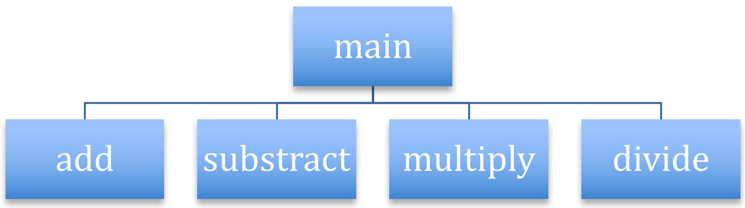


Figure 1: Five-method hierarchy of the calculator program

like, but make it descriptive. After the name, in brackets, we have a list of inputs, in this case two *int* variables *a* and *b*. Again you can give them any names you like.

Then, on lines 2 and 3 we have a body of the method, which adds the two numbers and returns the result - pretty simple stuff!

1) Write **substract(), multiply()** and **divide()** methods. Don't worry about exceptions e.g. handling the case of a division per zero. This will be covered in future tutorials.

2) Now modify your **main()** method to display the following message to the user:

Simple calculator. Please choose an operation:

1. *Add*
2. *Substract*
3. *Multiply*
4. *Divide*
5. *Exit*

Now take an input from the user, and if a number 1-4 is given, ask the user to input two numbers and call appropriate function with them as arguments. Then display the result and finish the program.

3) What happens when you try to divide 5 by 2? Do you know why the result is like it is?

4) Change the implementation so the result of the above operation is accurate;

Hint: *Is int the right primitive type for this type of operations?*

5) Now let's add something more interesting - calculating a factorial of a number. There is more than one way of writing this, but we will use recursion. Try to find materials online that explain what recursion is.

6) Write a method that calculates and returns a factorial of an *int* using recursion. Add another option to your calculator.

7) Write a method that exits the program. ***Hint***: To exit, use *System.exit(0)*;

* Exercise 3 Arrays

Array is a collection of values of the same type. To initialise an array of 20 *int*s write

```
int  numbers [ ]  = new int [20];
```

1) Loop trough the array and assign a number to each element. Let the first element *numbers[0] = 1;*. Then, each subsequent element *i* should be double the previous element *i-1*.

2) Create another array that stores mean values of neighbouring elements in numbers. That is the element with index *i* in array means should contain the mean of *i-1* and *i+1* from *numbers*. At the edges (the rest and the last elements) just store the rest and the last elements of the previous array.

** Exercise 4 Byte Conversion

Now we want to combine the usage of *Arrays* and *Loops* to create a small method, which converts bytes into bytes, kilobytes, megabytes or gigabytes and returns the correct unit.

In everyday life the metric system is used, meaning every conversion is as a multiple of 10. Kilo for example has a conversion factor of 103 = 1000, so 1 kilometer is 1000m. In computing the binary system is used in most cases, which uses a base of 2 instead of 10. Thus, 2^10 = 1024 is used to convert between bytes and kilobytes, kilobytes and megabytes, and so on.

Hint: Check out the *Math.pow()*method in the java documentation http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#pow(double,%20double)

1) First, we need to set up the units, so that the result can be printed with the units attached.

```java
final  String [ ]  UNITS = { "B","KB","MB", "GB"};
```

A short note on 'final': The keyword final makes a variable unchangeable. In case someone else uses your code, you do not want anybody to change the units of this method. The same goes for example for PI, PI is always roughly 3.1415926535, nobody should be able to change that.

2) At the beginning we know the initial value in bytes, after one iteration we should know the size in kilobytes, etc. So we will have to keep track of the size in the current unit. To get higher accuracy, we also want to know the decimal number of bytes, so that 512 bytes would return "0.5 KB".
   So for every iteration you need to keep track of the current size in the current unit and the remainder, which you will use to calculate the decimal place.

3) The method should return a string with the converted number and the unit. As a test you can use the following result. Entering 537395200 as bytes, should return "512.5 MB".

*** Exercise 5 Virtual Chessboard

Having acquired all the knowledge so far, it is important to put your skills to test. Let's build a virtual chessboard for two human players. If you don't know the rules, you can still do the exercise and look up the bits and pieces from the Internet as you go along.

1) Create an enumerated type Chessmen for all possible chess pieces. To signify an empty place on the board, add EMPTY too.

   Hint: *enums are used to express fixed sets of constants throughout your program, and because they are constants, we type-set them in UPPER_CASE. Similarly, you could have enum for days of week (MONDAY, TUESDAY…), compass directions (NORTH, WEST…) etc.*

```java
public  class   VirtualChess {
    public enum Chessmen  {
     WHITE_KING,
     WHITE_QUEEN,
     WHITE_ROOK,
     WHITE_BISHOP,
     WHITE_KNIGHT,
     WHITE_PAWN,
     BLACK_KING,
     BLACK_QUEEN,
     BLACK_ROOK,
     BLACK_BISHOP,
     BLACK_KNIGHT,
     BLACK_PAWN,
     EMPTY
    }
    public  static  void main (String[] args){
    ...
      }
}
```

Listing 2: Enumerated type of Chessmen

2) When the chessmen have been specified, you can create a chessboard matrix. It is an array of arrays (hence matrix) of size 8 X 8.

```java
...
public static void main(String[] args){
    Chessmen[][] chessboard = new Chessmen[8][8];
}
```

Listing 3: Chessboard matrix of type of Chessmen

3) Use two nested for loops to populate the chessboard with pieces and empty spaces. If…else statements should help to set the pieces where they belong. For example, empty spaces would be set as follows:

   chessboard[][] = Chessmen.EMPTY;

***Hint**: If you don't know the standard chessboard pieces layout, look it up online or peek at question 4 below.*

4) Write a separate function that will take the chessboard as an input and print it to the console. Do not forget to print labels for the chessboard so that horizontally it uses letters "a" to "h" and vertically numbers "8" to "1" assuming white player is at the bottom. For the time being you can

invent your own labeling for individual chessmen. You can use WK for WHITE_KING and so on.

```
/***
*  Prints chessboard  to  the console
*  @param   chessboard
*/
public static void printBoard (Chessmen [][]
chessboard) {
      // for i rows
            // for j columns
                  // switch ( chessboard [i][j] )
                    //  case  WHITE KING: …
                    //  . . .
            // end      for   j
            // print new line
      // end for  i
}
```

Listing 4: Chessboard print stub

***Hint**: Use ''\t'' for placing tabs in strings. This will help you align individual columns in your chessboard.*

The output could look similar to this:

```
        a    b    c    d    e    f    g    h

8.      bR   bK   bB   bQ   BK   bB   bK   bR

7.      bP   bP   bP   bP   bP   bP   bP   bP

6.

5.

4.

3.

2.      wP   wP   wP   wP   wP   wP   wP   wP

1.      wR   wK   wB   wQ   WK   wB   wK   wR
```

5) To display actual chess pieces on the screen you need to get *Unicode* characters support in console. If it does not work on lab computers, continue with the rest of the tutorial and try this at home later.

Open the main folder with your Eclipse installation and find *eclipse.ini* file. Add *-Dfile.encoding=UTF-8* at the end of this file and restart Eclipse. If the system complains the file is read-only, you need administrator privileges.

6) If you successfully modified the *eclipse.ini* file, you should be able to print Unicode characters to console through Eclipse. Try the following program to see whether you can display chess pieces.

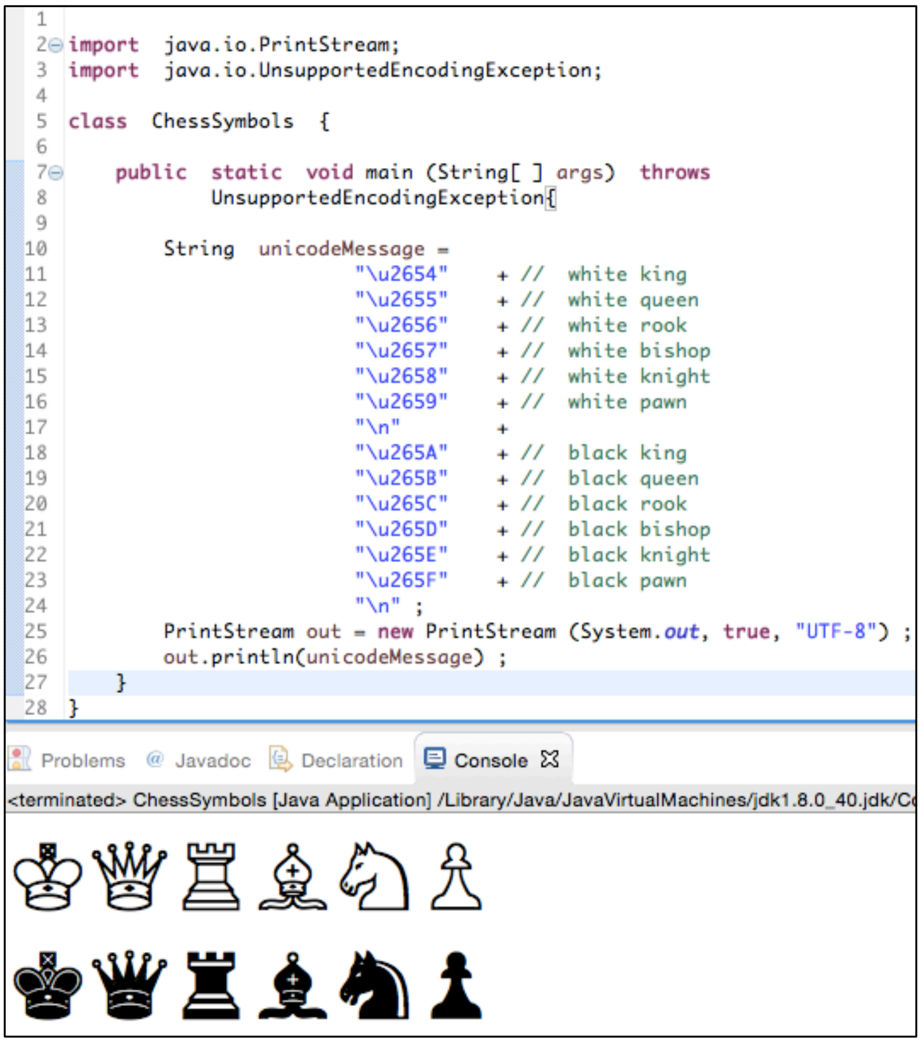
```
import  java.io.PrintStream;
import  java.io.UnsupportedEncodingException;

class  ChessSymbols{

      public  static  void main (String[ ] args)  throws
      UnsupportedEncodingException  {

String  unicodeMessage =
      "\u2654"    + //  white king
      "\u2655"    + //  white queen
      "\u2656"    + //  white rook
      "\u2657"    + //  white bishop
      "\u2658"    + //  white knight
      "\u2659"    + //  white pawn
      "\n"        +
      "\u265A"    + //  black king
      "\u265B"    + //  black queen
      "\u265C"    + //  black rook
      "\u265D"    + //  black bishop
      "\u265E"    + //  black knight
      "\u265F"    + //  black pawn
      "\n" ;

PrintStream out = new PrintStream (System.out, true,
"UTF-8");
out.println(unicodeMessage) ;
      }
}
```

Listing 5: Chess Unicode characters

Hint: *To increase the console font size, set it through the main menu in Eclipse: Eclipse → Preferences → General → Appearance → Colors and Fonts → Debug → Console Font.*

Note: *This is for MacOS Eclipse version. For other Oss, there might be a slight difference in accessing the menu.*

With font size 48, the output of the program above should look like Figure 2.



```java

import  java.io.PrintStream;
import  java.io.UnsupportedEncodingException;

class  ChessSymbols  {

    public  static  void main (String[ ] args)  throws
            UnsupportedEncodingException{

        String  unicodeMessage =
                        "\u2654"    + //  white king
                        "\u2655"    + //  white queen
                        "\u2656"    + //  white rook
                        "\u2657"    + //  white bishop
                        "\u2658"    + //  white knight
                        "\u2659"    + //  white pawn
                        "\n"        +
                        "\u265A"    + //  black king
                        "\u265B"    + //  black queen
                        "\u265C"    + //  black rook
                        "\u265D"    + //  black bishop
                        "\u265E"    + //  black knight
                        "\u265F"    + //  black pawn
                        "\n" ;
        PrintStream out = new PrintStream (System.out, true, "UTF-8") ;
        out.println(unicodeMessage) ;
    }
}
```

```
<terminated> ChessSymbols [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/C
♔♕♖♗♘♙
♚♛♜♝♞♟
```

Figure 2: Unicode chess characters in Eclipse console

7) If you confirmed you can print Unicode characters, replace your ASCII representation (such as ***WK*** for ***WHITE_KING***) with suitable pictures from Unicode above.

8) Inside the *main* function, create a *while* loop which will print the board and listen for user input (i.e. the next move) of each player in turn. User shall input the moves in standard chess notation, such as: “e1 to e5”. The *while* loop should exit when user types in “exit”.

9) Create a method that takes a chessboard and a move (a string representation of a move) as input and performs the move on the board.

```
public  static    void move(Chessmen[][] chessboard,  String  move)
{
      // parse move string into components
      String[] components = move.split(" ");

      // if you assume that move is "e1 to e5" then
      // components[0].chartAt(0) ='e'
      // components[0].charAt(1)    =='1'

      // make the move: replace original position with Chessmen.
      EMPTY
      //and place the piece into the new position
}
```

Listing 6: Move pieces

Hint*: To split a string, such as “e1 to e5” into individual components, use split(“ ”) function, which will return you an array of substrings.*

10) Modify the while loop inside the *main* function so that it passes the user input into move and prints the new chessboards on the screen until “exit” was put in.

11) Create a method which takes a chessboard and [i,j] integer coordinates of the old and new position and returns true if the move is valid and false otherwise.

Hint*: Chess move is valid, if the piece follows the rules (such as Knight moves in L shape) the target position of the move is not of the same colour and the piece does not end up outside of the board. Simple version of the chess rules is good enough for this exercise. For a list of all valid moves in chess, please refer to: https://en.wikipedia.org/wiki/Rules_of_chess*

```
/**
Returns a boolean true if the move is valid, false otherwise.

      @param chessboard
      @param oldI
      @param oldJ
      @param newI
      @param newJ
      @return
**/

public  static    Boolean isValid(Chessmen[][] chessboard, int
      oldI, int oldJ, int newI, int newJ) {
            // fill in the contents

                  return  false;
            }
```

Listing 7: Validation of a chess move

12) Use this validation method inside move method to make sure the move you are about to perform is indeed a valid one. If not valid, print error message.

***Hint**: System.err.println("error message");*

13) Add input validation, so that if a player types in wrong commands, your program displays suitable error message and asks for an input again.

14) Try to play your game with a friend and make sure it behaves as expected. Code testing and validation is an important part of programming and we shall return to this topic later.

~~~ 000~~~
~~~