**Introduction to Computer Science, IDC Herzliya, 2016**

# HW 1: Getting Started

---

If you haven't done it so far, we recommend going through the "Hello World" Application lesson and the Closer Look at the "Hello World" Application lesson in the "official" Java Tutorials (there are many Java tutorials on the web; this tutorial was written by the folks who developed the language). The Java Tutorials is a good self-learning resource, and it's quite helpful to consult with it when the need arises.

**Experimenting with Errors:** When writing and compiling Java programs, you will run into all sorts of problems. As you will soon discover, the error messages that the Java compiler generates are rather cryptic (this is true for most, if not all programming languages). One way to get used to, and understand these error messages, is to force common programming errors intentionally and then read and figure out the resulting error messages. That's what this exercise is all about.

Look at the `PrintSomeNumbers` program that was introduced in class. In this exercise you are required to make some changes to this program, observe how the Java compiler and run-time environment react to these changes, and explain what is going on.

To get started, type the program's code using some plain text editor, and save it using the file name `PrintSomeNumbers.java`. Make sure that your editor does not change the file name into `PrintSomeNumbers.java.txt` or something like this. Next, you have to compile and run the program.

Windows users: double-click the supplied `commandshell.cmd` file. This batch file will open the "command window" (also known as the "cmd window") from your current folder. Mac users: use Spotlight to find and activate the "terminal" app. Then drag the current folder onto the terminal icon.

Next, compile the program. If the program compiled successfully, proceed to execute (run) it. Your session should look something like this:

```
% javac PrintSomeNumbers.java

% java PrintSomeNumbers
1
2
3
4
5
Done
```

If you don’t get similar results, make sure that you've typed the program correctly. Next, proceed to make the following changes in the program's code (please read the instructions after the list of changes before making any modification) :

1. Change the class name to `printSomeNumbers` (but don't change the file name – keep it `PrintSomeNumbers.java` like before).
2. Change `Done` to `done`
3. Change `"Done"` to `"Done` (remove the closing quotation mark)
4. Change `"Done"` to `Done` (remove both quotation marks)
5. Change `main` to `man`
6. Change `System.out.println(i)` to `System.out.printline(i)`
7. Change `System.out.println(i)` to `println(i)`
8. Remove the semicolon at the end of the `System.out.println(i);` statement
9. Remove the last brace (`}`) in the program
10. Change `i = i + 1` to `i = i * 1`

For each one of the ten changes listed above, do as follows:

a. Make the change, using a text editor
b. Compile the changed program using the command
   `% javac PrintSomeNumbers.java`
c. Observe what is going on. You may well get an error message
d. If the program compiled successfully, execute it using the command
   `% java PrintSomeNumbers`
e. If you think that there's a problem, start by identifying what kind of problem it is: compile-time error? Run-time error? Logical error? Write a clear and concise explanation (not more than one sentence), describing what went wrong. If you think that there is no error, say it also.
f. Important: Fix the program – undo the change – in preparation for the next change. Or, keep a copy of the original error-free `PrintSomeNumbers.java` file and always start with it - we do not combine two modifications (although you may want to do it for your own curiosity).

This is a self-practice exercise – there is no need to submit anything. It's important though that you do it, write down your answers (for yourself), and make sure that you understand what went wrong.

**Stopping a program's execution in the command shell:** In some cases, typically because of some logical error, a Java program will not terminate its execution, going into what is known as an *infinite loop*. In such cases, you can stop the program's execution by pressing CTRL-c on the keyboard (press the "CTRL" key; while keeping it pressed, press also the "c" key).

**In the remainder of this homework assignment** you will write several Java programs. The purpose of this first exercise is to get you started with Java programming, learn how to read API documentation, and practice submitting homework assignments in this course. The programs that you will have to write are very simple. That is because we haven't yet covered commands like 'if', 'while', and 'for', which are essential for writing non-trivial programs. If you know how to use these commands (or any other Java command) you are welcome to use them, if and when it makes sense to do so. However, using these commands is not required in this first assignment, and will not give you any extra credit.

**API comment:** When you write programs in Java, you often have to use library constants and functions like, say, `Math.PI` and `Math.sqrt`. If you want to read the API documentation of any of these things, you can Google, say, "`java 8 Math`". This will open the API documentation of the Math class, and you can then proceed to search the constant or method that you wish to use within this web page.

1. **Popsicles**. Write a program (`Popsicles.java`) that calculates how much money is needed to buy a given number of popsicles. The program reads two command-line arguments. The program treats the first argument as a `String` that represents the buyers's name, and the second argument as an `int` that represents the number of popsicles bought. Assume that each popsicle costs 2.5 Shekels. The program calculates the total price and prints an output, as shown in the following examples. Assume that all the supplied inputs are valid. Here is an example of the program's execution:

   ```
   % java Popsicles Ron 3

   Ron, 3 popsicles will cost you 7.5 Shekels. Bon appetit.

   % java Popsicles Mor 10

   Mor, 10 popsicles will cost you 25.0 Shekels. Bon appetit.
   ```

   Note: the output of your program should be formatted exactly like the output in these examples.

2. **Sine function**. Write a program (`SinFunction.java`) that generates the following output:

   ```
   % java SinFunction
   sin(0)      = 0.0
   sin(1/4 PI) = 0.7071067811865475
   sin(1/2 PI) = 1.0
   sin(3/4 PI) = 0.7071067811865475
   sin(PI)     = 0.0
   sin(5/4 PI) = -0.7071067811865475
   sin(3/2 PI) = -1.0
   ```

```
sin(7/4 PI) = -0.7071067811865475
sin(2 PI)   = 0.0
```

The purpose of this simple exercise is to familiarize yourself with the *Sine* function, which will come to play later in the course. Use Java's `Math.PI` constant and `Math.sin()` function.

**Note:** Java uses *floating point arithmetic* to calculate operations on *double* values. We will have more to say about it later in the course. For now, note that your program may evaluate sin(0), sin(PI) and sin(2PI) not as 0 (the correct theoretical value), but as a tiny value like -2.4492935982947064E-16 = $-2.449 * 10^{-16}$, which is very very close to 0. That's ok.

3. **Random distance**. Write a program (`RandomDistance.java`) that calculates the distance between two randomly chosen points in the plane. The program reads two command-line arguments, *a* and *b*, and treats them as integers. The program generates four random integers, *x1, y1, x2, y2*, in the range *a* to *b* (inclusive). The program then computes and prints the distance between the two points (*x1*,*y1*) and (*x2*,*y2*). Assume that all the inputs are valid, and thet $a < b$. Here is an example of the program's execution:

```
% java RandomDistance 10 15

The distance between (14, 12) and (12, 14) is 2.8284271247461903

% java RandomDistance -200 100

The distance between (97, -145) and (-145, -26) is 269.675731203236
```

Make sure that you print a nicely spaced and formatted string, as shown in the output example above. Hint: You will need to use the Pythagorean theorem and the library method `Math.sqrt()`.

4. **Order**. Write a program (`Order.java`) that reads three integers and prints them in the right order, from the smallest to the largest. Here is an example of the program's execution:

```
% Order 9 1 7
1 7 9
```

Hint: You may use the library methods `Math.max()` and/or `Math.min()`.

**Submission:** Before submitting any program, take some time to inspect your code, and make sure that it is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Any deviations from these guidelines will result in points penalty.

Submit the following four files only:

- `Popsicles.java`
- `SinFunction.java`
- `RandomDistance.java`
- `Order.java`

**Deadline:** Submit your assignment no later than November 16, 23:55.

# Assignments Submission Guidelines

Checking over 100 submissions a week is a challenging task. In order to help us in completing this task on time, you must comply with the following guidelines. Take a minute to go through the guidelines; not following them will result in loss of points.

## What to Submit

Unless specifically instructed otherwise, assignments are submitted via the course website as a single Zip file that should include all your source files.
For more information regarding submission via the course website, please refer the appropriate part in the student guide: http://www.idc.ac.il/moodle/StudentGuide.pdf

## Soft Copy

Unless the assignment instructions state otherwise, follow these guidelines:

1. Pack all your files in one Zip file. You can use 7Zip, a free, open source software for that. There are other free software that can create Zip files (Google them). Do not include folders in your file, but only the files themselves. You can do so by selecting all the files, right-clicking them, and selecting "Add to Zip".
2. The zip file will be named in the following format: Exnn-id-fullname.zip, where 'nn' is the assignment number in two digits, ‘id’ is your identification number, and ‘fullname’ is your full name. For example, the files of your solution to assignment 3 will be packed in a Zip file called "Ex03-01234567-Umberto_Cohen.zip". Do not use the space character, instead, use “_”.
3. Include your source files (.java files), text solutions (if any) and any other solution file.
4. Do not include your byte-code (.class) files -- we don't need them.
5. Remove any testing code from your soft copy, unless explicitly instructed to do so.
6. Do not include any code that we provided, unless explicitly instructed to do so.
7. The submission box on the course website will be closed after the submission deadline, so make sure you upload any fixes beforehand. The latest version you upload before the submission time is the one we check.
8. For any issues regarding submission, use the Piazza forum (you can send us private messages there), or contact one of the course staff directly.

9. Each one of your class files must begin with the following submission block (the personal details are an example. Include your own personal details instead):

```
/*
Assignment number       :    3.1
File Name               :    FooBar.java
Name (First Last)       :    Umberto Cohen
Student ID              :    032392431
Email                   :    cohen.umberto@post.idc.ac.il
*/
```

## When to Submit

The deadline of each assignment is posted under the assignment page. Deadlines may change, so pay attention to the web site. Late submissions will not be accepted.

Last modified: Friday, 8th October 2013

Styling Guidelines

Section 1 - variables, expressions and comments

## 1.1 Variable naming

- Names of non-constant variables (reference types, or non-final primitive types) should use the *infixCaps* style. Start with a **lower-case** letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words.

- The names should be nouns or noun phrases and should be informative of the variable's purpose.

- Names should not be too long or too short.

- Avoid using special characters (like underscore) in names. Keep names as characters and numbers.

Examples:

```
boolean resizable;
char recordDelimiter;
```

Names of fields being used as *constants* should be all upper-case, with underscores separating words.
Examples:

```
MIN_VALUE, MAX_BUFFER_SIZE, OPTIONS_FILE_NAME
```

Avoid variable l (“el”) because it is hard to distinguish it from 1 (“one”) on some printers and displays.

## 1.2 Indentation

Line indentation is always 4 spaces, for all indentation levels. Every line should be indented the same as the line above it, except when for block statements.

The opening brace comes of a block statement comes at the end of the same line of the block statement (separated with a single space). Code should be Indented one level deeper than the statement line. The closing brace should be on a separate line right after the last non-blank line of the code block, indented the same as the statement line.

Example:

```
public class Example {
      public static void main (String[] args){
            System.out.print(“just an example”);
            System.out.print(“so is this”);
      }
}
```

### 1.3 Simple statements

- Each line should contain at most one statement.

```
a = b + c; count++;       // WRONG
a = b + c;                // RIGHT
count++;                  // RIGHT
```

- Generally, local variable declarations should be on separate lines; however, an exception is allowable for temporary variables that do not require initializers.

```
int i, j = 4, k;     // WRONG
int i, k;            // acceptable
int j = 4;           // RIGHT
```

- Assignment statements: separate the equal operator with a single space in both sides

```
someVar=1;        // WRONG
someVar = 1;      // RIGHT
```

### 1.4 Expressions

- Separate binary and ternary operators from operands with a single space on both sides.

```
x+y/z>0                  // WRONG
x + y / z > 0            // RIGHT
```

- Parentheses: Separate parentheses from the outside with a single space. Do not separate parentheses from the inside. If a parenthesized expression begins or ends with another parenthesis, attach them (no space).

```
( ( x + y ) / z )        // WRONG
((x + y) / z)            // RIGHT
```

- Avoid using "== true" and "== false" in boolean expressions.

```
if (x > 0 == true)       // WRONG
if (x > 0)               // RIGHT
```

### 1.5 Line wrapping

- Keep lines no longer than 80 characters (assuming a tab of 4 spaces long).
- If a statement is longer than 80 characters, break it according to these rules:
    - Prefer breaking a higher level statement than a lower level.
    - Before an operator.
    - After an opening parenthesis.
    - After a comma.
    - Within a long String and concatenate.
    - Preserve logics; if several breaks needed for a single statement, prefer keeping logical groups in each line.
- After a line break, indent the second line one level deeper than the statement (not the line) that was broken. If the second line requires more breaking, indent the next lines at the same level as the second line.

```
float z = Math.sqrt(Math.pow(x, 2)) * ((x + y)          //
      / 2)                                               // WRONG
            / Math.abs(x - y);                           //

float z = Math.sqrt(Math.pow(x, 2))                      //
            * ((x + y) / 2)                              // RIGHT
            / Math.abs(x - y);                           //
```

**1.6 Comments**

Here are some general guidelines for comment usage:

• Comments should help a reader understand the purpose of the code. They should guide the reader through the flow of the program, focusing especially on areas which might be confusing or obscure.

• Avoid comments that are obvious from the code, as in this famously bad comment example:

```
i = i + 1; // Add one to i
```

• Remember that misleading comments are worse than no comments at all.

• Avoid putting any information into comments that is likely to become out-of-date.

• Avoid enclosing comments in boxes drawn with asterisks or other fancy typography.

### 1.4.1 Single line comments

A single-line comment consists of the characters `//` followed by comment text. There is always a single space between the `//` and the comment text. A single line comment must be at the same indentation level as the code that follows it. More than one single-line comment can be grouped together to make a larger comment. A single-line comment or comment group should always be preceded by a blank line, unless it is the first line in a block. If the comment applies to a group of several following statements, then the comment or comment group should also be followed by a blank line. If it applies only to the next statement (which may be a compound statement), then do not follow it with a blank line. Example:

```
// Search for a maximal element in the array
for (int i = 0; i < array.length; i++) {
```

Single-line comments can also be used as *trailing* comments. Trailing comments are similar to single-line comments except they appear on the same line as the code they describe. At least one space should separate that last non-white space character in the statement, and the trailing comment. If more than one trailing comment appears in a block of code, they should all be aligned to the same column. Example:

```
if (length < size)
      return;     // nothing to do

length++;         // increase the length
```

Avoid commenting every line of executable code with a trailing comment.

## 3.2 Block comment

A regular block comment starts with the characters `/*` and endswith the characters `*/`.
A block comment is always used for the copyright/ID comment at the beginning of each file. It is also used to “comment out” several lines of code (i.e. eliminating code from the program without deleting it, can be usefull for debugging). **The use of block comments other than for the copyright/ID comment and commenting out code is strongly discouraged.**