Joanna Klukowska
joannakl@cs.nyu.edu

Code Conventions for CSCI 102

When you speak a language, you need to follow grammar rules and use commonly known vocabulary, otherwise nobody would understand you. Similar idea applies to code. Each programming language has its syntax and if you violate that syntax, the compiler cannot *understand* your code. Each language also comes with certain conventions (these do not matter to the compiler, but matter greatly to humans reading the code). Following certain conventions when writing code makes it easier to read and to debug. For the benefit of your code readers and your own, you should follow the coding conventions outlined in this document.

Naming

- Class names start with an uppercase letter: `Car`, `LinkedList`, `UndergraduateStudent` - each word in the name should start with an uppercase letter, no spaces, no underscores.
- Method and variable names use the camel case (first word is all lowercase, each consecutive word starts with an uppercase letter): `findLargestAverage`, `sortInReverseOrder`, `removeAll`.
- Constant names are written using all uppercase letters with underscores to separate the words: `MAX_NUMBER_OF_STUDENTS`, `SIZE`, `LARGEST_POSSIBLE_MOVE`.
- Package names (if you chose to use them) start with a lowercase letter.
- All names have to be descriptive, i.e. state what the identifier represents. For example, if the program computes student's average grade, then the name of the variable storing such average should be `averageGrade` (or something equally informative), not `x`.

Spacing

Your code has to be indented in a logical manner.

- Every time a new block of code starts, it should be indented with respect to the block of code in which it is contained. The indentation should be consistent:
  - do not use four spaces for some blocks and two spaces for others;
  - use either tabs or spaces, but not a mix - tabs can have different widths.
- The closing curly braces (brackets) should be aligned with the corresponding opening brace or the beginning of the line on which the corresponding opening brace occurs.
- Single line code blocks may be included in a set of curly braces or just indented - whichever way makes the code easier to read (note that future modifications may be easier if the braces are there).
- The blank lines should be used to separate logical parts of the code into more readable chunks (do not enter a blank like as every other line in the code).
- The lines of code should be limited in length to fit on a fairly small screen (for many programmers 80-100 characters should the the max for the line length) - do not let the code be wrapped by the editor.

The following two code snippets show two different ways of formatting a method according to the spacing rules above (note that they are missing proper comments, see more for that below):

Joanna Klukowska
joannakl@cs.nyu.edu

```
public static Potion findSmallest ( Potion[] listOfPotions) {

  Potion tmp = listOfPotions[0];

  for (int i = 1; i < listOfPotions.length; i++)
    if (tmp.toString().length() > listOfPotions[i].toString().length())
      tmp =  listOfPotions[i];

  return tmp;
}
```

or

```
public static Potion findSmallest ( Potion[] listOfPotions)
{

  Potion tmp = listOfPotions[0];

  for (int i = 1; i < listOfPotions.length; i++)
  {
    if (tmp.toString().length() > listOfPotions[i].toString().length())
      tmp =  listOfPotions[i];
  }

  return tmp;
}
```

The following snippets are not formatted properly, even though they all compile:

```
//BAD FORMATTING
public static Potion findSmallest ( Potion[] listOfPotions)
{
Potion tmp = listOfPotions[0];
for (int i = 1; i < listOfPotions.length; i++)
{
if (tmp.toString().length() > listOfPotions[i].toString().length())
tmp =  listOfPotions[i];
}
return tmp;
}
```

```
//BAD FORMATTING
public static Potion findSmallest ( Potion[] listOfPotions) {

  Potion tmp = listOfPotions[0];

  for (int i = 1; i < listOfPotions.length; i++)
      {
if (tmp.toString().length() > listOfPotions[i].toString().length())
            tmp =  listOfPotions[i];
  }
return tmp;
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

Access Modifiers

Java uses four different access modifiers: **private**, **pacakge**, **protected** and **public**. If you do not remember what they all mean, you should review their uses and meaning. See, https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html, for example.

In the spirit of encapsulation, all data fields of all classes should be **private**. The only exception to that rule is when you are writing a class that will serve as a base class for another class in your program. In that case, the data fields that need to be accessible from the derived class should be made **protected**. (Note that the derived class often does not need access to all data fields of the base class.)

The data fields should be *directly* accessed only within the class that they are defined in or within the derived class. Any other access to the data fields, if needed, should be provided by public accessor and mutator methods (also known as getters and setters). Note that this does not mean that there have to be getters and setters for all of the data fields. These methods should exist only if the values of data fields should be accessible to other classes.

Comments / Documentation

- All the code must be documented using Javadoc style comments (this is standard documentation style used in Java code). For a detail guide see, for example, *How to Write Doc Comments for the Javadoc Tool* at http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html. For what is required in your programs, see the following.
  - Header comments at the top of each class. These should contain description of the class (its purpose, general use, etc), the name of the author and (optionally) the version. For example:

```
/**
 * This class represents a magical potion that consists of multiple
 * ingredients (represented by characters).
 *
 * @author  Alexander Smithon
 * @version 04/11/2015
 */
```

    Notice that the opening of this comment **/**** is different than the standard multiline comment in Java that starts with **/***. This turns it into a Javadoc comment. This comment also uses special tags **@author** and **@version** that are interpreted by the javadoc program.
  - Each method must have a block comment. It should contain 1) the description of the method itself (what does it do) and state any assumptions that are made for the method to function properly; 2) a list and descriptions of all the parameters with any assumptions that are made about the parameters and the valid ranges (if applicable); 3) the description of the return value (if the method returns something); 4) the list of all thrown exceptions with the description of conditions under which they are thrown. For example, this is the documentation for the **addIngredient** method in the **Potion** class:

```
/**
  * Adds a specified ingredient to this Potion object.
  *
  * @param ingredient ingredient to be added.
  *
  * @return true if the ingredient is a valid one
  * (one of 'a', 'b', 'c', 'd', 'w'), false, otherwise.
  *
  * @throws PotionFullException if this Potion object is
  *         full (reached its maximum capacity)
  */
public boolean addIngredient( char ingredient ) {
  //body of the method
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

Notice the special javadoc tags above:

* `@param parameterName` followed by description of the parameter
* `@return` followed by description of the return value
* `@throws ExceptionType` followed by description of when the exception is thrown

There are other tags that can be used, but it is up to you.

- All the code must contain inline comments - short descriptions within the code that guide the author and the reader of the code as to what is going on. Do not comment every line of code, though.

Here is the previous method with complete documentation and proper formatting:

```
/**
  * Finds the shortest of the Potion objects in the list.
  * If there are multiple Potion objects with the same
  * length the first one of them is returned.
  *
  * @param listOfPotions a list of Potion objects
  *
  * @return the Potion object with the shortest length (or first one of them
  * on the list, if multiple ones have the same length)
  */
public static Potion findSmallest ( Potion[] listOfPotions) {

  //set the temporary shortest to the zeroth index item
  Potion tmp = listOfPotions[0];

  //go through the list to find the shortest element
  for (int i = 1; i < listOfPotions.length; i++)
    if (tmp.toString().length() > listOfPotions[i].toString().length())
      tmp =  listOfPotions[i];

  //return the reference to the shortest one
  return tmp;
}
```

Exceptions

Exceptions are used to communicate errors and other exceptional situations from one method to another method. For a review/tutorial on exception handling see https://docs.oracle.com/javase/tutorial/essential/exceptions/.

When a program throws an exception, it should always include a message specifying the reason for the exception. The types of exceptions indicate the reason for which they are thrown, but the more detailed information is attached to the exception, the easier it is to narrow down what heppend.

You should attempt to write programs that never crash with an exception once they are completed. All exceptions that may be thrown by functions that the program is using should be handled. It is much more professional to terminate the program with a human readable message than to have it crash.

Packages

Java projects can be organized into packages that provide means for grouping related types, avoiding name conflicts, allowing access, etc. For a complete tutorial on packages, you can see one of the Oracle tutorials on the topric at https://docs.oracle.com/javase/tutorial/java/package/index.html.

Joanna Klukowska
joannakl@cs.nyu.edu

For the purpose of the assignment in this class, all your code should be contained in the same package. The projects specification will often state the name of this package.

Things to Do

- Classes' data fields should be declared `private` (or `protected`) unless they are constant. The access to such data fields, if desired, should be provided by appropriate getters and setters (accessors and mutators).
  Data fields should only be accessed by the methods of the class in which they are defined, or methods of the inheriting classes.
- All methods that are designed to be used only internally by other methods of the same class should be defined with `private` or `protected` access specifiers.
- Classes' data fields and methods should not be declared `static` unless they are to be shared by all instances of the class or provide access to such data.
- In a class that represents a collection of tools (for example, `Math` class from Java API), all methods should be defined with `static` modifier, there should be no non-static data fields and the constructor should be defined with `private` access specifier to prevent instantiation of such class.

Things To Avoid

- Do not hardcode any values. All values should be saved in properly named constants that describe their meaning.
- Do not declare (and initialize) variables that are never used.
- Do not use loops with conditions that are always true and then use a `break` statement to terminate.
- Do not write the same code multiple times - turn it into a method.
- Do not write methods that do too many things - each method should perform a single concise task.
- Do not write classes that represent multiple things or that perform actions that are not representative of the objects that the class represents. For each method, ask yourself: "Does it really belong in this class?"

Unimplemented / Broken Parts of Code

The program that is submitted has to work (i.e., compile and run without errors or crashes). If some parts of the program are not implemented correctly or are broken, they should be commented out (excluded from the compilation) and clearly marked as non-functioning. You may need to leave a method stub (i.e., a blank method returning a dummy value) in some cases for the other methods/classes to compile.

In addition, the header for the actual program (the class with `main()` method) has to list all parts of code that are left unimplemented or are broken (and, hence, commented out).

If the program is running for an unreasonably long time or produces unreasonably large output file(s), then the program is NOT working (or contains parts that are not working properly and should be removed before you submit the final product).