Coding Style

(67125) Introduction to Object Oriented Programming

Among many things, a good program is:

- Correct
- Portable
- Efficient
- Readable and documented
- Flexible (i.e. changeable, modular)

Readable style (mainly) includes naming, indentation, and documentation. These items are ignored by the compiler, but have significant effect on how easily people can read and understand a program.
The first golden style rule is: **be consistent**. Once you have selected a style that conforms to the conventions used in the community of programmers in a given language (and to the conventions in the course), use it consistently throughout all your programs.
Similarly to the PEP8 set of Python coding conventions you were presented in the *intro2cs/p* course, Java also has an agreed upon set of conventions which has been adopted by most Java programmers. The main components of the conventions are described below (*Note that some of them are rather different from Python's*).

1. Naming Style Summary

1. Here are conventions common to names of classes, interfaces, variables, data members, and methods: Use meaningful names, consisting of one or more words, only in English. Use words, not abbreviations or acronyms, except as noted below.
2. The first letter of every word, except possibly the first, is capitalized (This is referred to as *camelCase*).
3. Do not use $ or _ in names, except as noted below.
   Note: Some books use slightly different conventions. For example, some might use "_" in class or variable names. We stick to the generally accepted Java conventions, and do not use "_" in these names.
4. Naming rules by category:
   - **Classes:** A good name for a class is usually nouns that describe the class. Here are a couple of rules for class names:
     i. The first letter of the first word is also capitalized:
     `Lincoln, Bucket, InputRequestor, QuadraticFunction`.
     ii. Use short, meaningful names: `RoundButton` is better than `AButtonWhoseShapeIsRound`. However, `Gate` is not an acceptable substitute for `ElectronicGate`. Don't be afraid of typing long names if necessary (`ArrayIndexOutOfBoundsException`).

   - **Variables:** The guidelines and rules for classes generally apply to variables as well, except that the first letter of a variable in Java is **always** in lower case. Examples: `size, width, numberOfElements, currentTime`.
     In some cases it is ok, even desirable and customary, to use a single letter name for a variable (e.g. loop counters: `i, j`).

- **Data Members:** The same conventions as for variables. For example: `x, y, center, vectorLength`.

- **Methods:** We use the same conventions for method names as for variable names. For example: `enqueue() , getX(), addVector()`. The exception is constructors, whose names are those of their class (including the first capital letter, as in `Vector()` or `RoundButton()`).
  Never use single letters for method names!

- **Constants (final variables):** Similar to variable's naming convention (meaningful names, etc.) but all letters are capitalized. For example: `PI, MAX_LENGTH`. Note that "_" is used here as a word separator.

- **Abbreviations and acronyms:**
  i. As a rule **don't use abbreviations**.
  For example, use `MessageHeaderAttribute` and not `MsgHeadAtr, MsgHAttr, MsgHdrAttrib`, etc.
  ii. An exception: abbreviations which are common in general, and in computing in particular, may be used. For example: `min, max, temp` are considered as words and are accepted as part of names. Both `TemporaryFile` and `TempFile` are OK (just be consistent!).
  iii. Similarly, you can use acronyms that are very well known.
  For example, it is better to use `TCPConnection` than `TransportLayerProtocolConnection`. If you are designing a package and there is a common concept appearing in many of your classes you can consider inventing a new acronym for it. Be sure to emphasize this in your documentation.

2. Indentation

The purpose of indentation is to emphasize the structure of the program - its division into blocks. You can read the discussion about white space characters in Lewis & Loftus p. 36. We use the following indentation rules:

- Use tabs to indent the code (and not spaces).

- The brackets ({) opening a block are placed at the same line of the statement or method that opens this block.

- The closing brackets (}) are placed at the same column as the beginning of the statement/method declaration that opens the block.

- The content of the block always starts at a new line, shifted one tab to the right.

For example:

```
if (some condition) {
        ... some code ...
} else if (other condition) {
        ... some other code ...
} else {
        ... last code ...
}
```

- An exception to this rule are cases of a switch statement:

```
switch (some variable) {
case value1:
        ... some code ...
        break;
case value2:
        ... some other code ...
        break;
default:
        ... last code ...
        break;
}
```

3. Method Design

Below are some rules concerning the partitioning of code into methods:

- Each method should have a (single) well defined role. If you cannot describe the functionality of the method in a single short sentence, it is an indication that the role of the method is not clear to you and that you should change your division into methods. If you use the word 'and' in your description of the method or in the method name, it is an indication that you should divide the method into two.

- A method should not be very long. If the method becomes too long, consider dividing it into several methods. The preferable size of a method is 5-20 lines of code. A method of 35-40 lines is generally too long. Exceptions to this rule are methods that consists of a switch-case/if-else statement with many cases (in which case each case should be at most 1-5 lines), methods for defining a GUI appearance, etc. In general, when your method exceeds 35-40 lines, verify that it is not possible to divide it into sub-methods.

- Don't be afraid of very short methods. A method of 1 or 2 lines is perfectly OK if it has a well-defined role. Often you have a code segment that appears in many places of your program which is very short; in such a case define a method for this code segment.

- Methods should normally have 0-4 parameters. If you write a method with too many parameters it becomes hard to remember which parameters the method gets and in which order, and it may be an indication that the method does too much and should be divided into several methods. Consider defining a class that encapsulates all the information the method should get. Use such a class only if you have a clear notion of what it represents. If you have many parameters, move some of them to a new line to not exceed line characters limit.

- Like methods, constructors also should not have too many parameters. If you need to specify a lot of information for the creation of an object, you can give the important information in the parameters of the constructor and the rest in setter methods or give the constructor an object with all the data you need.

4. Comments

There are two types of comments in Java: **implementation comments** and **documentation comments**. Implementation comments are intended for the programmer who maintains the code, and needs to understand how it works. Documentation comments on the other hand, are intended for programmers who want to use the code as a whole. They are not interested in understanding how it works, and often they do not have the source.

4.1 Implementation Comments

There are two forms for writing implementation comments in Java:

- In-line comments: the text from the // symbol till the end of the text line is a comment. These are usually used to describe the use of declared variables, or to give a short explanation for an unusual statement.

- Block comments: the text between the opening /* symbol and until the closing */ symbol is a comment. Block comments can spread across lines. They are usually used to describe a section of code that needs clarification and for temporary hiding of code while debugging.

Here are some guidelines about implementation comments (some of these are taken from **Lewis J. & Lottus W.** "Java software solutions" pp. 605):

- Assume the reader is computer literate and is familiar with the Java language. Do not explain the obvious, e.g. programming constructs.

- Assume the reader knows almost nothing about what the program is supposed to do. Remember that a section of code that seems intuitive to you when you write it might not seem intuitive to another reader or to yourself later.

- Add a comment only if it contributes to the understanding of the code. Keep in mind that in a good program, the code should be obvious. **Do** comment tricky parts, non-trivial algorithms, things that are not apparent at first glance, and any other fact you think would be useful to understanding the code or maintaining it in the future.

- Your comments must be accurate, and written in a simple language (the readers of your code may know only basic English). Otherwise the comment can do more damage than help. If a comment becomes inaccurate because a change is made to the source code, remove the comment or correct at once.

- Every method should have a comment block describing it. If the method is public or protected, use a documentation comment (see below); otherwise, use an implementation comment. If a method requires a lengthy explanation, try to put all the explanation in one block at the beginning of the method. It is easier to understand the explanation as a whole and the code as a whole.

4.2 Documentation Comments

Documentation Comments are special Java comments used to describe the API of a program. They are intended for generating API documentation for programmers who wish to *use* your classes and interfaces, and need to understand their interfaces, not for the programmer who maintains the code!

Use Documentation Comments only for entities that are part of your public interface:

- Before a public class or interface definition.

- Before any (and only) public or protected methods and fields. One way for programmers to use your class, is by subclassing it; in this case they need to know how to use its protected members.

A documentation comment begins with /** and ends with */. It may include plain text, HTML tags, and some special documentation tags. The latter are used to describe authors of a class, parameters of a method, and so on. They include: `@author, @param, @return, @throws, @see,` and a few more. A documentation comment starts with some general description, and **ends** with short comments tagged by documentation tags. It is important that these appear at the end!

You may use html tags in your comments. Useful ones are: `<b>` for bold; `<i>` for italics; `<code>` for code, including single words such as types; `<p>` for opening a new paragraph; `<br>` for a line break. For all of the above, except the last, the text for which the tag is intended should be surrounded by an opening and a closing tag. Example: `<b>this is bold</b>` will produce **this is bold**.

Here is a discussion of comments by program construct:

- A class comment should open with a short sentence describing the purpose of the class, followed by a more detailed description of its *use*. The implementation of the class should not be documented here, unless it affects the API of the class (in which case you should add a short note indicating this). If the use of the class is not obvious from the API, it is recommended to give an example for its use. At the **end** of the class documentation block, you should denote yourself as the author of the class, using `@author`, and (if needed) add references, using `@see`, to related classes or methods. For example:

```java
/**
 * A Deck of cards. This class is intended for use in card games.
 * Each player holds a Deck (or more) of cards containing several
 * cards. The content of the Deck changes as the game proceeds.
 * @author Shmulik London
 * @see games.card_games.Card
 * @see games.card_games.BridgePlayer
 */
public class Deck {
    // ...
}
```

- Every public and protected method must be preceded with a documentation comment. The comment should begin with a short sentence describing the method, ending with a period ('.'). Do not worry if this sentence seems to add no information - it probably indicates that your choice for the name of the method is good (This frequently happens for the constructors of the class). If one sentence is not enough for describing the use of the method, you can add several sentences of explanations. Only the first sentence (until the period) is displayed in the API's method index. The full comment appears in the detailed list of methods (together with the first). Explain only what the method does and NOT how it does it! Be brief! If you cannot describe the method in a short comment, then possibly it does more than one thing, and you should consider splitting it into several methods, or reconsider your design.

- Document **all** the parameters, in the same order they appear in the method's signature, using `@param` for each. If two parameters are related you can document them in the same line, using a single `@param`. If the return type is not void, use the `@return` tag to document the return value (A constructor has no return value, so its documentation does not use `@return`).

- Document each exception the method throws using the `@throws` tag.

- Examples:

```
/**
 * Adds a card to the top of the deck.
 */
public void addCard(Card card) {
    // ...
}

class Figure {
    /**
     * Constructs a new Figure.
     * @param x,y The location of the top-left corner of the Figure.
     * @param width, height The dimensions of the Figure.
     */
    public Figure(int x, int y, int width, int height) {
        // ...
    }
}

class  Deck {
    /**
     * Draws a card from the top of the Deck.
     * @return A card from the top of the Deck.
     * @throws DeckEmptyException If the Deck is empty.
     */
    public Card drawCard() throws DeckEmptyException {
        // ...
    }
}
```

- Data members of a class will usually be private. Public fields make sense if they denote a public constant or they are part of a class which is a wrapper for several values (such as `java.awt.Point`). You should have a reason for defining a field as protected and a good one for defining it as public. Every public or protected field must be preceded by a short documentation comment. If you cannot describe the function of the field in one sentence it may be a good indication that it is not suitable for it to be public or protected (consider an access method instead). Example:

```
/**
 * The double value that is closer than any other to pi, the ratio of the
 * circumference of a circle to its diameter.
 */
public static double PI = 3.14159265358979323846;
```

5. Misc

Line length should not exceed 110 characters, both in README and in your code. This restriction comes from the presubmission PDF file, and might be different in places outside of HUJI.