COM1027 Programming Fundamentals

Java Style Guidelines

Overview

Whenever you write code, you should always try to write code that is readable, is easy to understand and is as simple and well-designed as possible. This makes the code easier to debug and easier to maintain should it need to be changed. Most organisations will have their own standard design and code guidelines. Below is a summary of some simple guidelines for Java. More detailed guidelines can be found in

Vermeulen, A., Ambler, S.W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J. & Thompson, P. (2000). *The Elements of Java Style*. Cambridge, UK.: Cambridge University Press.

Formatting Java

An example of code formatted using the following simple guidelines can be found in Figure 1.

1. A Java class file should have a header comment stating the name of the file. The may also include source code control information, if available:

```
/**
 * Tiger.java
 */
```

2. A class definition should start with a documentation comment (`"/** … */"`) describing what the class is for and who it was authored by (`"@author"`). The version number for the class (`"@version"`), initial release (`"@since"`), and any cross-references (`"@see"`), should also be included.
3. Classes should be given sensible names that help describe their purpose. The name (and any other word within the name) should start with a capital letter.
4. Blocks of code surrounded by "`{`" and "`}`" should be indented by exactly 2 spaces (not tabs).
5. The start of a block of code ("`{`") should be at the end of the statement it refers to (for example, a class definition). The end of a block of code ("`}`") should be on a newline on its own.

```
/**
 * Defines the properties and behaviour of a tiger. This is partly based
 * upon the example used in Bates, K. & Bates, B. (2005). Head First Java.
 * Sebastopol, CA: O'Reilly Media, Inc.
 *
 * @author Stella Kazamia
 */
public class Tiger extends Feline {
  ...
}
```

6. If you use constant values, then these should be declared as fields at the top of a class as "`static`" and "`final`". The names of constants should be in upper case with each word separated by an underscore ("_").

```
  /** The default number of people killed by a tiger. */
  private static final int DEFAULT_NUMBER_OF_PEOPLE_KILLED = 0;
```

7. All fields in the class should be put at the top of the class definition, each with a documentation comment describing what the field is for. To promote responsibility-driven design, all fields

should be marked as "`private`" and be initialised with a value (`null`, `0` or a constant value, for example).

```
/** How many people has the tiger killed? */
private int numberOfPeopleKilled = DEFAULT_NUMBER_OF_PEOPLE_KILLED;
```

8. Field and local variable names should be given sensible names that help describe their purpose. The name should start with a lower case letter, but any other word within the name should start with a capital letter.

9. All constructors and methods should start with a documentation comment (`"/** … */"`) describing what the constructor or methods is for. This should be followed by a list of parameter descriptions ("`@param`"), a description of what is returned if appropriate (`"@return"`) and any exceptions that might be raised within the method (`"@exception`").

```
/**
 * Constructor. Sets the field values.
 *
 * @param howHungry
 *          How hungry is the animal?
 * @param howTired
 *          How tired is the animal?
 * @param strengthOfRoar
 *          How strong is the feline's roar?
 * @param numberOfPeopleKilled
 *          How many people has the tiger killed?
 */
public Tiger(double howHungry, double howTired, double strengthOfRoar,
    int numberOfPeopleKilled) {
  ...
}
```

10. Constructors should be placed directly after the field definitions. Methods should be placed after the constructors in alphabetical order.

11. If a line of code (such as a method definition or comment) is longer than 132 characters, then the line should be broken with a newline and the following code indented to show which line it belongs to by 4 spaces.

12. In-line comments should be used to describe code that is not obvious or needs extra notes.

```
  // Use the super constructor to set the feline's initial state.
  super(howHungry, howTired, strengthOfRoar);
```

13. Fields and methods that are accessed from within the class should be referenced using the "`this`" keyword to make it clear that they are not local variables or external methods.

```
  this.numberOfPeopleKilled = numberOfPeopleKilled;
```

14. When imports are used, the full class name should be imported, not all classes within a package. For example, you should not use "`import java.awt.*;`", but "import `java.awt.Component;`" instead.

15. Packages should be given sensible names that help describe their purpose. The name should all be in lower case, including any words that are contained within the name.

Good Design

A good design makes the code easier to understand and maintain. You should not be afraid of changing what you have written to make it better. Some simple design guidelines are:

1. If in doubt, make fields and methods "`private`". Fields should never be accessed publicly outside of a class. Instead accessors and mutators should be used to keep the class responsible for its own state. Only make methods public if they are needed outside the class (are part of its contract). Internal methods should be "`private`".
2. Try not to duplicate code, even if the code is slightly different. One method that is well-written and tested is better than having lots of methods doing the same (or similar) things. The more times you write the same code, the chances of error increase.
3. Keep methods shorter than one page (or screen) full. Methods that are longer than this are hard to read. Chop the method up into several methods. This will also help reduce code duplication.
4. Functions should contain only one return statement. Having multiple return statements makes it harder to read the code and trace through what it does.
5. When comparing values in an if statement, use "==" for native data types but "`.equals(…)`" for all other types. This avoids checking whether two references are the same when what you meant to compare was the content of those references (the objects).
6. Always compile and test your code before releasing it for use.
7. Testing of code should be extensive. All possible scenarios, boundaries and error conditions should be tested (black box and white box testing). JUnit should be used to automate testing to make it easily repeated.
8. Classes and methods should not include code that is commented out or is not used. Remove all redundant code so that it does not confuse the reader.

```
/**
 * Tiger.java
 */

package workingwithclasses.animal;

/**
 * Defines the properties and behaviour of a tiger. This is partly based
 * upon the example used in Bates, K. & Bates, B. (2005). Head First Java.
 * Sebastopol, CA: O'Reilly Media, Inc.
 *
 * @author Stella Kazamia
 */
public class Tiger extends Feline {

  /** The default number of people killed by a tiger. */
  private static final int DEFAULT_NUMBER_OF_PEOPLE_KILLED = 0;

  /** How many people has the tiger killed? */
  private int numberOfPeopleKilled = DEFAULT_NUMBER_OF_PEOPLE_KILLED;

  /**
   * Constructor. Sets the field values.
   *
   * @param howHungry
   *          How hungry is the animal?
   * @param howTired
   *          How tired is the animal?
   * @param strengthOfRoar
   *          How strong is the feline's roar?
   * @param numberOfPeopleKilled
   *          How many people has the tiger killed?
   */
  public Tiger(double howHungry, double howTired, double strengthOfRoar,
      int numberOfPeopleKilled) {

    // Use the super constructor to set the feline's initial state.
    super(howHungry, howTired, strengthOfRoar);

    this.numberOfPeopleKilled = numberOfPeopleKilled;
  }

  /**
   * @return How many people has the tiger killed.
   */
  public int getNumberOfPeopleKilled() {
    return this.numberOfPeopleKilled;
  }
}
```

Figure 1: Sample code formatted using these simple guidelines.