Coding Standards

Guy Wiener

1 Naming

The name of a type[1] or a feature[2] is the fastest way to convey information about this feature. Therefore, the name of a feature should encapsulate the most important information about it. This information should include:

1. The role and functionality of the feature
2. The scope of the feature

To supply this information, we use the following naming convension, based on the Java naming convension:

1. The name of a class or an interface should describe its role. If the name contains more then one word, the words are separated by capitalization. The first letter should be in uppercase. For example: `Vehicle`, `SportsCar`, `CarsPool`, `ElementsSortingStrategy`.
2. The name of a method should describe its functionality. If the name contains more then one word, the words are separated by capitalization. The first letter should be in lowercase. For example: `run`, `igniteEngine`, `sortByStrategy`.

   The description should be unambigious and avoid vague terms. For example, a method that reloads all the car records from a file should be named `reloadAllCarRecordsFromFile` and not `performRefresh`.

Variable[3] names should describe what data the variable holds. It is not necessary to encode the type of the variable into the variable name (a.k.a "Hungarian Notations") unless there is some ambiguity. Use the variable type as its name only if it makes sense.

1. Parameters are named similarly to methods. For example, `speed`, `position`, `sortingStrategy`.
2. temporary variable names begin with a lowercase '`t`', followed by the variable name starting with an uppercase letter. The '`t`' prefix indicates that this variable is temporary. For example: `tCounter`, `tAverageSpeed`.

[1] A class or an interface
[2] A method, field or a variable
[3] Variable: A field, parameter or a temporary variable

3. Field names begin with an underscore '_' prefix followed by the field name starting with a lowercase letter. The underscore indicates that the scope of the field is the entire class. For example: `_positon`, `_currentElenent`.

   The underscore can be replace by explicitly using "`this.`" notation. There is no need to add prefixes like "my" to the field name to indicate relation to the class.

4. Static fields and enums are global variables, and therefore should be marked in a way that differentiate them from other variables. Global variable names are all in uppercase, words separate by underscore. For example, `MAX_SPEED`, `DEFAULT_STRATEGY`.

2 Scope and Indentation

Different scopes must be clearly and unambigiously visible. Therefore we impose the following rules:

1. All scopes are sorrunded by curly brackets, even if not required to.
2. The opening bracket is on the same line as the command.
3. The closing bracket is on a new line
4. Commands after an opening bracket are indented by one level more.
5. The closing bracket and afterwards, are indented by one level less.
6. Intentation is a tab character with a width of 4 spaces.

3 Visibility Rules

Using the visibility modifiers – private, protected, public, package – on feaures helps to control their usage. Limiting the usage to the necessary minimum helps to keep track of the feature. Therefore we impose the following policy:

1. All non-global fields must be private.
2. Accessing fields is done only through getter and setter methods.
3. Private and protected visibility is better then public or package visibility. Prefer a more limited visibility when possible.

4 Size

Small features are easier to understand. We use the following heuristics:

1. A class should have a single role. If a class serves more then one main functionality, split it to several classes.
2. The entire body of a method should be visible without scrolling. If a mehtod is too long, split it to several methods.
3. Expressions should not be too long. A single line of code should be 80 characters long. If an expression is too long, split it to sub-expressions. Use temporary variables if necessary.

5 Packaging

Packages helps to control the namespace of classes in large projects and to avoid name collisions. Guidelines for using packages:

1. It is not allowed to use the default package. All classes must be in a named package.
2. Related classes should be in the same package.
3. If a package is very large (over a dozen classes), try to move the less-related classes to sub-packages or other packages.

6 Documentation

The documentation is an important auxiliary tool when trying to understand a piece of code. It is important to document the following things:

1. What is this code doing
2. How it is done
3. When is this code supposed to be used
4. Why is this code impelemented in this way and not another
5. The decisions that lead to this specific implementation

Therefore the code must include the following comments:

1. Javadoc (or similar) comments for all non-trivial features, including fields and variables.
2. Comments that explain what a piece of code does, if it is not stright-forward.

7 Example

```
package vehicles.land;

/**
 * A car with a motor.
 * @see vehicles.Motor
 */
public class MotorCar extends Car {

    /**
     * The legal speed limit.
     * We keep this limit as a constant and not a parameter
     * for efficient access.
     */
    public static final int MAX_SPEED = 120;

    /**
     * The motor model for this car, set by the sub-classes
     * @see setMotor
     */
    private Motor _motor;

    public Motor getMotor() {
        return _moror;
    }

    protected void setMotor(Motor motor) {
        _motor = motor;
    }

    /**
     * Increase the speed of the car if it is legal,
     * or alert if not.
     * @param moreSpeed additional speed in KPH
     */
    public void accelerate(int moreSpeed) {
        // New speed, possibly ilegal
        int tNewSpeed = getSpeed() + moreSpeed;
        if (tNewSpeed > MAX_SPEED) {
            alert("Speed limit violated");
        } else {
            setSpeed(tNewSpeed);
        }
    }
}
```