SOMETIMES STYLE REALLY DOES MATTER*

David Reed
Department of Computer Science
Creighton University
davereed@creighton.edu

ABSTRACT

Programming, like any creative endeavor, involves some personal style choices on the part of the programmer. Within the computer science education community, there are some programming style conventions that have been widely agreed upon, because following them unequivocally leads to programs that are easier to read and maintain. In other cases, a programmer might reasonably choose between competing styles, each of which provides similar advantages. This paper describes three instances where programming style is more than just aesthetics - following these conventions can actually help a beginning programmer to avoid mistakes and better understand the underlying programming concepts that they are utilizing.

1 INTRODUCTION

Programming is a fascinating blend of engineering and art. While software engineering provides tools and methodologies for building reliable and cost-effective software systems, the creative aspect of programming and the opportunities for personal expression in the programming process cannot be denied. In his 1974 Turing Award acceptance speech [1], Donald Knuth wrote:

> *"When I speak about computer programming as an art, I am thinking primarily of it as an art form, in an aesthetic sense. The chief goal of my work as an educator and author is to help people learn how to write beautiful programs. ... Programmers who subconsciously view themselves as artists will enjoy what they do and will do it better. "*

In both art and programming, style is not a purely subjective matter. For example, there are some foundational elements of painting (i.e., brush techniques, effective use of color) that must be mastered by the budding artist and integrated into his or her aesthetic.

Similarly, within the computer science education community there are some programming style conventions that have been widely agreed upon. For example, any introductory text on programming will mention the importance of selecting meaningful names for variables, since names that suggest their purpose make code easier to read and maintain.

In addition to universally accepted conventions, there are some instances where different programming style aesthetics compete, but without a clear advantage. For example, some programmers still argue religiously over where to put the opening curly-brace on an if statement or loop: some advocate placing the curly-brace at the end of the opening line for that statement/loop, while others advocate placing it alone on the next line. The debate persists because there is no clear advantage to either style. Proper indentation ensures that the code is easy to read, regardless of where the opening curly-brace to placed.

The following sections describe three style conventions whose benefits go beyond aesthetics. Each convention can assist the beginning programmer in avoiding errors and in developing a stronger understanding of the underlying programming concepts involved. These three conventions are specific to the Java language, although the general concepts can be applied to other object-oriented languages.

CATEGORY	NAMING CONVENTION
classes interfaces	Should be nouns, in mixed case, with the first letter uppercase, e.g., `CustomerDatabase`.
methods	Should be verbs, in mixed case, with the first letter lowercase, e.g., `getCustomerID`.
instance/class variables parameters local variables	Should be nouns, in mixed case, with the first letter lowercase, e.g., `numberOfEntries`. Variable names should not start with '_' or '$' characters, even though both are technically allowed.
class constants	Should be all uppercase with words separated by underscores, e.g., `DEFAULT_SIZE`.

Figure 1. Java naming conventions (from http://java.sun.com/docs/codeconv/).

2 JAVA NAMING CONVENTIONS

The first style convention that can be beneficial to beginning programmers is a non-controversial one: following consistent naming conventions. Naming conventions, guidelines for choosing the names of program elements, have been used extensively within the software industry to develop uniform code that is easier to read, maintain, and integrate across projects. Even in the smaller-scale programs typically encountered by beginning programmers, naming conventions can clarify the roles of different program elements and can provide guidance when making design choices. For example, if a student knows that constants are represented using all-uppercase names, then he can more easily identify them when scanning code. Similarly, following the convention that object and class names are nouns whereas method names are verbs provides design guidance to beginners, making name choices less arbitrary and leading to code that reads more

naturally. Interestingly, research described in [2] suggests that poorly chosen identifier names can not only hinder comprehension, but may also indicate confusion on the part of the programmer and potential bugs in the code.

While all programming languages support various naming conventions, Java is noteworthy for its use of a central, well-established set of naming conventions that is published by Sun along with other references (Figure 1). Since all official Java code, including the standard libraries, follows these conventions, a student reviewing code can rely upon the conventions to help her understand the code. For example, the following Java expressions, although similar syntactically, suggest very different meanings:

```
Foo.bar()                    foo.bar()
```

By knowing the Java naming conventions, it is possible to distinguish between these at first glance: the first represents a call to the static method bar in the `Foo` class, while the second represents an instance method call on an object named `foo`. A beginning programmer who learns these naming conventions and follows them in his own code will similarly experience the benefits of readability and will be able to seamlessly integrate his code with existing programs.

The value of following the Java coding standards is demonstrated by a survey of introductory Java texts. Some texts (e.g., [3,5,7]) explicitly refer to the Java standards when describing the naming conventions used in the text. And while others (e.g., [4,8]) may not explicitly refer to the Java standards, the conventions used throughout the texts match the Sun coding conventions. Despite the clear message being sent by textbook authors, however, it is clear that many students are not being taught the importance of following the Java naming conventions. A search of online introductory Java courses turns up numerous violations, most notably nouns for method names and inconsistent capitalization.

3 CONSISTENT USE OF "this."

The next style convention, the consistent use of the "this." prefix for internal method calls and instance variable references, is more controversial but has great potential for assisting beginning programmers. Recall that, in Java, a dot is used to denote object ownership when calling a method or accessing an instance variable of an external object. For example, `die6.roll()` specifies calling the `roll` method that *belongs to* (and is being applied to) an object named `die6`. Similarly, `Math.PI` references the public class variable `PI` that *belongs to* the predefined `Math` class. In cases such as these, the prefix before the method call or variable reference is required in order to identify the particular object being acted upon. In contrast, internal method calls and instance/class variable references do not require prefixes, as the current object is the implicit target of any actions/references. For example, another method within `die6`'s class definition could call the `roll` method as simply `roll()`. Likewise, a method of the `Math` class can access the `PI` class variable as simply `PI`.

In my 10+ years of teaching intro Java programming, I have found that students are often confused by the inconsistent structure of external and internal method calls and variable references. As soon as they begin to appreciate that the dot notation denotes ownership and the application of a method to a particular object, they encounter internal

method calls where the object is not explicit. Similar confusion arises when distinguishing between local variables and instance variables within a class. Since there is no dot prefix to identify instance variables as belonging to the object, references to instance variables within methods have the exact same form as accesses to local variables.

While Java does not require explicit prefixes on internal method calls and internal references to instance/class variables, it does allow for them using the "this." prefix. The Java keyword "this" denotes the current object, so adding it to the prefix of internal method calls and instance/class variable references does not change their meanings. It does, however, make the format of internal calls and references consistent with their external counterparts. For example, consider the implementation of the `Die` class shown in Figure 2. The class has a single instance variable, `numSides`, which stores the number of die sides (six by default). Whenever, that instance variable is referenced in the class, it is referenced as `this.numSides`, highlighting the fact that it belongs to the current object. Similarly, the internal method call `this.getNumberOfSides()` uses the "this." prefix to highlight that the method is being applied to the current object.

The cost of making "this." an explicit prefix on instance variables and method calls is a minor one - the additional length of the references. The benefits, however, are major. First, the consistent use of "this." makes object ownership explicit and consistent. Every method call, whether it be internal (applied to the same object) or external (applied to another object) has the same format: `OBJECT.METHOD()`. Similarly, every reference to an instance variable explicitly lists the object: `OBJECT.VARIABLE`. This consistent syntax makes object ownership more transparent to a beginning programmer and reinforces an object-oriented mindset. Second, the consistent use of the "this." prefix makes the distinction between instance variables and parameters/locals clear. When the student reviews a class definition and sees the "this." prefix on a variable, he knows that this is an instance variable that belongs to the object (and persists between method calls). A variable without the prefix is a parameter or local variable that belongs to a method and exists only while that method is executing. This explicit distinction makes it easier for a beginner to understand object state and variable scope. Third, the use of the "this." prefix for instance variables avoids naming conflicts that might otherwise arise. For example, in the `Die` class (Figure 2) there is an instance variable named `numSides` and a constructor parameter with the same name. Without the "this." prefix, different names would have to be chosen for one of the variables, which usually leads to an abbreviated and less meaningful name such as `sides`.

Despite the advantages of the explicit "this." prefix, no introductory texts consistently use this style. Most refer to "this" in a more limited context, especially with respect to avoiding naming conflicts (e.g., [8,9,10]). The strongest support for this convention comes in [5], where Horstmann encourages programmers to try out the style of explicitly listing the "this." prefix. In personal conversations, Horstmann has expressed a personal preference for this style, but admits to not including it in his text because it is not widely used. Anecdotally, I have observed less confusion and a deeper understanding of object concepts among my introductory-level students since I began emphasizing "this."

```
public class Die {
  private int numSides;

  /** Constructs a 6-sided die. */
    public Die() {
        this.numSides = 6;
    }

  /** Constructs an arbitrary die.
   *    @param numSides the number of die sides */
    public Die(int numSides) {
        this.numSides = numSides;
    }

  /** Accessor method for the number of die side.
   *    @return the number of sides */
    public int getNumberOfSides() {
        return this.numSides;
    }

  /** Simulates a roll of the die.
   *    @return a random int between 1 and the number of sides */
    public int roll() {
        int currentRoll = (int)(Math.random() *
       this.getNumberOfSides()) + 1;
        return currentRoll;
    }
}
```

Figure 2. A class for modeling dice (and demonstrating the use of "this.").

4 MAIN METHOD FOR CLASS TESTING

Over the past decade, studies have shown that integrating unit testing into introductory computer science courses can help students learn the process of designing and testing software (see [11,12], for example). While many Java IDEs, integrate unit testing functionality, unit testing has still not achieved widespread use at the introductory level in colleges. A justification commonly given by instructors is that introductory programming courses are already too full of syntactic and technical details - adding another programming tool and set of techniques is just not feasible.

Some of the benefits of unit testing can be achieved using pedagogically inspired IDEs such as BlueJ and Dr. Java. Using BlueJ, a student can call any method on an object by right-clicking on the object icon, selecting the method, and entering parameter values in text boxes. This direct manipulation of objects is very intuitive for beginning students, allowing them to test each method individually (and alleviating the need for "public static void main" altogether). BlueJ and Dr. Java also have interaction panes, where the student can enter snippets of code and test class methods without the overhead of a separate driver class (or unit test). While these features are handy, they do not replace the full functionality of unit testing. The student must manually test each method of each class, and there is no automatic process for retesting code after making modifications.

```
public class Die {
  private int numSides;

  /** Constructs a 6-sided die. */
    public Die() {
        this.numSides = 6;
    }

  /** Constructs an arbitrary die.
   *    @param numSides the number of die sides */
    public Die(int numSides) {
        this.numSides = numSides;
    }

  /** Accessor method for the number of die side.
   *    @return the number of sides */
    public int getNumberOfSides() {
        return this.numSides;
    }

  /** Simulates a roll of the die.
   *    @return a random int between 1 and the number of sides */
    public int roll() {
        int currentRoll =
(int)(Math.random()*this.getNumberOfSides()) + 1;
        return currentRoll;
    }

  /** Main driver method for testing Die objects.
   *    Constructs a 6-sided die and displays 20 rolls, then
   *    constructs a 4-sided die and displays 20 rolls.      */
    public static void main(String[] args) {
        Die d6 = new Die();
        System.out.print(d6.getNumberOfSides() + "-sided die: ");
        for (int i = 0; i < 20; i++) {
            System.out.print(d6.roll() + " ");
        }
        System.out.println();

        Die d4 = new Die(4);
        System.out.print(d4.getNumberOfSides() + "-sided die: ");
        for (int i = 0; i < 20; i++) {
            System.out.print(d4.roll() + " ");
        }
        System.out.println();
    }
}
```

Figure 3. Die class with a main method for testing.

In most programming texts, classes are tested using a separate "driver" class. The driver contains a main method for creating an object of the class and calling methods on that object. Thus, to test the class, the programmer must add the driver to the project and separately compile and execute that driver. Unfortunately, requiring one or more drivers for every class in a project can lead to a very cluttered project space. An alternative style that avoids project clutter while achieving some of the advantages of unit testing involves integrating the driver's main method directly into the class itself. For example, Figure 3

shows the `Die` class with a main method added. This method creates two different die objects and displays the results of method calls on those objects.

By embedding the main method in the class itself, the programmer obtains the advantages of the driver class without the clutter of additional classes in the project. Popular IDEs such as netBeans and Eclipse allow the programmer to separately compile files in the project and execute any file that has a main method. Thus, by embedding the testing code inside a main method in the class, the programmer achieves some of the benefits of unit testing without requiring any additional tools. Each class can be tested independently (by executing its main method), and it is straightforward to retest a class if any related code is updated.

While this style of class design/testing does avoid cluttering the project with driver classes, one drawback is that it clutters individual classes with main methods. For example, the `Die` class from Figure 2 models a real-world die object, capturing its state and behavior. Adding a main method for testing purposes (Figure 3) is not part of the software model, and seeing it in the class definition or accompanying javadoc documentation for the class might confuse some beginning programmers.

5 CONCLUSION

Sometimes, programming style is more than just aesthetics. The consistent use of pedagogically sound conventions can actually help the beginning programmer to avoid mistakes and better understand underlying concepts. For example, following the Java naming conventions can lead to code that is easier to read and maintain. Consistently using the "this." prefix for instance/class variables and internal method calls can help a student understand object-oriented concepts and more easily differentiate between variable types. Adding a main method to every class in a project can achieve some of the benefits of unit testing without introducing any new concepts or tools. Conventions such as these can not only assist the budding programmer in writing "beautiful" programs (as Knuth would put it), but can also help them to understand difficult object-oriented and software engineering concepts.

REFERENCES

[1] Knuth, Donald. Computer Programming as an Art, *Communications of the ACM*, 17(12), 1974.

[2] Butler, Simon. The Effect of Identifier Naming on Source Code Readability and Quality. In *Proceedings of the Doctoral Symposium for ESEC/FSE (Amsterdam)*, ACM, 2009.

[3] Dale, Nell and Weems, Chip. *Programming and Problem Solving using Java, 2nd edition*, Jones and Bartlett, 2007.

[4] Lewis, John and Loftus, William. *Java Software Solutions, 6th edition*, Addison-Wesley, 2009.

[5] Horstmann, Cay. *Big Java, 3rd edition*, John Wiley and Sons, 2008.

[6] Sun Developer Network. *Code Conventions for the Java Programming Language*, 1999. Accessed online at http://java.sun.com/docs/codeconv/.

[7] Morelli, Ralph and Walde, Ralph. *Java, Java, Java, Object -Oriented Problem Solving, 3rd edition*, Prentice Hall, 2006.

[8] Liang, Daniel Y. *Introduction to Java Programming - Comprehensive Version, 6th edition*, Prentice Hall, 2007.

[9] Deitel, Harvey and Deitel, Paul. *Java How to Program, 7th edition*, Prentice Hall, 2007.

[10] Wu, C. Thomas. *A Comprehensive Introduction to Object-Oriented Programming with Java*, McGraw-Hill, 2008.

[11] Olan, Michael. Unit Testing: Test Early, Test Often, *Journal of Computing Sciences in Colleges*, 19(2), 2003.

[12] Desai, Chetal, Janzen, David S. and Clements, John. Implications of Integrating Test-driven Development into CS1/CS2 Curricula. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education, ACM SIGCSE Bulletin*, 41(1), 2009.