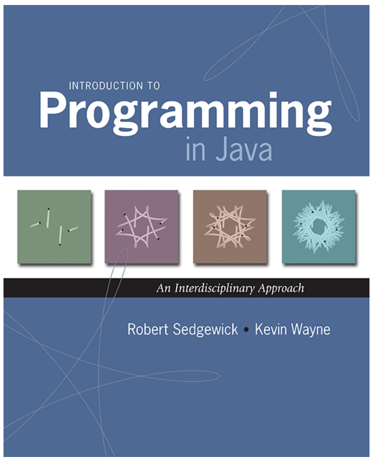
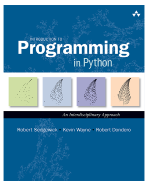
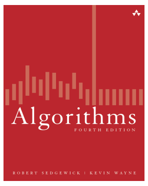

INTRODUCTION TO
Programming
in Java
An Interdisciplinary Approach
Robert Sedgewick • Kevin Wayne


Introduction to
Programming
in Python
An Interdisciplinary Approach
Robert Sedgewick Kevin Wayne Robert Dondero



Search

Appendix B: Writing Clear Code

The overarching goal when writing code is to make it easy to read and to understand. Well-written programs are easier to debug, easier to maintain, and have fewer errors. Writing a program is a lot like writing an essay. When writing an essay, your message is more convincing when it is accompanied by proper grammar and punctuation. When writing computer programs, you should follow the same principle. It is even more important when programming since someone may be assigned to maintain and support your code for long periods of time. You will appreciate the importance of good style when it is your task to understand and maintain someone else's code!

Coding.

- Keep programs and methods short and manageable.
- Use language-specific idioms.
- Use straightforward logic and flow-of-control.

- Avoid magic numbers (numbers other than `-1`, 0, 1, and 2); instead, give them meaningful symbolic names.

Naming conventions.

Here are some general principles when choosing names for your variables, methods, and classes.

- Use meaningful names that convey the purpose of the variable. Choose names that are easy to pronounce, and avoid cryptic abbreviations. For example, use `wagePerHour` or `hourlyWage` instead of `wph`. Use `polygon` instead of `p` or `poly` or `pgon`.

- Be consistent.

- Name `boolean` variables and methods so that their meaning is unambiguous, e.g., `isPrime` or `isEmpty()` or `contains()`.

- Use shorter names (e.g., `i`) for short-lived variables and loop-index variables. Use more descriptive names for variables that serve an important purpose.

- Avoid generic names like `foo` or `tmp` and meaningless names like `fred`. Use terminology from the application domain when possible.

- Name a constant by its meaning, not its value, e.g., name your variable `DAYS_PER_WEEK` instead of `SEVEN`.

IDENTIFIER	NAMING RULES	EXAMPLE

Variables	A short, but meaningful, name that communicates to the casual observer what the variable represents, rather than how it is used. Begin with a lowercase letter and use camel case (mixed case, starting with lower case).	`mass`
		`hourlyWage`
		`isPrime`

Constant	Use all capital letters and separate internal words with the underscore character.	`N`
		`BOLTZMANN`
		`MAX_HEIGHT`

Class	A noun that communicates what the class represents. Begin with an uppercase letter and use camel case for internal words.	`class Complex`
		`class Charge`
		`class PhoneNumber`

Method	A verb that communicates what the method does. Begin with a lowercase letter and use camelCase for internal words.	`move()`
		`draw()`
		`enqueue()`

Commenting.

Programmers use comments to annotate a program and help the reader (or grader) understand how and why your program works. As a general rule, the code explains to the computer and programmer *what* is being done; the comments explain to the programmer *why* it is being done. Comments can appear anywhere within a program where whitespace is allowed. The Java compiler ignores comments.

- *Line comments*. An end-of-line comment begins with `//` (two forward slashes) and ends at the end of the line on which the forward slashes appear. Any text from the `//` to the end of the line is ignored.
- *Block comments*. A block comment begins with `/*` (a forward slash and asterisk) and ends with `*/` (asterisk and a forward slash). Any text between these delimiters (even if it spans multiple lines) is ignored.
- *Bold comments*. A bold comment is a special case of a block comment designed to draw attention.

```
/*---------------------------------------------------------
 *  Here is a block comment that draws attention
 *  to itself.
 *---------------------------------------------------------*/
```

- *Javadoc comments*. A Javadoc comment is a special case of a block comment that begins with `/**` (a forward slash and two asterisks). They are typically used to automatically generate the API for a class. Here are guidelines for writing Javadoc comments.

There is no widely agreed upon set of rules. Good programmers write code that documents itself.

- Make sure that comments agree with the code. Be careful to update the comments when you update the code.
- Do not write comments that merely restate the code. Generally, comments should describe *what* or *why* you are doing something, rather than *how*.

```
i++;      //  increment i by one
```

- Comment any potentially confusing code, or better yet, rewrite the code so that it isn't confusing.
- Include a bold comment at the beginning of each file with your name, date, the purpose of the program, and how to execute it.

```
/*-----------------------------------------------------------------
 *  Author:        Kevin Wayne
 *  Written:       5/3/1997
 *  Last updated:  8/7/2006
 *
 *  Compilation:   javac HelloWorld.java
 *  Execution:     java HelloWorld
 *
 *  Prints "Hello, World". By tradition, this is everyone's
 *  first program.
 *
 *  % java HelloWorld
 *  Hello, World
 *
 *-----------------------------------------------------------------*/
```

- Comment every important variable name (including all instance variables) with a `//` comment, and align the comments vertically.

```
private double rx, ry;    //  position
private double q;         //  charge
```

- Comment each method with a description of what it does. Include what it takes as input, what it returns as output, and any side effects. Use the parameters names in your description.

```
/**
 *   Rearranges the elements in the array a[] in random order
 *   using Knuth's shuffling algorithm.
 *
 *   Throws a NullPointerException if a is null.
 */
public static void shuffle(String[] a)
```

Whitespace.

Programmers use whitespace in their code to make it easier to read.

- Don't put more than one statement on a line.
- Use blank lines to separate your code into logical sections.

- Put a space between all binary operators (e.g., <=, =, +) and their operands. One possible exception is to emphasize precedence.

```
a*x + b
```

- Include a space between a keyword (e.g., `while`, `for`, `if`) and its opening parenthesis.
- Put a space after each statement in a `for` loop.

```
for(int i=0;i<N;i++)    vs.      for (int i = 0; i < N; i++)
```

- Put a space after each comma in an argument list.
- Put space after each comment delimiter.

```
    //This comment has no space           //  This comment has two
    //after the delimiter and is          //  spaces after the delimiter
    //difficult to read.                  //  and is easier to read.
```

- Do not put spaces before a semicolon.
- Do not put spaces between an object name, the `.` separator, and a method name.
- Do not put spaces between a method name and its left parenthesis.
- Include blank lines to improve readability by grouping blocks of related code.
- Use spaces to align parallel code whenever it enhances readability.

```
int N      = Integer.parseInt(args[0]);      //  size of population
int trials = Integer.parseInt(args[1]);      //  number of trials
```

Indenting.

Programmers format and indent their code to reveal structure, much like an outline.

- Avoid lines longer than 80 characters.
- Do not put more than one statement on a line.
- Indent a fixed number of spaces. We recommend 3 or 4.
- Always use spaces instead of tabs. Modern IDEs (including DrJava) insert spaces when you type the tab key - these are known as *soft tabs*. Hard tabs are obsolete: in ancient times, they were used for data compression.
- Use a new indentation level for every level of nesting in your program.
- Follow either the K&R or BSD/Allman indentation styles for curly braces, and use it consistently. We consistently use the former for the booksite and the latter in the textbook.

```
//  K&R style indenting
public static void  main(String[] args) {
    System.out.println("Hello, World");
}

//  BSD-Allman style indenting
```

```
public static void main(String[] args)
{
    System.out.println("Hello, World");
}
```

Q + A

Q. Are there any official coding standards?

A. Here are Sun's Code Conventions for the Java Programming Language. However, this document was written in 1997 and is no longer being maintained.

Q. Any good references on programming style?

A. The Practice of Programming by Brian W. Kernighan and Rob Pike is a classic.

Q. Do Java comments nest?

A. No. So you cannot eliminate a block of code by simply surrounding it with the block comment delimiters (since the block itself may contain a `*/` delimiter).

Q. How can I autoindent my code?

A. Use an editor designed for writing code. For example, in DrJava, if you select a region of code and hit the Tab key, it will reindent it for you automatically.

Q. Are there tools for enforcing coding style?

A. Yes, we recommend Checkstyle. If you followed our Windows, Mac OS X, or Linux instructions, checkstyle will already be installer and set up to use our configuration file.

Q. How can I write unmaintainable code?

A. Here's one guide to designing unmaintainable code and here's another.

Exercises

1. **Fun with comments.** What is the value of `a` after the following code fragment is executed?

   ```
   //*/
   a = 17;
   /*/
   a = -17;
   //*/
   ```

   Repeat the question after deleting the first `/`.
2. **More fun with comments.** What does the following print?

   ```
   public static void main(String[] args) {
      boolean nesting = true;
      /* /* */ nesting = false; // */
      System.out.println(nesting);
   }
   ```

   Answer: this code prints `true` if `/*` comments can nest and `false` if they can't. Since Java comments don't nest, it prints `false`.

Last modified on September 16, 2014.