## **Java Code Style Guidelines**

## “*Instead of imagining that our task as programmers is to instruct computers what to do, let us concentrate rather on explaining to human beings what we want the computer to do*.”

## – Donald Knuth

In academic courses we normally require students to work on their homework assignments individually: Each student writes code on his or her own. This is done in order to build competence and self-confidence, and to learn how to learn and develop individually. In general, though, programming is a team effort: Typical software development projects engage dozens if not hundreds of collaborators, all working on the same code base. With that in mind, one of the objectives of our Introduction to CS course is to teach you how to write code which is a pleasure to read and debug, and a pleasure to maintain and extend.

Professional programmers are judged by their ability to write lucid and self-explanatory code. This is an acquired art which is best learned by seeing many well-written code examples, and by following a few simple styling rules. Every organization – be it a company, an open-source community, or a programming course – is advised to publish a standard set of rules that guide how members of the organization are expected to write code. This document is the styling guidelines of Reichman University’s Introduction to CS course.

Later in your career you may deviate from some of these guidelines, and develop your own coding style. This is fine, as long as other programmers will find your code clear and readable. Here is an example: We recommend to always put spaces around operators, resulting with statements like x = a + b, and not x=a+b; However, it may be argued that x= a + b is more self-descriptive: Dropping the space before the = operator hints that this is an assignment operator, unlike algebra’s = relation. So which is better: x = *someValue* , or x= *someValue*? Both styles are acceptable, but we prefer the former. Why? It’s a matter of taste, and habit – not terribly important in this particular case. What *is* important is to *be consistent*: If you choose to follow a styling rule, stick to it throughout your code.

As the course progresses you will see numerous code examples written by the course staff. In some cases we will violate some of the rules given below. This typically happens because we have to squeeze the code into a single lecture slide, or avoid a page break in the middle of a code segment. And, we’ll probably make some styling mistakes here and there.

This document is work in progress. The first version, given here, supports the first half of the course. Later in the course we will introduce more concepts, like object-oriented programming, and the document will be extended to cover them as well.

**Terminology:** Throughout this document, when we say “method” we mean what we call “function” in the first half of the course.

**English:** Like all other resources in this course, this document gives you an opportunity to improve not only programming skills, but also professional English skills. If you don’t understand some words, look them up, and develop your vocabulary.

**1. Formatting conventions**

Write one Java statement per line. In some cases we may break this rule and write two or more statements in a line, but this will be the exception. Make sure that your lines are not too long, and that you don’t force the code reader to use horizontal scrolling. A maximum of 80 characters per line is recommended.

If you have to break a line, do it sensibly:

|  |  |  |
| --- | --- | --- |
| // Good line break  x = a + b + c + d +  e + f + g; | // Less good  x = a + b + c + d  + e + f + g; | // Bad style  x = a + b + c + d  + e + f + g; |

Java-wise, all three code segments are equivalent, and work perfectly well. But if you’ll use the one of the left, people will appreciate your sensibility and attention to detail. If you’ll use the one in the middle, some people may think that you have no style. Use the one on the right, and people will start losing confidence in your code. If you can’t manage a simple indentation convention, what other problems are lurking in your code?

**2. White space**

Use single spaces to make your code readable:

|  |  |
| --- | --- |
| // Looks good  while (x < 100) {  sum = sum + x;  count = count + 1;  } | // Cluttered  while(x<100){  sum=sum+x;  count=count+1;  } |

The general rule is to insert a single space before and after each binary operator (meaning, an operator that has two operands), and before a left parenthesis (סוגר שמאלי). Here is another example:

|  |  |
| --- | --- |
| // Looks good  for (int i = 0; i < 100; i++) {  process(x, i);  } | // Cluttered  for(int i=0;i<100;i++){  process(x,i);  } |

Why is there no space before the left parenthesis in process(x, i)? This is an important exception: When you write a *method call*, don’t separate the method name from the left parenthesis. This makes the code more readable, allowing the reader to immediately discern that process is a method call.

Other exceptions to the “one space separator” rule are unary operators (operators that have one operand). For example, write -17, not – 17. Likewise, write n++ , not n ++.

**Blank lines:** Code segments can be long and difficult to read as one sequence. To help the reader follow the logical structure of your code, insert a blank line (and optionally a descriptive comment) before each major logical code subset.

**Comments**

There are three types of comments:

// Short comments, written for people who read the code and want to understand it.

/\* Multiline comments. Written for people who read the code and  
 want to understand it. \*/

/\*\* JavaDoc comments: Written for people who want to use a method or a class  
 \* without understanding how the code works. \*/

To these standard comment types we add a fourth type which we sometimes use:

//// A temporary, “thing to do” or “reminder” kind of a comment,

//// which is typically removed once the code development is completed.

**Don't over-comment:** Code that is obvious needs no explanation. For example:

i = i + 1; // adds 1 to i (silly)

Assume that the code reader is a Java programmer who has basic programming skills.

In general, if you write too many comments, it means that your code is not well-written. Ideally, the code should explain itself. Most of this document illustrates how to do exactly that: Write well-designed code that explains itself, requiring a minimum of comments.

**3. Indentation and curly braces** {}

Statements that form a logical block should appear one below the other, left justified. Use indentation (הָזָחָה) to clarify your code structure. The standard indentation is 4 spaces. Here are two equivalent examples of good indentation:

|  |  |
| --- | --- |
| if (x < y) {  x = y;  y = 0;  }  else {  x = 0;  y = y / 2;  } | if (x < y) {  x = y;  y = 0;  } else {  x = 0;  y = y / 2;  } |

Some programmers put the opening brace on a new line, as shown below. This is not a good practice: The number of lines that the code reader can see on the screen without vertical scrolling is limited; Putting braces in separate lines wastes this precious resource.

// Too long, too “airy”, and difficult to comprehend as one logical unit:

if (x < y)

{

x = y;

y = 0;

}

else

{

x = 0;

y = y / 2;

}

**Watch out:** The following code segment works, but opens the door for potential bugs:

while (x < 100)

doThis();

At some point in the future, either you or someone else may extend the code as follows:

while (x < 100)

doThis();

doThat();

Apparently, the programmer who made this addition thought that both doThis and doThat are inside the while loop (which will be true in Python, but not in Java). To prevent such latent bugs, always enclose every code block in curly braces, *even if it includes a single statement*:

while (x < 100) {

doThis();

}

This rule applies to all control structures: if, if-else, for-loops, and while-loops.

When it is obvious that a code block contains only one statement, you can break this rule. If you do so, put the single statement immediately after the condition, and not in a separate line:

if ( ... ) doThis();

**4. Structuring if…else statements**

Consider the following two equivalent code segments:

|  |  |
| --- | --- |
| if (*condition*) {  ...  *many lines of code*  ...  } else {  x = 100;  } | if (!*condition*) {  x = 100;  } else {  ...  *many lines of code*  ...  } |

The right version has a better style. As a rule, put the shorter code block (in this case, one statement only) as the first branch of the if … else statement. This make the code easier to read.

**5. Naming Conventions**

A computer program contains many names that are invented by the code writer: Class names, method names, variable names. Taken together, all these names are called *identifiers*. Choosing self-describing names that conform to naming conventions is the most important element in program readability. Therefore, pay special attention to the names that you choose to use in your programs.

A project development team consists of system architects, and developers. In this course, the course staff plays the role of system architects, and the students play the role of developers. When we ask you to write code, we normally provide a class skeleton with classes and methods that we already named, and ask you to complete the missing code. Therefore, most of the identifiers that you will have to invent in the first half of the course are *variable* names (and, occasionally, names of *private methods*).

**CamelCase:** The names of variables and methods are written in a style known as “camelCase”: Full or abbreviated English words are smashed together, and the first letter of each word except the first one is capitalized. For example: midValue, sumOfColumns, getPrice, sortInPlace, and so on.

Class names follow the same convention, except that the first letter is capitalized: Math, LinkedList, MatrixOps, and so on.

One exception to the camelCase convention is the name of *constants*. The naming convention of constants is upper-case letters, separated by underscores. Here are some examples:

public static final double PI = 3.141592653589;

public static final double SPEED\_OF\_LIGHT = 299792458; // in milliseconds

public static final double SHEKELS\_PER\_DOLLAR = 3.80; // as of Oct 1, 2023

public static final int MAX\_LIST\_SIZE = 1000;

In Java, constants are implemented as variables that don’t change throughout the program’s execution. This requirement is enforced by the keyword final. If a constant is defined at the class level (and not inside some method), it is said to be static. If we want to allow code from other classes use a constant, we make it public.

**Long names are fine:** A long name that describes what a variable or a method are doing is better than a short name that leaves doubt. Suppose that list is an ordered sequence of int values like (7, 5, 3, 2, 3, 5), and suppose that we are writing a method that removes the last occurrence (מוֹפָע) of a value. For example, removing the last occurrence of 3 results in the list (7, 5, 3, 2, 5). How should we name this method? Here are two possibilities:

removeLastOccurence(list) remLastOcc(list)

The left name is better. A self-describing name is preferred to a “cool” shorthand that forces the reader to guess what the method is doing.

**5.1 Variable names**

Variable names generally start with a lowercase letter: sum, count, midValue, etc. The variable name should be used to convey what the variable is used for. For example, suppose you use a local variable to accumulate a sequence of values, and another local variable for counting them. In this case it makes sense to name these variables sum and count. Java doesn’t care if you name them a and b, or perhaps s and c, but such arbitrary names make the code cryptic and difficult to work with.

As we just mentioned, it is perfectly ok to use long variable names, like numberOfLines. Yet in some cases long variable names can be annoying, making the code cumbersome. Here are some examples:

// Not clear what the variable names stand for:

c = (f – 32) \* 5 / 9

// Bad style (inconsistent naming):

celsius = (f – 32) \* 5 / 9

// Clear, but cumbersome:

tempratureInCelsius = (tempratureInFahrenheit – 32) \* 5 / 9

// A reasonable compromise:

tempInC = (tempInF – 32) \* 5 / 9

It is not easy to come up with variable names that are both descriptive and elegant. One has to exercise judgement, and taste. While we are at it, never use Hebrew names, or obscure names and private jokes. Here are some bad variable names: mispar, shever, sof, xMamashi, trickOfShimon, etc.

In the first 3-4 weeks of the course we use three kinds of variables: Static variables, local variables, and parameters of methods. Static variables are variables that are declared at the class level, and not at the method level. The scope of static variables is the entire class, meaning that every method can use them. The constants that we described earlier in this section are examples of (final, i.e. immutable) static variables. Local variables are variables that are declared in the body of a method, and parameters are variables that are declared in the method’s signature. *Parameter variables* are essentially local variables that are initialized to *argument values* that are passed by the method caller.

Typically, we use shorter names for *parameter* and *local variables*, and longer names for *static variables*. The reason is this: When you read a well-written method signature, you can understand quite well the roles that the parameters play in the method’s code. Likewise, when reading the declaration of a local variable, you can normally guess from the context what the variable is designed for. At the same time, static variables come into the method code out of the blue, since they are declared at the class level, which may be far away (vertically speaking) from the place in which they are used. Therefore, static variables must have self-explanatory names that are easy to read and understand, even if they are long and cumbersome:

/\*\* Returns the energy embedded in the given mass \*/

public double energy (int m) {

    return m \* SPEED\_OF\_LIGHT \* SPEED\_OF\_LIGHT;

}

As the example shows, it is quite possible to guess what m stands for. At the same time, if we used something like “c” for the speed of light constant, the code would be less readable. The use of upper-case letters and underscores tell the code reader that this is a class-level constant.

Local variables are declared in the method’s body. The scope of a local variable is usually short, and its meaning is often obvious either from a comment on its declaration or from the context in which it is used. Therefore, like parameter names, names of local variables may be short.

If a variable is “important”, or plays a major role in the code, we can name it using upper-case letters. Examples:

int T = getInput();  // Number of times we’ll run the simulation

int N = getInput();  // Length of the sequence

**Where to declare a local variable**: Consider the following code examples, each doing the same thing using the two local variables sum and x:

|  |  |
| --- | --- |
| // Version 1:  public void foo() {  int x;  int sum = 0;  ...  *many lines of code*  ...  while (...) {       x = getInput();       sum += x;  }  // More code follows  } | // Version 2:  public void foo() {  ...  *many lines of code*  ...  int sum = 0;  while (...) {       int x = getInput();       sum += x;  }  // More code follows  } |

Let’s start with Version 2: It looks like x is re-declared in each loop iteration. In fact, the Java compiler makes sure that the variable is declared only once. So don’t worry about “re-declaring” a local variable. With that in mind, which code version is better?

In Version 1, the two local variables are declared at the top of the method, and things look nice and tidy. But... there are problems. First, by the time readers of this code will get to read the loop, they will forget that variable x exists, and what purpose it serves in the program. Second, because x is declared before the loop, this variable will persist after the loop terminates. This may cause nasty bugs down the road. In contrast, Version 2 declares sum just before the loop, and x inside the loop, so the code is more readable. Also, once the loop terminates, the x variable ceases to exist, as it should.

So here is the general rule: A local variable should be declared close to the place in the code in which it is used. Also, when declaring a local variable, it is recommended to initialize it to some value. Although Java initializes local variables automatically to default values, a variable declaration like int sum = 0; is more readable than int sum;

**5.2 Class and interface names**

(We’ll discuss interfaces and the keywords extends and implements in the second half of the course). The name of a class or an interface is generally a noun (שם עצם), like Pixel, or noun phrase, like GreyScalePixel, beginning with a capital letter:

class Point { ... }

class ColoredPoint extends Point { ... }

interface List { ... }

class LinkedList implements List { ... }

**5.3 Method names**

Just like variable names, method names generally begin with a lowercase letter.

**Two kinds of methods:** Some methods are designed to compute and return a value, like sqrt(x), and other methods are designed to carry out an action, like print(). The former are sometimes called *functions*, and the latter *procedures*.

Technically speaking, the same method can be both a procedure and a function, implementing an action that also returns a value. The best practice though is to break such operations into two separate methods: One that does the action, and one that returns a value. For example, suppose you want to return the last value from a list (we’ll learn lists later in the course), and remove this value from the list. In principle, you can design a method like getAndRemoveLastValue(list). It is better though to design and use two separate methods: One that returns the list’s last value, say last(list), and one that performs removeLast(list). The resulting code will be more readable, and the method names more elegant.

**Naming functions** (methods that return values): The convention is to use a noun (שם עצם) or an adjective (שם תואר) that describes the value that the method returns. Examples:

/\*\* Returns the minimum of the two given values \*/

public int min(int x, int y) { ... }

/\*\* Returns an array that contains the elements of the given array, sorted \*/

public int[] sorted(int[] arr) { ... }

If the method is designed to return a boolean value, the convention is to use phrase starting with "is", or "has", describing what the returned value means:

/\*\* Checks if the given year is a leap year \*/

public boolean isLeapYear(int year) { ... }

/\*\* Checks if a < b < c \*/

public boolean isStrictlyAscending(int a, int b, int c) { ... }

/\*\* Checks if the given string has more lines to process \*/

public boolean hasMoreLines(String str) { ... }

If you focus only on the method signatures, some of these method names may look strange. However, if you shift your attention to the caller’s perspective, the names make perfect sense. Examples:

int z = min(x, y);

...

arr1 = sorted(arr2);

...

if (isLeapYear(2018)) { ... }

...

while (hasMoreLines(text)) {

String line = nextLine(text)

}

So, when naming a method, the general rule is that the method name should make sense from the caller’s perspective. A method can be called numerous times, in many different places in the class, as well as in other classes. If the method name makes sense, the code will be self-explanatory, and there will be no need to write comments that explain what the method call is doing.

**6. Documentation**

Make yourself a habit to document your code as you write it, instead of delaying the documentation to the end of the program’s development. The purpose of the documentation is to help you, and other programmers and reviewers, understand (i) how to use your code, and (ii) how the code works (if the code is not self-explanatory). So, document your program well and early, but don’t overdo it.

The [Elements of Style](https://www.gutenberg.org/files/37134/37134-h/37134-h.htm), a classical writing guide which is available on-line, gives writing rules that are also applicable to writing good documentation. Among them are:

Omit needless words.

Use the active voice.

Follow these rules when documenting methods. For example, don't write “In this method an array arr is searched for a value v. If the value is found, a true value is returned. If the value is not found, a false value is returned. ''. Instead, write succinctly:

/\*\* Checks if the array contains the given value. \*/

boolean isIn(int v, int[] arr)

When you write comments that explain what a method does, ask yourself if better method and parameter names could make the comments shorter. There is a general lesson here: If you’ll choose good names, and write your code clearly, you will have to write very few comments.

As a convention, each class and each method should start with a comment that describes what the class or method is doing. Do you need to write more comments? Here, again, is the general advice, which is also a good summary of this entire document:

**Whenever you write a comment that explains what your code does, ask yourself:  
Can I write the code in a way that makes the comment unnecessary?**

Since this is a style guide, we’ll conclude with a stylistic anecdote that demonstrates how brevity can communicate deeper meaning. British writer George Bernard Shaw sent Winston Churchill two theater tickets to the opening of his play with a note: "Here are two tickets for the first night of my new play; bring a friend, if you have one." Churchill responded, "Cannot attend first night; will attend second, if there is one."