

Style in Java

Basic principles

Readability is the most important attribute of style

Choose good names. The most direct way of explaining what a program is about is by selecting good names for variables, types, procedures, etc. If I'm trying to write some new code, and I can't figure out a good name for a class or method, I'll often be stuck until I have something. Many times, I find that the reason I can't pick a good name is that I don't understand enough about what I'm trying to say; with understanding comes good names, and vice versa. Don't use abbreviations

Code so that changes will be easier.

Formatting

1. Indent (four spaces) to show organizational structure of code.
2. Blank lines improve readability by setting off sections of code that are logically related.
3. Two blank lines should always be used in the following circumstances:
 - a. Between sections of a source file
 - b. Between class and interface definitions
4. One blank line should always be used in the following circumstances:
 - a. Between methods
 - b. Between the local variables in a method and its first statement
 - c. Before a block or single-line comment
 - d. Between logical sections inside a method to improve readability
5. Do not place multiple statements on the same line.
6. Do not exceed 80 characters on a line. Indent continuation lines.
7. One declaration per line is recommended since it encourages commenting.
8. Compound statements are statements that contain lists of statements enclosed in braces "{ statements }". The enclosed statements should be indented one more level than the compound statement.
9. switch Statements should have the following form:

```
switch (condition) {  
  case ABC:  
    statements;  
    /* falls through */  
  
  case DEF:  
    statements;  
    break;  
  
  case XYZ:  
    statements;  
    break;
```

```
default:
    statements;
    break;
}
```

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Naming

1. Interface/Class Names
 - a. Begin with an uppercase letter. Each class should be in a separate file named using the *ClassName.java*.
 - b. Keep your class names simple and descriptive. Use whole words - avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
2. Methods
 - a. Method names should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.
 - b. Write methods that only do "one thing".
 - c. Avoid overloading methods on argument type. If you need to specialize behavior according to the class of an argument, consider instead choosing a general type for the nominal argument type (often `Object`) and using conditionals checking `instanceof`.
 - d. Make methods as specialized and self-contained as possible. This will allow for code reuse and maintenance.
 - e. Methods to get and set an attribute variable `V` have names `getV` and `setV`.
 - f. A method to return the length of something should be named `length`.
 - g. A method that tests a boolean condition `V` should be named `isV`. Example: `isInterrupted`.
 - h. A method that converts its object to a particular format `F` should be named `toF`. Example: `toString`.
 - i. Whenever possible, base names of methods in a new class on names in an existing class that is similar.
3. Constant Names: all names must be explained via a comment
 - a. Names should be sequence of one or more words, acronyms, or abbreviations, **all uppercase**, components separated by underscore. Examples: `MIN_VALUE`, `MIN_RADIX`.
 - b. A group of constants that represents alternative values of a set are sometimes specified with a common prefix: `PS_RUNNING` and `PS_SUSPENDED`.
4. Local Names & Parameters: all names must be explained via a comment
 - a. Fields should have names which are nouns, noun phrases, or abbreviations for nouns.
 - b. Java classes should have very few public or protected fields.
 - c. Use abbreviations sparingly.
 - d. Bad names are often obvious: they're the ones which are hard to remember or don't seem to describe their purpose well. Until an interface is frozen or exposed to a body of users, it's

- a good idea to repeatedly go back to the names that bother you. When a name is good, it probably won't stick out at you. Eventually, you'll find a better name.
- e. Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios.
 - f. Avoid giving a variable the same name as one in a superclass as this is usually an error.
 - g. Minimize the use of abbreviations. If abbreviations must be used, be consistent in their use. An abbreviation should have only one meaning. If using *min* to abbreviate *minimum*, do so everywhere and do not later use it to abbreviate *minute*.
 - h. Use names that describe a parameter's meaning rather than names that describe a parameter's type.
 - i. Boolean variable names should contain *Is* or other words which implies Yes/No or True/False values, such as `PrintJobsFinished`. Often names can be chosen either in a positive (e.g., `isValid` or `finished`) or negative (`isInvalid` or `doContinue`) form. Always use the positive form (and ensure that the implementation matches the name).
 - j. One character local variable names should be avoided, except for looping variables, temporary variables. Conventional one-character names are
 - i. `b` for byte, `c` for char, `l` for long
 - ii. `i,j,k` for integers
 - iii. `d` for double, `f` for float
 - iv. `e` for Exception, `o` for Object, `s` for String
 - v. `v` for arbitrary value

Methods

1. Always validate procedure parameters. Validating parameters is the first task that should be performed by the procedure.
2. Methods should only have one exit point.
3. Always pass in the data/object(s) the procedure needs to perform the task instead of relying on global or module level variables inside of the procedure. This makes it easier to understand.
4. Document cases where the return value of a called method is ignored. These are typically errors. If it is by intention, make the intent clear. A simple way to do this is: `int unused = obj.methodReturningInt(args)`
5. Each method must have a few lines of comments which explain what parameters are needed and what the method accomplishes.

Readability

1. Declare all constants (except for 0, 1, and 2) as `final`. Mysterious numeric constants, termed *magic numbers*, hinder maintenance as well as readability. This means you cannot use the literal `5` (for example) in your code. The reader asks, "Why 5?" Defining a constant with the value of 5 allows the programmer to explain why the value of 5 is used.
2. It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems.
3. Use `XXX` in a comment to flag something that is awkward but works. Use `FIXME` to flag something that is broken.
4. Avoid assignments (`=`) inside `if` and `while` condition. These are almost always typos.

5. Write searchable code. For example, when breaking apart long lines of code, do not put a carriage return between the keyword `new` and the class name. Thus, you can find all places where instances of class `BigStuff` are created by searching for `new BigStuff`.
6. Use names you can pronounce. People talk about programs. It's easier to talk about code if you can pronounce the words inside of it.
7. Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {  
    return true;  
} else {  
    return false;  
}
```

should instead be written as

```
return booleanExpression;
```