# Formatting

You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. It helps to have an automated tool that can apply those formatting rules for you.

The Purpose of Formatting
Code formatting is about communication, an communication is the professional developer's first order of business.

Perhaps you thought that "getting it working" was the first order of business for a professional developer. I hope by now, however, that this book has disabused you of that idea. The functionality that you create today has a good chance of changing in the next release, but the readability of your code will have a profound effect on all the changes that will ever be made.

## Vertical Formatting

How big should a source file be? In Java, file size is closely related to class size.
the small difference in vertical position implies a very large difference in absolute size.

It appears to be possible to build significant systems out of files that are typically 200 lines long, with an upper limit of 500. Although this should not be a hard and fast rule, it should be considered very desirable. Small files are usually easier to understand than large files are.

## The Newspaper Metaphor

Think of a well-written newspaper article. You read it vertically. At the top you expect a headline that will tell you what the story is about and allows you to decide whether it is something you want to read. The first paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts. As you continue downward, the details increase until you have all the dates, names, quotes, claims, and other minutia.

We would like a source file to be like a newspaper article.

A newspaper is composed of many articles; most are very small. Some are a bit larger. Very few contain as much text as a page can hold. This makes the newspaper usable. If the newspaper were just one long story containing a disorganized agglomeration of facts, dates, and names, then we simply would not read it.

## Vertical Openness Between Concepts

Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines.
Each blank line is a visual cue that identifies a new and separate concept. As you scan down the listing, your eye is drawn to the first line that follows a blank line.

## Vertical Density

If openness separates concepts, then vertical density implies close association. So lines of code that are tightly related should appear vertically dense.

## Vertical Distance

Concepts that are closely related should be kept vertically close to each other . Clearly this rule doesn't work for concepts that belong in separate files. But then closely related concepts should not be separated into different files unless you have a very good reason. Indeed, this is one of the reasons that protected variables should be avoided.

## Variable Declarations

Variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear at the top of each function.
Control variables for loops should usually be declared within the loop statement.
In rare cases a variable might be declared at the top of a block or just before a loop in a long-ish function.

## Instance variables

There have been many debates over where instance variables should go. In C++ we commonly practiced the so-called scissors rule, which put all the instance variables at the bottom. The common convention in Java, however, is to put them all at the top of the class. I see no reason to follow any other convention. The important thing is for the instance variables to be declared in one well-known place. Everybody should know where to go to see the declarations.

## Dependent Functions

If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use.

## Conceptual Affinity

Certain bits of code want to be near other bits. They have a certain conceptual affinity. The stronger that affinity, the less vertical distance there should be between them.
there are other possible causes of affinity. Affinity might be caused because a group of functions perform a similar operation.

## Vertical Ordering

In general we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling.2 This creates a nice flow down the source code module from high level to low level.

## Horizontal Formatting

This suggests that we should strive to keep our lines short. The old Hollerith limit of 80 is a bit arbitrary, and I'm not opposed to lines edging out to 100 or even 120. But beyond that is probably just careless. I used to follow the rule that you should never have to scroll to the right. But monitors are too wide for that nowadays, and younger programmers can shrink the font so small that they can get 200 characters across the screen. Don't do that. I personally set my limit at 120.

## Horizontal Openness and Density
We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. Consider the following function:
Another use for white space is to accentuate the precedence of operators.

## Horizontal Alignment
```
private   Socket socket;
private   InputStream input;
private   OutputStream output;
private   Request request;
private   Response response;
private   FitNesseContext context;
protected long requestParsingTimeLimit;
private   long requestProgress;
```

that this kind of alignment is not useful. The alignment seems to emphasize the wrong things and leads my eye away from the true intent. For example, in the list of declarations above you are tempted to read down the list of variable names without looking at their types. Likewise, in the list of assignment statements you are tempted to look down the list of rvalues without ever seeing the assignment operator. To make matters worse, automatic reformatting tools usually eliminate this kind of alignment.

## Indentation
A source file is a hierarchy rather like an outline. There is information that pertains to the file as a whole, to the individual classes within the file, to the methods within the classes, to the blocks within the methods, and recursively to the blocks within the blocks.

To make this hierarchy of scopes visible, we indent the lines of source code in proportion to their position in the hiearchy. Statements at the level of the file, such as most class declarations, are not indented at all. Methods within a class are indented one level to the right of the class. Implementations of those methods are implemented one level to the right of the method declaration. Block implementations are implemented one level to the right of their containing block, and so on.
Without indentation, programs would be virtually unreadable by humans.

## Team Rules
Every programmer has his own favorite formatting rules, but if he works in a team, then the team rules. A team of developers should agree upon a single formatting style, and then every member of that team should use that style. We want the software to have a consistent style. We don’t want it to appear to have been written by a bunch of disagreeing individuals.

Remember, a good software system is composed of a set of documents that read nicely. They need to have a consistent and smooth style. The reader needs to be able to trust that the formatting gestures he or she has seen in one source file will mean the same thing in others. The last thing we want to do is add more complexity to the source code by writing it in a jumble of different individual styles.