**Chapter 5: Formatting**

**SUMMARY**

Code Formatting is crucial for it is about communication, and it's the professional developer's first order of business.

**The Purpose of Formatting**

* Readability of the code will have a profound effect on all the changes that will ever be made.
* Coding style and readability sets precedents that continue to affect maintainability and extensibility long after the original code has been changed beyond recognition.

**Vertical Formatting**

* A small difference in vertical position implies a very large difference in absolute size.
* Small files are usually easier to understand than large files.

**The Newspaper Metaphor**

* Source files should be like newspaper articles to read. Name should be simple but explanatory. It should be sufficient to tell us whether we are in the right module or not.
* Detail should increase as we make downward until at the end we find the lowest level function and details in the source file.

**Vertical Openness between Concepts**

* All code is read left to right and top to bottom represents an expression or a clause, and each group of lines represents a complete thought.
* Vertical Openness has blank lines that separates the package declaration, the imports and each of the functions. It has a profound effect on the visual layout of the code.
* Each blank line is a visual cue that identifies a new and separate concept.

**Vertical Density**

* It implies close association.
* Lines of the code are tightly related and should appear vertically dense.

**Vertical Distance**

* Concepts that are closely related should be kept vertically close to each other.
* Its vertical separation should be a measure of how important each is to the understability of the other.

**Variable Declarations**

* Variables should be declared as close to their usage as possible.
* Local variables should appear at the top of each function.
* Control variables for loops should be declared within the loop statement.
* Instance variables should be declared at the top of the class.

**Dependent Functions**

* If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible, for it gives the program a natural flow.

**Conceptual Affinity**

* Certain bits of code want to be near other bits.
* The stronger the affinity, the less vertical distance there should be between them.
* Affinity might be based on a direct dependence such as one function calling another, or a function using variables.
* Affinity might also be caused, because a group of functions perform a similar operation.

**Vertical Ordering**

* It creates a nice flow down the source code module from high level to low level.

**Horizontal Formatting**

* A linear appearance above 80 characters is very significant.
* Programmers clearly prefer short lines.
* We should strive to keep our lines short.
* We should set the limit at 120 at least.

**Horizontal Openness and Density**

* Using horizontal whitespace to associate things that are strongly related and disassociate things that are more weakly related.
* Separating arguments that are closely related makes them appear disjoined instead of conjoined.

**Horizontal Alignment**

* Horizontal alignment seems not useful in declaring variables because it seems to emphasize the wrong things and can lead the eye away from the true intent.
* Unaligned declarations and assignments point out important deficiencies.

**Indentation**

* Source file is a hierarchy rather than an outline. Information is stated and pertains to the file as a whole.
* To make source file a hierarchy of scope visible, indentation should be considered in proportion to their position.
* Using indentation, you can easily spot the important details needed.

**Breaking Indentation**

* Never be tempted to break the indentation rule.

**Dummy Scopes**

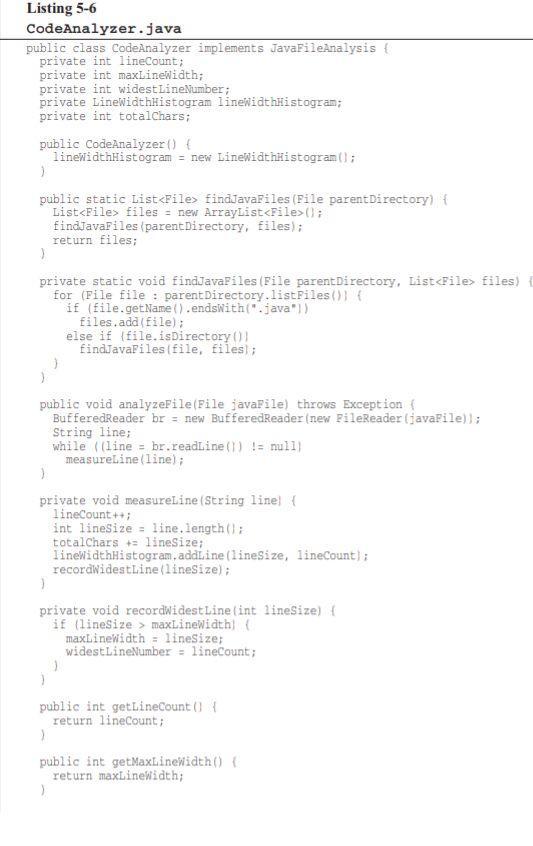
* Sometimes the body of a while or for statement is a dummy.

**Team Rules**

* Team developers should agree upon a single formatting style.
* A good software system is composed of a set of documents that is readable. They have to be consistent and smooth in style.

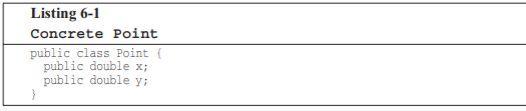
**Uncle Bob's Formatting Rules**

* Here is an example of how cose makes the best coding standard document.

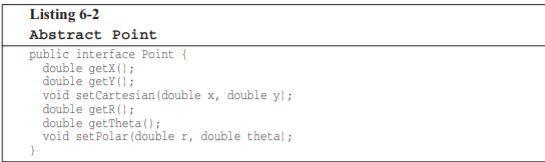


**Chapter 6: Objects and Data Structure**

**SUMMARY**

**Data Abstraction**

* Figure 1 represents more than just a data structure. Methods enforce an access policy. You can lead the individual coordinates independency but you must set the coordinates together as an atomic operation.



* Figure 2 represents a very clear implementation in rectangular coordinates and forces us to manipulate those coordinates independently. It exposes implementation.
* Hiding implementation is not just a matter of putting a layer of functions between the variables. It is about abstractions.

**Data/Object Anti-Symmetry**

* Objects hide their data behind abstractions and expose functions that operate on that data.
* Data structure exposes data and has no meaningful functions.
* Procedural code: makes it easy to add new functions without changing the existing data structures.
* OO code: makes it easy to add new classes without changing existing functions.
* In any complex system there are going to be times when we add new data types rather than new functions. For these cases objects and OO are most appropriate.

**The Law of Demeter**

* Module should not know about the inwards of the objects it manipulates.
* Methods should not invoke methods on objects that are returned by any of the allowed functions.

**Train Wrecks**

* It looks like a bunch of coupled train cars.
* Chains of calls like this are generally considered to be sloppy style and should be avoided.
* If they are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear violation of the Law of Demeter.

**Hybrids**

* They have functions that do significant things, and also have either public variables or public accessors and mutators that for all intents and purposes, make the private variables public, tempting other external functions to use those variables the way a procedural program would use a data structure.

**Data Transfer Objects**

* The quintessential form of a data structure is a class with public variables and no functions
* Data Transfer Objects are very useful structures especially when communicating with databases or passing messages from sockets and so on.
* They sometimes become the first in a series of translation stages that convert raw data in a database into objects in the application code.

**Active Record**

* These are special forms of DTOs. They are data structures with public variables; they have navigational methods like save and find. They are direct translations from database tables, or other data sources.
* We should treat Active Record as a data structure to create separate objects that contain the business rules and that hide their internal data.

Objects expose behavior or hide data. This makes it easy to add new kinds of objects without changing the existing behaviors.