**Triwits Coding Standards**

Purpose of Having Coding Standards:

* A coding standard gives a uniform appearance to the codes written by different engineers.
* It improves readability, and maintainability of the code and it reduces complexity also.
* It helps in code reuse and helps to detect error easily.
* It promotes sound programming practices and increases efficiency of the programmers.

Advantages of Coding Guidelines:

* Coding guidelines increase the efficiency of the software and reduces the development time.
* Coding guidelines help in detecting errors in the early phases, so it helps to reduce the extra cost incurred by the software project.
* If coding guidelines are maintained properly, then the software code increases readability and understandability thus it reduces the complexity of the code.
* It reduces the hidden cost for developing the software.

Drawbacks of not having proper coding conventions for a team:

* Without predefined conditions that all team member should follow can result in the following:
* Reduced engineers motivation
* Increased development time
* Complex codebase structure

Coding Principles

Source code should contain all the information necessary to re-create the object. This should include information about compile time overrides and object attributes.

* Source code should be edited and viewed interactively as much as possible. Source listings should only actually be printed in special circumstances. Use the browsing and scanning facilities of the series to examine source and compilation listings. Adopt layout conventions that facilitate this approach.
* Use the machine to find syntax errors and basic mistakes. The editors and compilers of the series give excellent diagnostics, and can be used to find low-level syntax errors.
* Take an incremental approach to development. OS/400 provides an interactive development environment. Rather than writing and testing programs as entirely separate steps, you can program from top, down. Write the main control structure of a program first, compile and test it, and then add the detailed coding, such as field validation.
* Strictly regulate source versions. One version of the source should be regarded as definitive. If changes are required, additional versions of the source should be copied to a separate development library and only be transferred back, together with the changed object, in a carefully controlled manner. For more information, refer to the section, Operating Environment Standards, in this section.
* Leave comments and prioritize documentation

Don’t assume that just because everyone else viewing the code is a developer, they will instinctively understand it without clarification. Devs are human, and it is a lot easier for them to read comments describing code function rather than scanning the code and making speculations.

Take an extra minute to write a comment describing the code function at various points in the script. Ensure that the comments guide any readers through the algorithm and logic implemented. Of course, this is only required when the code’s purpose is not apparent. Don’t bother leaving comments on self-explanatory code. Standardize headers for different modules

It is easier to understand and maintain code when the headers of different modules align with a singular format. For example, each header should contain:

1. Module Name
2. Date of creation
3. Name of creator of module
4. History of modification
5. Summary of what the module does
6. Functions in that module
7. Variables accessed by the module

**Naming conventions**

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier.

**(Inner) classes, interfaces, enums and annotations**

Names should be nouns, in mixed case with the first letter of each word capitalised. Try to keep your names simple and descriptive. Use whole words and avoid acronyms and abbreviations.  
Examples:  
class Raster  
class TreeFrame

**Interfaces**

Like class names, but if there is a name clash, the interface wins.  
Example:  
interface Remote Repository extends Repository

**Services**

Same as interfaces so don't append "Service" as you usually do not know if an interface is a service or not.  
Example:  
interface Connection Factory

**Implementation classes**

If a class implements an interface, it should use the name of the interface as part of its name, adding something specific for this implementation to it, or *Impl* if that does not make sense.  
Examples:  
class FileBasedRepository implements Repository  
class VersionServlet implements HttpServlet

**Exceptions**

Like class names; always ending in "Exception".  
Example:  
Input Exception

**Methods**

Methods should be verbs in mixed case with the first letter lowercase. Within each method name capital letters separate words. Property methods or get-set methods are used as follows: When a method is used to get a value start the method name with 'get'. When a method is used to set a value start the method name with 'set'. When a method returns a Boolean start the method name with 'is'.  
Examples:  
run();  
runFast();  
setBackground();  
isOptional();

**Variables (except for (constant) static final variables and member variables)**

All variables are in mixed case with a lowercase first letter. Words are separated by capital letters.  
Examples:   
int index;   
float myWidth;

**Member variables**

The same capitalisation as for normal variables prefixed with 'm\_'.  
Examples:  
int m\_index;  
float m\_myWidth;

**Constant (static final) variables, enum names**

Names should be all uppercase with words separated by underscores ('\_').  
Example:  
public static final int MAX\_LIMIT = 99;

**Packages**

Lowercase only; avoid lengthy package names; always start with *org.apache.ace*.  
Example:  
package org.apache.ace.demo.bundle;

For Reference:

https://developer.wordpress.org/coding-standards/wordpress-coding-standards/php/