Coding Guidelines

Java

We largely follow Hadoop coding guidelines:

- Use 2 (TWO) spaces to indent code.
- Use LF (Unix style) line endings.
- Do **not** use the `@author` Javadoc tag. (Be modest ! :-))
- For the rest, follow the original Sun Java coding conventions.
- If some file has different formatting, try to match the existing style.
- Use CheckStyle to verify your coding style.
- Avoid mixing both style and functionality changes in one commit.
- Make sure all files have the needed license header.

C#

We largely follow the coding guidelines of the .NET Core open source project.

Comments

We require to add comments to generate proper JavaDoc and XMLDoc documentation. In addition to this requirement, we encourage committers to add comments (if needed) to make code easier to understand.

- Class
- Interface
- Public constructor
- Public method
- Important fields
- Code part that captures complex logic

Deprecation and Obsolete

We strive not to break source compatibility between versions of REEF. Instead, we mark APIs as deprecated or obsolete in one version, and then remove it in the next. When marking something as deprecated:

- Provide an alternative users should switch to.
- Document the alternative, as well as the version of REEF in which the code was deprecated.
- File a JIRA for the removal of the deprecated API in the next version.
- Make sure that all current uses of that API in REEF have either been moved to the new API or been deprecated as well.

On immutability of why you should make everything `final` or `readonly`

REEF favors immutable objects over mutable ones. When you browse the code base, you see that there is an enormous number of `final` and `readonly` modifiers used. The reason for this stems from the distributed and parallel nature of REEF: What cannot be changed doesn't have to be guarded with locks. Hence, immutability is an excellent tool to achieve both thread safety and performance. Following this line of thought, we arrive at the following guidelines:

- Make all instance variables possible `final`.
- Use the Java Concurrency in Practice annotations for the instance variables that aren't `final`.
- When a class has more than 3 non-`final` instance variables, consider breaking the class in two to limit the lock complexity.