ISO IEC JTC 1 SC 22 WG 23 N0627
19 January 2016

Proposed top Ruby specific guidance:

1. Use the language's built-in mechanisms (`rescue`, `retry`) for dealing with errors.
2. Use symbols for enumerators rather than named constants.
3. Provide code to catch exceptions resulting from mismatches between objects and methods.
4. Knowledge of the types or objects used is a must. Compatible types are ones which can be intermingled and convert automatically when necessary. Incompatible types must be converted to a compatible type before use.
5. In most cases a break statement can be avoided by using another looping construct. These are abundant in Ruby.
6. Provide code to catch exceptions resulting from mismatches between objects and methods.
7. Use symbols for enumerators rather than named constants.

For reference, here is all of the Ruby specific guidance from 24773 (n0461):

- Knowledge of the types or objects used is a must. Compatible types are ones which can be intermingled and convert automatically when necessary. Incompatible types must be converted to a compatible type before use.
- Do not check for specific classes of objects unless there is good justification.
- Provide code to catch exceptions resulting from mismatches between objects and methods.
- For values created within Ruby the user need not concern themselves with the internal representation of data. In most situations using specific binary representations makes code harder to read and understand.
- Network packets that go on the wire are one case where bit representation is important. In situations like this be sure to use the `Array#pack` to produce network endian data.
- Binary files are another situation where bit representation matters. The file format description should indicate big-endian or little-endian preference.
- Use symbols for enumerators rather than named constants.
- Do not define named constants to represent enumerators.
- Be aware that use of Bignums can have performance and storage implications.
- Use names that are clear and visually unambiguous.
- Be consistent in choosing names.
- Check that each assignment is made to the intended variable identifier.
- Use static analysis tools, as they become available, to mechanically identify dead stores in the program.
- Enable detection of unused variables in the processor.
- Ensure that a definition does not occur in a scope where a different definition is accessible.
- Know what a module defines before including. If any definitions conflict, do not include the module, instead use the fully qualified name to refer to any definitions in the module.

- Avoid unnecessary includes.
- Do not access variables outside of a block without justification.
- Use parenthesis around operators which are known to cause confusion and errors.
- Break complex expressions into simpler ones, storing sub-expressions in variables as needed.
- Read method documentation to be aware of side-effects.
- Do not depend on side-effects of a term in the expression itself.
- Avoid assignments in conditions.
- Do not perform assignments within Boolean expressions.
- Include an else clause, unless the intention of cases not covered is to return the value nil.
- Multiple expressions (separated by commas) may be served by the same when clause.
- Do not modify loop control variables inside the loop body
- Use careful programming practice when programming border cases.
- Use static analysis tools to detect off-by-one errors as they become available.
- Instead of writing a loop to iterate all the elements of a container use the `each` method supplied by the object's class.
- While there are some cases where it might be necessary to use relatively unstructured programming methods, they should generally be avoided. The following ways help avoid the above named failures of structured programming:
  - Instead of using multiple return statements, have a single return statement which returns a variable that has been assigned the desired return value.
  - In most cases a break statement can be avoided by using another looping construct. These are abundant in Ruby.
  - Use classes and modules to partition functionality.
- Methods which modify their parameters should have the exclamation mark suffix. This is a standard Ruby idiom alerting users to the behaviour of the method.
- Make local copies of parameters inside methods if they are not intended to be modified.
- The Ruby interpreter will provide error messages for instances of methods called with an inappropriate number of arguments. All other aspects of consistency should be checked thoroughly in accordance with the general guidance found in 6.36.5.
- Extend Ruby's exception handling for your specific application.
- Use the language's built-in mechanisms (`rescue`, `retry`) for dealing with errors.
- Consult implementation documentation concerning termination strategy.
- Do not assume each implementation behaves handles termination in the same manner.
- Provide documentation of encapsulated data, and how each method affects that data.
- Inherit only from trusted sources, and, whenever possible check the version of the superclass during initialization.
- Provide a method that provides versioning information for each class.
- Develop wrappers around library functions that check the parameters before calling the function.
- Use only libraries known to validate parameter values.

- Implementations may provide a framework for inter-language calling. Be familiar with the data layout and calling mechanism of said framework.
- Use knowledge of all languages used to form names acceptable in all languages involved.
- Ensure the language in which error checking occurs is the one that handles the error.
- Verify dynamically linked code being used is the same as that which was tested.
- Do not write self-modifying code.
- Use tools to create signatures.
- Avoid using libraries without proper signatures.
- Use library routines which specify all possible exceptions.
- Use libraries which generate Ruby exceptions that can be `rescue`d.
- Do not rely on unspecified behaviour because the behaviour can change at each instance.
- Code that makes assumptions about the unspecified behaviour should be replaced to make it less reliant on a particular installation and more portable.
- Document instances of use of unspecified behaviour.
- Avoid using features of the language which are not specified to an exact behaviour.
- The abundant nature of implementation-defined behaviour makes it difficult to avoid. As much as possible users should avoid implementation defined behaviour.
- Determine which implementation-defined implementations are shared between implementations. These are safer to use than behaviour which is different for every.