Rationale for Programming Languages — Ruby

IPA Ruby Standardization WG Draft

February 4, 2010

Contents

0 Introduction

This Rationale summarizes the deliberations of Ruby Standardization Working Group, which has been established under Information-technology Promotion Agency, Japan (IPA) to codify the specification of the Ruby language. This document has been published along with the draft specification to assist the review process. In this Rationale, the unqualified "WG" refers to Ruby Standardization Working Group.

In drafting the specification, the WG has set the following guidelines.

- Use Ruby 1.8.7 as the primary reference implementation.
- Intend that existing implementations without modification can conform to the specification.
- Keep the specification as compact as possible.

First, Ruby 1.8.7 is used as the primary reference implementation. Ruby 1.8 remains widely in use and there are several implementations which implement Ruby 1.8 features only. Also, it seems likely that Ruby 1.8 will remain in use in the years to come. Some might argue that Ruby 1.9 should be used as the primary reference, but Ruby 1.9 is moving fast and its features change frequently. A specification based on Ruby 1.9 would get quickly out of sync.

Second, the WG intends that existing implementations such as Ruby 1.8.7, Ruby 1.8.6, Ruby 1.9, JRuby, Rubinius, and IronRuby can conform to the specification without modification. There are some features which are not implemented in some of the implementations or are different among the implementations. Those features are excluded from the specification, or described as either "implementation-defined" or "unspecified."

Finally, the WG tries to keep the specification as compact as possible. But the WG would not like the specification to be so compact that no useful program can be written. The current draft therefore includes those classes, modules, and methods which are widely used and necessary for basic programming tasks.

The WG will add additional classes, modules, and methods which are useful for advanced programming tasks, to a revised specification in the future. The WG will also revise the specification to support Ruby 1.9 features when Ruby 1.9 gets widely used. The addition of those classes, modules, and methods and the support of Ruby 1.9 may be pursued concurrently if Ruby 1.9 gets widely used until the next revision of the specification.

0.1 Organization of the document

This Rationale is organized to parallel the draft specification as closely as possible to facilitate finding relevant discussions. Some subclauses of the draft specification are absent from this Rationale; this indicates that the WG thought no special comment was necessary.

A reference prefixed by "Draft" (e.g. Draft §6.1) indicates that it is a reference to the draft specification, and a reference prefixed by "Rationale" (e.g. Rationale §6.1) indicates that it is a reference to this Rationale. An unqualified reference (e.g. §6.1) is a reference to both the draft specification and this Rationale.

1 Scope

2 Normative references

3 Conformance

Some syntactic constructs, including those of regular expressions, are omitted in the draft specification according to the drafting guidelines, but a conforming processor may support such syntactic constructs. In this case, however, the processor shall accept any conforming programs.

4 Terms and definitions

Terms defined in the draft specification follow conventional usage among Ruby users.

However, the term *eigenclass* is not widely accepted among Ruby users, and is arbitrarily selected from some candidates such as *singleton class* and *metaclass* because there is no term widely accepted among Ruby users. The term *eigenclass* is used in “The Ruby Programming Language”, which is a book written by David Flanagan and Yukihiro Matsumoto.

5 Notational conventions

5.1 Syntax

Most of the syntax of the language is specified by production rules. However, some syntax is specified by verbal descriptions because it is impossible or difficult to specify by production rules.

For example, the syntax of the nonterminal symbol *line-content* is specified with a verbal description as follows (see Draft §8.5):

```
line-content ::
        source-character+
```

Any characters that are considered as *line-terminator*s are not allowed within a *line-content.*

Because a *line-terminator* may contain two characters, it cannot be specified as follows:

```
line-content ::
        ( source-character but not line-terminator )+
```

Instead, it can be specified as follows, but it is more complicated than the definition with a verbal description.

line-content ::
    ((*source-character* *source-character*?) **but not** *line-terminator*)+

5.2 Conceptual names

Ruby has some sets of nonterminal symbols which share similar semantics but which cannot be syntactically reduced to the same nonterminal symbol. *Conceptual name*s are introduced to organize such sets of nonterminal symbols. For example, *assignment* is a conceptual name which represents *assignment-expression* or *assignment-statement* (see Draft §11.4.2.1). Note that *conceptual name*s are not produced from the start symbol *program*.

6 Objects

The lifetime of an object and garbage collection are not specified in the draft specification because the WG consider them implementation details. Object allocation and garbage collection may be implemented in an implementation-defined manner; however, it shall not violate any other specification in the draft specification.

6.5 Boolean values

The term *false* has two related but different meanings in the draft specification, and they are distinguished by font faces. `false` (in typewriter face) represents a keyword, but **false** (in bold sans serif face) is distinguished from `false` and represents the only instance of the class `FalseClass`, to which `false` evaluates. The same font faces are used for the terms *nil* and *true*.

The terms *trueish value* and *falseish value* are introduced to classify objects into two categories. Only **false** and **nil** are *falseish value*s, and any other objects are *trueish value*s.

7 Execution context

7.1 Contextual attributes

Other than the stacks specified in the draft specification, Ruby 1.8 has another stack called *ruby_class*. However, the draft specification employs a different model, where the head of the list on the top of ⟦class-module-list⟧ is used instead of the top of *ruby_class*. This is because the top of *ruby_class* and the head of the list on the top of ⟦class-module-list⟧ are the same class or module in most cases. The top of *ruby_class* and the head of the list on the top of ⟦class-module-list⟧ can be different during an invocation of the method `class_eval` of the class `Module`, but such a situation is described in Step b of Draft §15.2.2.4.15.

8 Lexical structure

8.2 Source text

The draft specification does not support ISO/IEC 10646 but ISO/IEC 646:1991 IRV because the support of ISO/IEC 10646 is different between Ruby 1.8 and Ruby 1.9.

Both Ruby 1.8 and Ruby 1.9 can handle multibyte characters, but in different ways. In Ruby 1.8, strings can be encoded in ASCII, UTF-8, EUC-JP, or Shift_JIS, but strings are basically treated as byte strings. Ruby 1.9 supports CSI (code set independent) multilingualization, and UTF-8 as one of supported encodings. Ruby 1.9 also supports various encodings such as ISO-8859-X, GB18030, BIG5, EUC-KR, CP949, KOI8-R, and KOI8-U, and strings are treated as character strings (or technically codepoint strings) in Ruby 1.9.

The support of multibyte characters and ISO/IEC 10646 should be specified according to the behavior of Ruby 1.9 because Ruby 1.9 supports multibyte characters and ISO/IEC 10646 better than Ruby 1.8; however, the current primary reference is Ruby 1.8, and the support of multibyte characters and ISO/IEC 10646 is thus not described in the draft specification. When the primary reference is switched to Ruby 1.9 or 2.0 in the future, the support of ISO/IEC 10646 should be described in the specification.

8.3 Line terminators

The draft specification supports the use of either CR+LF (0x0d 0x0a) or LF alone (0x0a) as a line terminator, but not CR alone (0x0d), to be in line with existing implementations.

8.4 Whitespace

The draft specification does not include a *line-terminator* in the production rules of a *whitespace* because the distinction between a *line-terminator* and a *whitespace* contributes to the clarity of the specification. However, in Ruby, a *line-terminator* is often forbidden where *whitespace* shall not occur. The draft specification therefore forbids a *line-terminator* to occur at the location indicated by "[no *whitespace* here]".

8.5 Comments

`=begin` and `=end` shall be at the beginning of a line in the draft specification because existing implementations accept them only at the beginning of a line. However, a conforming processor may permit whitespaces before them because such a syntax extension is permitted in §3.

8.6 End-of-program markers

In most of existing implementations, source characters after an *end-of-program-marker* can be read from an instance of the class `IO` which is bound to the constant `DATA`. However, the draft specification does not require this feature because it is not necessary for basic programming tasks.

8.7 Tokens

8.7.2 Keywords

`__LINE__`, `__ENCODING__`, `__FILE__`, `BEGIN`, and `END` are reserved for the future use.

8.7.6 Literals

8.7.6.1 General description

Character literals (e.g. `?a`) are omitted in the draft specification because they evaluate to instances of different classes between Ruby 1.8 and Ruby 1.9; a character literal evaluates to an instance of the class `Integer` in Ruby 1.8, but to an instance of the class `String` in Ruby 1.9. An expression such as `"a"[0]` can be used instead of `?a`.

8.7.6.2 Numeric literals

The draft specification does not require IEC 60559:1989 because Ruby is implemented on some platforms that does not support IEC 60559:1989 such as VAX. However, if the underlying platform of a conforming processor supports IEC 60559:1989, the representation of an instance of the class `Float` shall be the 64-bit double format in IEC 60559:1989.

9 Scope of variables

9.3 Global variables

Ruby has some built-in global variables such as `$1`, which are semantically not global, but local. The draft specification does not specify such built-in global variables because they may be removed from Ruby in the future.

10 Program structure

11 Expressions

11.2 Logical expressions

The unary operator `!` is a built-in operator in Ruby 1.8, but is a method invocation in Ruby 1.9. Whether `!` is a method invocation or not is thus implementation-defined.

11.3 Method invocation expressions

11.3.3 Blocks

In contrast to Ruby 1.8, Ruby 1.9 does not allow some syntactic constructs such as *constant-identifier*s in a *block-parameter*. Whether they are allowed is therefore implementation-defined.

11.4 Operator expressions

11.4.2 Assignments

11.4.2.4 Multiple assignments

The term *multiple assignment* is less popular than *parallel assignment* among English speakers; however *multiple assignment* is used in the draft specification because Yukihiro Matsumoto, the creator of Ruby, prefer it.

There are some incompatibilities in multiple assignments between Ruby 1.8 and Ruby 1.9. For example, `x = *[1]` sets `x` to `1` in Ruby 1.8, but to `[1]` in Ruby 1.9. Behaviors in such cases are described as implementation-defined in the draft specification.

11.4.4 Binary operators

The binary operators `!=` and `!~` are built-in operators in Ruby 1.8, but are method invocations in Ruby 1.9. Whether they are method invocations or not is thus implementation-defined.

12 Statements

12.5 The `while` modifier statement

If the *statement* of a *while-modifier-statement* is a *rescue-expression*, most of existing implementations evaluate the *statement* once before evaluating the *expression*. For example, `print("hello") while false` prints nothing, but `begin print("hello") end while false` prints "hello". The behavior when the *statement* of a *while-modifier-statement* is a *rescue-expression* is described as implementation-defined in the draft specification because the behavior of existing implementations is confusing, and may be changed in the future.

12.6 The `until` modifier statement

The behavior when the *statement* of an *until-modifier-statement* is a *rescue-expression* is described as implementation-defined in the draft specification for the same reason as a *while-modifier-statement* (see §12.5).

13 Classes and modules

13.3 Methods

13.3.1 Method definition

In existing implementations, the value of a *method-definition* is **nil**. However, it may be changed to return a more meaningful value such as a symbol which represents the method name in the future. The value of a *method-definition* is therefore described as implementation-defined in the draft specification.

13.4 Eigenclasses

Eigenclasses were originally introduced to implement singleton methods, and were hidden from users at that time. But once eigenclass definitions had been introduced, users began to use them for advanced meta programming. The draft specification specifies eigenclasses because they are widely used in these days.

However, some details of eigenclasses such as their superclasses are different between Ruby 1.8 and Ruby 1.9. Therefore, such details of eigenclasses are described as implementation-defined in the draft specification.

14 Exceptions

15 Built-in classes and modules

15.1 General description

For the sake of brevity, the draft specification includes only classes, modules and methods which can be implemented in pure C89 without platform-specific extensions. If a method can be implemented easily in terms of other methods included in the draft specification, it is omitted. But if it is impossible or difficult to implement in terms of other methods included in the draft specification, the method is included.

15.2 Built-in classes

The following built-in classes are omitted in the draft specification.

`Method`, `UnboundMethod`, `Binding`: These classes are omitted to keep the draft specification as compact as possible and because they are not necessary for basic programming tasks. However, they will be included in a future revision of the specification.

`Thread`: It is omitted because it is not necessary for basic programming tasks, and is hard to implement on platforms which does not support threads.

`Dir`: It is omitted because it cannot be implemented in pure C89.

`Continuation`: It is omitted because it is removed from built-in classes in Ruby 1.9, and is hard to implement on some platforms.

`Data`: It is omitted because it is used only for writing extensions using external languages such as C.

15.2.1 Object

15.2.1.2 Direct superclass

The class `Object` has the class `BasicObject` as a direct superclass in Ruby 1.9. The draft specification therefore permits implementations where the class `Object` is not the root of the class inheritance tree.

15.2.3 Class

15.2.3.3 Instance methods

15.2.3.3.3 Class#new

Most of existing implementations invoke the method `allocate` on the receiver in Step b of Draft §15.2.3.3.3. However, the method `allocate` is not intended to be invoked directly by user programs. The draft specification therefore does not specify it.

15.2.8 Integer

15.2.8.1 General description

Most of existing implementations have subclasses of the class `Integer`, the class `Fixnum` and the class `Bignum`. However, the draft specification does not require these subclasses of the class `Integer` because they are defined only in order to increase implementation performance.

15.2.9 Float

15.2.9.1 General description

If an arithmetic operation involving floating point numbers results in NaN, an exception may be raised in a future version of Ruby. Therefore, the behavior in this case is left unspecified.

15.2.9.3 Instance methods

15.2.9.3.12 Float#round

In some implementations, the method `round` is implemented as `(x + 0.5).floor`, where `x` is the value of the receiver, for positive instances of the class `Float`. However, if `x` is `0.49999999999999994`, `(x + 0.5).floor` returns `1`, in which case it is obvious that the resulting value is not “the nearest integer” to the value of the receiver. This behavior is caused by a rounding error when calculating `0.49999999999999994 + 0.5`. The WG had considered it a bug, and reported it to implementors. It is thus specified as “the nearest integer” in the draft specification.

15.2.10 String

15.2.10.1 General description

A character is represented by an instance of the class `Integer` in Ruby 1.8, but by an instance of the class `String` in Ruby 1.9. For example, `"abc"[0]` returns `97` in Ruby 1.8, but `"a"` in Ruby 1.9. The draft specification permits both representations.

15.3 Built-in modules

The following built-in modules are omitted in the draft specification.

`ObjectSpace`, `GC`: These modules are omitted because they provide implementation dependent features and are hard to implement on some platforms.

`FileTest`: It is omitted because it cannot be implemented in pure C89.

`Precision`: It is omitted because it has been removed in Ruby 1.9.