

# Encoding in Ruby 1.9

## The Tower of Babel Compatibility Kit (TBCK)

Florian Gilcher <florian.gilcher@asquera.de>

Asquera GbR

30. Mai 2010

# Contents

- 1 Introduction
- 2 Smooth Sailing
- 3 Rough Waters
- 4 Usage for Europeans
- 5 Pitfalls

# Introduction

# On Encoding

## Americans

„I don't want to know!”

## Japanese

„How can you not know?!”

## Europeans

„Unicode solves everything!”

# On Encoding

## Americans

„I don't want to know!”

## Japanese

„How can you not know?!”

## Europeans

„Unicode solves everything!”

# On Encoding

## Americans

„I don't want to know!”

## Japanese

„How can you not know?!”

## Europeans

„Unicode solves everything!”

# On Encoding

Americans

„I don't want to know!”

Japanese

„How can you not know?!”

Europeans

„Unicode solves everything!”

Thanks to @wycats for that one...

# The technical problem

- Encodings are the Tower of Babel of computing
- going away for the western world with UTF-8
- not so for the eastern world

Ruby 1.9 solves this problem by (potentially) supporting all encodings.

## String: 1.8 -> 1.9

String is the biggest change in Ruby 1.9:

- is not Enumerable anymore
- not a list of bytes but a list of characters
- every String has an associated Encoding object
- Strings in multiple Encodings can exist in one environment

# Smooth Sailing

# String.encode (1)

String now provides a way to convert strings between encodings using the `#encode` and `#encode!` methods. It fails if the conversion between both Encodings is not defined or not possible.

## Sample

```
"foo".encoding #=> #<Encoding:UTF-8>  
"foo".encode("US-ASCII")  
"föö".encode("US-ASCII") #=> Encoding::UndefinedConversionError
```

# String.encode (1)

String now provides a way to convert strings between encodings using the `#encode` and `#encode!` methods. It fails if the conversion between both Encodings is not defined or not possible.

## Sample

```
"foo".encoding #=> #<Encoding:UTF-8>  
"foo".encode("US-ASCII")  
"föö".encode("US-ASCII") #=> Encoding::UndefinedConversionError
```

Note: Conversion into the same encoding does nothing.

# Concatenating Strings

Strings of different Encodings can be concatenated just fine if they are compatible.

## Sample

```
s1.encoding #=> #<Encoding:ISO-8859-1>  
s2.encoding #=> #<Encoding:US-ASCII>  
(s1 + s2).encoding #=> #<Encoding:ISO-8859-1>  
Encoding.compatible?(s1,s2) #=> #<Encoding:ISO-8859-1>
```

# Regex.encoding

Regular Expressions do have an encoding as well. It cannot be changed, though!

## Sample

```
r = /foo/  
r.encoding #=> #<Encoding:US-ASCII>  
r.encode("UTF-8") #=> undefined method `encode'
```

# Regexp.encoding

Regular Expressions do have an encoding as well. It cannot be changed, though!

## Sample

```
r = /foo/  
r.encoding #=> #<Encoding:US-ASCII>  
r.encode("UTF-8") #=> undefined method `encode'
```

Note: Be wary of bugs in systems with incompatible encodings.

# IO

IO objects have two special encodings:

- `#external_encoding` is the encoding of the input or output stream.
- `#internal_encoding` is the encoding you want to use internally.

## Sample

```
#!/ruby1.9 -wKU
File.open("test.txt", "r:ISO-8859-1") do |f|
  f.external_encoding #=> #<Encoding:ISO-8859-1>
  f.internal_encoding #=> #<Encoding:UTF-8>
  string = f.read
end

string.encoding #=> #<Encoding:UTF-8>
```

## Specifying encodings: Source encoding

String literals have the encoding of the source file they are in. Ruby assumes this to be US-ASCII unless otherwise specified by the user.

### UTF-8 script, incorrect

```
puts "Hello Wörld!"
```

### UTF-8 script, correct

```
#{- encoding=utf-8 -}  
puts "Hello Wörld!"
```

## Specifying encodings: Source encoding

String literals have the encoding of the source file they are in. Ruby assumes this to be US-ASCII unless otherwise specified by the user.

UTF-8 script, incorrect

```
puts "Hello Wörld!"
```

UTF-8 script, correct

```
#{- encoding=utf-8 -}  
puts "Hello Wörld!"
```

Think about putting special strings in a localization file instead.

# Specifying encodings: External Encoding

The encoding of streams can be set using the mode string:

## Setting the external encoding

```
File.open("test.txt", "r:US-ASCII") do |f|  
  f.read  
end
```

This tells Ruby that test.txt uses US-ASCII.

- File.read has no way to set an encoding and uses the default!
- Ruby 1.8.7 understands r:encoding, 1.8.6 does not

# Specifying encodings: External Encoding

The encoding of streams can be set using the mode string:

## Setting the external encoding

```
File.open("test.txt", "r:US-ASCII") do |f|  
  f.read  
end
```

This tells Ruby that test.txt uses US-ASCII.

- File.read has no way to set an encoding and uses the default!
- Ruby 1.8.7 understands r:encoding, 1.8.6 does not

# Default encodings

Internal and external Encodings can be set to defaults using `Encoding.default_internal =` and `Encoding.default_external =`.

## Setting the external encoding

```
Encoding.default_internal #=> nil  
Encoding.default_external = Encoding.find("UTF-8")
```

Note: This emits a warning if `default_internal` is already set.

# Command line flags

The cleanest way to set default encodings is the command line.

## Commandline

`-Eex[:in]` default\_external and default\_internal

`-U` shorthand for `-Eutf8:utf8`

# Rough Waters

Working with Rubys Encoding-System is tedious, boring und error-prone.

Working with Encodings is tedious, boring und error-prone.

# Bytestrings

What if you need to use Strings as byte strings?

## Binary Encoding: ASCII-8BIT

```
Encoding.find("binary")  
#=> #<Encoding:ASCII-8BIT>
```

## Properties of ASCII-8BIT

- No undefined characters, no errors
- Basically behaves like Ruby 1.8 String

# Fringe problems

There are two cases where handling binary is a must have.

- the IO streams encoding is not known beforehand and must be determined
- reading input bitwise

```
File.open("Test.txt", "r:utf-8") {|f| f.read 16}.encoding  
#=> #<Encoding:ASCII-8BIT>
```

# String#force\_encoding

If you are absolutely sure that the string has a different encoding than it is flagged with, you can change it without conversion.

## API

```
s.encoding #=> #<Encoding:ASCII-8BIT>  
s.force_encoding("utf-8")
```

## Warnings

- think twice before doing this!
- source of hard to trace bugs if you expectation fails
- not a quick-fix! You tell the interpreter you know better, so be sure that you are right in any case!

# String#force\_encoding

If you are absolutely sure that the string has a different encoding than it is flagged with, you can change it without conversion.

## API

```
s.encoding #=> #<Encoding:ASCII-8BIT>  
s.force_encoding("utf-8")
```

## Warnings

- think twice before doing this!
- source of hard to trace bugs if you expectation fails
- not a quick-fix! You tell the interpreter you know better, so be sure that you are right in any case!

# The three steps guide to complex IO in Ruby 1.9

## Read input

```
s = iostream.read  
enc = determine_encoding(s) # here be unicorns
```

Force encoding to the encoding the input is provided in

```
s.force_encoding(enc)
```

Encode to internal\_encoding

```
if Encoding.default_internal  
  s.encode!(Encoding.default_internal)  
end
```

# The three steps guide to complex IO in Ruby 1.9

## Read input

```
s = iostream.read  
enc = determine_encoding(s) # here be unicorns
```

## Force encoding to the encoding the input is provided in

```
s.force_encoding(enc)
```

## Encode to internal\_encoding

```
if Encoding.default_internal  
  s.encode!(Encoding.default_internal)  
end
```

# The three steps guide to complex IO in Ruby 1.9

## Read input

```
s = iostream.read  
enc = determine_encoding(s) # here be unicorns
```

## Force encoding to the encoding the input is provided in

```
s.force_encoding(enc)
```

## Encode to internal\_encoding

```
if Encoding.default_internal  
  s.encode!(Encoding.default_internal)  
end
```

## String.encode (2)

*#encode* is also able to do error handling:

### Sample

```
"äü".encode("US-ASCII", :invalid => :replace,  
              :undef => :replace,  
              :replace => "?") #=> "??"
```

```
"äü".encode("US-ASCII", :xml => :text) #=> "&#xE4;&#xFC;"
```

*#encode* has some additional options, for example conversion of line endings. See `ri`!

# Usage for Europeans

## Quick Tips for users

- `RUBYOPTS="-wKU"` gives you the best Unicode support for 1.8 and 1.9
- Check IO-Libraries whether they return sane strings...
- ...if not, complain to the maintainer

# Quick Tips for library developers

- Use US-ASCII as file encoding. Offload other stuff to localization files.
- Expect UTF-8 and compatible encodings as input. Provide output accordingly.
- IO-Librarys should be able to read UTF-8, ISO-\* and Windows-Encodings (test for it!)
- Documentation is king!
- Consider failing if `#default_internal` does not match your expectations

# Encoding conversion for Ruby 1.8 and 1.9

Ruby 1.9 still ships with ICONV.

## Beispiel

```
Iconv.iconv("ISO-8859-1", "UTF-8", my_string)
```

# Pitfalls

# Your display lies

How do you encode español in Unicode?

Version 1

```
s1 = ["e", "s", "p", "a", "ñ", "o", "l"]
```

# Your display lies

How do you encode español in Unicode?

## Version 1

```
s1 = ["e", "s", "p", "a", "ñ", "o", "l"]
```

## Version 2

```
s2 = ["e", "s", "p", "a", "~", "n", "o", "l"]  
s1 == s2 #=> false
```

Remember: español might not be the same as español.

# Other common problems

Other common problems often wrongly considered bugs in Ruby:

- wrong console encoding, especially on Windows consoles
- inability to guess source encoding (not possible)
- source is already broken

# Thank you!

github <http://github.com/skade>

twitter Argorak

smtp [florian.gilcher@asquera.de](mailto:florian.gilcher@asquera.de)

RL Florian Gilcher