The Coral Language Specification

Version 0.1

Kateřina Nikola Lisová

April 11, 2014

Prague, Czech Republic

Contents

Lexical Syntax 6

1.1 Identifiers 6

1.2 Keywords 7

1.3 Newline Characters 8

1.4 Operators 8

1.5 Literals 9

1.5.1 Integer Literals 10

1.5.2 Floating Point Literals 12

1.5.3 Imaginary Number Literals 13

1.5.4 Units of Measure 13

1.5.5 Boolean Literals 13

1.5.6 String Literals 14

1.5.7 Symbol Literals 14

1.5.8 Type Parameters 14

1.5.9 Regular Expression Literals 14

1.5.10 Collection Literals 15

1.6 Whitespace & Comments 16

1.7 Preprocessor Macros 17

Identifiers, Names & Scopes 17

Types 18

3.1 Paths 19

3.2 Value Types 20

3.2.1 Value Type 20

3.2.2 Type Projection 20

3.2.3 Type Designators 20

3.2.4 Parameterized Types 21

3.2.5 Tuple Types 21

3.2.6 Annotated Types 21

3.2.7 Compound Types 21

3.2.8 Function Types 21

3.2.9 Existential Types 22

3.3 Non-Value Types 22

3.3.1 Method Types 22

3.3.2 Polymorphic Method Types 22

3.3.3 Type Constructors 23

3.4 Relations Between Types 23

3.4.1 Type Equivalence 23

3.4.2 Conformance 23

Basic Declarations & Definitions 24

4.1 Variable Declarations & Definitions 24

4.1.1 Property Declarations & Definitions 24

4.1.2 Instance Variable Definitions 24

4.1.3 Type Declarations & Aliases 24

4.1.4 Type Parameters 24

4.1.5 Variance of Type Parameters 24

4.1.6 Function Declarations & Definitions 24

Preface

Coral is a Ruby-like programming language which enhances advanced object-oriented programming with elements of functional programming. Every value is an object, in this sense it is a pure object-oriented language. Object blueprints are described by classes. Classes can be composed in multiple ways – classic inheritance, mixin composition, union and fusion types.

Coral is also a functional language in the sense that every function is also an object. Therefore, function definitions can be nested and higher-order functions are supported out-of-the-box. Coral also has a limited support for pattern matching, which can emulate the algebraic types used in other functional languages.

Coral has been developed from 2012 in a home environment out of pure enthusiasm for programming and out of a desire for a truly versatile language. This document is a work in progress and will stay that way forever. It acts as a reference for the language definition and some core library classes.

# 

Lexical Syntax

Coral programs are written using the Unicode character set; Unicode supplementary characters are supported as well. Coral programs are preferably encoded with the UTF-8 character encoding. While every Unicode character is supported, usage of Unicode escapes is encouraged, since fonts that IDEs might use may not support the full Unicode character set.

## Identifiers

**Syntax:**

simple\_id ::= lower [ id\_rest ]

variable\_id ::= simple\_id | '\_'

constant\_id ::= upper [ id\_rest ]

function\_id ::= simple\_id [ id\_rest\_ext ]

id\_rest ::= { letter | digit | '\_' }

id\_rest\_mid ::= id\_rest [ ( '/' | '+' | '-' ) id\_rest ]

id\_rest\_ext ::= id\_rest [ id\_rest\_mid ] [ '?' | '!' ]

There are three kinds of identifiers.

First, *variable identifiers*, which are simply a lower-case letter followed by arbitrary sequence of letters (any-case), digits and underscores, or just one underscore (which has special meaning).

Second, *constant identifiers*, which are just like variable identifiers, but starting with an upper-case letter and never just an underscore.

And third, *function identifiers*, which are the most complicated ones. They can start as a variable identifier, then optionally followed by one of “/”, “+” and “-”, and then optionally ended with “?” or “!”.

Coral programs are parsed greedily, so that a longest match rule applies. Letters from the syntax may be any Unicode letters, but English alphabet letters are recommended, along with English names.

## Keywords

A set of identifiers is reserved for language features instead of for user identifiers. However, unlike in most other languages, keywords are not being recognized inside of paths, except for a few specific cases.

The following names are the reserved words.

alias

annotation

as

begin

bitfield

break

case

cast

catch

class

clone

constant

constructor

declare

def

destructor

do

else

elsif

end

ensure

enum

for

function

fusion

goto

if

implements

in

include

interface

is

let

loop

match

memoize

message

method

mixin

module

native

next

nil

no

of

opaque

operator

out

property

protocol

raise

range

record

redo

refine

rescue

retry

return

self

skip

struct

super

template

test

then

this

throw

throws

transparent

type

undef

unless

until

union

use

val

var

void

yes

with

when

while

yield

Not every reserved word is a keyword in every context, this behavior will be further explained. For example, the bitfield reserved word is only recognized as a keyword inside an enumeration definition context, in a specific place. Every reserved word may be used as a function identifier, with a little work-around when used with an implicit receiver.

## Newline Characters

Syntax:

semi ::= nl { nl } | ';'

Coral is a line-oriented language, in which statements are expressions and may be terminated by newlines, as well as by semi-colon operator. A newline in a Coral source file is treated as the special separator token nl if the following criterion is satisfied:

1. The token immediately preceding the newline can terminate an expression.

Since Coral may be interpreted in a REPL fashion, there are no other suitable criteria. Such a token that can terminate an expression is, for instance, not a binary operator or a message sending operator, which both require further tokens to create an expression. Keywords that expect any following tokens also can not terminate expressions. Coral interpreters and compilers do not look-ahead beyond newlines.

If the token immediately preceding the newline can not terminate an expression and is followed by more than one newline, Coral still sees that as only a one significant newline, to prevent any confusion.

Keywords that can terminate an expression are: break, end, opaque, native, next, nil, no, redo, retry, return, self, skip, super, this, transparent, void, yes, yield.

## Operators

A set of identifiers is reserved for language features, some of which may be overridden by user space implementations. Operators have language-defined precedence rules that usually comply to user expectations, and may be easily restructured by putting expressions inside of parenthesis pairs.

The following character sequences are the operators recognized by Coral.

:=

+=

-=

\*=

\*\*=

/=

%=

||=

&&=

^^=

|=

&=

^=

~=

<<

>>

<<<

>>>

;

=

!=

==

!==

===

!===

=~

!~

<>

<

>

<=

>=

<=>

+

-

\*

\*\*

/

div

%

mod

||

or

&&

and

!

not

^^

xor

|

&

^

~

..

...

,

->

<-

~>

<~

<~>

=>

::

:

<:

:>

<<|

|>>

<|

|>

(

)

[

]

{

}

.

Some of these operators have several different meanings, usually up to two. Some are binary, some are unary, none is ternary.

## Literals

There are literals for numbers (including integer, floating point and complex), characters, booleans, strings, symbols, regular expressions and collections (including tuples, lists, dictionaries and bags).

**Syntax:**

literal ::= integer\_literal

| floating\_point\_literal

| complex\_literal

| character\_literal

| string\_literal

| symbol\_literal

| regular\_expression\_literal

| collection\_literal

| '**nil**'

### Integer Literals

**Syntax:**

integer\_literal ::= ['+' | '-'] (decimal\_numeral

| hexadecimal\_numeral

| octal\_numeral

| binary\_numeral)

decimal\_numeral ::= '0' | non\_zero\_digit {['\_'] digit}

hexadecimal\_numeral ::= '0x' hex\_digit {['\_'] hex\_digit}

digit ::= '0' | non\_zero\_digit

non\_zero\_digit ::= '1' | … | '9'

hex\_digit ::= '1' | … | '9' | 'a' | … | 'f'

octal\_numeral ::= '0' oct\_digit {['\_'] oct\_digit}

oct\_digit ::= '0' | … | '7'

binary\_numeral ::= '0b' bin\_digit {['\_'] bin\_digit}

bin\_digit ::= '0' | '1'

Integers are usually of type Number, which is a class cluster of all classes that can hold numbers. Unlike Java, Coral supports both signed and unsigned integers. Usually integer literals that are obviously unsigned integers result in being represented internally by a class that stores the integer unsigned, something like Integer\_64\_Unsigned. Math operations on numbers are handled internally in such a way that the user does not need to care about the actual types of the numbers – when an integer overflow would occur, the result is stored in a larger container type, and when the largest container type would be overflown, a decimal type is used automatically.

Underscores used in integer literals have no special meaning but to improve readability of larger literals, i.e., to separate thousands.

Integral members of the Number class cluster include the following container types.

1. Integer\_8 (-27 to 27-1)
2. Integer\_8\_Unsigned (0 to 28)
3. Integer\_16 (-215 to 215-1)
4. Integer\_16\_Unsigned (0 to 216)
5. Integer\_32 (-231 to 231-1)
6. Integer\_32\_Unsigned (0 to 232)
7. Integer\_64 (-263 to 263-1)
8. Integer\_64\_Unsigned (0 to 264)
9. Integer\_128 (-2127 to 2127-1)
10. Integer\_128\_Unsigned (0 to 2128)
11. Decimal (-∞ to ∞)
12. Decimal\_Unsigned (0 to ∞)

The special Decimal and Decimal\_Unsigned container types are also for storing arbitrary precision floating point numbers. All the container types are constants defined in the Number class. (Number::*Container\_Type*)

Aliases for these classes exist as follows:

1. Integer\_8 = Byte
2. Integer\_8\_Unsigned = Byte\_Unsigned
3. Integer\_16 = Short
4. Integer\_16\_Unsigned = Short\_Unsigned
5. Integer\_64 = Long
6. Integer\_64\_Unsigned = Long\_Unsigned
7. Integer\_128 = Double\_Long
8. Integer\_128\_Unsigned = Double\_Long\_Unsigned

Moreover, a helper type Number::Unsigned exists, which can be used for type casting in cases where an originally signed number needs to be treated as unsigned.

These inner classes of Number also have one important feature: if the actual number fits into the range of possible values of another inner class, then it will also pass when performing a type check on that other class.

Notice that some host platforms may not provide a native implementation of 128 bit values, thus in such a case, the class actually becomes an alias of Decimal or Decimal\_Unsigned respectively, or just has a different internal implementation. Users should not rely on these internal classes and rather use range types when constraining value ranges. As helper types for the range types, Number::Integer and Number::Integer\_Unsigned exist, to allow easy constraining of the range types to integral numbers while being machine-size-independent.

Also notice that Coral does not have a Char or Character number literal, since that is not considered to be a fixed-size number by Coral, with respect to Unicode standards being of variable byte length (i.e., UTF-8 characters are usually between 1 to 4 bytes long). This neatly solves another Java issue with their Characters being in UTF-16 with higher and lower surrogates being split into two successive Character instances (or primitive values!).

### Floating Point Literals

**Syntax:**

float\_literal ::= digit {['\_'] digit} '.' digit {['\_'] digit}

[exponent\_part] [float\_type]

| digit {['\_'] digit} exponent\_part [float\_type]

| digit {['\_'] digit} [exponent\_part] float\_type

exponent\_part ::= 'e' ['+' | '-'] digit {['\_'] digit}

float\_type ::= 'f' | 'd'

Floating point literals are of type Number as well as integer literals, and have fewer container types. Compiler infers the precision automatically, unless the float\_type part is present.

1. Float\_32 (IEEE 754 32-bit precision)
2. Float\_64 (IEEE 754 64-bit precision)
3. Decimal (-∞ to ∞)
4. Decimal\_Unsigned (0 to ∞)

The aliases for these classes exist as follows:

1. Float = Float\_32
2. Double = Float\_64

The letters in exponent type and float type literals have to be lower-case in Coral sources, but functions for parsing of floating point numbers indeed support them being upper-case for compatibility reasons.

### Imaginary Number Literals

**Syntax:**

imaginary\_literal ::= real\_number\_literal 'i'

complex\_literal ::= (real\_number\_literal ('+' | '-') complex\_literal)

| (complex\_literal ('+' | '-') real\_number\_literal)

real\_number\_literal ::= (integer\_literal | float\_literal)

number\_literal ::= real\_number\_literal

| imaginary\_literal

| complex\_literal

**Examples:**

0i

5i

5+1i

1-5.2i

0.5f+1.5fi

3.14159i

Coral supports imaginary number literals along with real number literals, by simply putting an extra “i” after the real number literal.

### Units of Measure

Coral implements an addition to number handling, called *units of measure*. Number instances can be annotated with a unit of measure to ensure correctness of arithmetic operations.

**Syntax:**

annotated\_number ::= number\_literal '[<' Units\_of\_Measure '>]'

TBD: add link to details about units of measure, how to declare them and how to use them

### Boolean Literals

**Syntax:**

boolean\_literal ::= 'yes' | 'no'

Both literals are members of type Boolean. The no literal has also a special behavior when being compared to nil: it equals to nil, while not actually being nil. Identity equality is indeed different.

### String Literals

**Syntax:**

string\_literal ::= simple\_string\_literal | interpolable\_string\_literal

simple\_string\_literal ::= ''' { string\_element } '''

string\_element ::= printable\_char | char\_escape\_seq

interpolable\_string\_literal ::= '"' { int\_string\_element } '"'

int\_string\_element ::= string\_element | interpolated\_expression

interpolated\_expression ::= '#{' expression '}'

String literals are members of the type String. Single quotes in simple string literals have to be escaped (\') and double quotes in interpolable string literals have to be escaped (\").

### Symbol Literals

**Syntax:**

symbol\_literal ::= simple\_symbol | quoted\_symbol

simple\_symbol ::= ':' simple\_id

quoted\_symbol ::= simple\_quoted\_symbol | interpolable\_quoted\_symbol

simple\_quoted\_symbol ::= ':'' { string\_element } ) '''

interpolable\_quoted\_symbol ::= ':"' { int\_string\_element } '"'

Symbol literals are members of the type Symbol.

### Type Parameters

**Syntax:**

type\_param ::= '$' (variable\_id | constant\_id)

### Regular Expression Literals

**Syntax:**

regexp\_literal ::= '%/' regexp\_content '/' [ regexp\_flags ]

| '%r/' regexp\_content '/' [ regexp\_flags ]

| '%r(' regexp\_content ')' [ regexp\_flags ]

| '%r~' regexp\_content '~' [ regexp\_flags ]

| '%r<' regexp\_content '>' [ regexp\_flags ]

| '%r[' regexp\_content ']' [ regexp\_flags ]

| '%r{' regexp\_content '}' [ regexp\_flags ]

regexp\_content ::= regexp\_element { regexp\_element }

regexp\_element ::= string\_element | interpolated\_expression

regexp\_flags ::= printable\_char { printable\_char }

Regular expression literals are members of the type Regular\_Expression with alias of Regexp.

### Collection Literals

**Syntax:**

collection\_literal ::= tuple\_literal

| list\_literal

| dictionary\_literal

| bag\_literal

tuple\_literal ::= '(' list\_expressions ')'

list\_literal ::= '%' collection\_flags '[' list\_expressions ']'

dictionary\_literal ::= '%' collection\_flags '{' dictionary\_expressions '}'

bag\_literal ::= '%' collection\_flags '(' list\_expressions ')'

list\_expressions ::= expression { ',' expression }

dictionary\_expressions ::= dict\_expression { ',' dict\_expression }

dict\_expression ::= expression ' => ' expression

| simple\_id ': ' expression

collection\_flags ::= printable\_char { printable\_char }

Tuple literals are members of the Tuple type family. List literals are members of the List type, usually Array\_List, but the collection flags may change the resulting class. Dictionary literals are members of the Dictionary type with alias of Map, usually Hash\_Dictionary with alias of Hash\_Map, but the collection flags may again change the resulting class. Bag literals are members of the Bag type, usually Hash\_Bag, but the collection flags may yet again change the resulting class. Order of each flag is not significant.

List literal collection flags are the following:

1. Flag i = immutable, makes the resulting list frozen.
2. Flag l = linked, makes the resulting list a member of Linked\_List instead.
3. Flag w = words, the following expressions are treated as words, converted to strings separated by whitespace.

Dictionary literal collection flags are the following:

1. Flag i = immutable, makes the resulting dictionary frozen.
2. Flag l = linked, makes the resulting dictionary a member of Linked\_Hash\_Dictionary.

Bag literal collection flags are the following:

1. Flag i = immutable, makes the resulting bag frozen.
2. Flag s = set, the resulting collection is a set instead of a bag (a special bag that does support multiple item occurences, and tally of each item is always 0 or 1). The resulting literal is a member of Hash\_Set.
3. Flag l = linked, the resulting collection is linked, so either a member of Linked\_Hash\_Bag or Linked\_Hash\_Set.

Linked collections have a predictable iteration order in case of bags and dictionaries, or are simply stored differently in case of lists.

## Whitespace & Comments

Tokens may be separated by whitespace characters and/or comments. Comments come in two forms:

A single line comment is a sequence of characters that starts with // and extends to the end of the line.

A multi-line comment is a sequence of characters between /\* and \*/. Multi-line comments may not be nested.

Documentation comments are multi-line comments that start with a /\*!, and multi-line comments that start with a /\*\* are also recognized as documentation comments, while the former is preferred.

## Preprocessor Macros

TBD

Identifiers, Names & Scopes

Names in Coral identify various types, values, methods and constants, which are the *entities*. Names are introduced by local definitions and declarations, inheritance, use clauses or module clauses, which are the *bindings*.

Bindings of different kinds have a different precedence defined on them:

1. Definitions and declarations that are local have the highest precedence.
2. Inherited definitions and declarations have the next highest precedence.
3. Definitions and declarations made available by module clause have the next highest precedence.
4. Explicit **use** imports have the next highest precedence.
5. Definitions and declarations that are not in the same compilation unit (a different script or a different module) have the next highest precedence.
6. Definitions and declarations that are not bound have the lowest precedence. This happens when the binding simply can’t be found anywhere, and probably will result in a name error, while being inferred to be of type Object.

There is only one root name space, in which a single fully qualified binding designates always up to one entity.

Every binding has a *scope* in which the bound entity can be referenced using a simple name (unqualified). Scopes are nested, inner scopes inherit the same bindings, unless shadowed. A binding in an inner scope *shadows* bindings of lower precedence in the same scope (and nested scopes) as well as bindings of the same or lower precedence in outer scopes. Shadowing is a partial order, and bindings can become ambiguous – fully qualified names can be used to resolve binding conflicts.

A reference to an unqualified identifier *x* is bound by a unique binding, which

1. defines an entity with name *x* in the same name scope as the identifier *x*, and
2. shadows all other bindings that define entities with name *x* in that name scope.

It is syntactically not an error if no such binding exists, thanks to the dynamic features of the language (unbound references are implicitly bound to the same scope and are resolved by dynamic method callbacks). The same applies to fully qualified bindings that don’t resolve into any entity. However, it is an error if a binding is ambiguous.

If *x* is bound by explicit **use** import clause, then the simple name *x* is considered to be equivalent to the fully qualified name to which *x* is mapped by the import clause. If x is bound by a definition or declaration, then *x* refers to the entity introduced by that binding, thus the type of *x* is the type of the referenced entity.

Types

When we say *type* in the context of Coral, we are talking about a blueprint of an entity, while the type itself is an entity. Every type in Coral is backed by a *class*, which is an instance of the type Class.

We distinguish a few different properties of types in Coral. There are first-order types and type constructors, which take type parameters and yield new types. A subset of first-order types called *value types* represents set of first-class values. Value types are either *concrete* or *abstract*.

Concrete value types can be either a *class type* (i.e. referenced with a type designator, referencing a class or a mixin), or a *compound type* representing an intersection of types, possibly with a refinement that further constrains the types of its members. Both class types and compound types may be bound to a constant, but only class types referencing a concrete class can be blueprints of values (objects). Compound types can only constrain bindings to a subset of other types.

Non-value types capture properties of identifiers that are not values. For instance, a type constructor does not directly specify a type of values, but a type constructor, when applied to the correct type arguments, yields a first-order type, which may be a value type. Non-value types are expressed indirectly in Coral. In example, a method type is described by writing down a method signature, which is not a real type itself, but it creates a corresponding method type.

To sum it up, type system in Coral is a way to describe sets of values (objects, instances of concrete classes – class types). Types can be applied to method arguments, return values, variables, as a way of binding them to a particular set of possible values with predictable properties and behaviors. However, Coral is a dynamically typed language, and therefore types are type-checked in runtime rather than bound forever by compile time – Coral is able to perform only limited type checking while compiling, the worst it can do is to issue a warning when a type mismatch is apparent.

## Paths

Paths are Coral’s way of referencing types. A path represents a binding in a particular scope. If a binding can not be found while compiling a program, then the path is bound to the first enclosing type.

**Syntax:**

Path ::= ['::'] Const\_Path ['::' Self\_Path]

| Self\_Path

Self\_Path ::= ('**this**' | '**self**' | '**super**' [Class\_Qualifier]) [Const\_Path]

Class\_Qualifier ::= '<' Const\_Path '>'

Const\_Path ::= constant\_id {'::' constant\_id}

Note that the :: token is an operator used while dynamically referencing not only types, but any properties or methods that happen to be found along the way through the path.

The this keyword refers to the same class that is directly enclosing its occurence. The self keyword refers to the actual class that the value is a member of. The super keyword refers to the direct superclass of the value, or the superclass designated by the class qualifier. If there is no constant path before the self path, then the directly enclosing class is implicitly a part of the constant path. A path that is prefixed with an extra :: operator is a stable root path, and therefore not bound to the scope.

## Value Types

Value types are entities, for which we can tell if a value is of that type or not.

### Value Type

To retrieve the actual type of a value, the method .class can be used. This method is defined on every object and its return type is covariant – it is always a Class (or a subtype, if any exists). Type projection to “**class**” can be used as a stable way to reference the real class, of any expression that has a value, referred to as “Value\_Expr” in the following syntax.

**Syntax:**

Simple\_Type ::= Path '.' 'class'

Simple\_Type ::= Simple\_Type '##' '**class**'

Simple\_Type ::= Value\_Expr '##' '**class**'

### Type Projection

**Syntax:**

Simple\_Type ::= Simple\_Type '##' constant\_id

A type projection *T##x* references type member named *x* of type *T*.

### Type Designators

A type designator refers to a named value type (constant) and can be qualified or unqualified. Every type designator is a shorthand for a type projection.

Every unqualified type designator *t* with a stable binding in scope of a type *C* is a shorthand for the type projection *C::this.class##t*. Unqualified type designators are bound to the same scope in which they appear: *C::this.class##t*.

### Parameterized Types

**Syntax:**

Simple\_Type ::= Simple\_Type Type\_Args

Type\_Args ::= '<' Types '>'

### Tuple Types

**Syntax:**

Simple\_Type ::= '(' Types ')'

### Annotated Types

**Syntax:**

Annotated\_Type ::= {Annotation} Simple\_Type

### Compound Types

Compound types can be used in two ways: to further describe values which possess behaviors of multiple types (up to one class, unlimited number of protocols and mixins and an optional refinement constraint), or to create types ad-hoc (if no class is given, Object is implied).

**Syntax:**

Compound\_Type ::= Annotated\_Type {'**with**' Annotated\_Type} [Refinement]

| Refinement

Refinement ::= '**refine**' '{' Refine\_Expr {semi Refine\_Expr} '}'

### Function Types

**Syntax:**

Function\_Type ::= Args\_List {'->' Args\_List} '->' Return\_Type

Args\_List ::= '(' [Arg\_Type {',' Arg\_Type}] ')'

Return\_Type ::= (Type | '**void**')

Multiple arguments lists declare function types that are (automatically) curried, so this way a value can be declared as a function that returns a function.

### Existential Types

**Syntax:**

Type ::= Compound\_Type [Existential\_Clauses]

| Function\_Type {\* for clarity \*}

Existential\_Clauses ::= '**for-some**' '{' Existential\_Decl

{semi Existential\_Decl} '}'

## Non-Value Types

The types described as non-value types do not represent sets of values, nor do they appear explicitly in programs, only implicitly: you can’t write them in your programs, they are the internal types of defined identifiers.

### Method Types

This type represents methods and functions that take (named) arguments and return a result of a type.

**Examples:**

Function declarations:

1. **def** a -> Integer
2. **def** b (x : Integer) -> Boolean
3. **def** c (x : Integer) -> (y : String, z : String) -> String
4. **def** d (:x : Integer) -> Integer
5. **def** e (\*x : Integer) -> Integer
6. {|x : Integer| -> Boolean }

… produce the typings:

1. :a () Integer
2. :b (Integer) Boolean
3. :c (Integer) (String, String) String
4. :d (:x Integer) Integer
5. :e (\*Integer) Integer
6. (Integer) Boolean

### Polymorphic Method Types

TBD

### Type Constructors

TBD

## Relations Between Types

There are two useful relations between types.

* *Type equivalence*: *T* ≡ *U*, thus *T* and *U* are interchangeable in all contexts.
* *Conformance*: *T* <: *U*, thus type *T* conforms to type *U*.

### Type Equivalence

### Conformance

The conformance relation (<:) is the smallest transitive relation that satisfies the following conditions.

* If *T* is equivalent to *U*, then *T* conforms *U*. If *T* ≡ *U*, then *T* <: *U* and *U* <: *T*.
* For every value type T, the following is true: ::Nothing <: T <: ::Any, where ::Nothing is the class of nil and has the special property that it conforms to every value type, and ::Any is a type, such that $A **for-some** { **type** $A } exists (which is only ::Object).

Basic Declarations & Definitions

## Variable Declarations & Definitions

### Property Declarations & Definitions

### Instance Variable Definitions

### Type Declarations & Aliases

### Type Parameters

### Variance of Type Parameters

### Function Declarations & Definitions