

## basics

indents

```
block statement
→ | this = is + a * long + wrapped(
→ | → | → | statement)
single tab (or 4 spaces) indicates block indent
two tabs (or 8 spaces) indicates wrapped line
```

comments

```
// line comment
/* block comment */
```

## primitive types & literals

Bool	Int	Float	String
true or false	64bit integer	64bit double precision	utf-8 string
true	123 -234	12.3 -0.234	"Text" '日本語' 'α'
false	0x1234_5678	3.141_592_653_589_79	r"\\x4d+(foo[a-z]*?)"
	0b01_001001_01001	1.3e+10	R<<<[LF]Raw String...[LF]>>>

## scope and imports

script root or class

```
var name [:initial_value]
def foo (parameter_list)
  name: initial_value
  for [iterator:] [loop_variables] in iterable
    name: initial_value
    an_expression_refers(name)
```

operator ":" allocates new variable onto current scope

4<sup>th</sup>: anonymous imported

3<sup>rd</sup>: class or package scope

2<sup>nd</sup>: outer scopes

1<sup>st</sup>: current scope

imports

top of script

```
import script_name [as scope_name]
import foo → Package scope in "foo" will be imported as anonymous scope
import foo as bar → Package scope in "foo" will be imported as scope explicitly named "bar"
```

## functions

definition

script root → package-global functions

class → class methods

any block → inner functions

```
def name (parameter_list) [as native]
  return [return_value(s)]
  ...
some_variable: 123
```

operator "&" can generate function references from any style of functions e.g. &func, &object.func, &inner\_func

inner functions only: function can refer variables defined in outer scope

parameter list

```
p := name [:type] [(constructor_arguments)]
name : default_value
```

Parameters with constructor arguments or default value can be omitted in calling time

primitive types (Bool, Int, Float, String) only

parameter\_list := [p [, p ...]] [args\_name...] [: p [, p ...]] [kwargs\_name...]

Array

Dict

parameters after semi-colon forces keyword arguments

arguments

```
{value | *value} [, values, ...]
```

has to Dict

has to Array

parameter list & arguments examples:

```
def foo()
def foo(a, b, c)
def foo(a, b, args...)
def foo(a, b, c)
def foo(a, b, c, kwargs...)
def foo(a, b, args..., c, kwargs...)
def foo(a: Int, b:String)
def foo(a, b: 'bar', c: SomeClass(1, 2, 3))
```

r = foo() or r = foo

r = foo(1, 'bar', true)

r = foo(1, 2, 3, '4', 5.0, false, key: 7)

r = foo(1, c: 'paramC', b: 'paramB')

r = foo(1, 2, c: 3, d: 4)

r = foo(1, 2, 3, 4, d: 5, c: 6)

r = foo(1, 'bar')

r = foo(1) or r = foo(1, b: 'hoge') etc.

operator override

script root

```
def operator (name [:type] op_suffix)
def operator (op_prefix name [:type])
def operator (name [:type] op_binary name [:type])
```

overridable operators:

```
++ -- !
+ = * & / - /- + % % - % - % + * + & + - & -
<< >> >>> & | ^ < > <= >= == !=
```

op=

examples:

```
def operator (a:SomeClass++)
def operator (b:SomeClass)
def operator (a:Class1 + b:Class2)
```

## classes

definition

script root

```
class name [extends super_class [, types, ...]]
1st ...
2nd def init [:variation] (parameter_list)
  var name [:initial_value]
  ...
def name (parameter_list) [as native]
def name (parameter_list) .set (r)
```

bare constructor

constructor [with variation specifier]

field definition

field assignment overload

class inheritance

```
class ... extends ... [:= def init ...]
```

called when instance.name(...) = r

Super

SubA

SubB

Class

1<sup>st</sup> call

2<sup>nd</sup> call

Only called once

Required to call all constructors on direct inherited classes

class SubA extends Super

```
def init (...)
super(...)
```

class Class extends SubA, SubB

```
def init (...)
super.SubA(...)
super.SubB.variation(...)
```

## loop-statements

for-loop

```
label_ext: for ...
continue label_ext
continue label
break break label
break label_ext
```

iterable.iterator() for single loop\_variables

iterable.keyedIterator() for two loop\_variables

[label:] for [iterator:] [loop\_variables] in iterable

continue [label]

break [label]

remove current element

iterator.remove()

iterator.hasNext()

if false

while-loop

```
[label:] while condition
...
[continue, break]
```

## conditional-branches

if-else

```
if condition1
...
else if condition2
...
else
...
```

any times

only once

if the object implements both .has() and .equals(), only .has() will be called

switch-case

```
switch target
case object_expression
case object_expression
case bool_expression
case scalar_expression
...
default
...
```

any order

optional should be last

implicit break

break

break

"scalar" means Int, Float, or String

if target == scalar\_expression

do

```
do
...
```

not a loop-statement

only constructs scope

## exceptions

any block (script root), for, while, if, else if, switch, case, default, do, def, class, catch, finally

```
throw exception_object
...
catch [ex_variable:] exception_type [, types, ...]
throw
throw another_exception
finally
...
```

throwing an exception

type match

any times

end of/text from block

optional

throwing an exception → abort execution

continue execution or scanning catch handler

continue scanning catch handler in upper frame

Pre-defined exception types (inherits from)			
Exception		RuntimeException	Exception
ParserErrorException	Exception	NullReferenceException	RuntimeException
UnsupportedOperationException	Exception	ArithmeticException	RuntimeException
PackageNotFoundException	Exception	OverflowException	ArithmeticException
ClassNotFoundException	Exception	DivideByZeroException	ArithmeticException
MemberNotFoundException	Exception	IllegalArgumentException	RuntimeException
IncompleteExpressionException	Exception	NativeException	Exception
TypeMismatchException	Exception		

## operators

note: precedence of operators are slightly different from C/C++

Gr	Notations	Operator name	Associativity	Gr	Notations	Operator name	Associativity
0	a . b	element selection	left to right	4	a << b	left shift	left to right
0	a ++	postfix increment	left to right	4	a >> b	right shift	left to right
0	a --	postfix decrement	left to right	4	a >>> b	right shift with zero extension	left to right
0	a ...	variadic arguments	left to right	5	a & b	bitwise AND	left to right
1	++ a	prefix increment	right to left	6	a   b	bitwise OR	left to right
1	-- a	prefix decrement	right to left	7	a ^ b	bitwise XOR	left to right
1	~ a	bitwise complement	right to left	8	a < b	less than	left to right
1	! a	logical NOT	right to left	8	a > b	greater than	left to right
1	not a	logical NOT	right to left	8	a <= b	less than or equal to	left to right
1	+ a	unary plus	right to left	8	a >= b	greater than or equal to	left to right
1	- a	unary minus	right to left	8	a is type	instance of	left to right
1	* a	tuple extraction	right to left	9	a == b	equal to	left to right
1	& a	function object	right to left	9	a != b	not equal to	left to right
1	async a	asynchronous evaluation	right to left	10	a && b	logical AND	left to right
2	a * b	multiplication	left to right	10	a and b	logical AND	left to right
2	a & * b	multiplication with ignoring overflow	left to right	11	a    b	logical OR	left to right
2	a / b	division	left to right	11	a or b	logical OR	left to right
2	a /- b	division with rounding down	left to right	12	a ^^ b	logical XOR	left to right
2	a /+ b	division with rounding up	left to right	12	a xor b	logical XOR	left to right
2	a % b	modulus	left to right	13	a ? b : c	conditional	right to left
2	a %- b	modulus with rounding down	left to right	14	a : b	pair / define new variable	right to left
2	a %+ b	modulus with rounding up	left to right	15	a , b	tuple	left to right
3	a ** b	exponent	left to right	15	a ; b	tuple delimiter	left to right
3	a + b	addition	left to right	16	a = b	assignment	right to left
3	a &+ b	addition with ignoring overflow	left to right	16	a op= b	binary operator and assignment	right to left
3	a - b	subtraction	left to right				
3	a &- b	subtraction with ignoring overflow	left to right				