.chatra CHEAT SHEET version 0.9.0

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 false	123 -234 0x1234_5678 0b01_001001_01001	12.3 -0.234 3.141_592_653_589_79 1.3e+10	"Text" '¥t日本語¥n' r"^¥d+(foo[a-z]*?)" R<<<[LF]Raw String...[LF]>>>

scope and imports

```
script root or class
    var name [:initial_value]                  ← 3rd : class or package scope
    def foo (parameter_list)
        name: initial_value                     ← 2nd : outer scopes
        for [iterator:] [loop_variables] in iterable
            name: initial_value                 ← 1st : current scope
            an_expression_refers(name)
```

4th : anonymous imported

operator ":" allocates new variable onto 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

operator "&" can generate function references from any style of functions
e.g. &func &object.func &inner_func

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

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

parameter list

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

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

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

```
parameter_list := [p [, p ...] ] [args_name...] [; p [, p ...] ] [kwargs_name...]
```

args_name... → Array

parameters after semi-colon forces keyword arguments

kwargs_name... → Dict

arguments

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

has toDict() (0, *a, b:4) → (0, x:1, y:2, z:3, b:4)

has toArray() (0, *a, 4) → (0, 1, 2, 3, 4)

parameter list & arguments examples:

```
def foo()                                 r = foo()  or  r = foo
def foo(a, b, c)                          r = foo(1, 'bar', true)
def foo(a, b, args...)                    r = foo(1, 2, 3, '4', 5.0, false, keyed: 7)
def foo(a, b; c)                          r = foo(1, c: 'paramC', b: 'paramB')
def foo(a, b; c, kwargs...)               r = foo(1, 2, c: 3, d: 4)
def foo(a, b, args...; c, kwargs...)      r = foo(1, 2, 3, 4, d: 5, c: 6)
def foo(a: Int, b:String)                 r = foo(1, 'bar')
def foo(a, b: 'bar', c: SomeClass(1, 2, 3))   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 (!a:SomeClass)
def operator (a:Class1 + b:Class2)
```

classes

definition

script root

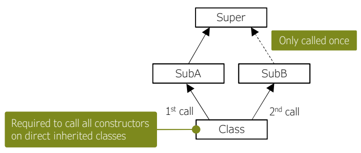
```
class name [extends super_class [, types, ...] ]
    1st  ...                                     bare constructor
    2nd  def init [.variation] (parameter_list)  constructor [with variation specifier]
         var name [:initial_value]               field definition

         def name (parameter_list) [as native]
         def name (parameter_list) .set (r)      field assignment overload
```

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

class inheritance

```
class ... extends ... [→| def init ...]
    super [.class_name] [.variation] (constructor_arguments)
```



Super
Only called once
SubA
SubB
1st call
2nd call
Class
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 ...
    → [label:] for [iterator:] [loop_variables] in iterable
            continue [label]
            break [label]
      → ...
→ ...
```

continue label_ext

continue | continue label

break | break label

break label_ext

iterable.iterator() for single loop_variables
iterable.keyedIterator() for two loop_variables

iterator.next()

iterator.hasNext() — if false

remove current element iterator.remove()

while-loop

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

conditional-branches

if-else

```
if condition1
        ...
else if condition2        any times
        ...
else                      only once
        ...
```

switch-case

```
switch target
        case object_expression      object_expression.has(target)     if true
        case object_expression      object_expression.equals(target)  if true
        case bool_expression        if true
        case scalar_expression      if target == scalar_expression
                ...                 [break]
            implicit break
        default                     optional, should be last
                ...                 [break]
...
```

any order

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

"scalar" means Int, Float, or String

do

not a loop-statement
only constructs scope

```
do
        ...
```

exceptions

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

```
throw exception_object
...                              throwing an exception
catch [ex_variable:] exception_type [, types, ...]    ← type match      any times
        throw
        throw another_exception
finally                                               optional
        ...                      throwing an exception → abort execution
```

end of/exit from block

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
2	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				