A Concise Reference of the Nit Language

This document attempts to be as short as possible while covering all features of the language in depth. It is not a real manual to learn the language since concepts are covered when required.

1 Basic Syntax of Nit

The syntax of Nit follows the Pascal tradition and is inspired by various script languages (especially Ruby). Its main objective is readability.

Indentation is not meaningful in Nit; blocks usually start with a specific keyword and finish with \texttt{end}. Newlines are only meaningful at the end of declarations, at the end of statements, and after some specific keywords. The philosophy is that the newline is ignored if something (a statement, a declaration, or whatever) obviously needs more input; while the newline terminates lines that seems completed. See the complete Nit grammar for more details.

Nit aims to achieve some uniformity in its usage of the common punctuation: equal (\texttt{=}) is for assignment, double equal (\texttt{==}) is for equality test, column (\texttt{::}) is for type declaration, dot (\texttt{.}) separates elements, and quad (\texttt{quad}) is for explicit designation.

1.1 Identifiers

Identifiers of modules, variables, methods, attributes and labels must begin with a lowercase letter and can be followed by letters, digits, or underscores. However, the usage of uppercase letters (and camelcase) is discouraged and the usage of underscores to vary the following is encouraged:

\begin{verbatim}
some_identifier

and some_virtual_type.
\end{verbatim}

1.2 Style

While Nit does not enforce any kind of source code formatting, the following is encouraged:

\begin{itemize}
\item indentation uses the tabulation character and is displayed as 8 spaces;
\item lines are less than 80 characters long;
\item binary operators have spaces around them: $a + b$, $x = 5$;
\item columns (\texttt{::}) and commas (,\texttt{,}) have a space after them but not before: \texttt{var x: [1, 2, 3]};
\item parenthesis and brackets do not need spaces around them;
\item superfluous parenthesis should be avoided;
\item the \texttt{do} of methods and the single \texttt{do} is on its own line and not indented;
\item the other \texttt{do} are not on a newline.
\end{itemize}

1.3 Comments and Documentation

As in many script languages, comments begin with a sharp (\texttt{#}) and run up to the end of the line. Currently, there is no multiline-comment.

A comment block right before any definition of module, class, or property, is considered as its documentation and will be displayed as such by the autodoc. At this point, documentation is displayed verbatim (no special formatting or meta-information).

2 Basic Types

2.1 Object

Nit is a full object language. Every value is an instance of a class. Even the basic types described in this section, \texttt{Object} is the root of the class hierarchy. All other classes, including the basic ones, are a specialization of \texttt{Object}.

Classes, methods and operators presented in this section are defined in the standard Nit library that is implicitly imported by every module. Many other classes and methods are also defined in the standard library. Please look at the specific standard library documentation for all details.

2.2 Int and Float

1, -1 are \texttt{Int} literals, and 1.0, -0.1 are \texttt{Float} literals. Standard arithmetic operators are available with a common precedence rules: \texttt{*, /, and \%} (modulo); then \texttt{+} and \texttt{-}. Some operators can be composed with the assignment (\texttt{=}).

\begin{verbatim}
var i = 5
i += 2
\end{verbatim}

Conversion from \texttt{Int} to \texttt{Float} and \texttt{Float} to \texttt{Int} must be done with the \texttt{to_f} and \texttt{to_i} methods.

2.3 String

Literal strings are enclosed within quotes (\texttt{"}). To insert a value inside a literal string, include the values inside braces (\texttt{\{\}}). Braces have to be escaped. \texttt{+} is the concatenation operator but is less efficient than the brace form.

\begin{verbatim}
var j = 5
print "j=\{\j\}; j+=\{\j+1\}" # outputs "j=5; j=6"
\end{verbatim}

Common escaping sequences are available (\texttt{\v, \n, \t, etc.}) plus the escaped brace \texttt{\v}.

\begin{verbatim}
print "hel\lo\wo\ril"
# outputs 'help world relief'
# and 'wo rld ' on a second line
\end{verbatim}

Multi-line strings are enclosed with triple quotes (\texttt{\"\"\"}). Values are inserted with a triple braces (\texttt{\{(\value\}\}}). The multi-line form thus allows verbatim new-lines, quotes and braces

\begin{verbatim}
print """\some text
with line breaks
and characters like " and {
but {{\{\1+2\}}} is rendered as 3
""
\end{verbatim}

All objects have a \texttt{to_s} method that converts the object to a \texttt{String}. \texttt{print} is a top-level method that takes any number of arguments and prints to the standard output. \texttt{print} always adds a newline, another top-level method, \texttt{print}, does not add the newline.

\begin{verbatim}
var x: String
x = 5.to_s # -> the String "5"
print x # outputs "5"
\end{verbatim}

2.4 Bool

\texttt{true} and \texttt{false} are the only two \texttt{Bool} values. Standard Boolean operators are available with the standard precedence rule: \texttt{not}; then \texttt{and}; then \texttt{or}.

Common comparison operators are available: \texttt{==} and \texttt{\!=} on all objects; <, >, <=, >= and \texttt{\!=} on \texttt{Comparable} objects (which include \texttt{Int}, \texttt{String} and others).

\begin{itemize}
\item \texttt{==} and \texttt{\!=} are standard Nit operators (so they are redefinable).
\item \texttt{and}, \texttt{or}, and \texttt{not} are not standard Nit operators: they are not redefinable, also they are lazy and have adaptive typing flow effects.
\item \texttt{==} is not for reference equality but for value equality (like \texttt{equals} in Java). There is a special reference equality operator, \texttt{is}, but it cannot be redefined and its usage is not recommended. Note also that while \texttt{==} is redefinable, it has a special adaptive typing flow effect when used with \texttt{null}.
\end{itemize}
2.5 Array

Array is a generic class, thus Array[Int] denotes an array of integers and Array[Array[Boolean]] denotes an array of array of Booleans. Literal arrays can be declared with the bracket notation ([ ]). Empty arrays can also be instantiated with the new keyword and elements added with the add method. Elements can be retrieved or stored with the bracket operator.

```
var a = [1, 2, 3, 4] # A literal array of integers
print a.join(";") # outputs "1;2;3;4"
```

Note that the type of literal arrays is deduced using the static type combination rule.

2.6 Range

Range is also a generic class but accepts only discrete types (Int is discrete). There are two kinds of literal ranges, the open one (1..5) that excludes the last element, and the closed one [1..5] that includes it.

```
print([1..5].join(";")) # outputs "1;2;3;4;5"
print([1..5].join(";")) # outputs "1:2:3:4:5"
```

Ranges are mainly used in for loops.

2.7 HashMap

HashMap is a generic class that associates keys with values. There is no literal hashmap, therefore the new keyword is used to create an empty HashMap and the bracket operators are used to store and retrieve values.

```
var h = new HashMap[String, Int] # h associates strings to integers
h["six"] = 6
print h["six"] + 1 # outputs "7"
```

3 Control Structures

Traditional procedural control structures exist in Nit. They also often exist in two versions: a one-liner and a block version.

3.1 Control Flow

Control structures dictate the control flow of the program. Nit heavily refers to the control flow in its specification:

- No unreachable statement;
- No usage of undefined variables;
- No function without a return with a value;
- Adaptive typing.

Some structures alter the control flow but are not described in this section: and, or, not, else and return.

Note that the control flow is determined only from the position, the order and the nesting of the control structures. The real value of the expressions used has no effect on the control flow analyses.

```
if true then
  return
else
  return
end
print 1 # Compile error: unreachable statement
```

Note that the following example is invalid since the first line is syntactically complete thus the newline terminate the whole if structure; then an error is signaled since a statement cannot begin with else.

```
if exp then stm
if exp then stm else stm
if exp then
  stm
else
  if exp then
    stm
  else
    if exp then
      stm
    else
      end
    end
else
  end
end
```

3.2 if

```
if exp then stm
if exp then stm else stm
if exp then
  stm
else
  if exp then
    stm
  else
    end
  end
else
  end
end
```

3.3 while

```
while exp do stm
while exp do stm
end
```

Note that the following is invalid:

```
while exp do
  stm
else
  exp
end
```

3.4 for

```
for x in [1..5] do print x # outputs 1 2 3 4 5
for x in [1..5] do print x # outputs 1 2 3 4 5
end
```

3.5 loop

```
loop
  stms
  if exp then break
  stms
end
```

Infinite loops are mainly used with breaks. They are useful to implement until loops or to simulate the exit when control of Ada.

Note that loop is different from while true because the control flow does not consider the values of expression.

3.6 do

```
do
  var x = 5
  print x
end
```

3.7 break, continue and label

Unlabeled break exits the current for, while, loop. Unlabeled continue skips the current for, while, loop.

label can be used with break or continue to act on a specific control structure (not necessary the current one). The corresponding label must be defined after the end keyword of the designated control structure.

```
for i in [0..width] do
  for j in [0..height] do
    if foo(i, j) then break label outer_loop # The 'break' breaks the 'for i' loop
  end
end label outer_loop
```

November 12, 2015
A Concise Reference of the Nit Language
2
label can also be used with break and single do structures.

do
  stmts
  if expr then break label block
end label block

3.8 abort

abort stops the program with a fatal error and prints a stack trace.
Since there is currently no exception nor run-time-errors, abort
is somewhat used to simulate them.

3.9 assert

assert verifies that a given Boolean expression is true, or else it
aborts. An optional label can be precised, it will be displayed
on the error message. An optional else can also be added and
will be executed before the abort.

assert bla: whatever else
  # "bla" is the label
  # "whatever" is the expression to verify
  print "Fatal error in module blablabla."
  print "Please contact the customer service."
end

4 Local Variables and Static Typing

var declares local variables. In fact there is no global variable in
Nit, so in this document variable always refers to a local variable.
A variable is visible up to the end of the current control structure.
Two variables with the same name cannot coexist: no nesting
nor masking.

Variables are bound to values. A variable cannot be used
unless it has a value in all control flow paths (À la Java).

4.1 Adaptive Typing

Nit features adaptive typing, which means that the static type of
a variable can change according to: the assignments of variables,
the control flow, and some special operators (and, or, or else, ==, !=, and isa).

var x = 5
# static type is Int
print x + 1 # outputs 6
z = [6, 7]
# static type is Array[Int]
print x[0] # outputs "6"

var x
  if whatever then
    x = 5
  else
    y = 6
end
# Static type is Int

4.2 Variable Upper Bound

An optional type information can be added to a variable declara-
tion. This type is used as an upper bound of the type of the
variable. When a initial value is given in a variable declaration
without a specific type information, the static type of the initial
value is used as an upper bound. If no type and no initial value
are given, the upper bound is set to nullable Object.

var x: Int # Upper bound is Int
x = "Hello" # Compile error: expected Int

var y: Object # Upper bound is Object
y = 5 # OK since Int specializes Object

var z: Int # Upper bound is Int
z = "Hello" # Compile error: expected Int

var t: Object # Upper bound is Object
t = "Hello" # OK

The adaptive typing flow is straightforward, therefore loops
(for, while, loop) have a special requirement: on entry, the upper
bound is set to the current static type; on exit, the upper bound
is reset to its previous value.

var x: Object = ...

# static type is Object, upper bound is Object
x = 5
# static type is Int, bound remains Object
while x > 0 do
  # static type remains Int, bound sets to Int
  x -= 1 # OK
  z = "Hello" # Compile error: expected Int
end
# static type is Int, bound reset to Object
x = "Hello" # OK

4.3 Type Checks

isa tests if an object is an instance of a given type. If the
expression used in an isa is a variable, then its static type is
automatically adapted, therefore avoiding the need of a specific cast.

var x: Object = whatever
if x isa Int then
  # static type of x is Int
  print x + 10 # OK
end

Remember that adaptive typing follows the control flow,
including the Boolean operators.

var a: Array[Object] = ...
for i in a do
  # the static type of i is Object
  if not i isa Int then continue
  # now the static type of i is Int
  print i + 10 # OK
end

An interesting example:

var max = null
for i in whatever do
  if i isa Int and i > max then max = i
  # the > is valid since, in the right part
  # of the "and", the static type of i is Int
end

Note that type adaptation occurs only in an isa if the target
type is more specific that the current type.

var a: Collection[Int] = ...
if a isa Comparable then
  # the static type is still Collection[Int]
  # even if the dynamic type of a is a subclass
  # of both Collection[Int] and Comparable
end ...

4.4 Nullable Types

null is a literal value that is only accepted by some specific
static types. However, thanks to adaptive typing, the static type
management can be mainly automatic.

nullable annotates types that can accept null or an expression
of a compatible nullable static type.

var x: nullable Int
var y: Int
  x = 1 # OK
  y = 1 # OK
  x = null # OK
  y = null # Compile error
  x = y # OK
  y = x # Compile error

Adaptive typing works well with nullable types.

var x
  if whatever then
    x = 5
  else
    x = null
end
# The static type of x is nullable Int

Moreover, like the isa keyword, the == and != operators can
adapt the static type of a variable when compared to null.

var x: nullable Int = whatever
if x != null then
  # The static type of x is Int (without nullable)
  print x + 6
end
# The static type of x is nullable Int

And another example:

```nit
var x: nullable Int = whatever
loop
  if x == null then continue
  # The static type of x is Int
end

or else can be used to compose a nullable expression with any other expression. The value of x or else y is x if x is not null and is y if x is null. The static type of x or else y is the combination of the type of y and the not null version of the type of x.

```nit
var i: nullable Int = ...
var j = i or else 0
# the static type of j is Int (without nullable)
```

Note that nullable types require a special management for attributes [?] and constructors [??].

### 4.5 Explicit Cast

as casts an expression to a type. The expression is either casted successfully or there is an abort.

```nit
var x: Object = 5 # static type of x is Object
print x.as(Int) * 10 # outputs 50
print x.as(String) # aborts: cast failed
```

Because of type adaptation, as is rarely used on variables. isa (sometimes coupled with assert) is preferred.

```nit
var x: Object = 5 # static type of x is Object
assert x isa Int
# static type of x is now Int
print x * 10 # outputs 50
```

as(not null) can be used to cast an expression typed by a nullable type to its non nullable version. This form keeps the programmer from writing explicit static types.

```nit
var x: nullable Int = 5 # static type of x is nullable Int
print x.as(not null) * 10 # cast, outputs 50
print x.as(Int) * 10 # same cast, outputs 50
assert x != null # same cast, but type of x is now Int
print x * 10 # outputs 50
```

4.6 Static Type Combination Rule

Adaptive typing, literal arrays, and or else need to determine a static type by combining other static types. This is done by using the following rule:

- The final type is nullable if at least one of the types is nullable.
- The final type is the static type that is more general than all the other types.
- If there is no such a type, and the thing typed is a variable, then the final type is the upper bound type of the variable; else there is a compilation error.

```nit
var d: Discrete = ...
# Note: Int < Discrete < Object
var x
if whatever then x = 1 else x = d
# static type is Discrete
if whatever then x = 1 else x = "i"
# static type is nullable Object (upper bound)
var a1 = [1, d] # a1 is an Array[Discrete]
var a2 = [1, "i"] # Compile error: # incompatible types Int and String
```

### 5 Modules

module declares the name of a module. While optional it is recommended to use it, at least for documentation purpose. The basename of the source file must match the name declared with module. The extension of the source file must be `nit`

A module is made of, in order:

- the module declaration;
- module importations;
- class definitions (and refinements);
- top-level function definitions (and redefinitions);
- main instructions.

### 5.1 Module Importation

import declares dependencies between modules. By default (that is without any import declaration), a module publicly imports the module standard. Dependencies must not produce cycles. By importing a module, the importer module can see and use classes and properties defined in the imported module.

- import indicates a public importation. Importers of a given module will also import its publicly imported modules. An analogy is using `#include` in a header file (.h) in C/C++.
- private import indicates a private importation. Importers of a given module will not automatically import its privately imported modules. An analogy is using `#include` in a body file (.c) in C/C++.
- intrude import indicates an intrusive importation. intrude import bypasses the private visibility and gives to the importer module a full access on the imported module. Such an import may only be considered when modules are strongly bounded and developed together. The closest, but insufficient, analogy is something like including a body file in a body file in C/C++.

### 5.2 Visibility

By default, all classes, methods, constructors and virtual types are public which means freely usable by any importer module. Once something is public it belongs to the API of the module and should not be changed.

- private indicates classes and methods that do not belong to the API. They are still freely usable inside the module but are invisible in other modules (except those that use intrude import).
- protected indicates restricted methods and constructors. Such methods belong to the API of the module but they can only be used with the self receiver. Basically, protected methods are limited to the current class and its subclasses. Note that inside the module (and in intrude importers), there is still no restriction.

Visibility of attributes is more specific and is detailed in its own section.
5.3 Visibility Coherence

In order to guarantee the coherence in the visibility, the following rules apply:

- Classes and properties privately imported are considered private: they are not exported and do not belong to the API of the importer.
- Properties defined in a private class are private.
- A static type is private if it contains a private class or a private virtual type.
- Signatures of public and protected properties cannot contain a private static type.
- Bounds of public generic class and public virtual types cannot contain a private static type.

6 Classes

interface, abstract class, class and enum are the four kinds of classes. All these classes can be in multiple inheritance, can define new whatever is the kind or the visibility of the class. The specialization between classes is transitive, therefore it is allowed to just write.

6.1 Class Specialization

super declares superclasses. Classes inherit methods, attributes and virtual-types defined in their superclasses. Currently, constructors are inherited in a specific manner. object is the root of the class hierarchy. It is an interface and all other kinds of classes are implicitly a subclass of object.

There is no repeated inheritance nor private inheritance. The specialization between classes is transitive, therefore super declarations are superfluous (thus ignored).

6.2 Class Refinement

redef allows modules to refine imported classes (even basic ones). Refining a class means:

- adding new properties: methods, attributes, constructors, virtual types;
- redefining existing properties: methods and constructors;
- adding new superclasses.

Note that the kind or the visibility of a class cannot be changed by a refinement. Therefore, it is allowed to just write redef class X whatever is the kind or the visibility of X.

In programs, the real instantiated classes are always the combination of all their refinements.

7 Methods

fun declares methods. Methods must have a name, may have parameters, and may have a return type. Parameters are typed; however, a single type can be used for multiple parameters.

fun foo(y, z: Int, s: String): Bool ...

do declares the body of methods. Alike control structures, a one-liner version is available. Therefore, the two following methods are equivalent.

fun next1(i: Int): Int
do return i + 1
end

fun next2(i: Int): Int do return i + 1
end

Inside the method body, parameters are considered as variables. They can be assigned and are subject to adaptive typing. self, the current receiver, is a special parameter. It is not assignable but is subject to adaptive typing.

return exits the method and returns to the caller. In a function, the return value must be provided with a return in all control flow paths.

7.1 Method Call

Calling a method is usually done with the dotted notation x.foo(y, z). The dotted notation can be chained.

A method call with no argument does not need parentheses. Moreover, even with arguments, the parentheses are not required in the principal method of a statement.

7.2 Method Redefinition

redef denotes methods that are redefined in subclasses or in class refinements. The number and the types of the parameters must be invariant. Thus, there is no need to reprecise the types of the parameters, only names are mandatory.

The return type can be redefined to be a more precise type. If same type is returned, there is no need to reprecise it.

The visibility, also, cannot be changed, thus there is also no need to reprecise it.

7.3 Abstract Methods

is abstract indicates methods defined without a body. Subclasses and refinements can then redefine it (the redef is still mandatory) with a proper body.

Concrete classes may have abstract methods. It is up to a refinement to provide a body.
7.4 Call to Super

super calls the “previous” definition of the method. It is used in a redefinition of a method in a subclass or in a refinement. It can be used with or without arguments; in the latter case, the original arguments are implicitly used.

The super of Nit behave more like the call-next-method of CLOS that the super of Java or Smalltalk. It permits the traversal of complex class hierarchies and refinement. Basically, super is polymorphic: the method called by super is not only determined by the class of definition of the method but also by the dynamic type of self.

The principle it to produce a strict order of the redefinitions of a method (the linearization). Each call to super call the next method definition in the linearization. From a technical point of view, the linearization algorithm used is based on C3. It ensures that:

- A definition comes after its redefinition.
- A redefinition in a refinement comes before a redefinition in its superclass.
- The order of the declaration of the superclasses is used as the ultimate disambiguation.

```nit
class A
  fun derp: String do return "A"
end

class B
  super A
  redef fun derp do return "B" + super
end

class C
  super A
  redef fun derp do return "C" + super
end

class D
  super B
  super C
  redef fun derp do return "D" + super
  # Here the linearization order of the class D is DBCA
  # B before C because D specializes B
  # B before A because C specializes B
  # C before A because C specializes A
  # B before C because in D 'super B' is before 'super C'
end

var b = new B
print b.derp # outputs "BA"
var d = new D
print d.derp # outputs "DBCA"
```

7.5 Operators and Setters

Operators and setters are methods that require a special syntax for their definition and their invocation.

- binary operators: +, -, *, /, %, =, \, >=, >, >=, <<, >> and <=. Their definitions require exactly one parameter and a return value. Their invocation is done with x + y where x is the receiver, + is the operator, and y is the argument.

- unary operator: -. Its definition requires a return value but no parameter. Its invocation is done with -x where x is the receiver.

- bracket operator: [>. Its definition requires one parameter or more and a return value. Its invocation is done with x[y, z] where x is the receiver, y the first argument and z the second argument.

- setters: something= where something can be any valid method identifier. Their definitions require one parameter or more and no return value. If there is only one parameter, the invocation is done with x.something = y where x is the receiver and y the argument. If there is more that one parameter, the invocation is done with x.something(y, z) = t where x is the receiver, y the first argument, z the second argument and t the last argument.

```nit
class Foo
  fun +=(a: Bar): Baz do ... end
  fun -=: Baz do ... end
  fun []=(a: Bar): Baz do ... end
  fun derp(a: Bar, b: Baz) do ...
  fun []=(a: Bar, b: Baz) do ...
end

var a: Foo = ...
var b: Bar = ...
var c: Baz = ...
c = a + b
c = -b
c = a[b] # The bracket operator '{/}
c = a.derp(b) # A normal method 'derp'
c = a. derp(b) = c # A setter 'derp'
c[b] = c # The bracket setter '{/}'
```

7.6 Variable Number of Arguments

A method can accept a variable number of arguments using ellipsis (...). The definition use x: Foo... where x is the name of the parameter and Foo a type. Inside the body, the static type of x is Array(Foo). The caller can use 0, 1, or more arguments for the parameter x. Only one ellipsis is allowed in a signature.

```nit
fun foo(x: Int, y: Int, ...: z: Int) do return "Hello World !"
end

foo(1, 2, 3, 4, 5) # outputs "1;2;3;4;5"
foo(1, 2, 3) # outputs "1;2;3"
```

7.7 Top-level Methods and Main Body

Some functions, like print, are usable everywhere simply without using a specific receiver. Such methods are just defined outside any classes. In fact, these methods are implicitly defined in the Object interface, therefore inherited by all classes, therefore usable everywhere. However, this principle may change in a future version.

In a module, the main body is a bunch of statements at the end of a file. The main body of the main module is the program entry point. In fact, the main method of a program is implicitly defined as the redefinition of the method main of the Sys class; and the start of the program is the implicit statement (Sys.new).main. Note that because it is a redefinition, the main part can use super to call the “previous” main part in the imported modules. If there is no main part in a module, it is inherited from imported modules.

Top-level methods coupled with the main body can be used to program in a pseudo-procedural way. Therefore, the following programs are valid:

```nit
fun sum(i, j: Int): Int do return i + j end
print sum(4, 5)
```

7.8 Intern and Extern Methods

Intern and extern indicate concrete methods whose body is not written in Nit.

The body of intern methods is provided by the compiler itself for performance or bootstrap reasons. For the same reasons, some intern methods, like + in Int are not redefinable.

The body of extern methods is provided by libraries written in C; for instance, the system libraries required for input/output. Extern methods are always redefinable. See FFI [??] for more information on extern methods.
8 Attributes

**var**, used inside concrete and abstract classes, declares attributes. Attributes require a static type and can possibly have an initial value (it may be any kind of expression, even including **self**)

class Foo
    var i: Int = 5
    fun dec(x: Int)
        do
            var k = self.i
            if k > x then self.i = k - x else self.i = 0
        end
    end

Note that from an API point of view, there is no way to distinguish the read access of an attribute with a normal method neither to distinguish a write access of an attribute with a setter. Therefore, the read access of an attribute is called a getter while the write access is called a setter.

```i = foo.bar # Is bar an attribute or a method?
foo.bar = y # Is bar an attribute or a setter?
# In fact, we do not need to know.```

### 8.1 Visibility of Attributes

By default, a getter is public and a setter is private. The visibility of getters can be precised with the **private** or **protected** keywords. The visibility of setters can be specified with an additional **writable** keyword.

class Foo
    var pub_pri: X
    protected var pro_pri: X
    var pub_pub: X is writable
    private var pri_pro: X is protected writable
    var pub_pri2: X is private writable # the default
end

### 8.2 Redefinition of Attributes

Getters and setters of attributes behave like genuine methods that can be inherited and redefined. Getters and setters can also redefine inherited methods. **redef var** declares that the getter is a redefinition while **redef writable** declares that the setter is a redefinition.

interface Foo
    fun derp: Int is abstract
end
class Bar
    super Foo
    redef var derp: Int redef writable
end
class Baz
    super Bar
    redef fun derp do ... redef fun derp*(o) do ...
end

### 9 Constructors

Constructors in Nit behave differently.

Their objective is double:
- be compatible with full multiple-inheritance
- be simple enough to be KISS and compatible with the principle of least surprise.

#### 9.1 new construction and simple classes

Classes in OO models are often a simple aggregates of attributes and methods.

By default, the **new** construction require a value for each attribute defined in a class without a default value.

class Product
    var id: String
    var description: String
    var price: Float
end
var p = new Product("ABC", "Bla bla", 15.95)
assert p.id == "ABC"

In subclasses, additional attributes are automatically collected.

class Book
    super Product
    var author: String
end
var book = new Book("ABC", "Bla bla", 15.95, "John Doe")

#### 9.2 special init method

The special init method is automatically invoked after the end of a **new** construction. It is used to perform additional systematic tasks.

Because the **init** is run at the end of the initialization sequence, initialized attributes are usable in the body.

class OverpricedProduct
    super Product
    init
do
        price = price * 10.0
    end
end
var op = new OverpricedProduct("ABC", "Bla bla", 15.95)
assert op.price == 159.50

#### 9.3 Uncollected attributes

There is three cases for an attributes to not be collected in the **new**.
- Attributes with a default value
- Attributes with the annotation **noinit**
- Attributes introduced in refinement of classes

class TaxedProduct
    super Product
    var tax_rate = 9.90
    var total_price: Float is noinit
    init
do
        total_price = price * (1.0 + tax_rate/100.0)
    end
end
var tp = new TaxedProduct("ABC", "Bla bla", 15.95)
assert tp.total_price == 17.82905

Note: The orchestration here is important. In order, the following is executed:
1. All defaults values are computed and set
2. Setters are invoked.
3. **init** is invoked.

Therefore, **total_price** cannot be initialised with a default value, because at the time of the computation of the default values, the attribute **price** in not yet initialised.

#### 9.4 Generalized initializers

Initializers are methods that are automatically invoked by the **new**. In fact, by default, the setter of an attribute is used as a initializer.

**autoinit** is used to register a method as a setter.

class FooProduct
    super Product
    fun set_x_y(x, y: Int) is autoinit do z = x * 10 + y
    var z: Int is noinit
end
var fp = new FooProduct("ABC", "Bla bla", 15.96, 1, 3)
assert fp.z == 13

Generalized setters are a powerful tool but often needed in only rare specific cases. In most case, there is no reason that an argument of a **new** construction is not stored in the object as a real attribute.

#### 9.5 Inheritance

As explained above, one of the main advantage of these constructors is their compatibility with multiple inheritance.

class MultiProduct
    super OverpricedProduct
    super TaxedProduct
    super Product
end
var mp = new MultiProduct("ABC", "Bla bla", 15.96, 1, 3)
assert mp.id == "ABC"
assert mp.price == 159.6
assert mp.total_price == 175.4
assert mp.z == 13

Nov 12, 2015
A Concise Reference of the Nit Language
9.6 Named init

Named init are less flexible through inheritance, thus should no be used. They allow to have additional constructor for classes and more control in the construction mechanism.

```nit
class Point
  var x: Float
  var y: Float
  init origin do
    init(0.0, 0.0)
  end
  init polar(r, phi: Float) do
    var x = r * phi.cos
    var y = r * phi.sin
    init(x, y)
  end
redef fun to_s do return "((x),(y))"
end
var p1 = new Point(1.0, 2.0)
assert p1.to_s == "(1,2)"
var p2 = new Point.origin
var p3 = new Point.polar(1.0, 2.0)
assert p3.to_s == "(-0.4161,0.9092)"
```

9.7 Legacy init

nameless init defined with argument or with an explicit visibility are still accepted as a fallback of the old-constructors. They should not be used since they will be removed in a near future.

9.8 new factories

new factories permit to completely shortcut the class instantiation mechanism. It could be used to provide new syntax on non-concrete class (mainly extern class).

new factories behave like a top-level function that return the result of the construction. It is basically some kind of syntactic sugar.

```nit
abstract class Person
  var age: Int
  new(age: Int) do
    if age >= 18 then
      return new Adult(age)
    else
      return new Child(age)
  end
end
```

10 Generic Classes

Generic classes are defined with formal generic parameters declared within brackets. Formal generic parameters can then be used as a regular type inside the class. Generic classes must always be qualified when used.

```nit
class Pair[E] var first: E
  var second: E
def is_same: Bool do
  return self.first == self.second
end
def p1 = new Pair[Int](1, 2)
print p1.second * 10
# outputs "20"
print p1.is_same
# outputs "false"
def p2 = new Pair[String]("hello", "world")
p2.first = "world"
print p2.is_same
# outputs "true"
```

Unlike many object-oriented languages, generic classes in Nit yield a kind of sub-typing. For example, `Pair[Int]` is a subtype of `Pair[Object]`.

11 Virtual Types

type declares a virtual types in a class. A bound type is mandatory. Virtual types can then be used as regular types in the class and its subclasses. Subclasses can also redefine it with a more specific bound type. One can see a virtual type as an internal formal generic parameter or as a redefinable typedef.

```nit
class Foo type E: Object
  var derp: E
def new
end
class Bar super Foo
def new
  redef type E: Int
def new
end
def b = new Bar(5)
print b.derp + 1
# outputs 6
```

November 12, 2015 A Concise Reference of the Nit Language