Nim Tutorial (Part II) nimversion

Andreas Rumpf

July 26, 2019

Contents

1 Introduction

"Repetition renders the ridiculous reasonable." – Norman Wildberger

This document is a tutorial for the advanced constructs of the *Nim* programming language. **Note that this document is somewhat obsolete as the** manual **contains many more examples of the advanced language features.**

2 Pragmas

Pragmas are Nim's method to give the compiler additional information/ commands without introducing a massive number of new keywords. Pragmas are enclosed in the special `{.` and `.}` curly dot brackets. This tutorial does not cover pragmas. See the manual or user guide for a description of the available pragmas.

3 Object Oriented Programming

While Nim's support for object oriented programming (OOP) is minimalistic, powerful OOP techniques can be used. OOP is seen as *one* way to design a program, not *the only* way. Often a procedural approach leads to simpler and more efficient code. In particular, preferring composition over inheritance is often the better design.

3.1 Objects

Like tuples, objects are a means to pack different values together in a structured way. However, objects provide many features that tuples do not: They provide inheritance and information hiding. Because objects encapsulate data, the `T()` object constructor should only be used internally and the programmer should provide a proc to initialize the object (this is called a *constructor*).

Objects have access to their type at runtime. There is an `of` operator that can be used to check the object's type:

```
type
  Person = ref object of RootObj
    name*: string  # the * means that `name` is accessible from other modules
    age: int       # no * means that the field is hidden from other modules

  Student = ref object of Person # Student inherits from Person
    id: int                      # with an id field

var
  student: Student
  person: Person
assert(student of Student) # is true
# object construction:
student = Student(name: "Anton", age: 5, id: 2)
echo student[]
```

Object fields that should be visible from outside the defining module have to be marked by `*`. In contrast to tuples, different object types are never *equivalent*. New object types can only be defined within a type section.

Inheritance is done with the `object of` syntax. Multiple inheritance is currently not supported. If an object type has no suitable ancestor, `RootObj` can be used as its ancestor, but this is only a convention. Objects that have no ancestor are implicitly `final`. You can use the `inheritable` pragma to introduce new object roots apart from `system.RootObj`. (This is used in the GTK wrapper for instance.)

Ref objects should be used whenever inheritance is used. It isn't strictly necessary, but with non-ref objects assignments such as `let person: Person = Student(id: 123)` will truncate subclass fields.

Note: Composition (*has-a* relation) is often preferable to inheritance (*is-a* relation) for simple code reuse. Since objects are value types in Nim, composition is as efficient as inheritance.

3.2 Mutually recursive types

Objects, tuples and references can model quite complex data structures which depend on each other; they are *mutually recursive*. In Nim these types can only be declared within a single type section. (Anything else would require arbitrary symbol lookahead which slows down compilation.)

Example:

```
type
  Node = ref object  # a reference to an object with the following field:
    le, ri: Node     # left and right subtrees
    sym: ref Sym     # leaves contain a reference to a Sym

  Sym = object       # a symbol
    name: string     # the symbol's name
    line: int        # the line the symbol was declared in
    code: Node       # the symbol's abstract syntax tree
```

3.3 Type conversions

Nim distinguishes between type casts and type conversions. Casts are done with the `cast` operator and force the compiler to interpret a bit pattern to be of another type.

Type conversions are a much more polite way to convert a type into another: They preserve the abstract *value*, not necessarily the *bit-pattern*. If a type conversion is not possible, the compiler complains or an exception is raised.

The syntax for type conversions is `destination_type(expression_to_convert)` (like an ordinary call):

```
proc getID(x: Person): int =
  Student(x).id
```

The `InvalidObjectConversionError` exception is raised if `x` is not a `Student`.

3.4 Object variants

Often an object hierarchy is overkill in certain situations where simple variant types are needed.

An example:

```
# This is an example how an abstract syntax tree could be modelled in Nim
type
  NodeKind = enum  # the different node types
    nkInt,          # a leaf with an integer value
    nkFloat,        # a leaf with a float value
    nkString,       # a leaf with a string value
    nkAdd,          # an addition
    nkSub,          # a subtraction
    nkIf            # an if statement
  Node = ref object
    case kind: NodeKind  # the ``kind`` field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: Node
    of nkIf:
      condition, thenPart, elsePart: Node

var n = Node(kind: nkFloat, floatVal: 1.0)
# the following statement raises an `FieldError` exception, because
# n.kind's value does not fit:
n.strVal = ""
```

As can been seen from the example, an advantage to an object hierarchy is that no conversion between different object types is needed. Yet, access to invalid object fields raises an exception.

3.5 Methods

In ordinary object oriented languages, procedures (also called *methods*) are bound to a class. This has disadvantages:

- Adding a method to a class the programmer has no control over is impossible or needs ugly workarounds.
- Often it is unclear where the method should belong to: is `join` a string method or an array method?

Nim avoids these problems by not assigning methods to a class. All methods in Nim are multi-methods. As we will see later, multi-methods are distinguished from procs only for dynamic binding purposes.

3.6 Method call syntax

There is a syntactic sugar for calling routines: The syntax `obj.method(args)` can be used instead of `method(obj, args)`. If there are no remaining arguments, the parentheses can be omitted: `obj.len` (instead of `len(obj)`).

This method call syntax is not restricted to objects, it can be used for any type:

```
import strutils

echo "abc".len # is the same as echo len("abc")
echo "abc".toUpperAscii()
echo({'a', 'b', 'c'}.card)
stdout.writeLine("Hallo") # the same as writeLine(stdout, "Hallo")
```

(Another way to look at the method call syntax is that it provides the missing postfix notation.)

So "pure object oriented" code is easy to write:

```
import strutils, sequtils

stdout.writeLine("Give a list of numbers (separated by spaces): ")
stdout.write(stdin.readLine.splitWhitespace.map(parseInt).max.`$`)
stdout.writeLine(" is the maximum!")
```

3.7 Properties

As the above example shows, Nim has no need for *get-properties*: Ordinary get-procedures that are called with the *method call syntax* achieve the same. But setting a value is different; for this a special setter syntax is needed:

```
type
  Socket* = ref object of RootObj
    h: int # cannot be accessed from the outside of the module due to missing star

proc `host=`*(s: var Socket, value: int) {.inline.} =
  ## setter of host address
  s.h = value

proc host*(s: Socket): int {.inline.} =
  ## getter of host address
  s.h

var s: Socket
new s
s.host = 34  # same as `host=`(s, 34)
```

(The example also shows `inline` procedures.)

The `[]` array access operator can be overloaded to provide array properties:

```
type
  Vector* = object
    x, y, z: float
```

```
proc `[]=`* (v: var Vector, i: int, value: float) =
  # setter
  case i
  of 0: v.x = value
  of 1: v.y = value
  of 2: v.z = value
  else: assert(false)

proc `[]`* (v: Vector, i: int): float =
  # getter
  case i
  of 0: result = v.x
  of 1: result = v.y
  of 2: result = v.z
  else: assert(false)
```

The example is silly, since a vector is better modelled by a tuple which already provides `v[]` access.

3.8 Dynamic dispatch

Procedures always use static dispatch. For dynamic dispatch replace the `proc` keyword by `method`:

```
type
  Expression = ref object of RootObj ## abstract base class for an expression
  Literal = ref object of Expression
    x: int
  PlusExpr = ref object of Expression
    a, b: Expression

# watch out: 'eval' relies on dynamic binding
method eval(e: Expression): int =
  # override this base method
  quit "to override!"

method eval(e: Literal): int = e.x
method eval(e: PlusExpr): int = eval(e.a) + eval(e.b)

proc newLit(x: int): Literal = Literal(x: x)
proc newPlus(a, b: Expression): PlusExpr = PlusExpr(a: a, b: b)

echo eval(newPlus(newPlus(newLit(1), newLit(2)), newLit(4)))
```

Note that in the example the constructors `newLit` and `newPlus` are procs because it makes more sense for them to use static binding, but `eval` is a method because it requires dynamic binding.

In a multi-method all parameters that have an object type are used for the dispatching:

```
type
  Thing = ref object of RootObj
  Unit = ref object of Thing
    x: int

method collide(a, b: Thing) {.inline.} =
  quit "to override!"

method collide(a: Thing, b: Unit) {.inline.} =
  echo "1"

method collide(a: Unit, b: Thing) {.inline.} =
  echo "2"

var a, b: Unit
new a
new b
collide(a, b) # output: 2
```

As the example demonstrates, invocation of a multi-method cannot be ambiguous: Collide 2 is preferred over collide 1 because the resolution works from left to right. Thus `Unit, Thing` is preferred over `Thing, Unit`.

Performance note: Nim does not produce a virtual method table, but generates dispatch trees. This avoids the expensive indirect branch for method calls and enables inlining. However, other optimizations like compile time evaluation or dead code elimination do not work with methods.

4 Exceptions

In Nim exceptions are objects. By convention, exception types are suffixed with 'Error'. The system module defines an exception hierarchy that you might want to stick to. Exceptions derive from `system.Exception`, which provides the common interface.

Exceptions have to be allocated on the heap because their lifetime is unknown. The compiler will prevent you from raising an exception created on the stack. All raised exceptions should at least specify the reason for being raised in the `msg` field.

A convention is that exceptions should be raised in *exceptional* cases: For example, if a file cannot be opened, this should not raise an exception since this is quite common (the file may not exist).

4.1 Raise statement

Raising an exception is done with the `raise` statement:

```
var
  e: ref OSError
new(e)
e.msg = "the request to the OS failed"
raise e
```

If the `raise` keyword is not followed by an expression, the last exception is *re-raised*. For the purpose of avoiding repeating this common code pattern, the template `newException` in the `system` module can be used:

```
raise newException(OSError, "the request to the OS failed")
```

4.2 Try statement

The `try` statement handles exceptions:

```
from strutils import parseInt

# read the first two lines of a text file that should contain numbers
# and tries to add them
var
  f: File
if open(f, "numbers.txt"):
  try:
    let a = readLine(f)
    let b = readLine(f)
    echo "sum: ", parseInt(a) + parseInt(b)
  except OverflowError:
    echo "overflow!"
  except ValueError:
    echo "could not convert string to integer"
  except IOError:
    echo "IO error!"
  except:
    echo "Unknown exception!"
    # reraise the unknown exception:
    raise
  finally:
    close(f)
```

The statements after the `try` are executed unless an exception is raised. Then the appropriate `except` part is executed.

The empty `except` part is executed if there is an exception that is not explicitly listed. It is similar to an `else` part in `if` statements.

If there is a `finally` part, it is always executed after the exception handlers.

The exception is *consumed* in an `except` part. If an exception is not handled, it is propagated through the call stack. This means that often the rest of the procedure - that is not within a `finally` clause - is not executed (if an exception occurs).

If you need to *access* the actual exception object or message inside an `except` branch you can use the getCurrentException() and getCurrentExceptionMsg() procs from the system module. Example:

```
try:
  doSomethingHere()
except:
  let
    e = getCurrentException()
    msg = getCurrentExceptionMsg()
  echo "Got exception ", repr(e), " with message ", msg
```

4.3 Annotating procs with raised exceptions

Through the use of the optional `{.raises.}` pragma you can specify that a proc is meant to raise a specific set of exceptions, or none at all. If the `{.raises.}` pragma is used, the compiler will verify that this is true. For instance, if you specify that a proc raises `IOError`, and at some point it (or one of the procs it calls) starts raising a new exception the compiler will prevent that proc from compiling. Usage example:

```
proc complexProc() {.raises: [IOError, ArithmeticError].} =
  ...

proc simpleProc() {.raises: [].} =
  ...
```

Once you have code like this in place, if the list of raised exception changes the compiler will stop with an error specifying the line of the proc which stopped validating the pragma and the raised exception not being caught, along with the file and line where the uncaught exception is being raised, which may help you locate the offending code which has changed.

If you want to add the `{.raises.}` pragma to existing code, the compiler can also help you. You can add the `{.effects.}` pragma statement to your proc and the compiler will output all inferred effects up to that point (exception tracking is part of Nim's effect system). Another more roundabout way to find out the list of exceptions raised by a proc is to use the Nim `doc2` command which generates documentation for a whole module and decorates all procs with the list of raised exceptions. You can read more about Nim's effect system and related pragmas in the manual.

5 Generics

Generics are Nim's means to parametrize procs, iterators or types with type parameters. They are most useful for efficient type safe containers:

```
type
  BinaryTree*[T] = ref object # BinaryTree is a generic type with
                              # generic param ``T``
    le, ri: BinaryTree[T]     # left and right subtrees; may be nil
    data: T                   # the data stored in a node

proc newNode*[T](data: T): BinaryTree[T] =
  # constructor for a node
  new(result)
  result.data = data

proc add*[T](root: var BinaryTree[T], n: BinaryTree[T]) =
  # insert a node into the tree
  if root == nil:
    root = n
  else:
    var it = root
    while it != nil:
```

```
      # compare the data items; uses the generic ``cmp`` proc
      # that works for any type that has a ``==`` and ``<`` operator
      var c = cmp(it.data, n.data)
      if c < 0:
        if it.le == nil:
          it.le = n
          return
        it = it.le
      else:
        if it.ri == nil:
          it.ri = n
          return
        it = it.ri

proc add*[T](root: var BinaryTree[T], data: T) =
  # convenience proc:
  add(root, newNode(data))

iterator preorder*[T](root: BinaryTree[T]): T =
  # Preorder traversal of a binary tree.
  # Since recursive iterators are not yet implemented,
  # this uses an explicit stack (which is more efficient anyway):
  var stack: seq[BinaryTree[T]] = @[root]
  while stack.len > 0:
    var n = stack.pop()
    while n != nil:
      yield n.data
      add(stack, n.ri)  # push right subtree onto the stack
      n = n.le          # and follow the left pointer

var
  root: BinaryTree[string] # instantiate a BinaryTree with ``string``
add(root, newNode("hello")) # instantiates ``newNode`` and ``add``
add(root, "world")          # instantiates the second ``add`` proc
for str in preorder(root):
  stdout.writeLine(str)
```

The example shows a generic binary tree. Depending on context, the brackets are used either to introduce type parameters or to instantiate a generic proc, iterator or type. As the example shows, generics work with overloading: the best match of `add` is used. The built-in `add` procedure for sequences is not hidden and is used in the `preorder` iterator.

6 Templates

Templates are a simple substitution mechanism that operates on Nim's abstract syntax trees. Templates are processed in the semantic pass of the compiler. They integrate well with the rest of the language and share none of C's preprocessor macros flaws.

To *invoke* a template, call it like a procedure.

Example:

```
template `!=` (a, b: untyped): untyped =
  # this definition exists in the System module
  not (a == b)

assert(5 != 6) # the compiler rewrites that to: assert(not (5 == 6))
```

The `!=`, `>`, `>=`, `in`, `notin`, `isnot` operators are in fact templates: this has the benefit that if you overload the == operator, the `!=` operator is available automatically and does the right thing. (Except for IEEE floating point numbers - NaN breaks basic boolean logic.)

`a > b` is transformed into `b < a`. `a in b` is transformed into `contains(b, a)`. `notin` and `isnot` have the obvious meanings.

Templates are especially useful for lazy evaluation purposes. Consider a simple proc for logging:

```
const
  debug = true
```

```
proc log(msg: string) {.inline.} =
  if debug: stdout.writeLine(msg)

var
  x = 4
log("x has the value: " & $x)
```

This code has a shortcoming: if `debug` is set to false someday, the quite expensive `$` and `&` operations are still performed! (The argument evaluation for procedures is *eager*).

Turning the `log` proc into a template solves this problem:

```
const
  debug = true

template log(msg: string) =
  if debug: stdout.writeLine(msg)

var
  x = 4
log("x has the value: " & $x)
```

The parameters' types can be ordinary types or the meta types `untyped`, `typed`, or `type`. `type` suggests that only a type symbol may be given as an argument, and `untyped` means symbol lookups and type resolution is not performed before the expression is passed to the template.

If the template has no explicit return type, `void` is used for consistency with procs and methods.

To pass a block of statements to a template, use 'untyped' for the last parameter:

```
template withFile(f: untyped, filename: string, mode: FileMode,
                  body: untyped): typed =
  let fn = filename
  var f: File
  if open(f, fn, mode):
    try:
      body
    finally:
      close(f)
  else:
    quit("cannot open: " & fn)

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

In the example the two `writeLine` statements are bound to the `body` parameter. The `withFile` template contains boilerplate code and helps to avoid a common bug: to forget to close the file. Note how the `let fn = filename` statement ensures that `filename` is evaluated only once.

6.1 Example: Lifting Procs

```
import math

template liftScalarProc(fname) =
  ## Lift a proc taking one scalar parameter and returning a
  ## scalar value (eg ``proc sssss[T](x: T): float``),
  ## to provide templated procs that can handle a single
  ## parameter of seq[T] or nested seq[seq[]] or the same type
  ##
  ## .. code-block:: Nim
  ##  liftScalarProc(abs)
  ##  # now abs(@[@[1,-2], @[-2,-3]]) == @[@[1,2], @[2,3]]
  proc fname[T](x: openarray[T]): auto =
    var temp: T
    type outType = type(fname(temp))
    result = newSeq[outType](x.len)
    for i in 0..<x.len:
      result[i] = fname(x[i])
```

```
liftScalarProc(sqrt)   # make sqrt() work for sequences
echo sqrt(@[4.0, 16.0, 25.0, 36.0])   # => @[2.0, 4.0, 5.0, 6.0]
```

7 Compilation to JavaScript

Nim code can be compiled to JavaScript. However in order to write JavaScript-compatible code you should remember the following:

- `addr` and `ptr` have slightly different semantic meaning in JavaScript. It is recommended to avoid those if you're not sure how they are translated to JavaScript.
- `cast[T](x)` in JavaScript is translated to `(x)`, except for casting between signed/unsigned ints, in which case it behaves as static cast in C language.
- `cstring` in JavaScript means JavaScript string. It is a good practice to use `cstring` only when it is semantically appropriate. E.g. don't use `cstring` as a binary data buffer.

8 Part 3

Next part will be entirely about metaprogramming via macros: Part III