Operno

- Dynamic Power -

Dokumentation

Vorwort

Operno ist ein Token-Interpreter, welcher imperativ, abstrakt, objektorientiert und dynamisch ist. Mit „dynamisch“ ist gemeint, dass viele Dinge in Operno zur Laufzeit verwaltet werden, was z.B. das neudefinieren von Objekten ermöglicht.

Das Ziel des Operno-Interpreters ist, ein übersichtliches Skript und höchste Dynamik zu ermöglichen. Die Syntax ist auch möglichst so aufgebaut, dass Einsteiger möglichst schnell ein Skript in Operno analysieren können. Unter der Dynamik leidet natürlich die Geschwindigkeit des Interpreters, welche jedoch nicht im Auge von Operno liegt.

Der beste gebrauch eines solchen Interpreters ist z.B. für Datenbanken oder für Konsolenanwendungen, die komplexe Ausdrücke zulassen. Für Spiele und andere große Applikationen ist Operno eher ungeeignet.

Bisher stehen als Schnittstelle für den Operno-Interpreter nur die Konsole und eine DLL zur Verfügung. Deshalb währe ich sehr dankbar, wenn jemand bereit ist, ein Editor speziell für Operno zu entwickeln; ein Eintrag in den Credits ist natürlich sicher.

Für den Einstieg in Operno würde ich vorschlagen, die Kapiteln „Syntax-Regeln“ und „Objekte“ durchzulesen. Für einen schnellen Einsteig bietet sich das Tutorial an, welches sich durch die Konsole und die Eingabe „.tut“ ausführen lässt.

Wenn es weitere Fragen, Bugs, Anregungen oder Interessen an der Entwicklung des Operno-Interpreters gibt, dann meldet euch bitte auf meiner E-Mail-Adresse „josef-operno-mail@t-online.de“. Auch unter meiner Homepage findet ihr Kontakt zu mir und weitere Informationen zu dem Interpreter.

Viel Spaß mit Operno wünscht

Euer Josef Sniatecki

Allgemeines

Geschichte

Operno 5.10:

Hinzugefügt:

- **Externe Bibliotheken** als "DLL" können importiert und genutzt werden
- Die Datei "LibrarySDK.pbi" im "Library"-Ordner bietet die Möglichkeit selbst externe Bibliotheken für Operno zu entwickeln.
- Die logische **XOr**-Verknüpfung wurde hinzugefügt
- **Switch**-Kontrollstruktur wurde für das schnellere Abzweigen hinzugefügt
- Optionen für den HTML-Generator können durch die Konsole vorgenommen werden

Geändert:

- **Standardklassen** bestehten nun nur noch **als Referenzen** in der Standard-Bibliothek (aus Übersichtlichkeits-Gründen)
- Klassenmethoden für Operatoren mit einer Sofortzuweisung (z.B. "+=") besitzen nun ein **"!" am Ende** des Namens **anstatt "Set"**
- Die **"file"**-Bibliothek wurde auf **"fileStream"** umbenannt
- **Tutorials** sind jetzt nicht mehr in Operno geschrieben (stattdessen wird ein spezieller Parser zum Ausführen genutzt)
- Das Startfenster kann mit **allen Mausklicks und Tasten** ausgeblendet werden

Repariert:

- Blöcke werden **besser ausgegeben** (z.B. bekommen logische Verknüpfungen Leerzeichen links und rechts rangesetzt)
- Error-Handling wurde **optimiert**, da immer noch viele IMAs ausgelöst wurden
- Kleinere Bugs beseitigt

Operno 5.00:

Hinzugefügt:

- Ein Programm kann durch "Operno.dll" die **Engine von Operno nutzen**
- **For**-Schleife für die gleichzeitige Auswertung von mehreren Werten
- Operatoren für die **direkte Zuweisung**: "+=", "-=", "*=", "/="
- Queue-Klasse für das Verwalten von **dynamischen Listen**
- **Bool**-Klasse für die Auswertung von Aussagen und anderen Fällen
- Funktionen können eine feste **Rückgabewert-Klasse** besitzen
- "pop" kann einen Wert **unabhängig von einer Variable** zurückgeben

Geändert:

- Jeder Wert besitzt nun eine **Klasse, die als Objekt** vorliegt
- Auf Klassenmethoden wird durch **":"** zugegriffen

Syntax-Regeln

Namensgebung

In Operno ist die Namensgebung von Objekten sehr flexibel, da man alle Umlaute und auch manchmal Leerzeichen benutzen darf, was in vielen Sprachen aus technischen Gründen nicht möglich ist.

Grundregeln

- Zahlen, Ausrufezeichen und Fragezeichen dürfen erst nach dem ersten Zeichen benutzt werden
- Leerzeichen können benutzt werden, solange die einzelnen Wörter keine Schlüsselwörter sind
- Groß- und Kleinschreibung wird beachtet

Erlaubte Zeichen

A-Z, a-z, 0-9, _, À, Á, Â, Ã, Ä, Å, Æ, Ç, È, É, Ê, Ë, Ì, Í, Î, Ï, Ð, Ñ, Ò, Ó, Ô, Õ, Ö, ×, Ø, Ù, Ú, Û, Ü, Ý, Þ, ß, à, á, â, ã, ä, å, æ, ç, è, é, ê, ë, ì, í, î, ï, ð, ñ, ò, ó, ô, õ, ö, ÷, ø, ù, ú, û, ü, ý, þ, ÿ

Leerzeichen

In Operno können auch Leerzeichen bei der Namensgebung benutzt werden, wobei diese nach dem Lexer zu zusammengesetzten Wörter werden. Z.B. wird „my name“ zu „myName“ und „the house of“ zu „theHouseOf“.

Schlüsselwörter werden jedoch nicht mit eingebunden. Bsp: „my output channel“ wird zu „my“, „output“ (das Schlüsselwort) und „channel“.

Beispiele:

```
dies ist ein test     '= diesIstEinTest
index
variable2
höhe von D            '= höheVonD
```

Zahlen

Zahlen können in folgenden drei Formen dargestellt werden:

Dezimalzahlen

Dezimalzahlen werden mit den normalen Ziffern von 0-9 dargestellt. Kommata sind in Operno Punkte („.“). Die Größe der Dezimalzahl entscheidet über den Typ. Zahlen die kleiner als 8 Zeichen sind, werden als Integer oder Float (mit Komma) definiert. Alle die gleich oder größer als 8 Zeichen groß sind, werden als Quad oder Double (mit Komma) definiert.

Beispiele:

```
100         'Integer
5.25        'Float
12345678    'Quad
0.120000    'Double
```

Binärzahlen

Binärzahlen werden mit einem „0$“ anfangs angedeutet. Folgende Zeichen sind für Nullen und Einsen erlaubt:

- **Null:** `0, o, O, Q`
- **Eins:** `1,i,I`

Beispiele:

```
0$11001
0$IQQQII    '= 0$100011
0$ioiio     '= 0$10110
```

Hexadezimalzahlen

Hexadezimalzahlen werden durch ein „0#“ am Anfang angedeutet. Die Buschstaben dürfen klein oder groß geschreiben werden.

Beispiele:

```
0#FF5
0#c0c0c0
0#91AB
```

Umwandlung in Bogenmaß:

Wird ein „°“ nach einer Zahl geschrieben, so wird diese vom Lexer in Bogenmaß umgewandelt. Bsp: „180° = pi“. Das Symbol „°“ wird ohne eine Verbindung mit Zahlen als „/ 180 * pi“ übersetzt. Bsp: „x° = x / 180 * pi“.

Strings

Zeichenketten werden, wie in vielen anderen Sprachen auch, mit einem Anführungszeichen begonnen und beendet. In Operno sind innerhalb eines Strings alle Zeichen, bis auf das Präprozessorsymbol (siehe unten), erlaubt.

Mehrzeilige Strings

In Operno ist es möglich Zeichenketten über mehrere Zeilen zu erstrecken. Dazu muss das beendende Anführungszeichen bis zur letzten Zeile des Strings ausgelassen werden. Die dazwischenliegenden Zeilenumbrüche werden ignoriert. Die Leerzeiche und Tabs am Anfang jeder Zeile werden genauso ignoriert. Ein „+“ signalisiert manuell den Anfang des nächsten Strings, wobei dieses „+“ selbst ignoriert wird. Dadurch können Leerzeichen und Tabs am Anfang einer Zeile berücksichtigt werden.

Beispiele:

```
text1 = „abc
         def
         ghi“

text2 = „Hello
        + world
         !“
```

„text1“ = „abcdefghi“ und „text2“ = „Hello world!“

Symbole

Zeichenketten können auch unschreibbare Zeichen beinhalten. Dazu dienen die Symbole, welche für ein Zeichen stehen, welches nicht direkt mit der Tastatur eingegeben werden kann. Diese Symbole werden durch den Präprozessorsymbol („#“) angedeutet. Wird das „#“ in einem String benötigt, so gibt man stattdessen „##“ ein.

Symbol	Beschreibung
n	Zeilenumbruch (ASCII 13,10)
q	Anführungszeichen
t	Tabulator (ASCII 8)
c	Copyright (©)
xxx	Dreistelliger* ASCII-Code eines Zeichens

*Ist der ASCII-Code kleiner als 3 Stellen, so müssen führende Nullen zur Füllung der drei Stellen genutzt werden.

Kommentare

Kommentare werden vom Lexer ignoriert und sind damit zur Ausführung nicht mehr vorhanden. Man kann in Operno Kommentare auf zwei verschiedene Weisen setzen:

Einzeilige Kommentare

Einzeilige Kommentare starten mit einem Apostroph und enden mit dem nächsten Zeilenumbruch. Alle weiteren Zeichen nach dem Apostroph werden ignoriert.

Beispiel:

```
x = 5 'Ein kurzer Kommentar
```

Block-Kommentare

Mehrzeilige Kommentare werden durch ein „'{“ gestartet und mit einem „}'“ beendet. Verschachtelungen sind möglich.

Beispiel:

```
'{
     Dies ist
     ein Kommentar, der
     sich über
     mehrere Zeilen
     erstreckt.
}'

'{
     '{
          Verschachtelt
     }'
}'
```

Objekte

Gruppen

Gruppen (engl. groups) erlauben in Operno die Gruppierung von mehreren Objekten. Diese Objekte werden als Unterobjekte (engl. sub-objects, child-objects) in der jeweiligen Gruppe gespeichert.
Diese Gruppen dienen gleichzeitig als Klassen, welche Träger von Klassen-Methoden und statischen Variablen sein können.

Definition

Gruppen werden mit angehängten geschweiften Klammern definiert. In diesen geschweiften Klammern können optional Unterobjekte definiert werden. Auch Gruppen sind als Unterobjekte erlaubt, wodurch Verschachtelungen entstehen können.

Beispiel:

```
'Definition einer leeren Gruppe:
friends{}

'Sofort-Definition von Unterobjekten:
person{
      name = „Larry“
      age  = 35
}
```

Zugriff

Auf Unterobjekte einer Gruppe wird durch ein Back-Slash zugegriffen. Das Unterobjekt steht nach dem Slash und die dazugehörende Gruppe davor.

Wird eine Klammer direkt nach dem Slash angegeben, so wird ein variabler Zugriff ermöglicht. Folgende Klassen als Ergebnisse des Ausdrucks einer solchen Klammer, werden folgend ausgewertet:

Klasse	Beschreibung
integer	Zugriff über Index. Das erste Unterobjekt trägt den Index 0. Existiert ein Objekt nicht unter dem angegebenen Index, dann wird ein Objekt unter dem Namen „item_“ plus dem Index definiert.
string	Zugriff über den Namen eines Unterobjekts. Ist das Objekt unter dem Namen nicht zu finden, so wird ein neues mit dem angegebenen Namen definiert.

Beispiel:

```
anybody{
      name = „Sally“
      age  = 21
}

out anybody\name           'Gibt „Sally“ aus

out anybody\(„na“ + „me“) 'Gibt „Sally“ aus
out anybody\(1)           'Gibt 21 aus
```

Variablen

Variablen speichern in Operno Werte, welche immer wieder mittels der Variable gelesen und beschrieben werden können.

Definition

Man kann Variablen auf fünf verschiedene Weisen definieren:

1) Durch das Setzen eines undefinierten Objekts wird eine Variable mit dem angegebenen Wert definiert.

Beispiel:

```
x = 61
```

2) Wird ein literaler Wert (darf kein Ausdruck sein) direkt nach dem Namen eines undefinierten Objekts angegeben, so wird eine Variable mit diesem Wert erstellt.

Beispiel:

```
name „Josef“
age  15
```

3) Anhand der Angabe einer Klasse durch ein Suffix-“.“/-“as“ wird eine Variable mit der Klasse definiert. Dabei sind bekannte Abkürzungen für Standardklassen erlaubt.

Beispiel:

```
num.integer
pos as float
text.str
```

4) Wird eine Parameter-Klammer nach der Definition Nr. 3 angehängt, so wird die Variable mit den angegebenen Parametern definiert.

Beispiel:

```
num.integer(123)        'Setzt „num“ auf 123
bad.queue(„black“, 13) 'Eine Queue mit [„black“, 13]
```

5) Durch das einfache erwähnen eines undefinierten Objekts wird eine Variable als Variant erstellt. D.h. die Variable besitzt bis zur ersten Zuweisung keine bestimmte Klasse und keinen Wert. (siehe Kapitel „Variant“)

Funktionen

Durch Funktionen kann man in Operno eine reihe von Anweisungen gruppieren und diese immer wieder durch das angeben der Funktion ausführen. Optinal lassen sich Parameter angeben.

Definition

Eine Funktionen wird immer durch eine Klammer definiert. Dabei können in dieser Klammer Variablen für Parameter angegeben werden. Nach der Klammer kann entweder eine einzeilige Anweisung stehen, oder ein Block von Anweisungen, welcher zwischen geschweiften Klammern steht.
Soll der Rückgabewert von einer bestimmten Klasse sein, so gibt man diese Klasse nach der Parameter-Klammer durch ein „as“/“.“ an.

Beispiele:

```
say hello to(name.s) out „Hello, “ + name + „!“

say goodbye to(name.s){
     out „Goodbye, “ + name + „!“
     out „I'll miss you!“
}

sum(a, b).float{
     a + b 'Wird als Float zurückgegeben
}
```

Rückgabewert

Der Rückgabewert ist das Ergebnis des zuletzt ausgeführten Ausdrucks, oder das Ergebnis des Ausdrucks nach einer „return“-Anweisung. Wurde eine Rückgabe-Klasse angegeben, dann wird der zurückzugebende Wert als angegebene Klasse kopiert. Ansonsten wird das Ergebnis direkt zurückgegeben.

Beispiel:

```
true? one(state){
     if(state) („True“) else(„False“)
} 'Gibt entweder „True“, oder „False“ aus

true? two(state){
     if(state){
          return „True“
     }
     „False“
} 'Macht das Gleiche wie „true? one“
```

Parameter

In Operno können Parameter optinal bei der Definition einer Funktion angegeben werden. Wobei dies nur zur Übersichtlichkeit dient, da die Parameter durch ein „pop“ in den Block der jeweiligen Funktion eingefügt werden.
Parametern können auch ein Default-Wert durch „=“ zugewiesen werden.

Beispiel: Beide Definitionen sind Equivalent:

```
average(a, b, c){
     return (a + b + c) / 3
}

average(){
     pop a, b, c
     return (a + b + c) / 3
}
```

Zugriff auf Klassenmethoden

Auf Klassenmethoden kann man durch ein Doppelpunkt nach dem jeweiligen Wert zugreifen. Dabei kann dieser Wert in einer Variable verankert, literal oder im Stack geladen sein.

Beispiel für verankerte Werte:

```
my name = "Josef"
my name:reverse! 'Zugriff auf eine Klassenmethode der "String"-Klasse
out my name      'Gibt "fesoJ" aus
```

Beispiel für Literale:

```
out -10:abs          'Gibt 10 aus
out "abc":upper case 'Gibt "ABC" aus
```

Beispiel für Werte im Stack:

```
out ("a" + "b" + "c"):reverse 'Gibt "cba" aus
```

Superklasse

Werden zwei Doppelpunkte angegeben, so greift man auf die Klasse des Headers zu (= Superklasse). Mit dem Header ist das Trägerobjekt einer Variable, einer Gruppe oder einer Funktion gemeint. Man kann somit z.B. ein Objekt umbenennen, oder den Block einer Funktion auslesen und überschreiben.

Beispiel für Variablen:

```
my name = "Josef"
my name::rename("otherName") 'Zugriff auf die Superklasse
out other name               'Gibt "Josef" aus
```

Beispiel für Gruppen:

```
myself{ name = "Josef"; age = 16 }
myself::clear
out myself 'Gibt myself{} aus, da die Gruppe geleert wurde
```

Beispiel für Funktionen:

```
sum(a, b) return a + b
out sum::block 'Gibt { pop a, b; return a + b } aus
```

Klassendefinition

Eigene Klassen werden genauso wie Gruppen definiert. Eine Klasse besteht also aus einer Gruppe von Klassenmethoden, wie einem Konstruktor, Dekonstruktor und Operatoren. In Operno sind Vererbungen und Kommunikationen zwischen mehreren Klassen möglich.

Auf die Daten eines Objekts kann man durch ein Back-Slash zugreifen. Der Zugriff ist also gleich dem einer Gruppe und dessen Unterobjekte.

Beispiel:

```
person{
  _new(){
    this\name = "Unknow"
    this\age  = 0
  }
  _form(){
    return "My name is " << this\name << "."
  }
}

myself.person{
  name = "Josef"
  age  = 16
}

out myself      'Gibt "My name is Josef." aus
out myself\age  'Gibt 16 aus
```

Standard-Klassenmethoden

Jede der folgenden optionalen Klassenmethoden wird automatisch zu einem bestimmten Ereignis aufgerufen und hat am Anfang des Namens ein Unterstrich. Die Referenz "this" ist in jeder Klassenmethode auf das jeweilige Objekt der Klasse gesetzt. In diesem Objekt sind alle Instanz-Variablen aufgestellt.

this:_new()

"_new" ist der Konstruktor einer Klasse. In dieser Funktion werden alle Instanz-Variablen definiert.

Beispiel für eine Vektor-Klasse:

```
_new(){
  this\x = 0
  this\y = 0
  this\z = 0
}
```

this:_def(...)

Wird eine Parameter-Klammer bei der Definition eines Objekts angegeben, so werden diese Parameter an die "_def"-Funktion übergeben.

Beispiel für eine Person-Klasse:

```
_def(name.s, age.i){
  this\name = name
  this\age  = age
}
```

this:_del()

"_del" ist der Dekonstruktor eines Objekts. In dieser Funktion können Aufräumarbeiten erledigt werden.

this:_true?()

Diese Klassenmethode wird aufgerufen, wenn geprüft werden soll, ob das Objekt "wahr" oder "falsch" ist. Dies ist z.B. für die Auswertung von Aussagen bei "If"-Zweigen nötig.

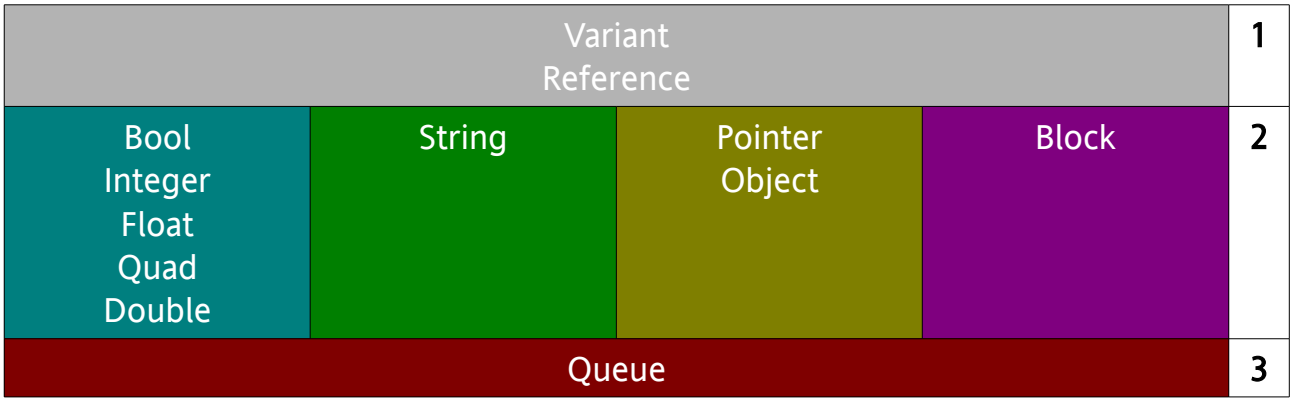
Beispiel für eine Person-Klasse:

```
_true?(){
  (#true) if(this\name <> "Unknow") else(#false)
}
```

Standardklassen

Übersicht

In Operno gibt es folgende Standardklassen, welche bei jedem Start von Operno automatisch in der Standard-Library Initialisiert werden:



Jede Klasse wird in eine Priorität eingestuft (von oben nach unten), die darüber entscheidet, welche Klasse nach einer Operation mit einer Anderen zurückgegeben wird.

Beispiel:

```
integer = 1
float = 0.2

result = integer + float
```

„result“ wird von der Klasse „float“ sein, da ein Float eine höhere Priorität als ein Integer besitzt.

Alle Frabblöcke von Klassen die in der Tabelle nebeneinander stehen, haben keinen Unterschied in ihrer Priorität und geben bei Operationen untereinander unterschiedliche Ergebnisse zurück.

Kurzzeitige Klassen

Die Standardklassen „Reference“ und „Variant“ habe deshalb die niedrigste Priorität, weil sie die Klasse und alle anderen Attribute eines zugewiesenen Wertes annehmen. Sie bestehen also nur innerhalb des Zeitraums in dem sie unbenutzt sind.

Reference

Eine Referenz kann einen Wert annehmen, ohne diesen vorerst zu kopieren. D.h. Referenzen beinhalten genau den Wert, den sie zugewiesen bekommen haben. Dies ist z.B. besonders für Funktionen nützlich, die einen angegebenen Parameter direkt modifizieren möchten, ohne diesen zu kopieren.
Löscht man eine Referenz, dann wird der unkopierte Wert nicht gelöscht. Wird jedoch der originale Wert gelöscht, obwohl noch Referenzen auf diesen bestehen, so können unerwartete Fehler auftreten.

Definition

Eine Referenz trägt vor der ersten Zuweisung noch keinen Wert. Wird trotzdem versucht, einen Wert aus einer leeren Referenz auszugeben, so wird „<UNTYPED>“ angezeigt.

Bekannte Abkürzungen: „ref“, „r“.

Beispiel:

```
x = 5
my reference.ref
out my reference 'Gibt „<UNTYPED>“ aus
my reference = x
out my reference 'Gibt „5“ aus
my reference = 123
out x            'Gibt „123“ aus
```

Referenz ändern

Man kann den Wert einer Referenz zu einem anderen Wert wechseln. Dazu wendet man die „set reference“-/“set ref“-Funktion an. Als erster Parameter wird die Referenz angegeben und als Zweiter der neue Wert.

Beispiel:

```
x = 1
y = 2
a.ref = x
out a 'Gibt „1“ aus
set ref(a, y)
out a 'Gibt „2“ aus
```

Bool

Ein Bool besitzt nur zwei Werte. Diese sind "0" (für "True") und "1" (für "False"). Diese Klasse wird zur Auswertung von Aussagen und zur Aktivierung einer Option verwendet.

Bekannte abkürzungen: "b"

```
inverted.bool = this:invert
                     inv
```
Gibt den umgekehrten Wert des jeweiligen Bools zurück. D.h. "True" ergibt "False" und "False" ergibt "True".

Beispiel:
```
state.bool = 1
out state        'Gibt "True" aus
out state:invert 'Gibt "False" aus
```

```
this:invert!
     inv!
```
Kehrt den Wert des Bools direkt um. Es wird also keine Kopie angelegt.

```
value.reference = this:select(value one, value two)
                        sel
```
Ist der Wert des jeweiligen Bools "True", so wird eine Referenz auf "value one" zurückgegeben. Andersfalls wird eine Referenz auf "value two" zurückgegeben.

Beispiel:
```
state.bool = 1
out state:select("One", "Two") 'Gibt "One" aus
state:invert!
out state:select("One", "Two") 'Gibt "Two" aus
```

```
#true, #false
```
Beide Konstanten sind als Bool definiert. "#true" besitzt den Wert 1 und "#false" den Wert 0.

Queue

Queue ist eine Containerklasse, die eine unbestimmte Anzahl an Werten besitzt. Man kann zur Laufzeit Werte einfügen, modifizieren und löschen.

Definition

Queues können als Literale durch Eckige Klammern definiert werden. Für die normale Definition können Anfangswerte angegeben werden.

Beispiel:

```
[11, 4, 8]              'Lieterale Queue mit drei Elementen
my queue.queue(9, 12) 'Neue Queue mit zwei Elementen
```

`this:append(...)`

Fügt die angegebenen Werten an das Ende der Queue ein. Dabei werden die Werte vorerst kopiert. Für das Hinzufügen von Referenzen siehe nächster Befehl.

Beispiel:

```
number.queue(1, 2)
out number 'Gibt [1, 2] aus
number:append(3, 4)
out number 'Gibt [1, 2, 3, 4] aus
```

`this:appendReference(...)`
`     appendRef`

Fügt Referenzen der angegebenen Werte an das Ende der Queue ein. Dadurch ist es möglich eine Selektion aus Werten zu bilden, ohne diese zu kopieren.

Beispiel:

```
selection.queue
x = y = z = 0
selection:appendReference(x, z)
selection = 1
out x, y, z 'Gibt 1, 0, 1 aus
```

`this:appendValue(class*, ...)`

Fügt einen Wert der angegebenen Klasse an das Ende der Queue ein. Die nachfolgenden Parameter werden als Parameter für die Sofortdefinition genutzt.

Beispiel:

```
number.queue(1, 2)
number:append value(integer, 3)
out number 'Gibt [1, 2, 3] aus
```

`this:prepend(...)`

Fügt die angegebenen Werte an den Anfang der Queue ein. Die Werte werden vorerst kopiert.

`this:prependReference(...)`
`prependRef`

Fügt Referenzen der angegebenen Werte an den Anfang der Queue ein.

`this:prependValue(class*, ...)`

Fügt einen Wert der angegebenen Klasse an den Anfang der Queue ein. Die nachfolgenden Parameter werden als Parameter für die Sofortdefinition genutzt.

`this:remove(...)`

Löscht alle Elemente, die an den angegebenen Positionen stehen.

Beispiel:

```
friend.queue("Max", "Nico", "Nick", "Leo")
friend:remove(1, 2)
out friend 'Gibt [Max, Leo] aus
```

Tipps und Tricks

Literale sind „Writeable“?

Folgender Beispiel-Code wird ausgeführt:

```
is it true?(x){
     return („Yes“) if(x) else(„No“)
}

out is it true?(1) 'Gibt „Yes“ aus
out is it true?(0) 'Gibt „No“ aus
```

Bis jetzt ist noch alles klar. Der Code definiert eine Funktion, die bei einem wahren Parameter „Yes“ zurückgibt und bei einem Falschen „No“.

Nun wird folgendes gemacht, was auf den ersten Blick unmöglich scheint:

```
is it true?(1) = „Yes, it is!“
is it true?(0) = „No, it isn't!“
```

Und hier der Beweis, dass der obere Code wirklich etwas bewirkt hat:

```
out is it true?(1) 'Gibt „Yes, it is!“ aus
out is it true?(0) 'Gibt „No, it isn't!“ aus
```

Erklärung

Der obere Beispiel-Code ist deswegen in Operno möglich, weil die Funktion aus Performance-Gründen die Werte „Yes“ und „No“ nicht bei der Rückgabe kopiert, sondern Referenzen auf diese Werte anlegt. Beide Werte liegen direkt im Token-Code und sind damit nicht temporär im Stack geladen. Die Referenzen bleiben also solange gültig, bis die Funktion gelöscht wird.
Wendet man nun den „=“-Operator an diese Referenzen an, so ändert man direkt die Werte, die in der Funktion verankert sind. Somit werden bei einem erneuten Aufruf die neuen Werte zurückgegeben.

Warnung

Diese Möglichkeit soll wirklich nur selten angewendet werden, weil die Verwendung später oft zu Verwirrungen führen kann. Um solche unerwarteten Veränderungen zu überprüfen, kann man den Code einer Funktion durch „out funktions name::block“ ausgeben.

Glossar

Operatoren

Hier sind alle Operatoren und deren Klassenmethoden aufgelistet. Die Rot gekennzeichneten Operatoren besitzen keine eigene Klassenmethode, sondern nur eine andere Art der Verarbeitung von anderen Klassenmethoden.

Zeichen	Bezeichnung	Klassenmethode
=	Zuweisung	`this:_set(other)`
+	Addition	`this:_add(other)`
-	Subtraktion	`this:_sub(other)`
*	Multiplikation	`this:_mul(other)`
/	Division	`this:_div(other)`
**	Potenzieren	`this:_pow(other)`
//	Radizieren	`this:_pow(1 / other)`
+=	Addition + Zuweisung	`this:_add!(other)`
-=	Subtraktion + Zuweisung	`this:_sub!(other)`
*=	Multiplikation + Zuweisung	`this:_mul!(other)`
/=	Division + Zuweisung	`this:_div!(other)`
==	Äquivalenz	`this:_eq(other)`
<>	Unäquivalenz	`not this:_eq(other)`
<	Kleiner	`this:_le(other)`
>	Größer	`this:_gr(other)`
<=, =>	Kleiner gleich	`this:_le(other) or this:_eq(other)`
>=, =>	Größer gleich	`this:_gr(other) or this_eq(other)`
++	Inkrement	`this:_inc()`
--	Dekrement	`this:_dec()`
<<	Umwandlung in leserliche Form	`this:_form()`

Klassenmethoden

Klassenmethode	Beschreibung
`this:_new()`	Konstruktor
`this:_def(...)`	Sofort-Definition
`this:_del()`	Dekonstruktor
`this:_true?()`	Auswertung für die Wahrhaftigkeit einer Aussage
`this:_set(other)`	Zuweisung
`this:_fSet(friend)`	
`this:_add(other)`	Addition
`this:_fAdd(friend)`	
`this:_sub(other)`	Subtraktion
`this:_neg()`	Negativen Wert bestimmen
`this:_fSub(friend)`	
`this:_mul(other)`	Multiplikation
`this:_fMul(friend)`	
`this:_div(other)`	Division
`this:_fDiv(friend)`	
`this:_pow(other)`	Potenzieren
`this:_fPow(friend)`	
`this:_add!(other)`	Addition + Zuweisung
`this:_fAdd!(friend)`	
`this:_sub!(other)`	Subtraktion + Zuweisung
`this:_fSub!(friend)`	
`this:_mul!(other)`	Multiplikation + Zuweisung
`this:_fMul!(friend)`	
`this:_div!(other)`	Division + Zuweisung
`this:_fDiv!(friend)`	
`this:_eq(other)`	Äquivalenz
`this:_le(other)`	Kleiner
`this:_fLe(friend)`	
`this:_gr(other)`	Größer

`this:_fGr(friend)`	
`this:_inc()`	Inkrement
`this:_dec()`	Dekrement
`this:_form()`	Umwandlung in eine lesbare Form (als String)
`this:_args(...)`	Zugriff durch eckige Klammer
`this:_fileRead(file)`	Daten aus einer Datei auslesen
`this:_fileWrite(file)`	Daten in eine Datei schreiben

Reservierte Namen

Einige Namen sind speziell für Operno reserviert. Es ist nicht möglich Schlüsselwörter in eigene Namen einzubinden. Dabei gibt es aber Reservierte Wörter, die nur manchmal in eigenen Namen eingebunden werden können.

Schlüsselwörter

Schlüsselwort	Kurzbeschreibung
`define/def`	Für eine explizite Definition von Objekten
`defined?/def?`	Überprüfung, ob ein Objekt existiert
`delete/del`	Manuelle Zerstörung von Objekten
`pop`	Ermittlung eines Arguments
`if, else, elseIf`	If-Kontrollsturktur
`while`	While-Schleife
`repeat`	Wiederholung eines Blocks
`for`	For-Schleife für eine Reihe von Werten
`switch`	Switch-Kontrollstruktur für schnelle Abzweigungen
`and, or, xor, not`	Logische Verknüpfungen
`break`	Abbruch einer Schleife
`return`	Sofortige Rückgabe eines Wertes für eine Funktion
`output/out`	Ausgabe von Werten in der Konsole
`end`	Sofortige Beendung des jeweiligen Programms

Sonstige Wörter/Objekte

Diese Objekte bestehen nur in bestimmten Blöcken bzw. Schleifen. Man kann also die Namen dieser Objekte in manchen anderen Bereichen benutzen.

Schlüsselwort	Kurzbeschreibung und Bereich
`this`	Zeiger auf das klassifizierte Objekt in einer Klassenmethode
`current`	Referenz auf den aktuellen Wert einer For-Schleife
`case`	Der aktuelle Wert im Switch-Block
`self`	Referenz auf das Objekt der jeweiligen Funktion
`result`	Eine Referenz auf das Ergebnis einer Funktion