The NewtonScript Programming Language

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

11/95

Contents

Figures, Tables, and Listings

About This Book

The NewtonScript Programming Language is the definitive reference for anyone learning the NewtonScript programming language. If you are planning to begin developing applications for the Newton platform, you should read this book first. After you are familiar with the NewtonScript language you should read the *Newton Programmer's Guide* for implementation details and the *Newton Toolkit User's Guide* to learn how to install and use Newton Toolkit, which is the development environment for writing NewtonScript programs for the Newton platform.

About the Audience

This book is for programmers who have experience with high level programming languages, like C or Pascal, and who already understand object-oriented programming concepts.

If you are not familiar with the concepts of object-oriented programming there are many books available on the subject available at your local computer bookstore.

Related Books

This book is one in a set of books included with Newton Toolkit, the Newton development environment. You'll also need to refer to these other books in the set:

- *Newton Programmer's Guide: System Software*. This set of books is the definitive guide and reference for Newton programming topics other than communications.

- *Newton Programmer's Guide: Communications*. This book is the definitive guide and reference for Newton communications programming.
- *Newton Toolkit User's Guide*. This book introduces the Newton development environment and shows how to develop Newton applications using Newton Toolkit. You should read this book first if you are a new Newton application developer.
- *Newton Book Maker User's Guide*. This book describes how to use Newton Book Maker and Newton Toolkit to make Newton digital books and to add online help to Newton applications. You have this book only if you purchased the Newton Toolkit package that includes Book Maker.
- *Newton 2.0 User Interface Guidelines*. This book contains guidelines to help you design Newton applications that optimize the interaction between people and Newton devices.

Sample Code

The Newton Toolkit product includes many sample code projects. You can examine these samples, learn from them, experiment with them, and use them as a starting point for your own applications. These sample code projects illustrate most of the topics covered in this book. They are an invaluable resource for understanding the topics discussed in this book and for making your journey into the world of Newton programming an easier one.

The Newton Developer Technical Support team continually revises the existing samples and creates new sample code. You can find the latest collection of sample code in the Newton developer area on eWorld. You can gain access to the sample code by

participating in the Newton developer support program. For information about how to contact Apple regarding the Newton developer support program, see the section "Developer Products and Support," on page xv.

Conventions Used in This Book

This book uses various conventions to present information.

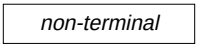
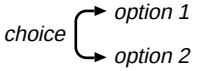
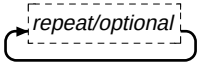
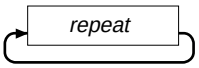
Special Fonts in Text

The following special fonts are used:

- **Boldface**. Key terms and concepts appear in boldface on first use. These terms are also defined in the Glossary.
- `Courier typeface`. Code listings, code snippets, and special identifiers in the text such as predefined system frame names, slot names, function names, method names, symbols, and constants are shown in the Courier typeface to distinguish them from regular body text. If you are programming, items that appear in Courier should be typed exactly as shown.
- *Italic typeface*. Italic typeface is used in code to indicate replaceable items, such as the names of function parameters, which you must replace with your own names. The names of other books are also shown in italic type, and *rarely*, this style is used for emphasis.

Syntax Conventions

In this manual, syntax is presented in two formats, as an extended BNF, and as bubble diagrams defined as follows:

Bubble Diagram	Extended BNF	Description
	`terminal`	Oval boxes / courier text indicates a word or character that must appear exactly as shown. Ambiguous terminal characters are enclosed in single quotes ('').
	*nonterminal*	Rectangular boxes / italics indicate a word that is defined further.
	[]	Dashed lines / brackets indicate that the enclosed item is optional.
	{choose | one}	Forked arrows / a group of words, separated by vertical bars (|) and grouped with curly brackets, indicates an either/or choice.
	[]*	A dashed box with a repeating arrow / an asterik (*) indicates that the preceding item(s), which is enclosed in square brackets, can be repeated zero or more times.
	[]+	A solid box with a repeating arrow / a plus sign (+) indicates that the preceding item(s), which is enclosed in square brackets, can be repeated one or more times.

Developer Products and Support

APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications for Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order product or to request a complimentary copy of the *Apple Developer Catalog*:

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

Undocumented System Software Objects

When browsing in the NTK Inspector window, you may see functions, methods, and data objects that are not documented in this book. Undocumented functions, methods, and data objects are not supported, nor are they guaranteed to work in future Newton devices. Using them may produce undesirable effects on current and future Newton devices.

CHAPTER 1

Overview

NewtonScript is a state-of-the-art, dynamic, object-oriented programming language, developed for the Newton platform.

Introduction

The goal of NewtonScript is to enable developers to create fast, smart applications easily. This calls for a language that is

- expressive, flexible, and straightforward to use
- consistent enough to allow reuse of concepts and structures
- portable enough to permit exploration of different architectures, and
- sufficiently compact to work with limited RAM

The constraints of the Newton system require a language capable of producing reusable code libraries, which uses memory efficiently, and collects garbage automatically.

NewtonScript is based on principles first used in Smalltalk and LISP, and was also influenced by Self, a language developed at Stanford University.

Semantic Overview

This section briefly introduces some special features of the NewtonScript language.

Expressions

NewtonScript is an expression-based language, rather than statement-based, as many other programming languages are. Almost everything in NewtonScript returns a value. Therefore, we talk about expressions rather than statements or commands in this manual.

The Object Model

NewtonScript is built on an object model. All data is stored as **objects,** or typed pieces of data. This differs from other object-oriented languages like C++ or Object Pascal, where data is a hybrid of objects and regular data types.

NewtonScript also differs from Smalltalk, although, like Smalltalk, it represents all data as objects. Only one kind of NewtonScript object, the frame**,** can receive messages**.**

The Newton object model structures data by using two kinds of 32-bit values to represent objects. These values are

- **immediates**—in which the 32 bits contain immutable data
- **references**—in which the 32 bits refer indirectly to an object

This is explained in greater detail in the section “Immediate and Reference Values” beginning on page 2-5 of Chapter 2, “Objects, Expressions, and Operators.”

Data Types and Classes

NewtonScript uses a **class** as a semantic type as opposed to a typical data type. The Newton platform uses classes to let parts of the system, like the Intelligent Assistant (which is described in the *Newton Programmer's Guide*) determine properties of the object at run time. Thus it can treat particular types of objects in interesting and different ways.

For instance, you can set up a data object containing personal data as shown in Listing 1-1. Note that the curly braces surrounding Listing 1-1 denote a frame object, that contains places, or slots, for objects that have the identifiers `name`, `company`, and `phones`.

Listing 1-1 A simple frame

```
{  name:    "Walter Smith",
   company: "Apple Computer",
   phones:  ["408-996-1010", "408-555-1234"] }
```

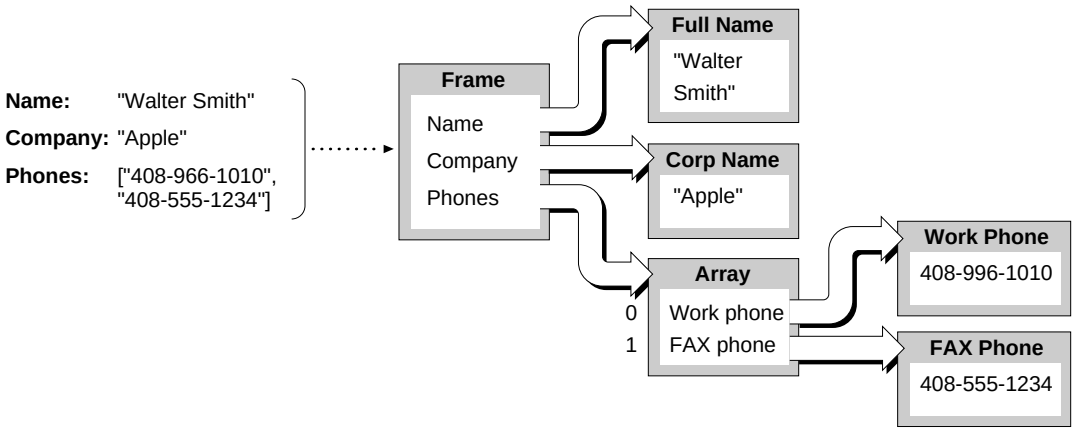
In the Newton system, objects can be typed. For instance, the values of `name` and `company` are plain strings. However, you can further define `phones` as being of type `workPhone` and `faxPhone`. These user-defined objects can then be manipulated by the Newton system in different ways. For instance, when the person using your application uses a fax phone number, a set of actions different from those for a work phone number is initiated.

The data object constructed in Listing 1-1 is shown in Figure 1-1.

The facility that lets the system know about an object's type is known as latent typing. Types are associated with objects. This means that a variable can hold any kind of object and can hold different types of objects at different times.

The class system is explained in further detail in "Objects and the Class System" beginning on page 2-1.

Figure 1-1 A sample data structure



Note

Smalltalk enthusiasts should keep in mind that NewtonScript classes have nothing in common with those used in class-based programming in a language such as Smalltalk. You can, however, use class-based programming concepts to organize parts of your NewtonScript application. For more information about this see Appendix C, "Class-Based Programming." ◆

Scope

The part of a program within which a variable can be used is called the scope of the variable. Normally a variable is available within the function where it is defined, although slots that are used like variables can also be inherited from proto and parent frames. See the section "Frame" beginning on page 2-17, and Chapter 5, "Inheritance and Lookup," for more information about frames and **inheritance,** respectively.

When looking up the value of a variable, NewtonScript first searches **local** variables, then **global** variables, and finally inherited variables, through the proto and parent chains.

Consider, for example, the following code segment:

```
aFrame:= {foo: 10,
         bar: func(x)
         begin
            if foo then Print ("hello");
            if x > 0 then
               begin
                  local foo; //local variable to function
                  foo:= 42;
               end
            return foo;
         end;
      }
```

Here the local variable `foo` is restricted in scope to the function definition `bar`. Even though `foo` is declared within a `begin ... end` code segment, its scope is not confined by that construct. When `foo` is used in the expression,

```
if foo then Print ("hello");
```

before it is declared as a local variable, it is already defined by the compiler as an implicit local and is initialized to the value of `nil`.

If the slot variable, `foo`, with the value of 10, is to be used as the value for `foo`, that is stated explicitly as

```
if self.foo then Print ("hello");
```

Otherwise, the search for the value of the variable goes by the rules, first to local variables, (the local variable `foo` was initialized to `nil`), then to global variables named `foo`, and then to inherited slot variables named `foo`.

In this example, the compiler creates a local variable `foo`, which is local to the method `bar` and is not initialized until the `if` expression that assigns it the value 42 executes. Therefore, nothing is ever printed.

When you send the message, `bar`, with a parameter value of ten, to the `aFrame` object, in the expression

```
aFrame:bar(10)
```

a value of 42 is returned. The same message with a parameter value of negative five

```
aFrame:bar(-5)
```

returns `nil`. See the section "Function Invocations" in Chapter 4 to learn more about message sending.

Extent

The extent of a variable refers to the period of time in which it may be used. In many languages, the scope of a variable is the same thing as its extent. However, in NewtonScript a variable has data storage space allocated for it while it is referenced anywhere in executing code.

Automatic garbage collection occurs only after an object is no longer referenced anywhere. Therefore, storage allocated for data structures is available until no references to the structure exist.

Make sure not to leave any references to large data structures you've allocated after you're finished using them. If you do, NewtonScript will not reclaim the associated memory.

An example of where you might think about this on the Newton platform is when you close an application, you can set slots to `nil` in the application's base frame to conserve memory.

Garbage Collection

In NewtonScript, garbage collection—that is, reclaiming the storage of objects that are no longer used—is carried out automatically by the system. Thus, the programmer does not need to worry about memory management.

In fact, in NewtonScript it's impossible to have "dangling pointers," which often cause the most insidious and hard to find bugs in an application.

If you've had to do garbage disposal manually in another language you can relax; the Newton system reclaims memory for you sometime after the last reference to an object goes away. Setting the value of all slots and variables referring to an object to `nil` allows the Newton garbage collector to reclaim the memory used by the object.

Automatic garbage collection is triggered every time the system runs out of memory. There's not really any reason to invoke garbage collection manually. However, if you must do so, you can call the global function `GC`. For more information on this function, see the chapter "Debugging" in *The Newton Toolkit User's Guide.*

How Is NewtonScript Dynamic?

In general, the term "dynamic" refers to the ability of the language to change properties of objects at run time. Therefore, the NewtonScript dynamic model is useful when you want to change an object at run time. For instance, it's possible to change an object to another kind of object in response to a user's actions while the application is running, if needed.

You can also add new data to objects while an application is in use. For instance, you can write NewtonScript code that dynamically adds a new variable to an executing object at run time and uses it, then adds a method to the same object and uses it, and finally changes the inheritance structure of the object by adding a special reference to another object. (This "special reference" is what is denoted as `_parent` in Listing 1-2 on page 1-11). The object can now use a method it inherits from the parent frame.

All of these operations are impossible in a static language, and they require a great deal of thought and discipline in dynamic languages. Though this powerful feature enables you to interactively program in a way that is impossible in static languages, it should be used sparingly and with caution.

Basic Syntax

Rather than invent an entirely new syntax, NewtonScript was designed with Pascal in mind. Wherever possible, its syntax is modeled closely on Pascal's.

Semicolon Separators

The semicolon (`;`) is used to separate lines, not to terminate them. Though semicolons are not required at the end of a line, you may spread one expression over several lines or enter multiple expressions on a single line by using the semicolon.

Expressions can be entered in a free-form manner, but we recommend that standard indentation be used for enhanced readability, as in this example:

```
if expression then
      expression
else
      another_expression;
```

NewtonScript syntax allows you to use as much white space as you wish; it is ignored.

Note

If you forget to add an important semicolon at the end of a NewtonScript expression, the interpreter uses whatever is on the next line as it tries to interpret a larger statement than intended, thus causing unusual error reports. ◆

In-Line Object Syntax

NewtonScript has two syntax features that make it easy to create objects: object literals and object constructors.

The object literal syntax lets you put a complex object into your program as easily as an integer. This syntactic mode is entered by writing a single quote, as shown in this simple example of a frame containing two strings:

```
x := '{name: "xxx", phone: "yyy"};
```

The object is constructed at compile time, and a reference to the same object results each time the object literal is evaluated.

The object constructor syntax makes it easy to construct objects at run time. The syntax is similar to object literals, but without the quote. In the object constructor syntax the slot value positions are evaluated expressions rather than nested literals. Each time the constructor is evaluated, a new object is created, and its slots are filled in with the results of the slot expressions. An object constructor is much easier to read than the equivalent operation written without it, as shown in the following two examples.

First, the example with an object constructor:

```
x := {name: first && last, tax: wage * taxRate};
```

Next, the equivalent operation without the object constructor:

```
x := {};
x.name := first && last;
x.tax := wage * taxRate;
```

Character Set

NewtonScript uses the standard 7-bit ASCII character set rather than the enhanced ASCII character set used by Macintosh computers. This ensures that your code will work in any Newton development environment.

Comments

NewtonScript uses the same convention for delineating comments as the C++ programming language. Multiline comment text needs to be surrounded by an opening right slash, asterisk (`/*`) and a matching asterisk, right slash (`*/`) at the end. For example:

```
/* This is an example of
a comment. It can be on one line
or as many lines as you need. */
```

If your comment is short enough to keep to one line, you can use two back slashes (`//`) before the text to signal that the rest of the line is a comment and should be ignored. This is often useful for putting a comment on the same line as a line of code, as in this example:

```
x:= 5 ; //This is a single-line comment.
```

Note that nested multiline comments are not allowed. However, within a `/* ... */` comment block, comments using the `//` notation are allowed.

A Code Example

If you like to try to understand code before reading the manual, continue on through this section. Otherwise, stop here and go to the next chapter.

Note that in the code the curly brackets (`{}`) denote a frame object, colon-equal (`:=`) is the assignment operator, and the colon (`:`) is the message-send operator, which sends the message following it to the frame expression that appears before it.

The code shown in Listing 1-2 is partially described in the section “How Is NewtonScript Dynamic?” beginning on page 1-7.

Listing 1-2 A dynamic example

```
y := { YMethod: func () print("Y method"), yVar: 14 };
x := { Demo: func () begin
      self.newVar := 37;
      print(newVar);
      self.NewMethod := func () print("hello");
      self:NewMethod();
      self._parent := y;
      print(yVar);
      self:YMethod();
      end
};

x:Demo();

37
"hello"
14
"Y method"
#2        NIL
```

Compatibility

There are two main enhancements to the 2.0 version of the NewtonScript language:

- new subclassing mechanism
- native functions

The new subclassing mechanism is described in the section “Classes and Subclasses” on page 2-3. This new subclassing mechanism allows user-defined classes to have more precise semantic definitions, while preserving the logical structure of these categories.

Native functions, described in the section “Native Functions” on page 4-16, are executed directly by the Newton processor, instead of going through the interpreter. This can increase the speed of your functions, but can also slow them down.

In addition, two type identifiers have been added to the language to speed up processing of native functions: `int` and `array`. There are two places where these type identifiers can be used: in declaring local variables, and in the argument list of a function declaration. For information on their syntax, see the section “Local” on page 2-23, and “Function Constructor” on page 4-2. For a detailed discussion of native functions and the use of type identifiers, see the chapter “Tuning Performance,” of the *Newton Toolkit User’s Guide.*

The 2.0 version of the language also includes new built-in functions, for a list of these see the section “Compatibility” beginning on page 6-2 of Chapter 6, “Built-In Functions.”

CHAPTER 2

Objects, Expressions, and Operators

This chapter discusses objects, expressions, and operators.

Objects and the Class System

The semantic type of an object is identified by a **class**. The Newton object system has four built-in primitive classes which describe an object's basic type. They are:

- Immediate
- Binary
- Array
- Frame

You can determine the primitive class of an object, `obj`, by executing the expression `PrimClassOf(obj)`. Similarly, you can determine the class of an object, `obj`, by executing the expression `ClassOf(obj)`. A number of functions exist to check if an object is of a particular type, which are faster than `ClassOf` and `PrimClassOf`. These are `IsArray`, `IsFrame`, `IsInteger`, `IsSymbol`, `IsCharacter`, `IsReal`, and `IsString`. These and the other Newton built-in functions are documented in Chapter 6, "Built-In Functions."

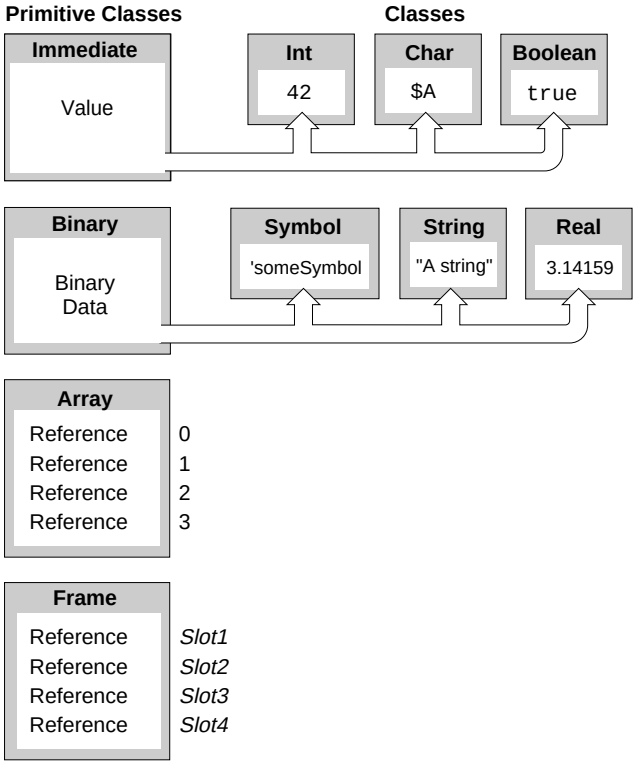
The primitive classes are of two categories: immediates and reference objects. The reference object category is composed of the **binary**, array, and frame classes. See "Immediate and Reference Values" beginning on page 2-5 for a more detailed discussion of the differences between these two categories of objects.

Objects with `Immediate` as their primitive class can be further identified as belonging to a class of `Int`, `Char`, or `Boolean`. System-defined objects with `Binary` as their primitive class can also be further identified as belonging to a class of `Symbol`, `String`, or `Real`. NewtonScript also allows user-defined classes for reference objects.

The NewtonScript class structure is shown in Figure 2-1

Classes function as semantic types that inform the system about the data in a reference object. For example, the class `'string` indicates a binary object containing a string and the class `'phoneNumber` indicates a string containing a phone number. With this knowledge, a Newton device could use phone numbers in ways it would not use other strings (to dial a phone, for instance.)

Figure 2-1 NewtonScript built-in classes



Classes and Subclasses

NewtonScript provides the `SetClass(`*obj*, *classSymbol*`)` function to assign a class, *classSymbol*, to a reference object, *obj*. Arrays and frames also have internal mechanisms for setting user-defined classes, but for binary objects the `SetClass` function must be used. These mechanisms are described in "Array" beginning on page 2-15, and "Frame" beginning on page 2-17.

Class symbols are arranged in a hierarchy; that is, some classes have subclasses. This allows objects to have more precise semantic definitions, while preserving the logical structure of these definitions.

To create a subclass add a period (.) and a symbol to the class name. For example, `'|rectangle.square|` is a subclass of `'rectangle`. Thus, a class symbol X is a subclass of a class symbol Y if either X is the same as Y, or Y is a prefix of X at a period (`.`) boundary. Everything is a subclass of the empty symbol `||`.

The symbol added after the period cannot, of course, itself contain a period (`.`). And neither symbol can contain a semicolon (`;`), which is reserved for future expansion. Furthermore, a class symbol that contains periods (`.`), must be surrounded by vertical bars (`|`), this is required by the syntax of a symbol which is described in "Symbol" beginning on page 2-12.

You can use the built-in function `IsSubclass(x, y)` to determine if `x` is a subclass of `y`. You can determine if an object `obj` is of class `x` or any subclass of `x` by using the function `IsInstance(obj, x)`. Note that this is just shorthand for `IsSubclass(ClassOf(obj), x)`. For more information about these functions see Chapter 6, "Built-In Functions."

Note

The period method of creating subclasses is new to NewtonScript; it is not supported by the NewtonScript interpreter on 1.x Newton devices. If your application might be run on a 1.x machine, do not use this mechanism. ◆

Adding classes to objects increases the complexity of your application. If you do not need to, do not add a class to your objects.

Note also that the there is no subclass relationship in the sense being discussed in this section between the built-in primitive classes and those classes that are derived from them. `String`, for example, is not a subclass of `Binary`.

The only class whose subclasses are important to the NewtonScript built-in functions is `'string`. There are several string-manipulation functions that require their arguments to have either the class `'string` or a subclass of `'string`.

A number of class symbols are automatically understood by the Newton system to be subclasses of `'string`. These are: `'company`, `'address`, `'title`, `'name`, `'phone`, `'homePhone`, `'workPhone`, `'faxPhone`, `'otherPhone`, `'carPhone`, `'beeperPhone`, and `'mobilePhone`.

To create other subclasses of `'string`, e.g. `'firstName`, define the class explicitly as `'|string.firstName|`.

Immediate and Reference Values

In NewtonScript, values are stored in 32 bits, two of which are used for class information. **Immediate** objects (integers, characters, and Booleans) contain their values within the remaining 30 bits. **Reference** objects, (binaries, arrays, and frames) on the other hand, contain a reference to the area of memory where their data resides.

This is an important distinction to keep in mind when assigning values to a variable. When a variable is assigned an immediate object, that object is copied directly into the variable. When a variable is assigned a reference object, on the other hand, only a reference to the object is copied in.

The behavior caused by this can be somewhat confusing. Consider, for example the following code fragment:

```
local a := {x: 1, y: 3};
local b := a;
a.x := 2;
// at this point b.x = 2
```

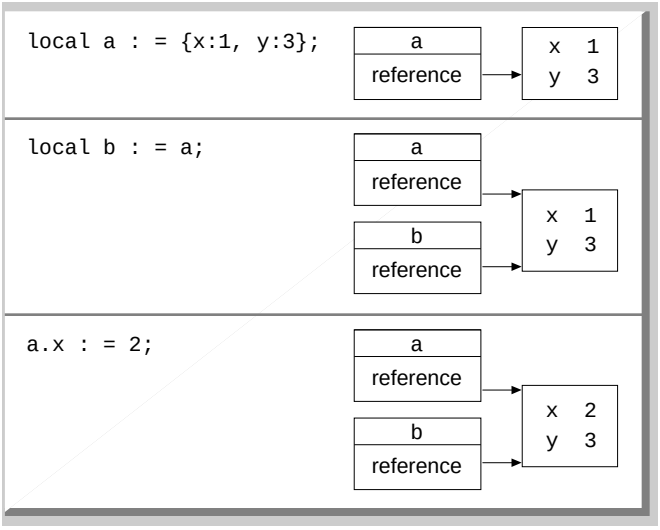
The first line declares a local variable, `a`, and assigns to it a frame with two slots – named `x` and `y` – whose values are 1 and 3, respectively.

The second line creates another local variable, `b`, and assigns to it the value of `a`, which is a reference to the frame object created in the first line. Thus both local variables `a` and `b` now refer to the same are of memory.

The third line (`a.x := 2;`) changes the value of the `x` slot. Since both variables `a` and `b` refer to the same frame, the value of `b.x` now also equals

2, though it was not explicitly assigned. You can see these results in Figure 2-2.

Figure 2-2 NewtonScript code sample

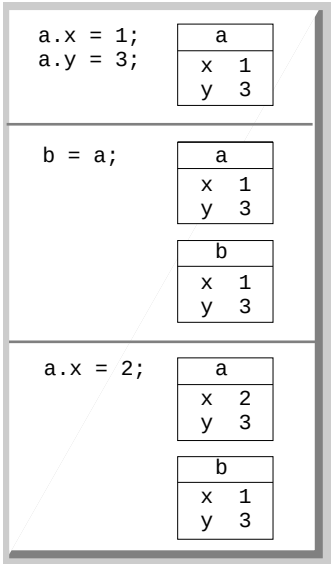


Consider now the C code:

```
struct foo {
            int x,y;
          };
foo a;
foo b;
a.x = 1;
a.y = 3;
b = a;
a.x = 2;
// at this point b.x = 1
```

In this example there are two separate `struct` objects, `a` and `b`, residing in separate areas of memory. Each `struct` contains two integers, `x` and `y`. The integer `a.x` is assigned the value 1. The value of `struct a` is saved in `struct b` and `a.x` then is set to 2. As you would expect, the value of `b.x` is unchanged at this point (it is still 1.) The results of this code example are shown in Figure 2-3.

Figure 2-3 C code sample



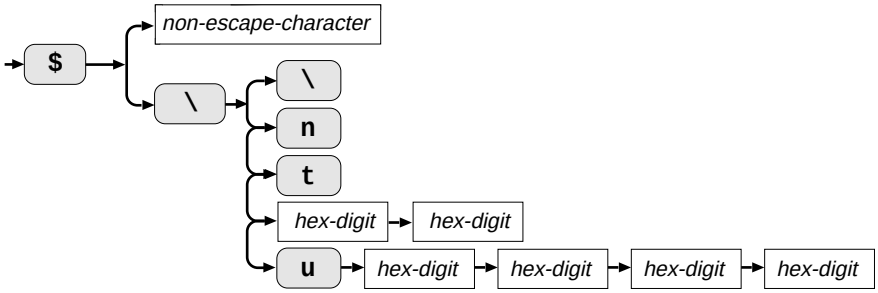
Note that assignments of references in NewtonScript are handled in the same manner as assignments of arrays and strings in C, since arrays and strings in C are pointers. Thus NewtonScript does not fundamentally differ from C in this respect.

The NewtonScript Objects

This section individually discusses the objects listed in "NewtonScript built-in classes" on page 2-3: characters, Booleans, integers, reals, symbols, strings, arrays, and frames.

Character

$ { *nonEscapeCharacter* | \ { \ | n | t | *hexDigit hexDigit* |
u *hexDigit hexDigit hexDigit hexDigit* } }



Characters in the standard character set are specified in your code by the dollar sign ($) and

- a backslash escape character (\) followed by a special character specification such as, \, n, and t, or by 2 hexadecimal digits
- a backslash escape character (\) followed by u (for Unicode) and four hexadecimal digits
- a non-escape character

The character set in Newton is stored as Unicode, in two bytes, to facilitate international conversions. By design, the first 128 characters match the ASCII character set. You must use Unicode character codes to specify special characters other than the ASCII character set.

Characters are immediate objects. (For more information about immediate objects see “Immediate and Reference Values” beginning on page 2-5.)

nonEscapeCharacter	Consists of any ASCII character with code 32–127 except the back slash (\).
hexDigit	Consists of: {`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `a` | `b` | `c` | `d` | `e` | `f` | `A` | `B` | `C` | `D` | `E` | `F`}

For example, `$a` or `$7` represent the characters “a” and “7”, respectively.

Special characters like “π” must be specified as Unicode (16-bit) characters by using the four-digit hex character code preceded by `$\u`. For example, the Unicode equivalent of “π” is: `$\u03C0`.

You specify a new line by imbedding the code `$\n` in a string. Special character codes are summarized in Table 2-1.

Table 2-1 Characters with special meanings

Character code	Meaning
`$\n`	newline character
`$\t`	tab character
`$\\`	backslash character
`$\` *hexDigit hexDigit*	hexadecimal
`$\u` *hexDigit hexDigit hexDigit hexDigit*	Unicode

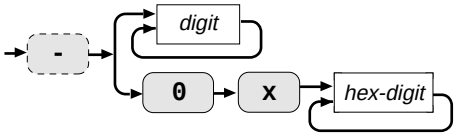
See Appendix B, “Special Character Codes,” for a list of the special characters and their Unicode equivalents.

Boolean

NewtonScript defines only one **Boolean** constant, `true`. Functions and control structures use `nil` as false and anything else as true. When you don't have anything else to use as true, use the special immediate `true`.

Integer

[-] {[*digit*]+ | 0x [*hexDigit*]+}



All integers in NewtonScript can be written in either decimal or hexadecimal. When a digit is prefixed with zero and the letter x (0x) it signifies a hexadecimal value. The optional minus sign (-) before a digit signifies a negative integer. Here are some examples of integers:

```
13475     -86    0x56a
```

Integers range from 536870911 through -536870912. When that limit is exceeded behavior is undefined.

Integers are immediate objects. (For more information about immediate objects see "Immediate and Reference Values" beginning on page 2-5.)

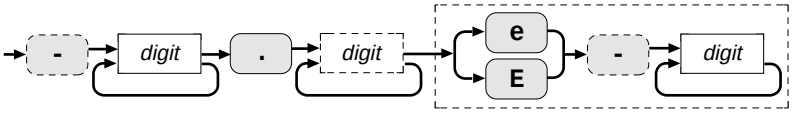
digit	Consists of: {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
hexDigit	Consists of: {*digit* | a | b | c | d | e | f | A | B | C | D | E | F}

Note

The integer -536870912 can't be specified as a literal but it can be computed. ◆

Real

[-] [*digit*]+.[*digit*]* [{ e | E } [-] [*digit*]+



A real number consists of one or more digits followed by a decimal point with zero or more additional digits. The optional minus (-) at the start indicates a negative number. You can specify scientific notation by placing the letter e (upper or lower case) directly after the last digit and following it with a negative or positive digit in the range of -308 to +308.

digit Consists of: {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}

NewtonScript floating point real numbers are represented internally in double precision; 64 bits. They have approximately 15 decimal digits of precision. Some examples of real numbers include:

```
-0.587 123.9 3.141592653589
```

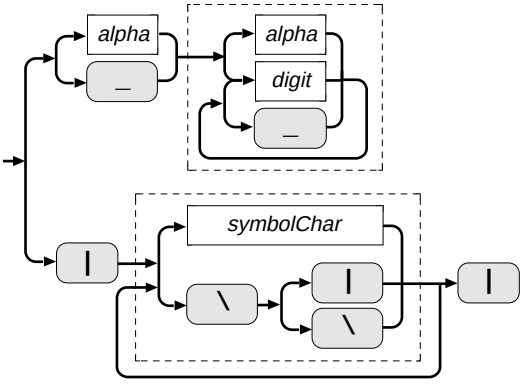
Here are some examples of exponential notation used to represent real numbers:

```
763.112e4   87.3789E-45    -34.2e6     69.e-5
```

Real numbers are stored as binary objects and have the class `Real`.

Symbol

{ { *alpha* | _ } [{ *alpha* | *digit* | _ }]* |
'|' [{ *symbolChar* | \ { '|' | \ }]* '|' }



A symbol is an object used as an identifier. NewtonScript uses symbols to name variables, classes, messages, and frame slots. You can also use symbols as simple identifying values, as you would use enumerated types in other languages.

Symbol names may be up to 254 characters long and may include any printable ASCII character; for instance, `|Weird%Symbol!|` is valid. A symbol can be written by itself, without being enclosed in vertical bars, if it begins with an alphabetic character or an underscore and contains only alphabetic characters, underscores, and digits. NewtonScript is case insensitive, though it preserves case.

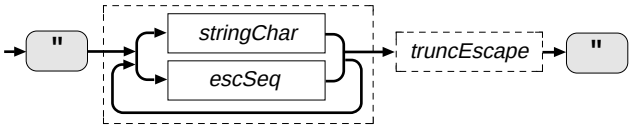
alpha	Consists of: {A–Z and a–z}.
digit	Consists of: {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}.
symbolChar	Consists of any ASCII character with code 32–127 except | or \.

One place where the Newton system requires symbols is in exception handling. An example of an exception symbol is: `|evt.ex.fr.intrp|` Note that vertical bars are required because of the dots in the symbol. You can read more about them in “Defining Exceptions” beginning on page 3-15.

Symbols appearing in expressions are normally evaluated as variable references. You can prevent this by preceding the symbol with a single quote (`'`). The quoted symbol evaluates to the symbol itself. See also “Quoted Constant” beginning on page 2-28.

String

" [{ *stringChar* | *escSeq* }]* [*truncEscape*]] "



A string constant is written as a sequence of characters enclosed in double-quotation marks.

stringChar — Consists of a tab character or any ASCII character with code 32–127 except the double quote (") or backslash (\).

escSeq — Consists of either a special character specification sequence or a unicode specification sequence.The special character specification sequence is: backslash (\) followed by a quote ("), backslash (\), the letter `n` or the letter `t`. The escape sequence for specifying Unicode begins with backslash-u (`\u`), is followed by any number of groups of four *hexDigits*, and ends with backslash-u (`\u`).

truncEscape — Consists of the shortened unicode specification sequence. It is: backslash-u (`\u`), is followed by any number of groups of four *hexDigits*.

Here are some simple examples of strings:

```
"pqr"    "Now is the time"    ""
```

Within strings you can include Unicode characters that are not in the standard character set by inserting the escape code, `\u`, to toggle on the Unicode hex mode. Follow the `\u` with any number of groups of four-

number codes specifying the special character. You can add another `\u` to toggle the Unicode hex mode off and return to the regular character set, though, you are not required to toggle the hex mode off. (See Appendix B, "Special Character Codes," for a list of these characters.)

For example, you could specify the French phrase, "Garçon, l'addition, s'il vous plaît!", by embedding Unicode in the string to specify the special characters as follows:

```
"Gar\u00e7\uon, l'addition, s'il vous pla\u00ee\ut!"
```

Other codes you use within strings to specify special characters are summarized in Table 2-2.

You can also use array accessor syntax to refer to a character in a string. See "Array Accessor" beginning on page 2-16 for more information. For example, you can define a string

```
aString := "ABCDE";
```

and then refer to the `B` by using an array accessor

```
aLetter := aString[1];
```

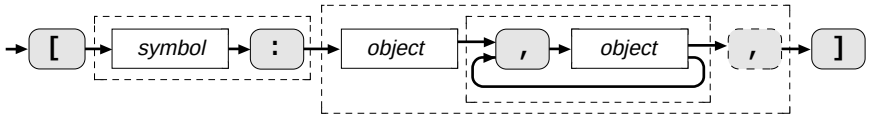
Note that the index to `aString` is one, because array indices are numbered beginning with zero.

Table 2-2 Codes for specifying special characters within strings

Character code	Meaning
`\n`	newline character
`\t`	tab character
`\u`	toggles Unicode on and off
`\\`	backslash character
`\"`	double quote character

Array

'[' [*symbol* :] [*object* [, *object*]* [,]] ']'



An **array** is a collection of zero or more objects enclosed in square brackets ([]) and separated by commas. The user-defined class for the array may optionally be specified by beginning the array with a symbol followed by a colon (:).

symbol — Consists of an identifier that is syntactically a symbol. If present, sets a user-specified class for the array. See the section "Symbol" beginning on page 2-12, for more information about their syntax.

object — May consist of any NewtonScript object. NewtonScript numbers the elements in an array by beginning with zero for the first element. Objects are separated by commas if there are more than one in the array.

Note

The syntax [*symbol*: *object* (…)] is ambiguous; the first symbol could be either a class for an array, or a variable to be used as the receiver for a message send. NewtonScript uses the first interpretation. (Message sends are described inChapter 4, "Functions and Methods.") ◆

Semantically, the array is an ordered collection of objects which are indexed numerically, beginning with zero. As with frames, the array can hold any type of object, including methods, frames, and arrays. Like other non-immediate objects, arrays can have a user-specified class, and can have their size changed dynamically.

Here is a simple example of an array literal:

```
[1,2,3]
```

You can specify a class name for an array by preceding the first array element with any arbitrary identifier that specifies a class and a trailing colon, as shown here:

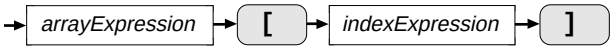
```
[RandomData: [1,2,3], 0, "Last element"]
```

Note that this array, which has the class `RandomData`, holds a mixture of objects. It contains another array as its first element, the integer zero as the second element, and a string as the third element.

Note
NewtonScript allows an optional trailing comma after the last array element. The trailing comma can be useful if you are going to add more elements to the array, or move elements around within an array, when editing source code. The presence or absence of this comma, does not affect the program. ◆

Array Accessor

arrayExpression '[' *indexEpression* ']'



Array elements are accessed with an expression, that evaluates to an array, and an index expression, that evaluates to an integer and is enclosed in square brackets.

arrayExpression	An expression that evaluates to an array.
indexEpression	An expression that evaluates to an integer. The *indexExpression* corresponds to the element of the array you wish to access. Note that arrays are indexed starting with zero.

For example, with the array `myArray`, which is defined as

```
myArray := [123, [4,5,6], "Alice's Restaurant"];
```

you access the second element in the array by using the expression:

```
myArray[1];
```

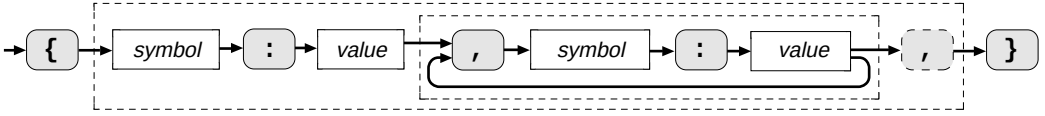
This expression evaluates to `[4,5,6]`.

You can also access array elements by using a path expression. Read about these in the section "Path Expression" beginning on page 2-20.

Note that array accessors are actually operators and are included here for your convenience. The rest of the NewtonScript operators are documented in "Operators" beginning on page 2-29.

Frame

'{' [*symbol* : *value* [, *symbol* : *value*]* [,]] '}'



A **frame** is a collection of zero or more slots enclosed in curly brackets and separated by commas. A **slot** consists of a symbol followed by a colon (`:`) and a slot expression. The symbol refers to the value of the slot expression.

symbol — A symbol giving the name of the slot. Note that slot symbols beginning with the underscore character (_) are reserved for system use; do not begin your slot symbol with the underscore character.

value — Can be any object, including another frame or a method.

A frame is an unordered collection of slots which consist of a name and value pair. As with arrays, the value of a slot can be any type of object, including methods and even other frames and arrays.

Frames can be used as repositories for data, like records in Pascal and structs in C, but can also be used as objects which respond to messages. As such, the frame is the basic programming unit in NewtonScript.

A simple record-like frame containing names and phone numbers might appear as

```
{name:"Joe Bob", phone:"4-5678", employee:12345}
```

Here is an example frame that contains integers in the first three slots, and a method in the fourth slot:

```
Jupiter := {size:491,
            distance:8110,
            speed: 34,
            position: func(foo) speed*foo/3.1416}
```

You may specify an optional class name for a frame by using the slot name `class`. Inserting the class slot:

```
class : 'planet
```

into the Jupiter frame gives it an appropriate class name. Just as for array classes, the class of a frame gives it a type, not special properties. However, if you wanted to give all objects of class `planet` special characteristics and functionality you could use the NewtonScript inheritance structure to set up relationships that allow objects to inherit data from other objects. (See Chapter 5, “Inheritance and Lookup.”)

You specify these relationships to other frames by referencing them from slots named `_proto` and `_parent`. The relationships these slots establish allow you to take advantage of the NewtonScript double inheritance scheme to construct object-oriented applications. Chapter 5, “Inheritance and Lookup,” describes these concepts.

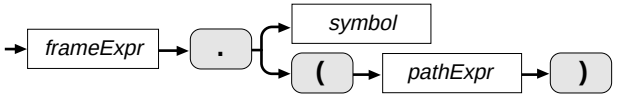
There are several slot names which are recognized by the system and used for special purposes. All these slots are optional. They are described in Table 2-3.

Table 2-3 Special slot names and their specifications

Slot Name	Specification
`class`: *identifier*	You use the special slot name `class` to specify a semantic type for your frame. The `class` of your object must be a symbol.
`_parent`: *frame*	You use the special slot name `_parent` to designate another *frame* as a parent frame to this frame. You can repeat this process, as necessary with other frames, to construct a parent inheritance chain. For information about inheritance see Chapter 5, "Inheritance and Lookup."
`_proto`: *frame*	You use the special slot name `_proto` to designate another *frame* as a prototype frame to this frame. Repeat this process, as necessary with other frames, to construct a prototype inheritance chain. For information about inheritance see Chapter 5, "Inheritance and Lookup."

Frame Accessor

frameExpr `.` {*symbol* | `(` *pathExpr* `)`}



Frame values are accessed with an expression–that evaluates to a frame–and either a symbol, or an expression enclosed in parentheses, that evaluates to a path expression.

A frame accessor expression returns the contents of the specified slot, or if the slot does not exist the expression evaluates to `nil`.

frameExpr Any expression that evaluates to a frame.

symbol A symbol reference to a slot. See "Symbol" beginning on page 2-12 for the syntax of symbols.

pathExpr Any expression that evaluates to a path expression object. The *pathExpr* corresponds to the slot of the frame you wish to access. Note that arrays are indexed starting with zero. See also "Path Expression" on page 2-20.

Slots in a particular frame can be accessed by using a dot (`.`) followed by a symbol. For example, the expression:

```
myFrame.name;
```

evaluates to the contents of the `name` slot from the frame referenced by the variable `myFrame`.

If the slot is not found in the specified frame using this syntax, the search for the slot continues through the inheritance chain. The next place NewtonScript looks is in the prototype frame and then in any of the prototype frame's prototypes until the end of the prototype chain is reached. If the slot is not found, the search stops; it does not continue up through the parent frames. If the slot does not exist the expression evaluates to `nil`.

The built-in functions `GetVariable` and `GetSlot` provide similar kinds of slot access but with different inheritance behavior. For more information see Chapter 6, "Built-In Functions."

For more information about the inheritance mechanism see Chapter 5, "Inheritance and Lookup."

Note that frame accessors are actually operators; they are included here for your convenience. The rest of the NewtonScript operators are documented in "Operators" beginning on page 2-29.

You can also access frame slots by using a path expression. Read about these in the section "Path Expression."

Path Expression

A **path expression** object encapsulates an access path through a set of objects. These objects are necessarily arrays or frames, since these are the only objects in NewtonScript that can contain other objects.

A path expression can take one of three forms:

- an integer
- a symbol
- an array of class `pathExpr`

A path expression which is an integer necessarily refers to an array element, since frame slot names must be symbols. The following code sample shows how an integer path expression can be used to refer to an array element.

```
anArray := ["zero", "one", "two"];
aPathExpression := 1;
anArray.(aPathExpression);
"one"
```

Similarly, a symbol path expression necessarily refers to a frame slot, as in this code fragment:

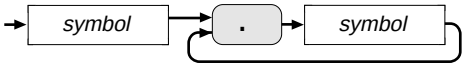
```
aFrame := {name: "Fred", height: 6.0, weight: 150};
aPathExpression := 'height;
aFrame.(aPathExpression);
6.0
```

The third kind of path expression can refer to any object, whether it is nested in arrays, frames, or both. The following code sample shows how a path expression can encapsulate an access path to an object within both arrays and frames:

```
myFrame:={name:"Matt",info:{grade:"B",tests:[87,96,73]}};
myPath:='[pathExpr: info,tests,1];
myFrame.(myPath);
96
```

If a path expression consists entirely of symbols, then the following syntax can be used:

symbol[.*symbol*]+



symbol Any valid NewtonScript symbol.

This syntax will actually create an array of class `pathExpr`, and a path expression written in this syntax will be printed out in the Inspector as an array of class `pathExpr`. The following code sample illustrates how this syntax is used.

```
myFrame := {kind:"Cat",type:{hair:"Long",color:"Black"}};
myPath := 'type.color;
myFrame.(myPath);
"Black"
```

Note that you can also use path expressions to set the value of a slot. For instance, to change the color of the cat, use an expression like:

```
myFrame.(myPath):= "White";
```

Expressions

A simple expression consists of values and an operator, as shown in the code:

```
12 + 3;
```

The values (`12` and `3`) and the infix operator plus (`+`), that appears in the line between them, are evaluated to return the value `15`.

Variables are often used in expressions as named containers in which to store values. For example, you can use a variable on the left side of an assignment expression, as in

```
currentScore := 12;
```

to store a value. The variable `currentScore` then becomes the identifier for the value. For more information about the assignment operator (`:=`), see the section "Operators," in this chapter.

Variables

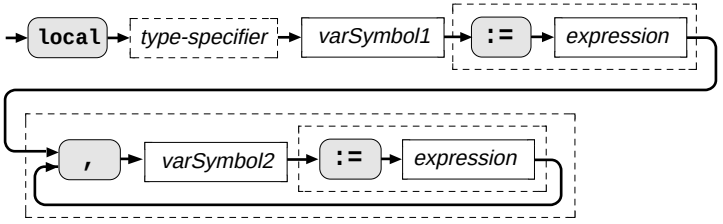
A variable is named by a symbol. You can use this symbol to refer to any kind of value; from numbers to frames.

When a method is executing and a variable reference is encountered, the system evaluates the value of the variable. This is done following the variable lookup rules of NewtonScript's double inheritance scheme. Variable lookup is discussed in Chapter 5, "Inheritance and Lookup."

The next section discusses the use of the `local` keyword to declare local variables.

Local

`local` [*typeIdentifier*]*varSymbol1* [`:=` *expression*]
[, *varSymbol2* [`:=` *expression*]]*



The **local** declaration consists of the keyword `local`, and any number of initialization clauses – an optional type identifier, a symbol, and optionally an assignment operator (`:=`), followed by an expression.

varSymbol	Consists of an identifier that is, syntactically, a symbol. The symbol names a variable that may be initialized with the optional expression. For more information on symbols see the section "Symbol" beginning on page 2-12.
typeIdentifier	Either of the keywords `int` or `array`. It is important to include a *typeIdentifier* when declaring local arrays or integers in a native function, since this will improve performance. For more information on native functions, see "Native Functions" on page 4-16.
expression	Consists of any valid NewtonScript expression. If a local variable is not explicitly initialized, NewtonScript will initialize it to `nil`.

Use of the `local` keyword is optional. If it is omitted the variables are still declared and initialized—so long as no other variables have these names. This keyword should never be omitted, however, for the following reasons:

- Performance is improved; the system has to search globals and the inheritance structure before declaring the local variable.
- Possible hard-to-find bugs are avoided. If a global variable or an inherited slot has this name, that variable will have its value changed; a new variable will not be declared. When the value of the global variable or inherited slot is then accessed, unexpected results might occur.
- Explicitly declaring local variables makes code easier to read and maintain.
- The native compiler cannot handle undeclared locals.

The scope of a local variable is restricted to the function definition in which it is declared. You can refer to it only within that function.

You may use the local expression to identify a variable as local without initializing it as in the example:

```
myFunc: func (x)
   begin
      local myVar, counter;
      ...
   end
```

This example declares the variables `myVar` and `counter` as local variables and initializes them to `nil`. Then, each time the function definition for `myFunc` is executed, new local variables are created that are unique to that function.

You may optionally use a `local` expression with one or multiple assignment clauses to assign a value to a variable or variables, as shown in the expression:

```
local x:=3, y:=2+2, z;
```

This expression creates three local variables, `x`, `y`, and `z`, and initializes them to the values of `3`, `4`, and `nil`, respectively.

The declaration of the local variable is processed at compile time but the values are assigned at run time, when the expression is encountered. For example, the expressions

```
x := 10;
local x, y := 20;
```

result in a value of 10 for `x` and a value of 20 for `y`. This works because `local` definitions work anywhere in the function.

By contrast, a run-time error is produced by the following code fragment:

```
x := y + 10;
local x, y := 20;
```

This is because at compile time `x` and `y` are declared and initialized to `nil`. When the assignments are made at run time, `y` evaluates to `nil`, and an error is produced in the computation of `nil+10`.

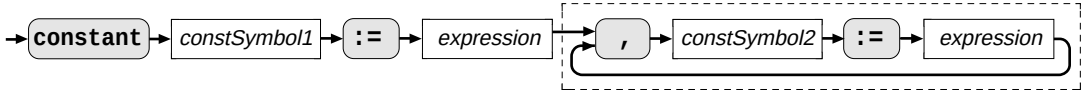
Constants

There are several ways to get unchangeable objects in NewtonScript. You can

- use the keyword `constant`
- put a single quote character (`'`) before an object literal
- initialize a variable to a literal value

Constant

`constant` *constSymbol1* `:=` *expression* [`,` *constSymbol2* `:=` *expression*]*



The **constant** declaration consists of the keyword `constant` and one or more initialization clauses consisting of a symbol, assignment operator (:=), and expression.

constSymbol	Consists of an identifier that is, syntactically, a symbol. For more information see the section "Symbol" beginning on page 2-12.
expression	Any expression consisting of operators and constants. The value of this expression becomes the value of *constSymbol.*

When a constants is used as a value, NewtonScript effectively substitutes literal values for the constant. This means that if you declare a constant, as in the expression

```
constant kConst := 32;
```

then when you use `kConst` as a value in your code, NewtonScript automatically substitutes the value 32. If you write

```
sum := kConst + 10;
```

it's exactly as if you had written

```
sum := 32 + 10;
```

However, if you use the same identifier as anything other than a value, as in the expressions:

```
x:kConst(42);
kConst(42);
x.kConst;
```

the value you defined is not substituted. This can be a problem if you define a constant to have a function or method as its value. For these cases there are built-in functions you can use as work-arounds, as shown in Table 2-4.

Table 2-4 Constant substitution work-arounds

No Substitution	Work-around
`x:kConst(42);`	`Perform(x,kConst,[42]);`
`kConst(42);`	`call kConst with (42);`
`x.kConst;`	`x.(kConst);`

You should also note that the constant value you assign does not get substituted when used in a quoted expressions like:

```
'{foo: kConst};
'[kConst];
```

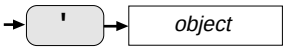
See the next section to learn more about the single quote syntax.

▲ **WARNING**

You can create local constants by putting the declaration in a function. Since they are in the same name space of the compiler as local variables, a local variable with the same name can override constants and vice versa. ▲

Quoted Constant

'*object*



The single quote character (') begins a kind of expression called a quoted constant.

You use the quote to create a literal object. The object is constructed at compile time, and a reference to the same object results each time the object literal is evaluated.

Here are several examples of literal objects formed with the quoted constant syntax:

```
'{name: "Joe Bob", income: yearTotal};
'myFrame.someSlot;
'[foo, 1234, "a string"];
```

When a quote appears outside of the brackets or braces, it applies to every element in the array or frame, and symbols within the array or frame do not need individual quotes. For instance, if you try to create this seemingly correct frame:

```
storyFrame := {Bear1: Mama, Bear2: Papa, Bear3: Baby};
```

you find that an error is caused when the value `Mama` gets interpreted as an undefined variable. One way to fix this is to put quotes before each name, or more simply to put one quote before the whole frame, as shown in the expression:

```
storyFrame := '{Bear1: Mama, Bear2: Papa, Bear3:Baby};
```

You can pass a quoted frame literal as a parameter to apply the object as needed.

Operators

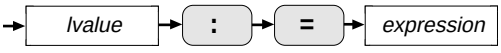
The NewtonScript operators are:

- assignment
- arithmetic
- Boolean
- equality
- relational
- unary
- message-send
- array and frame accessors

All NewtonScript operators, except the message-send operators and array and frame accessors are discussed in this section. The message-send operators are described in Chapter 4, "Functions and Methods." The access operators are described in "Frame Accessor" beginning on page 2-19 and "Array Accessor" beginning on page 2-16.

Assignment Operator

lvalue `:=` *expression*



In an assignment expression, the symbol, frame accessor, or array accessor that is the left-hand value for the assignment operator (`:=`), is assigned the value of the expression appearing to the right of assignment operator. An

assignment expression evaluates to the expression on the right-hand side of the assignment operator (`:=`).

lvalue	Consists of a symbol, a reference to an array element, or a reference to a frame slot.
expression	Consists of any valid NewtonScript expression.

You use assignment expressions to change the value of variables and slots. A simple assignment expression looks like this:

```
a := 10;
```

However, you can use any NewtonScript expression on the right-hand side of an assignment expression; when it is evaluated its value is assigned to the *lvalue*. For example, the variable x is set to refer to the value of the `if` expression in this assignment expression:

```
x := if a > b then a else b;
```

Here is an example of the assignment of a frame to a variable:

```
myFrame := {name: "", phone: "123-4567"}
```

Now you can assign a value to the `name` like this:

```
myFrame.name := "Julia"
```

Note that the NewtonScript inheritance rules affect the ultimate behavior of assignment expressions in frames. For more information about inheritance and setting slot values see “Inheritance Rules for Setting Slot Values,” in Chapter 5.

You can assign values to array slots in a similar manner. The second line of this code fragment changes the value 789 to 987.

```
myArray := [123, 456, 789, "a string"];
myArray [2] := 987;
```

Note that assignments of reference objects to variables will only copy a pointer to the object into the variable, see “Immediate and Reference Values” beginning on page 2-5. The built in functions `Clone`, `DeepClone`, and

`TotalClone` allow you to work around this behavior. See Chapter 6, "Built-In Functions" for an explanation of how these functions work.

An assignment expression, *lvalue* `:=` *expression*, evaluates to the value of *expression*. Furthermore, the assignment operator associates right-to-left. Thus, you can write an expression such as:

```
aVariable := anotherVariable := anExpression;
```

This will parse as:

```
aVariable := (anotherVariable := anExpression);
```

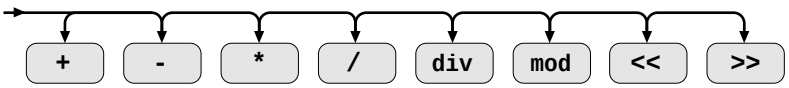
The `(anotherVariable := anExpression)` part will evaluate to `anExpression`, and both `aVariable` and `anotherVariable` will be set to the value of `anExpression`.

Note

If you accidentally write an equality operator (=) in an assignment expression as rather than the assignment operator (:=), your expression becomes a simple relational expression. For example, the expression `x = 5;` evaluates to `true` or `nil` and leaves the value of `x` unchanged. ◆

Arithmetic Operators

{ + | - | * | / | div | mod | << | >> }



NewtonScript provides the standard set of binary arithmetic operators. They are: the addition (+) and subtraction (-) operators, the multiplication (*) and

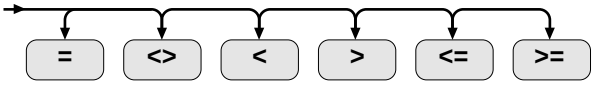
division (/) operators, the truncating operators `div` and `mod`, and the bitwise shift operators, bitwise left shift (<<) and bitwise right shift (>>).

`+`	The plus operator adds the two numbers it appears between.
`-`	The minus operator subtracts the number to its right from the number to its left. Note that minus can also be used as a unary operator to negate an expression. See "Unary Operators" beginning on page 2-35.
`*`	The multiply operator multiplies the two numbers it appears between.
`/`	The division operator divides the number to its left by the number to its right.
`div`	The divide and truncate operator divides the number to its left by the number to its right and truncates the remainder so that a whole number is returned.
`mod`	The modulo operator divides the number to its left by the number to its right and returns only the remainder.
<<	The bitwise shift left operator is an infix operator. In the expression: `x << y ;` `x` is shifted left by `y` bits.
>>	The bitwise shift right operator is an infix operator. In the expression: `x >> y ;` `x` is shifted right by `y` bits. The most significant bit is duplicated in the shift operation.

See Table 2-5 on page 2-39 for a summary of operator precedence.

Equality and Relational Operators

{ = | <> | < | > | <= | >=}



NewtonScript provides the standard set of binary equality and relational operators. The equality operators are: equal (=) and not equal (<>). The relational operators are less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=).

=	The equal operator tests the value of immediates and reals, and the identity of references; returning `true` if they are equal and `nil` if they are not equal.
<>	The not equal operator tests the value of immediates and reals, and the identity of references; returning `true` if they not equal and `nil` if they are equal.
<	The less than operator compares the values of numbers, characters, and strings; returning `true` if the operand on the left of the operator is less than the operand on its right and `nil` otherwise. An error is signalled if you try to compare arrays or frames.
>	The greater than operator compares the values of numbers, characters, and strings; returning `true` if the operand on the left of the operator is greater than the operand on its right and `nil` otherwise. An error is signalled if you try to compare arrays or frames.
<=	The less than or equal to operator compares the values of numbers, characters, and strings; returning `true` if the operand on the left of the operator is less than or equal to the operand on its right and `nil` otherwise. An error is signalled if you try to compare arrays or frames.

>=	The greater than or equal to operator compares the values of numbers, characters, and strings; returning `true` if the operand on the left of the operator is greater than or equal to the operand on its right and `nil` otherwise. An error is signalled if you try to compare arrays or frames.

This expression tests the identity of two reference objects:

```
"abc"<> [1, 2];
```

It evaluates to `true` because the two objects are not the same object.

In the same way, the equality operator (=), when applied to two array objects, compares their identity, as shown when you execute the code:

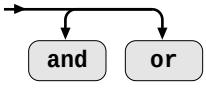
```
[1,2] = [1,2];
```

This expression evaluates to `nil` even though at first glance it looks as though it should be true. This is because each time the `[1,2]` expression is evaluated a new object is created.

The relational operators work with numbers, characters, and strings. If you try to use these operators with arguments that are arrays or frames, an error is signalled.

Boolean Operators

{ and | or }



The Boolean operators, `and` and `or`, are binary logical operators that compare two expressions.

`and`	The `and` operator tests the logical truth of its two operands; returning `true` if both expressions are true and `nil` otherwise.

`or`	The `or` operator tests the logical truth of its two operands; returning `true` if either expression is true and `nil` if both expressions are false.

Expressions involving the Boolean operators, `and` and `or`, like their counterparts, `&&` and `||`, in the C programming language, short circuit or stop evaluation as soon as the truth of an expression is determined. For instance, if

```
x < length(someArray)
```

evaluates to `nil` in a conditional expression like

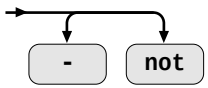
```
if x < length(someArray) and someArray[x]
   then doSomething
   else doSomethingElse;
```

processing is stopped immediately and goes on to the `else` clause. If one part of an `and` operation is not true, the whole `and` expression is not true. Therefore, it is not necessary to evaluate the second half of the `and` expression when the first half is not true. When this occurs it is said that execution is short-circuited. Similarly, if the first part of an `or` operation is true, than the whole `or` expression is true, and the second part is not evaluated.

The return value of an expression using any of the logical operators is either `nil`, which is false, or anything other than `nil`, which is true.

Unary Operators

{ - | not }



The unary prefix operators, minus (`-`) and `not`, precede any single expression.

`-`	The minus (`-`) operator returns the additive inverse of the expression it precedes.
`not`	The `not` operator is used before an expression to perform a logical negation of that expression.

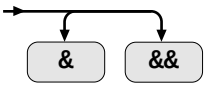
Here are some examples of these operators in use:

```
-x;      not x;   -(1 + 5);
not(a and -f(12) > 3);
```

`Exists` is another NewtonScript unary (but postfix) operator. It is described in the section"Exists" beginning on page 2-37.

String Operators

string1 { `&` | `&&` } *string2*



The two string operators, && and &, create a new string from the contents of two strings you provide. The single & operator creates a new string that has no spaces between the two objects, while the double && operator adds a space between the two strings in the new string.

If you use an object that is not a string in an && or & expression, NewtonScript converts the object to a string and uses it to construct a new string. This works for symbols, characters, and numbers.

An & and && expression returns a new string.

&	An & expression concatenates the string yielded by the expression on its left to the string yielded by the expression on its right.
&&	An && expression creates a new string by concatenating, with a space, the string yielded by the expression on its left to the string yielded by the expression on its right.

The single & operator concatenates the two strings by leaving no space, in the new string, between what was the second string and the first string. For instance, the expression:

```
"foo" & 17
```

creates the new string:

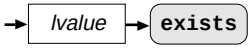
```
"foo17"
```

In contrast, the && operator creates a new string by copying the first string, adding a space, and then copying the second string, as you can see, in the following expression and its result:

```
"happy" && "days"
"happy days"
```

Exists

lValue `exists`



The `exists` operator is a special postfix operator that follows a single variable that can be a symbol, a frame accessor, or a message send.

The `exists` operator is used to check for the existence of variables, slots, or methods. When you apply the `exists` operator to an identifier, such as a variable, it returns `true` if the variable is defined in the current context and `nil` otherwise. (For more information about the scope of variables see "Scope" beginning on page 1-4.)

lValue — Consists of an expression that evaluates to a symbol, frame accessor, or a message-send.

All of these are legal forms for an `exists` expression to take:

```
x exists;
x.y exists;
```

```
x.(y) exists;
x:m exists;
```

Here is an example of a simple `if…then` structure that uses `exists`:

```
if myVar exists then myVar else 0;
```

When you apply `exists` to a frame accessor, `exists` returns `true` if the slot exists in the frame or in any of its prototype frames, and `nil` otherwise. (For more information about prototype frames and inheritance, see Chapter 5, "Inheritance and Lookup.") `If…then` expressions are described in Chapter 3, "Flow of Control."

This operator can be useful if you want to check for the existence of a slot that may or may not be in a proto frame.

```
if myFrame.aSlot exists then
      if not hasSlot(myFrame, 'aSlot) then
      print("'aSlot slot is in a prototype of myFrame")
```

The built-in function `HasSlot` provides similar functionality to the `exists` operator though they differ in the type of inheritance that is used to search for slots. See also "Inheritance Rules for Testing for the Existence of a Slot," in Chapter 5, and Chapter 6, "Built-In Functions."

Note

The `exists` operator is not guaranteed to work for local variables. ◆

Operator Precedence

In case it is not inherently apparent in an expression, a set of ratings tells the compiler which operator to evaluate first (or which operator takes precedence). Table 2-5 lists the order of precedence of all the NewtonScript operators in order from top to bottom. Note that operators grouped together in the table are of equal precedence.

Table 2-5 Operator precedence and associativity

Operator	Action	Associativity
`.`	slot access	left-to-right
`:` `:?`	message send conditional message send	left-to-right
`[]`	array element	left-to-right
`-`	unary minus	left-to-right
`<<` `>>`	left shift right shift	left-to-right

`*`
`/`
`div`
`mod`
	multiply
	float division
	integer division
	remainder
		left-to-right

`+` `-`	add subtract	left-to-right

`&`
 	
`&&`
	concatenate (string copy of expression value)
	concatenate with 1 space between
		left-to-right

`exists`	variable & slot existence	none

`<`
`<=`
`>`
`>=`
`=`
`<>`
	less than
	less than or equal
	greater than
	greater than or equal
	equal
	not equal
		left-to-right

`not`	logical not	left-to-right
`and` `or`	logical and logical or	left-to-right
`:=`	assignment	right-to-left

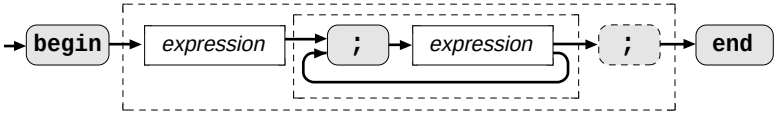
Flow of Control

This chapter discusses the syntax and semantics of the standard flow-of-control mechanisms including compound expressions, conditional expressions and iterators.

Also included is a discussion of the non-standard flow-of-control mechanism called exception handling that NewtonScript uses to control exceptional situations or errors.

Compound Expressions

```
begin
    expression1;
    expression2;
    …
    expressionN[;]
end
```



In NewtonScript, the keywords `begin` and `end` are used to group expressions, not to create structured blocks that define the scope of variables, as they are in some languages.

This construct is useful in conditional expressions, in any of the looping expressions, and in function definitions. In fact, anywhere the syntax specifies a single expression, you can use a compound expression instead.

The compound expression returns the result of the last expression.

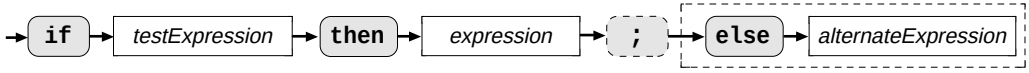
expression Consists of any valid NewtonScript expression; it must be separated from the next expression by a semicolon, unless it is the last expression in the compound expression. In this case, the keyword `end` separates it from any following expressions.

If you want to execute more than one expression in a conditional expression, use the keywords `begin` and `end` to group the expressions as shown in this example:

```
if x=length(myArray) then
   begin
      result := self:read(x);
      print(result)
   end
```

If…Then…Else

`if` *testExpression* `then` *expression* [`;`]
[`else` *alternateExpression*]



The `if`…`then`…`else` construct allows you to dictate programmatic flow of control using test conditions.

As is standard in other programming languages, you use an `if` expression to carry out one set of operations, when the condition you set up in a test expression is true, and another set of operations when the text expression evaluates to `nil`.

The `if` expression returns the value of its *expression* or *alternateExpression.* clause, unless the test condition is not true and there is no `else` clause. In that case, `nil` is returned.

testExpression	Consists of an expression that tests for the truth of a condition. If the test expression evaluates to anything other than `nil`, the expression following it is executed.
expression	Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section "Compound Expressions.") This *expression* is executed if the test is true; its value is returned as the value of the `if` expression.
alternateExpression	Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section "Compound Expressions.") The `else` clause, along with the *alternateExpression* that follows it, is executed if the test expression evaluates to `nil`.

An `else` clause binds to the nearest unmatched `if-then` clause.

Iterators

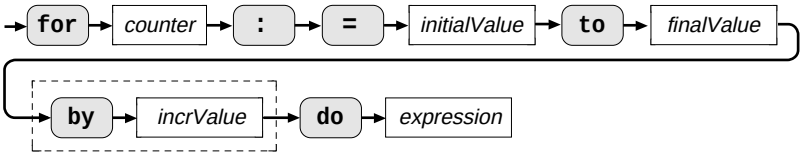
NewtonScript includes these iterators:

- `For`
- `Foreach`
- `Loop`

- `While`
- `Repeat`
- `Break`

For

`for` *counter* `:=` *initialValue* `to` *finalValue* [`by` *incrValue*] `do` *expression*



The `for` loop performs a set of expressions repeatedly, until a loop counter variable equals or exceeds a specified final value or a `Break` expression is reached. A counter variable keeps track of the number of times the loop has executed. You specify the initial value and final value of the counter variable. If you choose not to specify the amount to increase (or decrease) the counter by using the optional keyword `by` followed with an incremental value, the counter gets incremented by the default value of 1.

A for loop expression returns `nil` (or the argument of a `break` expression, if one is used to terminate the loop.)

counter — This symbol is set to the value of the *initialValue* expression when a `for` loop starts. After each repetition of the loop, the *counter* variable is incremented by the value of *incrValue*. or by the default value of 1 if an incremental value is not specified.

The *counter* symbol is automatically declared as a local by the NewtonScript compiler.

On loop exit the value of *counter* is undefined. It is an error to change its value from within the loop body. If you do so, loop behavior is undefined.

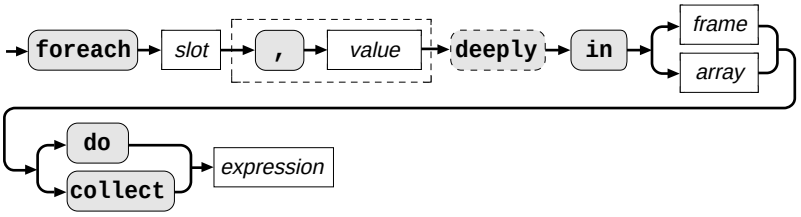
initialValue	This expression must evaluate to an integer. It is evaluated once before the loop starts and is used as the initial value of the counter.
finalValue	This expression must evaluate to an integer. It is evaluated once before the loop starts and is used as the final value of the counter.
incrValue	This expression follows the keyword `by`. It is evaluated once before the loop starts and used throughout the loop execution to increment (or decrement) the counter variable. The incremental value expression must evaluate to an integer (either positive or negative); a value of zero generates a run-time error. If you do not specify an incremental expression, after the keyword `by`, a default value of 1 is used to increment the counter.
expression	Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section “Compound Expressions.”)

Here is an example of a `for` loop.

```
for x:=1 to 10 by 2 do print(x);

1
3
5
7
9
```

Foreach

`foreach` [*slot*`,`] *value* [`deeply`] `in` {*frame* | *array*}
{`collect` | `do`} *expression*



Using the `foreach` iterator is one way you can access data stored in frames or arrays. This iterator executes once for each element in an array or frame allowing you to either iterate an expression over each value stored in an array or frame, or to collect data during each iteration.

In an array, iteration begins with the first element of the array, element 0. In a frame the starting point and subsequent order of iteration is unpredictable since frame slots are not stored in any particular order.

This iterator also has a special option, `deeply`, for frame iteration over values in the target frame and in its proto chain of frames as well. For information on prototype inheritance see Chapter 5, “Inheritance and Lookup.”

A `foreach` expression returns `nil` (or the argument of a `break` expression, if one is used to terminate the loop.)

slot — This symbol is set to the name or index of the next slot on each loop iteration over the elements of an array or frame. The value of this variable is undefined on loop exit. Using the *slot* variable is optional. If you specify just one variable of the *slot*`,` *value* pair, it’s assumed to be as the *value*.

The *slot* symbol is automatically declared as a local by the NewtonScript compiler.

value	Set to the value of the next array element or frame slot on each loop iteration over the elements of an array or frame. The value of this variable is undefined on loop exit. Using the *value* variable is mandatory. If you specify just one variable, of the *slot*, *value* pair, it's value is assigned as the element value. The *value* symbol is automatically declared as a local by the NewtonScript compiler.
array	An expression that evaluates to an array.
frame	An expression that evaluates to a frame.
expression	Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section "Compound Expressions" beginning on page 3-1)
`deeply`	Optional. If this keyword is included, iteration occurs over values in the immediate frame first and then in the `_proto` frame(s) as well. (For information on prototype inheritance see Chapter 5, "Inheritance and Lookup.") If you specify the `deeply` option and the frame you are concerned with does not have a `_proto` frame, no error is produced. Instead, the slot values evaluate to those of the current frame.

You use `foreach` to access data stored in a frame that functions like a Pascal record or a C struct. The data used in the following example is the `nameFrame` frame, defined as:

```
nameFrame := { name:"Carol",
               office:"San Diego",
               phone:"123-4567"};
```

You can use the `foreach do` loop to access and print the slot names and values stored in a `nameFrame` by writing a method like `reportIt`:

```
report:= { reportIt: func(frameName)
               foreach slot, value in frameName do
                  print(slot && ":" && value);
         }
```

When you send the message `reportIt` with the argument `nameFrame` to the `report` object, as shown here:

```
report:reportIt(nameFrame);
```

this is the output produced:

```
"name : Carol"
"office : San Diego"
"phone : 123-4567"
```

Using `collect` with the `foreach` iterator makes it easy to collect the data and manipulate it. Consider a `dataFrame` which is defined as:

```
dataFrame := {1,3,5,7,9}
```

You can collect the squares of each value in `dataFrame` and print the results, with the code shown here:

```
result := foreach value in dataFrame collect value*value;
print(result);
```

The values are collected in an array, as shown in the output:

```
[1,9,25,49,81]
```

Note
The behavior of `foreach` is undefined when the array or frame is modified inside the loop body, except for the specific case of deleting the current element. In this specific case, the loop will continue on to the next element as expected. ◆

If you want the `foreach` iterator to look up slot values in the prototype frame in addition to the current frame, use the `deeply` option. To make use of the `deeply` option with this iterator, your data set must include a frame that references a prototype frame. For purposes of example we can use the data defined here as:

```
x := {one:1, two:2, three:3};
y := {four:4, five:5, combo:x};
z := {six:6, _proto:y};
```

You can consult the picture of this data, shown in Figure 3-1, while looking at the accompanying table, Table 3-1, which shows the different results produced by using `foreach` both alone and with the `deeply` option.

Figure 3-1 Data objects and their relationships

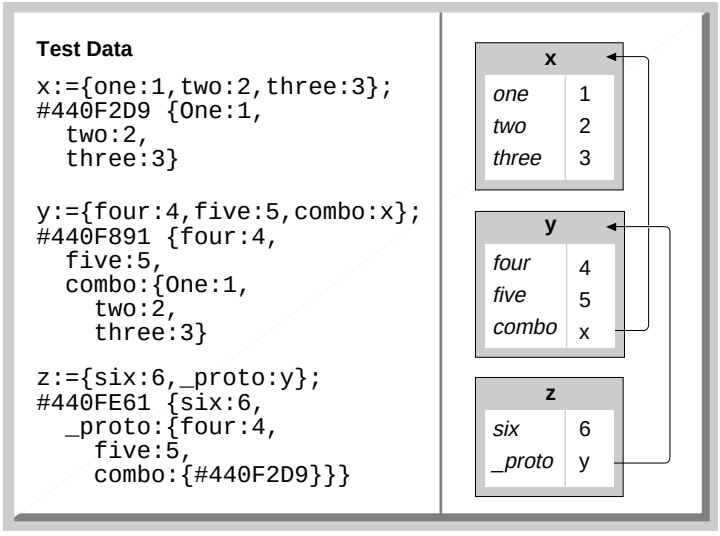


Table 3-1 allows you to compare the results produced by two functions, `normalList` and `deeplyList`, as applied to the data in Figure 3-1.

Table 3-1 Result comparison for the iterators `foreach` and `foreach deeply`

foreach	foreach deeply

`normallist := func (param) begin`
`  foreach tempItem in param`
`  collect tempItem;`
`end;`
	`deeplylist := func (param) begin`
	`  foreach tempItem deeply in param`
	`  collect tempItem;`
	`end;`

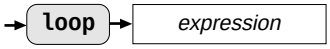
`:normallist(x)` `#4413441  [1, 2, 3]`	`:deeplylist(x)` `#44137D9  [1, 2, 3]`

`:normallist(y)  // same`
`#4413A11  [4,`
`  5,`
`  {One: 1,`
`  two: 2,`
`  three: 3}]`
	`:deeplylist(y)`
	`#4413C49  [4,`
	`  5,`
	`  {One: 1,`
	`  two: 2,`
	`  three: 3}]`

`:normallist(z)`
`#4416E29  [6,`
`  {four: 4,`
`  five: 5,`
`  combo: {#4415D79}}]`
	`:deeplylist(z)`
	`#4416FE1  [6,`
	`  4,`
	`  5,`
	`  {One: 1,`
	`  two: 2,`
	`  three: 3}]`

Loop

`loop` *expression*



This mechanism simply repeats any expressions that occur within the `loop` expression until a `break` expression is encountered; if no `break` expression is reached the loop never ends.

The `loop` expression returns the argument of the `break` expression that is used to terminate the loop.

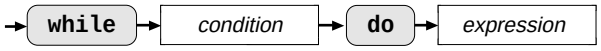
expression Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section "Compound Expressions" beginning on page 3-1) This expression is evaluated during each loop iteration.

This example prints the value of the variable `x` until it reaches the value of 0 and the `break` expression is executed.

```
local x:=4;
loop
   if x = 0 then
      break
   else
      begin
         print(x);
         x:=x-1
      end
4
3
2
1
```

While

`while` *condition* `do` *expression*



The `while` loop evaluates the conditional expression first. If it evaluates to a non-`nil` value (`true` or any other value that is not `nil`) the expression after the keyword `do` executes. This sequence repeats until the conditional expression evaluates to `nil` and ends loop execution.

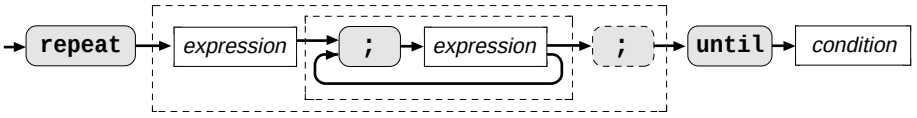
A `while` expression returns `nil` (or the argument of a `break` expression, if one is used to terminate the loop.)

condition — Consists of an expression that tests for the truth of a condition. If the test expression evaluates to `nil`, loop execution ends.

expression — Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section "Compound Expressions" beginning on page 3-1) This expression is evaluated during each loop iteration.

Repeat

```
repeat
    expression1;
    expression2;
    ...
    expressionN[;]
until condition
```



The `repeat` loop executes the expression(s) inside the loop first and then evaluates the test expression. If the expression at the end of the loop evaluates to `nil`, the expressions repeat and the test expression is evaluated again. This continues until the expression evaluates to non-`nil`, at which point the loop ends.

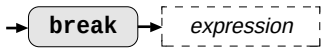
A `repeat` expression returns `nil` (or the argument of a `break` expression, if one is used to terminate the loop.)

condition Consists of an expression that tests for the truth of a condition. If the conditional expression evaluates to anything other than `nil`, loop execution ends.

expression Consists of any valid NewtonScript expression.

Break

`break` [*expression*];



While not an iterator itself, the `Break` expression interrupts the execution of any of the iterative structures. You must use the `Break` expression to stop the simple `Loop` structure, which has no built-in constructs to stop it.

If a expression follows `Break`, it is evaluated and returned as the value of the loop. If you use the `Break` with no expression following it, the loop returns the value `nil`.

expression Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section "Compound Expressions" beginning on page 3-1) The value of this expression is returned as the value of the `Break` expression.

Exception Handling

This section describes exception handling in NewtonScript. Exception handling is a non-standard flow-of-control mechanism that NewtonScript inherits from Newton system software.

NewtonScript exception handling allows you to respond to exceptional conditions that arise during the execution of your program. An exceptional condition, or **exception,** is a condition that either the Newton system

software or your own code raises when something unexpected or erroneous happens at run time.

When an exception is raised at run time, the system can transfer control to an **exception handler,** which is a block of code that attempts to handle the condition gracefully, rather than allowing the application to crash. An exception handler can respond by displaying an error message, reverting the state of a computation, or taking some other action or actions.

The act of raising an exception is known as throwing an exception. An exception handler catches the exception and responds in some manner. Each exception has a unique name and each exception handler responds to a specific exception or class of exceptions.

Newton system software throws and catches a number of built-in exceptions. You can define, throw, and catch your own exceptions, and you can also catch and handle the built-in exceptions.

Working with Exceptions

Working with exceptions in NewtonScript involves a number of entities. You can perform the following actions to work with exceptions:

- define an exception symbol for a specific exception or class of exceptions
- enclose a list of statements within a `try` statement to catch any exceptions that occur during execution of those statements
- catch a specific exception or class of exceptions with an `onexception` statement
- use the `CurrentException` function to examine the frame associated with the exception that you are handling
- throw an exception when you detect a condition that request handling
- rethrow an exception from within your exception-handling code to allow the next handler for the exception to respond to it

You can provide exception handling for any list of statements in your NewtonScript programs. You can also nest an exception-handling block

of code inside of another exception-handling block of code to provide a hierarchical chain of exception handlers.

Each exception handler can specify which exception or class of exceptions it processes by naming the symbol or symbol prefix that it handles. An exception handler can also reraise (rethrow) the exception that it is handling to allow other exception handlers in the chain an opportunity to process the exception.

The basic process for implementing exception handling is as follows:

1. Decide on a name for the exceptions that you are going to define and how you are going to respond when each exception is raised.
2. Write your code and use the `Throw` function to raise exceptions where appropriate.
3. Write an `onexception` clause for each exception. Each clause names an exception and provides a statement to handle that exception.
4. Enclose the list of statements in which you are raising and handling exceptions with a `Try` statement.

Defining Exceptions

Each exception is named with an **exception symbol.** You must adhere to the following format rules when defining an exception symbol. Each symbol

- must be enclosed in vertical bars (`|`)
- must contain at least one part that begins with the prefix `evt.ex`
- can contain up to 127 characters
- can contain multiple parts that are separated by semicolons (;)

A few example of exception symbols are shown in Listing 3-1.

Listing 3-1 Exception symbols

```
|evt.ex|
|evt.ex.fr.intrp|
|evt.ex.div0|
|evt.ex.msg;type.ref.frame|
```

IMPORTANT
Do not leave a space between the parts of exception symbols, since the vertical bars make the space part of the exception symbol. ◆

The prefixes contained in the exception symbols are used to define the hierarchy of exception handlers, as described in the section "Catching Exceptions" beginning on page 3-21. These prefixes are also important for defining exception types, as described in the section "Exception Frames" on page 3-16.

Exception Symbol Parts

Each exception symbol can contain multiple parts, enclosed in vertical bars and named as described earlier in this section. For example, the symbol

```
|evt.ex;type.ref.something|
```

contains two parts. The parts of an exception symbol must be separated by a semicolon.

When an exception symbol contains multiple parts, the exception is still processed as a single exception. This means that the first exception handler to catch any part of the exception symbol handles the exception. That handler can rethrow the exception to allow other handlers to catch it.

Exception Frames

A frame is associated with each exception. This **exception frame** contains two slots: a name slot and a data slot. The name slot is always named `name` and always contains the exception symbol. The name and contents of the

other slot, which contains the data, depend on the composition of the exception symbol, as shown in Table 3-2.

Table 3-2 Exception frame data slot name and contents

Exception symbol	Slot name	Slot contents
contains part with prefix `type.ref`	`data`	a data object, which can be any NewtonScript object
contains part with prefix `evt.ex.msg`	`message`	a message string
any other	`error`	an integer error code

Table 3-3 shows several examples of exceptions and the frames associated with them.

Table 3-3 Exception frame examples

Exception symbol	Exception frame
`|evt.ex;type.ref|`	`{name: '|evt.ex;type.ref|, data: {type: 'inka, size: 42, weight: 177}}`
`|evt.ex.msg|`	`{name: '|evt.ex.msg|, message: "there seems to be a problem"}`
`|evt.ex|`	`{name: '|evt.ex|, error: -48666}`

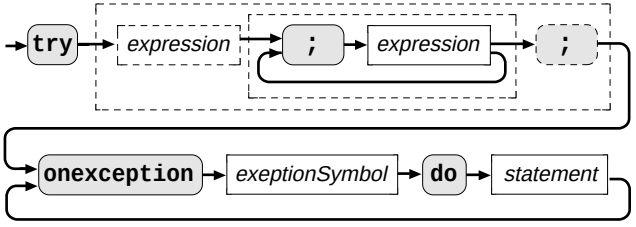
You can access the frame that is associated with an exception from within your exception handler by calling the built-in function `CurrentException`, described in Chapter 6, “Built-In Functions.”

The `CurrentException` function returns the frame that is associated with the current exception. You can examine the frame returned by `CurrentException` to determine what kind of exception you are handling. For example, you can call the `HasSlot` function to determine if the frame contains a slot named `error` and you can then take appropriate action.

The Try Statement

You use the `try` statement to enclose a list of statements in which you want to handle exceptions. The syntax of a `try` statement is

```
try
    expression1;
    expression2;
    ...
    expressionN
onexception exceptionSymbol do
    statement
onexception exceptionSymbol do
    statement...
```



The try statement encloses *statement1* through *statementN* and transfers control to one of the `onexception` clauses when an exception is raised. If no exceptions are raised, the value of the `try` statement is the value of its final statement. If an exception is raised and an `onexception` clause handles that exception, the value of the `try` statement is the value of the executed `onexception` clause's statement.

expression	Any valid NewtonScript expression.
exceptionSymbol	An exception symbol that can contain multiple parts separated by semicolons. The symbol is enclosed in vertical bars and can contain up 127 characters.

Exception symbols are described in "Exception Symbol Parts" beginning on page 3-16. Examples of the `try` statement and `onexception` clauses are found in Listing 3-4 on page 3-22.

Throwing Exceptions

To raise an exception in NewtonScript, you need to call the `Throw` function and to include the exception name and data as parameters. The form of the data that you send as a parameter must match the type of exception you are throwing.

The `Throw(`*name*`,`*data*`)` function raises an exception and creates an exception frame with the specified *name* and *data*. The possible values for the *data* parameter depend on the composition of *name*, and are shown in Table 3-2 on page 3-17. The `Throw` function is described in Chapter 6, "Built-In Functions."

You call the `Throw` function from within a list of statements that are enclosed by a `Try` statement. NewtonScript transfers control to the `onexception` clause whose symbol matches *name*. Listing 3-2 shows several examples of calls to the `Throw` function.

Listing 3-2 The `Throw` function

```
Throw('|evt.ex.foo|, -12345);
Throw('|evt.ex.msg|, "This is my message");
Throw('|evt.ex;type.ref.something|, ["a", "b", "c"]);
```

Note that the composition of the exception symbol that you pass as the first parameter to the `Throw` function defines the kind of data that you pass as the second parameter:

- The first statement in Listing 3-2 requires an error number as its second parameter.
- The second statement in Listing 3-2 contains the prefix `evt.ex.msg` and thus requires a message string as its second parameter.
- The third statement in Listing 3-2 contains the prefix `type.ref` and thus requires a data object (in this case, an array) as its second parameter.

Throwing an Exception to Another Handler

You can pass control from within an exception handler to the next enclosing `Try` statement by reraising the exception. To do this, you call the `Rethrow` function. This function is described in Chapter 6, "Built-In Functions."

The `Rethrow` function reraises the current exception to allow the next enclosing `Try` statement an opportunity to handle it. The `Rethrow` function also passes along the same parameters as were passed with the original call to the `Throw` function. The following example illustrates the use of `Rethrow`:

```
onexception |evt.ex.msg| do
   if StrEqual (CurrentException().message, someString)
      then self:doSomething();
   else Rethrow()
```

IMPORTANT

You can call the `Rethrow` function only from within an `onexception` clause. ▲

Catching Exceptions

When an exception is thrown during the execution of a list of statements, execution of that list of statements is terminated and control is transferred to the first exception handler that matches the exception. Each exception handler is an `onexception` clause enclosed within a `try` statement, as shown in Listing 3-3 and Listing 3-4.

Each `onexception` clause specifies the symbol of the exception or the class of exceptions that it handles. The first exception handler that matches the symbol of the exception that has been raised is the handler that is invoked. This happens as follows:

1. When an exception is raised, Newton system software examines the `onexception` clauses of the `try` statement that is currently active. The `onexception` clauses are examined in order, from first defined to last defined.
2. The first matching `onexception` clause is executed and the value of the clause becomes the value of the `try` statement. A matching `onexception` clause is one whose exception symbol is a prefix of any of the parts of the exception that was raised.
3. If the active `try` statement does not contain a matching `onexception` clause, the exception is passed onto the next enclosing `try` statement.
4. The exception is passed along to enclosing `try` statements until it is handled. If no `onexception` clause in your application handles it, the exception will be handled by the system, which responds by displaying an error alert.

There are two logical points that should be considered in structuring code with exception handlers.

First, since exceptions are handled by the first `onexception` clause that contains a prefix of a part of the exception symbol, you need to order your `onexception` clauses from most specific to least specific. For example, the code in Listing 3-3 contains three `onexception` clauses ordered improperly.

Listing 3-3 Several onexception clauses ordered improperly

```
try
   c := x:myFunc(p, q);
   :anotherFunc(c)
onexception |evt.ex.pgm.fnerr| do
   begin
      print(“function error”);
      c := nil;
   end
onexception |evt.ex.pgm| do
      print(“program error”)
onexception |evt.ex.pgm.dataerr| do
      print(“data error”);
```

The final `onexception` clause in Listing 3-3 will never be executed because the second `onexception` clause catches any exceptions that contain the `evt.ex.pgm` prefix. Changing the order of the clauses to make the least specific (the `|evt.ex.pgm|` symbol) clause last fixes the problem. This improved version of the code is shown in Listing 3-4.

Listing 3-4 The onexception clauses properly ordered

```
try
   c := x:myFunc(p, q);
   :anotherFunc(c)
onexception |evt.ex.pgm.fnerr| do
   begin
      print(“function error”); do
      c := nil;
   end
onexception |evt.ex.pgm.dataerr| do
      print(“data error”)
onexception |evt.ex.pgm| do
      print(“program error”);
```

Second, an `onexception` clause is matched with the nearest `try` statement, just as an `else` clause is matched to the nearest `if-then` clause. However, unlike the `if-then` case, a single `try` statement can bind to multiple `onexception` clauses. Listing 3-5 illustrates how this can cause problems when nesting `try` blocks. Listing 3-6 then shows how this problem can be avoided by explicitly declaring `try` blocks with the keywords `begin` and `end`.

Listing 3-5 Improperly nested `try` blocks

```
func f()
begin
   try
      try
         self:doSomething()
      onexception |evt.ex| do
         print( CurrentException() );
      self:doSomethingElse()
   onexception |evt.ex| do
      print( “There was a problem.” );
end
```

Listing 3-6 Nested `try` block problem fixed using `begin` and `end` (shown in bold)

```
func f()
begin
   try
      try
      begin
         self:doSomething()
      onexception |evt.ex| do
         print( CurrentException() );
      end
```

```
      self:doSomethingElse()
   onexception |evt.ex| do
      print( “There was a problem.” );
end
```

IMPORTANT

The `onexception` syntax is not forgiving about extra semicolons. Never include a semicolon (`;`) before an `onexception` clause. ▲

Responding to Exceptions

This section shows and describes several examples of using exception handling in a NewtonScript application program.

Listing 3-7 shows an exception handler that catches the exception raised by the Newton system software when there is not enough memory to store a new date in the Datebook soup.

Listing 3-7 Handling a soup store exception

```
onException |evt.ex.fr.store| do
   :Notify(kNotifyAlert, "Dates",
               "Not enough memory to save changes.");
```

Listing 3-8 shows an exception handler that examines the exception frame to determine if the exception represents a certain error. If so, the handler takes an action; otherwise, the handler rethrows the exception so that it can be caught by another handler.

Listing 3-8 An exception handler checking the exception frame

```
onException |evt.ex| do
   if HasSlot(CurrentException(), 'error) then
      begin
         if CurrentException().error = -48211 then
            Print(“The string you entered is too large”)
         else Rethrow();
      end
   else Rethrow();
```

CHAPTER 4

Functions and Methods

This chapter describes the way you encapsulate and access code in functions and methods, as well as related topics, including:

- method and function definition
- messages
- passing parameters
- function objects
- native functions

About Functions and Methods

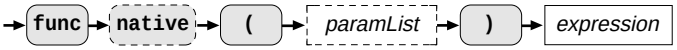
Most functions in NewtonScript are really methods; that is, they are defined within the context of a frame that can receive messages. In fact, a **method** in NewtonScript is nothing more than a function referenced by a frame slot and invoked with a message send.

You send **messages** to objects to execute methods, as in other object-oriented languages. In NewtonScript, the frame is the only type of object that receives messages. (See the section “Frame” beginning on page 2-17.)

NewtonScript also has built-in global functions that are part of the system. These are discussed in Chapter 6, “Built-In Functions.”

Function Constructor

`func [native](`*paramList*`)` *expression*



The `func` expression is used to create a function or method.

The syntax of a function constructor consists of the reserved word, `func`, and the optional keyword, `native`, followed by parentheses that surround zero or more parameters in a comma-separated list, and a body of code consisting of one expression. The keyword `native` denotes a native function; see the section “Native Functions” on page 4-16.

A function constructor returns a **function object,** which, when executed, returns the value of its *expression*. This is the last expression executed, if *expression* is a compound expression. See the section “Function Objects” beginning on page 4-9.

paramList — An optional list of parameter identifiers that are separated by commas and enclosed in parentheses. If your function does not use parameters you must still include an empty set of parentheses following the keyword `func`, or `native` (if it appears). Any identifiers in *paramList* can be preceded by the keywords `int` or `array`, which automatically declares them as variables of the respective types.

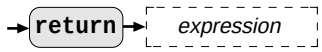
expression — Consists of any valid NewtonScript expression. A compound expression may be substituted. (For more information see the section “Compound Expressions” on page 3-1.)

When a function is executed, it returns the value of the *expression* evaluated. The following example, `myFunction`, simply returns the value of the difference between its two parameters, that is, the value of the `if` expression.

```
myFunction := func(n1, n2)
                  if n1 > n2 then n1 - n2;
                  else n2 - n1
```

Return

`return` [*returnValue*]



The `return` expression is used to exit a function and return a value.

When an expression appears following the keyword, `return`, it is evaluated and its value is returned as the value of the function. If you do not specify a return value, `nil` is returned on function exit.

returnValue Optional. Consists of any valid NewtonScript expression or compound expression. If no expression follows the `return` keyword, the `return` expression evaluates to `nil`.

Function Invocations

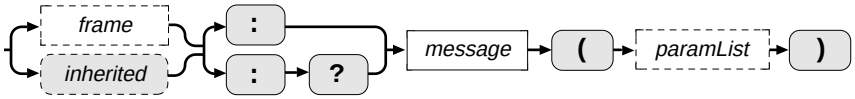
There are three ways function objects can be executed in NewtonScript:

- as the result of a message-send
- by using the `call with` syntax
- with a global function invocation

This section describes each of these in turn.

Message-Send Operators

[{`inherited`|*frame*}] {`:`|`:?`} *message*`(`*paramList*`)`



Most code is executed in response to messages you send to a frame. Messages are sent by using either the colon (`:`), which is the message-send operator, or the colon-question mark (`:?`), which is the conditional message-send operator.

The message-send operator (`:`) sends a message and its arguments, if any, to a frame object. The conditional message-send (`:?`) first checks to see if a method exists anywhere in the inheritance chain before sending the message.

The optional frame expression, *frame*, appears before the operator and specifies the frame where the message is sent. If a frame expression is specified, the message is sent directly to the frame you specify, and it becomes the **receiver** of the message.

When nothing appears before the message-send operator, the message is sent to the current receiver, which you can refer to using the pseudo-variable, **`self`.** Rather than leaving a blank before the message-send operator, you can make your code more readable by putting `self` there, to specify explicitly the current receiver. (See "Note" on page 5-10 for a discussion of the pros and cons of this usage of `self`.)

If you want to call an inherited method instead of the method that overrides it, use the keyword, `inherited`, before the message-send operator. This forces NewtonScript to bypass the receiver and look up the value of the method in the prototype chain, starting after the frame where the currently executing method was found. Note that lookup stops at the end of the prototype chain and does not continue up the parent chain. For more information on lookup see Chapter 5, "Inheritance and Lookup."

The message that follows the message-send operator is a symbol. The message-send operator looks for a frame slot with that name. The frame slot

must reference a function with the same number of parameters used in the message-send parameter list.

frame — Any valid NewtonScript expression that evaluates to a frame. The *frame* specified becomes the receiver of the message. The message is sent to the current receiver when a *frame* does not appear before the colon, as in the following expression.

`:`*message*`(`*argList*`);`

`inherited` — A keyword specifying that the message is being sent to an inherited version of the method code residing somewhere in the prototype chain. Using the `inherited` keyword forces method lookup to start in the prototype chain rather than in the receiver.

message — A symbol used to look up the method using the standard inheritance rules, beginning with the receiver, at run time. For more information on lookup see Chapter 5, "Inheritance and Lookup."

paramList — Consists of a list of zero or more parameters, separated by commas and enclosed by parentheses. The number of parameters must match the number of parameters expected by the method.

When the following message-send executes, the message, `msg1`, is sent to the object, `frame4`.

```
frame4:msg1();
```

If you send the same message without specifying which frame is the receiver, the message is sent to the current receiver, as in the expression:

```
:msg1();
```

The same operation could be written as:

```
self:msg1();
```

If you are not sure if a method exists, send the message using the conditional message-send operator (`:?`). This operator insures that the message is sent only if NewtonScript can find the method.

Note that the following two expressions are equivalent:

```
if frameName:messageName exists
   then frameName:messageName()
```

and,

```
frameName:?messageName()
```

The second is preferable, however, since the first message will be looked up twice; once to evaluate the `exists` expression, and once for the message-send.

You can also use the built-in function `Perform()` to send a message with a run-time argument list. There is also a function, `PerformIfDefined`, which mimics the conditional message send operator (`:?`). Two functions also exist which will send a message, but only search the prototype (and not the parent) inheritance chains; these are `ProtoPerform` and `ProtoPerformIfDefined`. All these functions are described in Chapter 6, "Built-In Functions."

Call With

`call` *function* `with (`*paramList*`)`

The `call with` expression executes the specified *function* object and its parameters, using the value of the environment that was captured when the function object was created. Thus, the function object executes as a closure would in a language like Lisp. See the section "Function Objects" beginning on page 4-9 for more information.

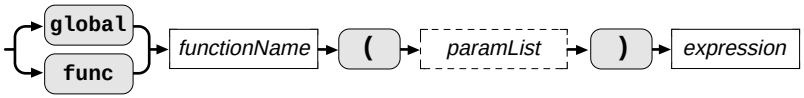
You can call a function with a run-time argument list by using the built-in function `Apply`. For more information on this function see Chapter 6, "Built-in Functions."

function	Consists of any valid NewtonScript expression that evaluates to a function.
paramList	Consists of a list of zero or more parameters, separated by commas and enclosed by parentheses. The number of parameters must match the number of parameters expected by the function.

A `call with` expression returns the result of the function it executes.

Global Function Declaration

`{ global|func }` *functionName* `(`*paramList*`)` *expression*



You can use the global function definition syntax to define global functions in some NewtonScript implementations. Note that except for the keyword `global`, this syntax is the same as the regular function definition syntax discussed in the section "Function Constructor" beginning on page 4-2.

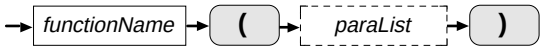
Global functions can only be defined at the top level, not inside another function. The scope of a global function definition includes every NewtonScript function.

Note

When programming in the NTK environment, this use of the keyword `global` creates a function that is global within the NewtonScript environment in NTK. If you then attempt to call this function inside another function executing in the NewtonScript environment on a Newton device, an "unknown global function" error is generated. ◆

Global Function Invocation

functionName (*paramList*)



The global function call syntax has the same effect as a `call with` expression: it executes the specified function object *functionName* and its parameters using the message environment that was captured when the function object was created. However, the function is determined by name rather than by evaluating an expression.

A global function call expression returns the result of the function object it executes.

functionName	A symbol naming a global function.
paramList	A list of zero or more parameters, separated by commas and enclosed by parentheses. The number of parameters must match the number of parameters expected by the function.

Many global function definitions are built into NewtonScript (see Chapter 6, “Built-In Functions,” for a list of them).

Passing Parameters

Parameters are passed by value in NewtonScript. In other languages this sometimes means that these parameters are unchangeable by the function. However, you should note that some NewtonScript values are references, and when a reference is the value of a parameter the function can modify that object.

For more information about NewtonScript immediates and references, see the section “Immediate and Reference Values” beginning on page 2-5.

Function Objects

A function object is constructed when code of the following form executes:

`func(`*paramList*`)` *expression*`;`

Functions are first-class objects in NewtonScript. They can be assigned to local variables, array elements, or frame slots. They can also be stored in soups, or passed as arguments to a function. For information on soups, see *The Newton Programmer's Guide*.

The term "function object," rather than just "function," is used to emphasize the fact that one `func` statement can give rise to many different function objects. When a function object is created it saves the environment that exists at that time. Therefore, multiple function objects that are created by one function constructor differ if the environments that existed when the `func` statement was executed differ.

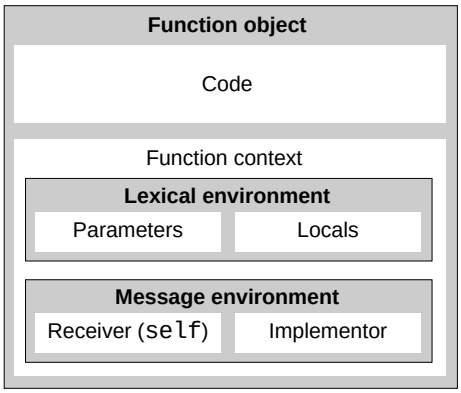
A function object consists of two main parts: its code, and the function context, which is where the environment that existed at the time of its creation is saved. The function context itself consists of two parts: the lexical environment, and the message environment.

- The lexical environment is a list of locals and parameters in the function and in any enclosing functions.
- The message environment consists of references to the frame in which the function is defined (the **implementor**) and to the frame to which it is sent (the receiver).

By saving the environment in which it was created, the function object has access to local variables and to parameters that existed when it was constructed. In addition, the function object has access to variables in the inheritance chain of the frame that was the value of `self` when the function object was created. The NewtonScript inheritance mechanism is described in Chapter 5, "Inheritance and Lookup."

The parts of a function object are shown in Figure 4-1.

Figure 4-1 The parts of a function object



Function Context

NewtonScript uses the context of a function--the lexical and message environments--to establish values for any variables used in a function without being defined there.

The Lexical Environment

The lexical environment consists of a list of locals and parameters in the function, and any enclosing functions. For instance, the lexical environment of the function shown in the following example is the value of the parameter `e` when the function was called:

```
frame1:={task1: func(e)
         begin
            //do something
         end }
```

The lexical environment of the function, `task2`, shown below in boldface type, consists of the values of the local variable `total`, and the parameters `e`, and `a` when the function was called.

```
frame2:={task1: func(e)
         begin
            local total:= e;
            e := 20;
            task2:= func(a) … ;
            total
         end }
```

Note

Some implementations of NewtonScript optimize the memory allocated to the lexical environment by saving only those variables that are actually used within the function body. ◆

The Message Environment

The message environment of a function consists of the implementor of the message and the receiver of a message.

The frame in which a method is defined is called its implementor. Note that a method could be defined in a number of places within a frame's inheritance chain. The implementor is the frame in the inheritance chain where the method is found using the inheritance rules described in Chapter 5, "Inheritance and Lookup."

When a message is sent, the frame to which it is sent is the receiver. The implementor and the receiver will differ when the method is found in a frame that is in the inheritance chain of the receiver.

To illustrate this last point consider the following two frames, `frame1` and `frame2`:

```
frame1 := { greeting : "HI!",
            sayHi : func() print(greeting) };
frame2 := { greeting: "Hello!",
            _proto : frame1 };
```

In the following expression `frame1` is both the receiver and the implementor:

```
frame1:sayHi();
"HI"
```

In this next expression, however, `frame2` is the receiver and `frame1` is the implementor:

```
frame2:sayHi();
"Hello!"
```

Note that the value of the variable, `greeting`, is based on the receiver, not the implementor. See the section "Inheritance Rules for Slot and Message Lookup" on page 5-7 for a discussion of this issue.

Invoking a function by using the `call with` syntax sets the value of `self` (the receiver) to the value saved in the function's message environment. This is in contrast to sending a message, where the receiver is changed to the frame specified in the message-send expression.

Self

The value of the pseudo-variable `self` is always set to the value of the receiver. Therefore, you can use `self` to reference the receiver in your code. Note that you cannot set `self` as you could a real variable (in an assignment for instance), hence its designation as a pseudo-variable.

For example, when `sayHi` executes in the following assignment, the value of `self` will be `frame1`:

```
x := frame1:sayHi();
```

An Example Function Object

The following example illustrates how the context of a function object is used to find values for the variables in the function. This example is complicated in that functions are nested, and the inheritance mechanism is utilized; this is to demonstrate how every part of a function object is used. You may want to skip this section, and come back to it after having read Chapter 5, "Inheritance and Lookup."

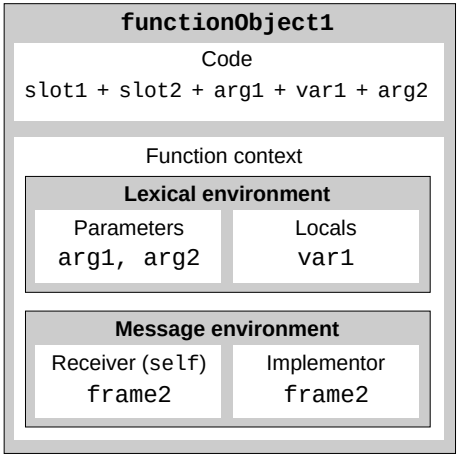
```
frame1 := {slot1 : 5};
frame2 := {
   _parent : frame1,
   slot2 : 40,
   outerMethod : func (arg1)
            begin
              local var1 := 2000;
              local nestedMethod := func (arg2)
                     slot1 + slot2 + arg1 + var1 + arg2;
             nestedMethod;
            end;
   }
```

When `outerMethod` executes through the following message-send

```
functionObject1 := frame2:outerMethod(300);
```

it returns the function object `nestedMethod`. This function object is stored in the variable `functionObject1`, shown in Figure 4-2.

Figure 4-2 functionObject1 dissected



This message-send saves the environment in which it was created. This contextual information provides values for the parameters `arg1` (300) and the local variable `var1` (2000). It also provides a value for `self`, the receiver of the message-send. This allows NewtonScript to provide values for `slot2` (40) and the inherited `slot1` (5).

When `functionObject1` is executed, as in the following function call, NewtonScript can properly lookup the value of all the addends:

```
call functionObject1 with (10000);
```

This returns `12345`.

Note

The following message-send does not work:

```
aFrame := {aSlot : functionObject1}
aFrame:aSlot(10000);
```

This is because the message-send changes the receiver to `aFrame`, and NewtonScript is unable to produce values for `slot1` and `slot2`.

`call aFrame.aSlot with (10000)` still functions properly, however. ◆

Using Function Objects to Implement Abstract Data Types

One use of function objects is to implement abstract data types. These are types that can only be modified procedurally; their actual data is hidden. Though it might appear so, frames with methods don't provide the same functionality. In a frame, the data values in the slots are visible and can be modified even when not using the appropriate methods. Consider the following account generator:

```
MakeAccount := func()begin
   local balance := 0;
   local d := func(amount) begin
      balance := balance + amount;
   end;
   local c := func() begin
      balance := 0;
   end;
   {Deposit: d, Clear: c};
end;
myAccount := call MakeAccount with ();
```

Calling `MakeAccount` returns a frame containing two function objects, `d` and `c`. The function objects in this frame reference the local variable `balance` from `MakeAccount`. Even though `MakeAccount` is no longer executing, the `balance` variable continues to exist, because the nested functions `Deposit` and `Clear` reference it. Thus, calling `myAccount` modifies the hidden variable `balance`, as you can see in the following Inspector output:

```
call myAccount.Deposit with (50);
#C8       50
call myAccount.Deposit with (75)
#1F4      125
call myAccount.Clear  with ();
#0        0
```

Also since neither `Deposit` nor `Clear` utilizes the value of `self`, message-sends can be used as well as the `call/with` syntax:

```
myAccount:Deposit (20);
#50       20
```

Native Functions

When the keyword `native` appears in a function constructor, some compilers generate native code for that function. Native code is machine language code executed directly by the Newton processor.

There are a number of considerations involved in deciding whether to declare a function `native`. For a detailed discussion of these issues, see the chapter "Tuning Performance," in the *Newton Toolkit User's Guide*.

Inheritance and Lookup

NewtonScript supports several object-oriented features and concepts through its double **inheritance** scheme. Frames are the basic data structure in NewtonScript. Inheritance between frames is set up through slots named `_parent` and `_proto`. This chapter describes **parent** and prototype (**proto**) inheritance. It also tells you

- how to set up frames with these relationships
- the rules associated with parent and prototype inheritance
- how inheritance affects slot and method lookup
- how inheritance affects setting slot values
- the uses of parent and prototype inheritance

Inheritance

There are two kinds of inheritance in NewtonScript: prototype inheritance and parent inheritance.

Prototype Inheritance

A frame can have a prototype, which is simply another frame it names as the value of a `_proto` slot. A frame inherits slots from its prototype if it does not contain them in itself. If a frame contains a slot with the same name as a slot in the prototype, it overrides the value of the prototype slot.

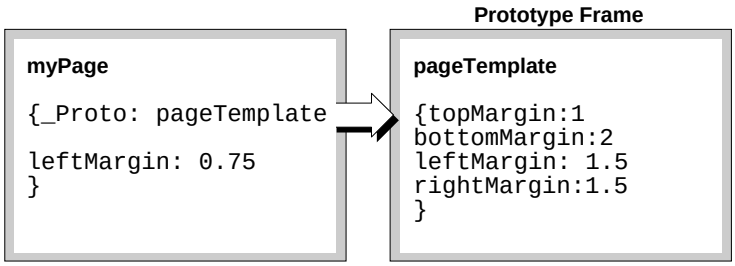
You use inheritance from prototype frames (abbreviated as protos) for

- object refinement—the Newton system has many user interface elements that are system protos you can use or modify in your own interface
- persistent storage of data—these values are commonly stored in ROM or on a PCMCIA card

Creating Prototype Frames

You create a prototype relationship between frames by using a special slot named `_proto`. The value of this slot must be a reference to the frame you intend to use as your prototype frame. For example, to use a frame called `pageTemplate` as a prototype for a frame called `myPage`, you include a `_proto` slot that evaluates to a reference to the `pageTemplate` frame. This is illustrated in Figure 5-1.

Figure 5-1 A prototype frame



Prototype Inheritance Rules

If a function in the frame `myPage` references the slot named `topMargin` during run time, as shown in Figure 5-1, the interpreter looks for the value of `topMargin` in the frame `myPage` first. It doesn't find the slot there it follows the `_proto` reference to the frame, `pageTemplate`. There it finds a `topMargin` slot and its value, 1. In this case, the frame `myPage` **inherits** that slot.

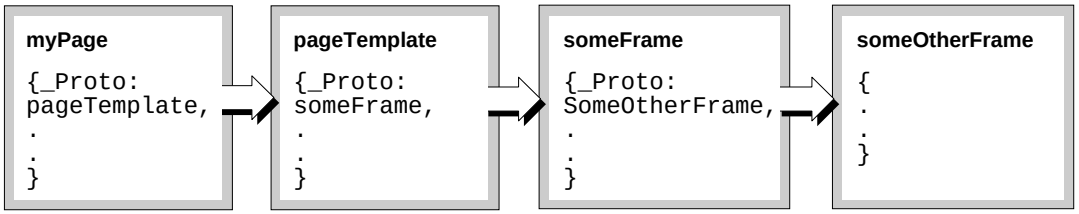
If a function in `myPage` references the slot named `leftMargin`, that slot is found in the current frame and evaluates to the value 0.75. In this case, the value in the current frame overrides the value found in the prototype.

Note that methods in frames can also be inherited and overridden.

The system obtains values during run time by following the prototype inheritance rules for looking up slot references. NewtonScript looks first in the current frame for a slot name. If the slot is not found, it looks at the prototype frame, and if the slot is still not found, it looks at the prototype frame of that frame, and so on, through all the prototypes in the chain.

An example of a prototype chain is shown in Figure 5-2. In this figure, the inheritance chain starts with the current frame, `myPage`, and follows the arrows to its prototype frames on the right.

Figure 5-2 A prototype chain



Parent Inheritance

Besides prototypical relationships between frames, you can set up hierarchical parent-**child** relationships.

Inheritance from parent frames is used for

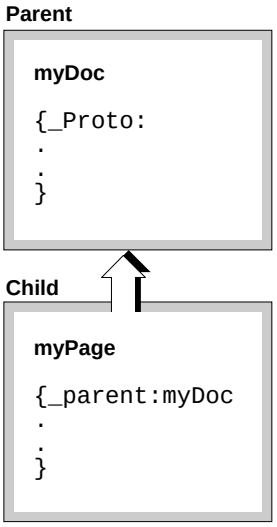
- sharing information between objects, both behavior and data objects
- creating hierarchies, like the view hierarchy of Newton applications

Creating Parent Frames

The parent-child link between frames exists by way of a special slot named `_parent`, which resides in the child frame. You can set this slot directly in your code or you can use the drawing tools in the Newton Toolkit to create view hierarchies automatically. See the *Newton Toolkit User's Guide* for more information about how to do this. Figure 5-3 shows an example of a parent-child relationship between two frames.

Frames that serve as parents can themselves be children of other frames, thereby forming an inheritance chain extending upwards.

Figure 5-3 Parent-child relationship



Parent Inheritance Rules

When you create parent-child hierarchies between frames, NewtonScript uses an inheritance mechanism that works similarly to prototype inheritance.

As in prototype inheritance, a child frame inherits slots from its parent that it does not itself contain. However, if a child frame contains a slot name that is the same as one in a parent frame, the child slot overrides the parent.

In this sense parent inheritance rules are like prototype inheritance rules; the same mechanism is involved. They differ, however, in that the prototype inheritance chain is searched in some instances where the parent inheritance chain is not, and assignment of inherited slots is handled differently in the two types of inheritance.

Combining Prototype and Parent Inheritance

In practice, most frames have prototypes and parents. When a slot is referenced during run time, the parent inheritance mechanism interacts with the prototype inheritance mechanism.

The basic rules for inheritance order are

1. NewtonScript looks first in the initial frame for a referenced slot. In variable lookup the initial frame is the current receiver; in message lookup it is the given receiver.
2. If the slot is not found, the prototype chain of the initial frame is searched.
3. If the slot is still not found, the search moves up one level to the parent frame. The parent and its prototype chain are searched in order. The search then moves up another level (to the parent's parent) and continues in the same way until the slot is found.

The numbered arrows in Figure 5-4 indicate the order in which frames are searched for a slot reference that is made from a function in the current frame, when that frame has both parent and proto frames.

Basically, prototype inheritance takes precedence over parent inheritance; all prototype frames on one level are searched before moving up to search a parent frame and its prototypes on another level.

Remember that these rule take effect within the context of the rules for variable lookup. When looking up the value of a variable, NewtonScript searches for the variable first as a local, then as a global, and finally through the inheritance structure.

Figure 5-4 Prototype and parent inheritance interaction order

Inheritance Rules for Slot and Message Lookup

There are a number of ways in which a slot can be accessed in NewtonScript. Some of these ways search both inheritance chains, some search only the prototype chain, and some search neither.

If the slot name appears by itself, then both inheritance chains are searched. In the following expressions, for example, the values of `chapterNum`, and

`rightMargin` are searched in the order shown by the arrows in Figure 5-4 (after being searched for as locals and globals, of course).

```
presentChapter := chapterNum;
if rightMargin > 1.0 then ...
```

However, if the `frame.slot` or `frame.(pathExpression)` syntax is used, as in the following examples, then only the prototype chain will be searched.

```
if myChap.header then ...
x:= self.topMargin;
```

A message-send searches both inheritance chains, whether the receiver is explicitly mentioned, as in `frame:Message()`, or is omitted, as in `:Message()`. An exception to this rule is that if the keyword `inherited` is used, as in `inherited:Message()`, the search will begins with the current frame's prototype frame, and only follows the prototype chain.

Note

Arguments can be made both for and against using `self:Message()` as opposed to `:Message()`. On the one hand, `self:Message()` looks like `self.slot`, which does not follow the parent inheritance chain and thus might cause confusion. On the other hand, `self:Message()` is arguably more readable, and a common bug can be avoided by always using this format. Consider these two seemingly correct lines of code:

```
:Message1()
:Message2()
```

This sends `Message2` to whatever `:Message1()` evaluates to, which is not what was intended. Including `self` would have prevented this bug. Another way to avoid this type of bug is to always include a semicolon (`;`) after an expression. ◆

NewtonScript also has two built-in functions that can be used for accessing frame slots: `GetVariable` and `GetSlot`. `GetVariable` searches both inheritance chains, and `GetSlot` searches neither. See Chapter 6, "Built-In Functions," for a description of these functions.

Appendix E, "Quick Reference Card," summarizes the information in this section.

Inheritance Rules for Testing for the Existence of a Slot

Inheritance rules for testing for the existence of a slot are basically the same as those for slot lookup. When the slot name appears by itself, as in `slot exists,` the full inheritance chain is searched. If a frame is explicitly mentioned however, as in `frame.slot exists` or `self.slot exists`, only the prototype chain is searched.

A method is searched in both inheritance chains, whether a frame appears before the colon, as in `frame:Message exists`, or not, as in `:Message exists`.

NewtonScript also provides two built-in functions for testing whether a slot exists: `HasVariable,` and `HasSlot`. `HasVariable` searches both parent and prototype chains, and `HasSlot` searches neither. See Chapter 6, "Built-In Functions," for a description of these functions

Appendix E, "Quick Reference Card," summarizes the information in this section.

Inheritance Rules for Setting Slot Values

Inheritance rules apply not only when a slot is referenced, but also when its value is set. However, the rules are slightly different for setting the value of a slot.

The basic difference is that slot values are changed in parent frames only during run time. One reason for this is that prototype frames often exist in ROM and, therefore, cannot be changed. (Of course, when you first create the prototype frames their slots can be set, but not when the application is running.)

When setting a slot, the inheritance search is the same as for slot lookup, except that the slot is not always set where it is found.

These are the rules for where a slot is set:

1. If a slot exists in the currently executing frame, its value is set there.
2. If the slot exists in the prototype chain of the current frame, a new slot is made in the currently executing frame and its value is set there.
3. If the slot exists in the parent of the currently executing frame, its value is set in that parent frame.
4. If the slot exists in the prototype chain of the parent, a new slot is made in the parent frame at the same level at which it was found, and its value is set in that parent frame.

Note that if you create a variable from within a function, it is created as a local variable that is restricted to the scope of the function. If you want to make sure the slot is made in the receiver, you must specify that by using `self`, in an expression like `self.slotName := aValue;`

If you want to set the value of a slot explicitly in the parent of the current frame, you can use the expression `self._parent.theSlot` to force the slot to be created there.

Note

It is unsafe to reference the `_parent` slot directly as a simple expression. A few work arounds are available, however. You can use the view system message `:Parent()`, which returns the current receiver's parent frame. Also, using `frame._parent` or `self._parent` avoids this problem. In summary:

`_parent`	is risky
`frame:Parent()`	is OK
`self._parent`	is OK (and should be the same as `:Parent()`)
`frame._parent`	is OK (and should be the same as `frame:Parent()`)

See the *Newton Programmer's Guide* for more information on `:Parent()` method. ◆

An Object-Oriented Example

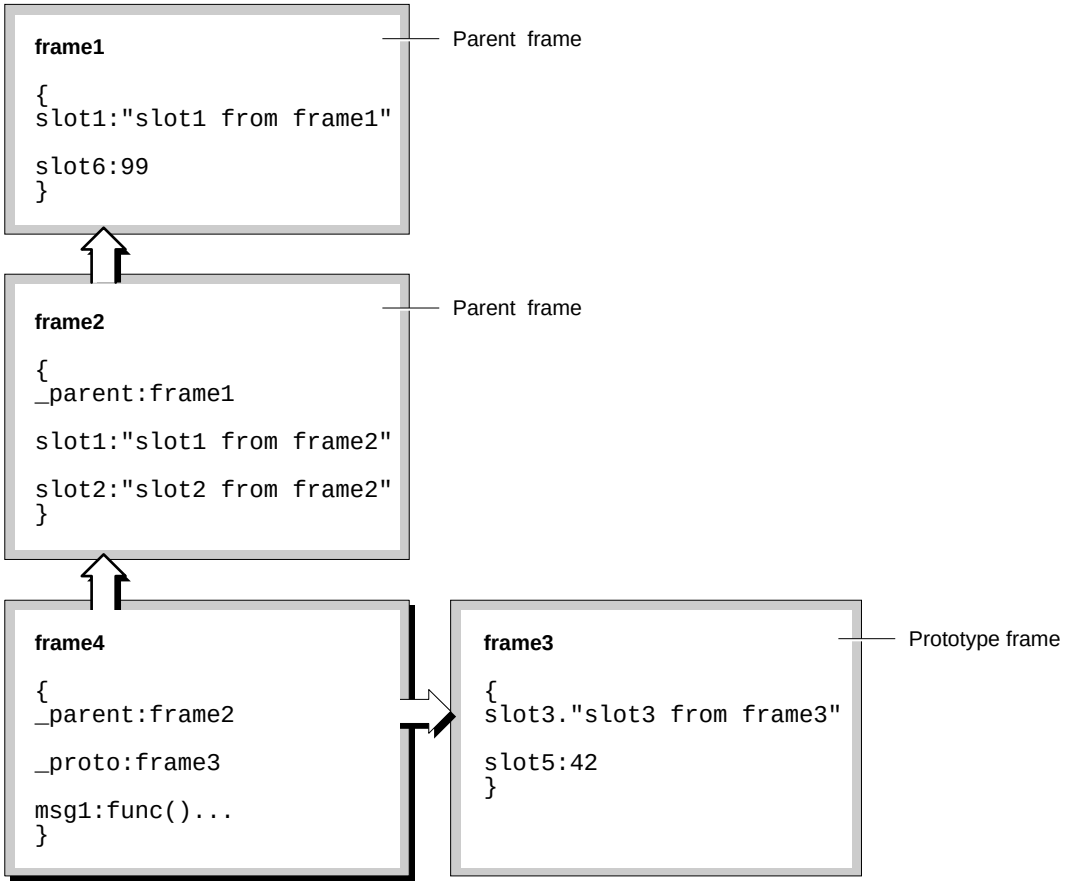
You may understand inheritance better if you construct an inheritance structure on which to experiment. You can use the following code:

```
frame1 := {
      slot1: "slot1 from frame1",
      slot6: 99};

frame2 := {
      _parent: frame1,
      slot1: "slot1 from frame2",
      slot2: "slot2 from frame2"};

frame3 := {
      slot3: "slot3 from frame3",
      slot5: 42};

frame4 := {
      _parent: frame2,
      _proto: frame3,
      msg1: func()
            begin
            //show slot from parent inheritance
            Print(slot1);

            //show slot from proto inheritance
            Print(slot3);

            //show slot from parent inheritance - again -
            //but doesn't work because
            // self.slot1 only searches proto chain
            Print(self.slot1);

            //show slot from proto inheritance - again
            Print(self.slot3);
      end }
```

This produces the inheritance structure shown in Figure 5-5. When the message `frame4:msg1()` is sent, the following output is produced:

```
"slot1 from frame2"
"slot3 from frame3"
NIL
"slot3 from frame3"
```

Figure 5-5 An inheritance structure



Built-In Functions

NewtonScript supports a number of built-in functions. The following groups of functions are included here:

- Object system
- String
- Bitwise
- Array and sorted array
- Math
- Floating point math
- Control of floating point math
- Financial
- Exception Handling
- Message sending
- Data extraction
- Data stuffing
- Getting and Setting Global Variables and Functions
- Miscellaneous

Note
The inspector examples used throughout this document often include a number after a pound sign; for example, #4945. This information can be ignored as it is an internal pointer to data in the system. ◆

Compatibility

This section describes the changes made to the built-in functions for Newton System Software 2.0.

New Functions

The following new functions have been added for this release.

New Object System Functions

The following new object system functions have been added.

`GetFunctionArgCount`
`IsCharacter`
`IsFunction`
`IsInteger`
`IsNumber`
`IsReadOnly` (existed in 1.0 but now documented)
`IsReal`
`IsString`
`IsSubclass` (existed in 1.0 but now documented)
`IsSymbol`
`MakeBinary`
`SetVariable`
`SymbolCompareLex`

New String Functions

The following new string functions have been added.

```
CharPos
StrExactCompare
StrTokenize
StyledStrTruncate
```

New Array Functions

The following new array functions have been added.

```
ArrayInsert
InsertionSort
LFetch
LSearch
NewWeakArray
StableSort
```

New Sorted Array Functions

The following new functions have been added that operate on sorted arrays. These functions are based on binary search algorithms, hence the "B" prefix to the function names.

```
BDelete
BDifference
BFetch
BFetchRight
BFind
BFindRight
BInsert
BInsertRight
BIntersect
BMerge
BSearchLeft
BSearchRight
```

New Message Sending Functions

The following new utility functions for sending immediate messages have been added.

```
PerformIfDefined
ProtoPerform
ProtoPerformIfDefined
```

New Data Stuffing Functions

The following functions have been added to stuff data.

```
StuffCString
StuffPString
```

New Functions to Get and Set Globals

The following new functions that get, set, and check for the existence of global variables and functions have been added.

```
GetGlobalFn
GetGlobalVar
GlobalFnExists
GlobalVarExists
DefGlobalFn
DefGlobalVar
UnDefGlobalFn
UnDefGlobalVar
```

New Miscellaneous Functions

The following function has been added.

```
BinEqual
```

Obsolete Functions

Some built-in functions previously documented in the *NewtonScript Programming Language* are obsolete, but are still supported for compatibility with older applications. Do not use the following utility functions, as they may not be supported in future system software versions:

`ArrayPos` (use `LSearch` instead)
`StrTruncate` (use `StyledStrTruncate` instead)

Object System Functions

The functions described in this section operate on NewtonScript objects. They perform operations such as removing slots, cloning frames, and so forth.

ClassOf

`ClassOf(`*object*`)`

Returns the class of an object.

object The object whose class to return.

The return value is a symbol. Some of the common object classes are: `'int`, `'char`, `'boolean`, `'string`, `'array`, `'frame`, `'function`, and `'symbol`. Note that this is not necessarily the same as the primitive class of an object. For binary, array, and frame objects, the class can be set differently from the primitive class.

Frames or arrays without an explicitly assigned class are of the primitive class `'frame` or `'array`, respectively. If a frame has a class slot, the value of the class slot will be returned. Here are some examples:

```
f:={multiply:func(x,y) x*y};
classof(f);
#1294    Frame
```

```
f:={multiply:func(x,y) x*y, class:'Arithmetic};
classof(f);
#1294     Arithmetic

s:="India Joze";
classof(s);
#1237    String
```

See also "PrimClassOf" on page 6-13.

Clone

`Clone(`*object*`)`

Makes and returns a "shallow" copy of an object; that is, references within the object are copied, but the data pointed to by the references is not.

object The object to copy.

Here is an example:

```
SeaFrame :=  {Ocean: "Pacific", Size: "large" , Color: "blue"};
seaFrameCopy := clone(seaFrame);
seaFrameCopy.Deep := true;
seaFrame
#441896D  {Ocean: "Pacific", size: "large", Color: "blue"}
seaFrameCopy
#4418B0D  {Ocean: "Pacific", size: "large", Color: "blue",
   Deep: TRUE}
```

DeepClone

`DeepClone(`*object*`)`

Makes and returns a "deep" copy of an object; that is, all of the data referenced within the object is copied, including that referenced by magic pointers (pointers to ROM objects).

object The object to copy.

It is not guaranteed that every part of the data structure is in RAM. (Certain information, such as the symbols naming frame slots, may be shared with the original object.)

Contrast this function with `Clone` that only makes a "shallow" copy, and the `EnsureInternal` function that ensures that the object exists entirely in internal RAM.

GetFunctionArgCount

`GetFunctionArgCount(`*function*`)`

Returns the number of arguments expected by a function.

function	The function whose number of arguments you want to get.

GetSlot

`GetSlot(`*frame*`, `*slotSymbol*`)`

Returns the value of a slot in a frame. Only the frame specified is searched.

frame	A reference to the frame in which to look for the slot.
slotSymbol	A symbol naming the slot whose value you want to get.

If the slot doesn't exist, this function returns `nil`.

Unlike `GetVariable`, `GetSlot` searches for a slot only in the indicated frame. Inheritance is not used to find the slot.

The use of the NewtonScript dot operator is similar to the `GetSlot` function in that it also returns the value of a frame slot. For example, the expression `frame.slot` returns the value of the specified slot. However, when using the dot operator, if the slot is not found in the specified frame, proto frames are also searched for the slot (but not parent frames).

GetVariable

`GetVariable(`*frame*`, `*slotSymbol*`)`

Returns the value of a slot in a frame. If the slot is not found, `nil` is returned.

frame A reference to the frame in which to begin the search for the slot.

slotSymbol A symbol naming the slot whose value you want to get.

This function begins its search for the slot in the specified frame and makes use of the full proto and parent inheritance.

HasSlot

`HasSlot(`*frame*`, `*slotSymbol*`)`

Returns non-`nil` if the slot exists in the frame, otherwise, returns `nil`. Inheritance is not used to find the slot.

frame The name of the frame in which to look for the slot.

slotSymbol A symbol naming the slot whose value you want to get.

This function begins its search for the slot in the specified frame and makes use of the full proto and parent inheritance.

HasVariable

`HasVariable(`*frame*`, `*slotSymbol*`)`

Returns non-`nil` if the slot exists in the frame, otherwise, returns `nil`. This function searches proto and parent frames of the specified frame if the slot is not found there.

frame The name of the frame in which to begin the search for the slot.

slotSymbol A symbol naming the slot whose existence you want to check. You must use a single quote before the slot name because it is a symbol.

Intern

`Intern(` *string* `)`

Creates and returns a symbol whose name is given as the string parameter *string*. If a symbol with that name already exists, the preexisting symbol is returned.

string The name of the symbol.

IsArray

`IsArray(`*obj*`)`

Returns non-`nil` if *obj* is an array.

obj The object to test.

IsBinary

`IsBinary(`*obj*`)`

Returns non-`nil` if *obj* is a binary object.

obj The object to test.

IsCharacter

`IsCharacter(`*obj*`)`

Returns non-`nil` if *obj* is a character, and returns `nil` otherwise.

obj The object to test.

IsFrame

`IsFrame(`*obj*`)`

Returns non-`nil` if *obj* is a frame.

obj The object to test.

IsFunction

`IsFunction(`*obj*`)`

Returns non-`nil` if *obj* is a function, and returns `nil` otherwise.

obj The object to test.

IsImmediate

`IsImmediate(`*obj*`)`

Returns non-`nil` if *obj* is an immediate.

obj The object to test.

IsInstance

`IsInstance(`*obj*`,` *class*`)`

Returns non-`nil` if *obj*'s class symbol the same as *class* or a subclass of *class*.

obj The object to test.

class A class symbol.

Note that this is equivalent to:

`IsSubclass(ClassOf(`*obj*`),` *class*`)`

IsInteger

`IsInteger(`*obj*`)`

Returns non-`nil` if *obj* is an integer, and returns `nil` otherwise.

obj The object to test.

IsNumber

`IsNumber(`*obj*`)`

Returns non-`nil` if *obj* is a number (integer or real), and returns `nil` otherwise.

obj The object to test.

IsReadOnly

`IsReadOnly(`*obj*`)`

Returns non-`nil` if *obj* is read-only, and returns `nil` otherwise. You can use `IsReadOnly` to determine if an array, frame, or binary object is writable.

obj An array, frame, or binary object to test. (Immediate objects such as integers are never read-only.)

Here is an example:

```
if IsReadOnly(viewBounds) then
   viewBounds := Clone(viewBounds);
```

This function should not be used to determine the location of an object, that is, whether it is in the heap, in ROM, or in protected memory. The NewtonScript language could permit read-only objects in the NS heap, or writable objects that exist in other locations.

IsReal

`IsReal(`*obj*`)`

Returns non-`nil` if *obj* is a real number, and returns `nil` otherwise.

obj The object to test.

IsString

`IsString(`*obj*`)`

Returns non-`nil` if *obj* is a string, and returns `nil` otherwise.

obj The object to test.

IsSubclass

`IsSubclass(`*sub*, *super*`)`

Checks if a class is a subclass of another class.

sub A class symbol you want to test.

super A class symbol.

This function returns non-`nil` if *sub* is a subclass of *super*, or is the same as *super*. Returns `nil` if *sub* is not a subclass of *super*. See also the related function `IsInstance` on page 6-10.

IsSymbol

`IsSymbol(`*obj*`)`

Returns non-`nil` if *obj* is a symbol, and returns `nil` otherwise.

obj	The object to test.

MakeBinary

`MakeBinary(`*length*`, `*class*`)`

Allocates a new binary object of the specified *length* and *class*.

length	The size of the binary object in bytes.
class	A symbol specifying the class

Map

`Map(`*obj*`, `*function*`)`

Applies a function to the slot name and value of each element of an array or frame.

obj	An array or frame.
function	Returns `nil`. A function to apply to the elements or slots in *obj*. The function is passed two parameters: *slot* and *value*. The *slot* parameter contains an integer array index if *obj* is an array, or it contains a symbol naming a slot, if *obj* is a frame. The *value* parameter contains the value of the array or frame slot referenced by the *slot* parameter.

This is equivalent to:

`foreach slot,value in` *obj* `do call` *function* `with (slot,value)`

PrimClassOf

`PrimClassOf(`*obj*`)`

Returns the primitive class of an object.

obj	The object whose primitive class to return.

Returns a symbol identifying the primitive data structure type of the object, one of: `'immediate`, `'binary`, `'array`, or `'frame`.

See also "ClassOf" on page 6-5.

RemoveSlot

`RemoveSlot(`*obj*`, `*slot*`)`

Removes a slot from a frame or array.

obj	The name of the frame or array from which to remove the slot.
slot	A symbol naming the frame slot you want to remove, or the index of the array slot to remove. Note that no inheritance look-up is used to find this slot in *obj*.

This function returns the modified frame or array. If *slot* is not found, nothing is done and the unmodified frame or array is returned. Note that the system throws an exception if *obj* is read-only.

ReplaceObject

`ReplaceObject(`*originalObject*`, `*targetObject*`)`

Causes all references to an object to be redirected to another object.

originalObject	The original object.
targetObject	The object to which you want to redirect references to *originalObject*.

This function always returns `nil`.

Note that you cannot specify immediate objects as parameters to this function.

Here is an example:

```
x:={name:"Star"};
y:={name:"Moon"};
replaceobject(x,y);
x;
#469E69   {name: "Moon"}

y;
#46A1E9   {name: "Moon"}
```

SetClass

`SetClass(`*obj*`, `*classSymbol*`)`

Sets the class of an object.

obj	The object whose class to set.
classSymbol	A symbol naming the class to give to the object.

This function returns the object whose class was set.

You can set the class of the following kinds of objects: frames, arrays, and binary objects. Note that you cannot set the class of an immediate object.

When setting the class of a frame, if a `class` slot doesn't exist, one is created in the frame. For example:

```
x:={name: "Star"};
setclass(x, 'someClass);
#46ACC9   {name: "Star",
           class: someClass}
```

SetVariable

`SetVariable(`*frame*`, `*slotSymbol*`, `*value*`)`

Sets the value of a slot in a frame. The value is returned.

frame	A reference to the frame in which to begin the search for the slot.
slotSymbol	A symbol naming the slot whose value you want to set. If the slot is not found, it is created in *frame*.
value	The new value of the slot.

This function begins its search for the slot in the specified frame and makes use of the full proto and parent inheritance.

Note that if the slot is found in the proto chain, it is not set there, but is created and set in *frame*, or in its parent chain, following the usual inheritance rules as they apply to setting a value.

SymbolCompareLex

`SymbolCompareLex(`*symbol1*`, `*symbol2*`)`

Compares symbols lexically. This function returns a negative number if symbol *symbol1* is less than symbol *symbol2*. Returns zero if the two symbols are equal. Returns a positive number if *symbol1* is greater than *symbol2*. Case is not significant (that is, `'Hello` and `'hello` are equal).

symbol1	A symbol.
symbol2	A symbol.

TotalClone

`TotalClone(`*obj*`)`

Makes and returns a “deep” copy of an object; that is, all of the data referenced within the object is copied.

obj	The object to copy.

This function is similar to `DeepClone`, except that this function guarantees that the object returned exists entirely in internal RAM. Also, unlike

`DeepClone`, `TotalClone` does not follow magic pointers, so that objects referenced through magic pointers are not copied.

String Functions

These functions operate on and manipulate strings.

BeginsWith

`BeginsWith(` *string*`,` *substr* `)`

Returns non-`nil` if *string* begins with *substr,* or returns `nil` otherwise. This function is case and diacritical-mark insensitive. An empty *substr* matches any *string*.

string	The string to test.
substr	A string.

Capitalize

`Capitalize(` *string* `)`

Capitalizes the first character in *string* and returns the result. *string* is modified.

string	The string to modify.

CapitalizeWords

`CapitalizeWords(` *string* `)`

Capitalizes the first character of each word in *string* and returns the result. *string* is modified.

string	The string to modify.

CharPos

`CharPos(`*str*`, `*char*`, `*startpos*`)`

Returns the position of the next occurrence of character in the specified string, starting from the *startPos* (or `nil` if it's not found).

str	The specified string.
char	The specified character in the string.
startpos	The starting position of the character to return.

Downcase

`Downcase(` *string* `)`

Changes each character in *string* to lowercase and returns the result. *string* is modified.

string	The string to modify.

EndsWith

`EndsWith(` *string*`, `*substr* `)`

Returns non-`nil` if *string* ends with *substr,* or returns `nil` otherwise. This function is case and diacritical-mark insensitive. An empty *substr* matches any *string*.

string	The string to test.
substr	A string.

IsAlphaNumeric

`IsAlphaNumeric(`*char*`)`

Returns non-`nil` if *char* is a number or a letter; otherwise, this function returns `nil`.

char	A character to test.

IsWhiteSpace

`IsWhiteSpace(`*char*`)`

Returns non-`nil` if *char* is a space ($\20), tab ($\09), linefeed ($\0A), or carriage return ($\0D) character; otherwise, this function returns `nil`.

char A character.

SPrintObject

`SPrintObject( `*obj*` )`

Returns a string of the object passed in. Numbers, strings, characters, and symbols are converted to their natural string representation. For frames, arrays, and Booleans, this function returns an empty string.

To convert a Boolean into a string, you must check for non-`nil` or `nil` and return the appropriate string.

Note
This function changes the number format depending on the current locale setting. Real numbers may be formatted unexpectedly. ◆

StrCompare

`StrCompare( `*a*`, `*b*` )`

Returns a negative number if string *a* is less than string *b*. Returns zero if string *a* and *b* are equal. Returns a positive number if string *a* is greater than string *b*. Case is not significant (that is, “Hello” and “hello” are equal).

a A string.

b A string.

Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrExactCompare` to do a case-sensitive comparison of strings.

StrConcat

`StrConcat( a, b )`

Concatenates string *b* onto string *a* and returns the result as a new string.

a A string.

b A string.

StrEqual

`StrEqual( a, b )`

Returns non-`nil` if the two strings, *a* and *b*, are equal.

a A string.

b A string.

Case is not significant. Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrExactCompare` to do a case-sensitive comparison of strings.

StrExactCompare

`StrExactCompare( a, b )`

Returns a negative number if string *a* is less than string *b*. Returns zero if string *a* and *b* are equal. Returns a positive number if string *a* is greater than string *b*. Case and diacritical marks are significant (that is, “Hello” and “hello” are not equal).

a A string.

b A string.

Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrCompare` or `StrEqual` to do a case-insensitive comparison of strings.

StrLen

`StrLen(` *string* `)`

Returns the number of characters in a string, excluding the null terminator (if one exists).

string A string.

StrMunger

`StrMunger(` *dstString, dstStart, dstCount, srcString, srcStart, srcCount* `)`

Replaces characters in *dstString* using characters from *srcString* and returns the destination string after munging is complete. This function is destructive to *dstString*.

dstString The destination string. The string must be writable, you can't specify a string literal, or an exception will be thrown. You'll have to use `Clone` (page 6-6) or a similar function to make a writable copy from a string literal.

dstStart The starting position within *dstString*.

dstCount The number of characters to be replaced in *dstString*. You can specify `nil` for *dstCount* to go to the end of the string.

srcString A string. This can be `nil` to simply delete the characters.

srcStart The starting position in *srcString* from which to begin taking characters to place into *dstString*.

srcCount The number of characters to use from *srcString*. You can specify `nil` to go to the end of *srcString*.

Here is an example:

```
StrMunger("abcdef", 2, 3, "ZYXWV", 0, nil)
"abZYXWVf"
```

`StrMunger` can also be used to concatenate large strings; for example:

```
StrMunger(str1, StrLen(str1)+1, nil, str2, 0, nil);
```

StrPos

`StrPos(` *string*`,` *substr*`,` *start* `)`

Returns the position of *substr* in *string*, or `nil` if *substr* is not found. The search begins at character position *start*. (The first character position in a string is zero.) This function is not case sensitive.

string	A string.
substr	A string.
start	An integer.

Here is an example:

```
StrPos("abcdef", "Bcd", 0)
1
```

StrTokenize

`StrTokenize(`*str*`,` *delimiters*`)`

Breaks up a string into chunks for you as defined by the *delimiters* argument. Each time you call the closure (passing it no arguments) you will get back the next token, until there are no more tokens and it returns `nil`.

str	A string to be broken up into tokens
delimiters	Either a character or string (list of characters) that are the delimiters separating the pieces of the string.

For example, to break up a sentence into space separated words you do something like the following:

```
fn := StrTokenize("the quick green fox", $ );
#441BE8D  <function, 0 arg(s) #441BE8D>
      while x := call fn with () do Print(x);
"the"
"quick"
"green"
"fox"
#2        NIL
```

StyledStrTruncate

`StyledStrTruncate(`*string*`,` *length*`,` *font*`)`

Truncates a string to the indicated length, in pixels. (Of course, the length does not include the null terminator.) Returns the truncated string.

string	A string.
length	An integer specifying the length, in pixels, at which to truncate the string.
font	A font specification, which is used to determine how many characters of the string will fit in the specified length. For details on specifying a font, refer to the section "Specifying a Font" in the chapter "Text Input and Display,"of the *Newton Programmer's Guide*.

This function adds an ellipsis (...) to the end of the truncated string.

SubStr

`SubStr(` *string*`,` *start*`,` *count* `)`

Returns a new string containing *count* characters from *string*, starting at position *start*. Character positions begin with zero for the first character.

string	A string.
start	An integer.
count	An integer.

TrimString

`TrimString(` *string* `)`

Removes any white space (spaces, tabs, and new line characters) from the beginning and end of *string* and returns the result. *string* is modified.

string	A string.

Upcase

`Upcase(` *string* `)`

Capitalizes each character in *string* and returns the result. *string* is modified.

string A string.

Bitwise Functions

These functions perform logical operations on bits.

Band, Bor, Bxor, and Bnot

`Band(`*a*`, `*b*`)`
`Bor(`*a*`, `*b*`)`
`Bxor(`*a*`, `*b*`)`
`Bnot(`*a*`)`

These bitwise functions each return an integer result of their operation on one or two integer parameters. They perform bitwise AND, OR, XOR, and NOT, respectively.

a An integer.

b An integer.

Array Functions

These functions operate on and manipulate arrays.

AddArraySlot

`AddArraySlot (`*array*`, `*value*`)`

Appends a new element onto an array.

array	An array.
value	A value to be added as new element in the array.

For example:

```
myArray := [123, 456]
#1634 myArray
addArraySlot (myArray, "I want chopstix")
#12 "I want chopstix"
myArray
#1634 [123, 456, "I want chopstix"]
```

Array

`Array(` *size*`, `*initialValue* `)`

Returns a new array with *size* number of elements that each contain *initialValue*.

size	An integer.
initialValue	A value.

ArrayInsert

`ArrayInsert(`*array*`, `*element*`, `*position*`)`

Inserts an element into an array and returns the modified array.

array	The array to be modified.
element	The element to be inserted into the array.
position	The index where the new element is to be inserted. Specify zero to insert the element at the beginning of the array. Specify the result of `Length(`*array*`)` to insert the element at the end of the array.

The length of the array is increased by one.

ArrayMunger

`ArrayMunger(` *dstArray*`,` *dstStart*`,` *dstCount*`,` *srcArray*`,` *srcStart*`,` *srcCount* `)`

Replaces elements in *dstArray* using elements from *srcArray* and returns the destination array after munging is complete. This function is destructive to *dstArray*.

dstArray	The destination array.
dstStart	The starting element in the destination array.
dstCount	The number of elements to be replaced in *dstArray*. You can specify `nil` for *dstCount* to go to the end of the array.
srcArray	An array. You can specify `nil` for *srcArray* to delete the elements.
srcStart	The starting position in the source array from which to begin taking elements to place into the destination array.
srcCount	The number of elements to use from the source array. You can specify `nil` to go to the end of the source array.

Here is an example:

```
ArrayMunger([10,20,30,40,50], 2, 3, [55,66,77,88,99], 0, nil)
[10, 20, 55, 66, 77, 88, 99]
```

Using `ArrayMunger` is the most efficient way to join two arrays.

To put B at the front of A:

```
ArrayMunger(A, 0, 0, B, 0, nil)
```

To put B at the end of A:

```
ArrayMunger(A, Length(A), 0, B, 0, nil)
```

You can also do this with `SetUnion` (page 6-34), which has the additional property that it eliminates duplicates, but `ArrayMunger` is much faster if you don't need that property.

ArrayRemoveCount

`ArrayRemoveCount(` *array*`,` *startIndex*`,` *count* `)`

Removes one or more elements from an array.

array	The array from which to remove elements.This parameter is modified by this function.
startIndex	An integer that is the index of the first element to remove.
count	An integer specifying the number of elements to remove.

Any elements following those removed are shifted left so that no empty elements remain.

InsertionSort

`InsertionSort(`*array*`,` *test*`,` *key*`)`

Sorts an array, preserving the original relative ordering of equivalent elements.

array	The array to modify by sorting.
test	Indicates how the array is to be sorted. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This sort performs very well on arrays that are nearly sorted already and on very small arrays. This sort is an $O(n^2)$ sort. To sort larger arrays, use `Sort` or `StableSort`.

Length

`Length (`*array*`)`

Returns the number of elements in an array, the number of slots in a frame, or the size, in bytes, of a binary object.

array An array or frame or binary object.

For example:

```
myArray := [123, 456, "I want chopstix"]
length (myArray)
#12    3
```

Note that arrays are indexed from 0, but length returns a count of the number of characters. Therefore, the last element of this example is element 2.

Note

If you pass a string to this function, you will get the number of bytes that a string occupies. To get the length of strings, use `StrLen` instead. ◆

LFetch

`LFetch(`*array*`, `*item*`, `*start*`, `*test*`, `*key*`)`

Linearly searches an array for the specified element and returns the element, or `nil` if it is not found or if *start* is equal to or greater than the length of the array.

array The array in which to search.

item The key value for which to search.

start The array index at which to begin searching.

test Indicates how to compare key values to test for a match. Specify one of the following symbols for *test*:

`'|=|`	If the objects being compared are immediates and reals, their values are compared for equivalency. For reference objects, their identity is compared.
`'|str=|`	For string objects, the contents of the strings are compared for equivalency.

Alternatively, for nonstandard sorting situations, you can specify a function object that compares two key values and returns a Boolean or integer value indicating whether or not they are equivalent. This function will be called to test for matches. The function is passed two parameters, *A* and *B*, where *A* is the *item* parameter passed to `LFetch` and *B* is the array element being tested.

The function must return a non-`nil` value (or zero) if the items are equivalent, or `nil` (or a non-zero integer) if the items are not equivalent.

Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This function works just like `LSearch`, except that `LSearch` returns the index of the found item.

If you know that the array you are working with is sorted, you can use the function `BFetch` to search for an element. This function, based on binary search algorithms, is much faster on large arrays than `LFetch` or `LSearch`, though it can be used only on sorted arrays.

LSearch

`LSearch(`*array*`, `*item*`, `*start*`, `*test*`, `*key*`)`

Linearly searches an array for the specified element and returns the index of the element, or `nil` if it is not found or if *start* is equal to or greater than the length of the array.

array — The array in which to search.

item — The key value for which to search.

start — The array index at which to begin searching.

test — Indicates how to compare key values to test for a match. Specify one of the following symbols for *test*:

- `'|=|` — If the objects being compared are immediates and reals, their values are compared for equivalency. For reference objects, their identity is compared.
- `'|str=|` — For string objects, the contents of the strings are compared for equivalency.

Alternatively, for non-standard sorting situations, you can specify a function object that compares two key values and returns a Boolean or integer value indicating whether or not they are equivalent. This function will be called to test for matches. The function is passed two parameters, *A* and *B*, where *A* is the *item* parameter passed to `LSearch` and *B* is the array element being tested. The function must return a non-`nil` value (or zero) if the items are equivalent, or `nil` (or a non-zero integer) if the items are not equivalent. Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key — Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This function works just like `LFetch`, except that `LFetch` returns the found item instead of its index.

If you know that the array you are working with is sorted, you can use the function `BFind` to search for an element. This function, based on binary search algorithms, is much faster than `LSearch`, though it can be used only on sorted arrays.

NewWeakArray

`NewWeakArray(`*length*`)`

Returns a new weak array with *length* number of elements, which are initialized to `nil`.

length An integer specifying the size of the array to create.

A weak array is an array that does not prevent the objects it refers to from being garbage-collected. That is, if the only references to an object are from weak arrays, the object is destroyed during the next garbage collection cycle. When that happens, the references in the weak arrays are replaced with `nil`.

The purpose of weak arrays is to cache objects without preventing them from being garbage collected. For example, if you wanted to keep an array of all objects in existence of a certain type, you could add each object to an array as it's created. If you use a regular array, those objects will never be garbage-collected, because there will always be references to them in your array, and the system will eventually run out of memory. However, if you use a weak array, its references don't affect garbage collection, so the objects will be garbage-collected normally, freeing memory when it is needed.

SetAdd

`SetAdd (`*array*`,`*value*`,`*uniqueOnly*`)`

Appends an element to the specified array and returns the modified array, or `nil` if the element was not added.

array	The array to which `SetAdd` appends the element in *value*.
value	The element to append to the array specified by *array*.
uniqueOnly	Whether only unique elements are to be added to the array; if the value of this parameter is non-`nil`, `SetAdd` appends *value* to the array only if it is not already present in the array. If the element specified by the *value* parameter is already present in the array, `SetAdd` returns `nil` and does not append the element. If *uniqueOnly* is `nil`, the item is appended to the array without checking whether it is unique.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetContains

`SetContains(` *array*`,` *item* `)`

array	An array.
item	An item that may be in the array.

Searches each element of an array to determine if *item* is equal to one of the array elements. If a match is found, this function returns the array index of the matching array element. If *item* is not found in the array, `nil` is returned.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetDifference

`SetDifference(` *array1*`,` *array2* `)`

Returns an array that contains all of the elements in *array1* that do not exist in *array2*.

array1	An array.
*array*2	An array.

If *array1* is nil, `nil` is returned.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetLength

`SetLength (`*array*`,` *length*`)`

Sets the length of an array.

array	An array.
length	An integer.

This function is useful for increasing or decreasing the size of an array. If you increase the size of the array, new elements are filled with a `nil` value. For example:

```
myArray := [123, 456, "I want chopstix"]
#1634 myArray
setLength (myArray, 4)
#1634 [123, 456, "I want chopstix", NIL]
myArray [3] := 789
#3156 789
myArray
#1634 [123, 456, "I want chopstix", 789]
```

SetOverlaps

`SetOverlaps(` *array1*`,` *array2* `)`

Compares each element in *array1* to each element in *array2*, and returns the index of the first element in *array1* that is equal to an element in *array2*. If no equivalent elements are found, `nil` is returned.

array1	An array.
*array*2	An array.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetRemove

`SetRemove (`*array*`,` *value*`)`

`SetRemove` removes the specified element from the specified array and returns the modified array. The length of the array is shifted left by one and all of the elements after the deleted element are shifted by one to the next lowest numbered array position. If the item is not found in the array, this function returns `nil`.

array	The array from which `SetRemove` removes the specified element.
value	The element to remove from the array specified by *array*.

Note
The type of comparison used in this function is identity comparison, not pointer comparison. ◆

SetUnion

`SetUnion(` *array1*`,` *array2*`,` *uniqueFlag* `)`

Returns an array that contains all of the elements in *array1* and all of the elements in *array2*.

array1	An array.
*array*2	An array.
uniqueFlag	If any non-`nil` value, `SetUnion` will not include any duplicate items in the array it returns. If *uniqueFlag* is `nil`, all elements from both arrays are included, even if there are duplicates.

If both of the arrays are `nil`, an empty array is returned.

`SetUnion` can eliminate duplicates. If you do not need that property, you can combine two arrays more efficiently using `ArrayMunger` (page 6-25).

Note

The type of comparison used in this function is identity comparison, not pointer comparison. ◆

Sort

`Sort(` *array*`,` *test*`,` *key* `)`

Sorts an array and returns it after it is sorted. The sort is destructive; that is, the array you give it is modified. The sort also is not stable; that is, elements with equal keys won't necessarily have the same relative order after the sort.

array	An array.
test	Defines the sort order. It can be a function object that takes two parameters *A* and *B* and returns a positive integer if *A* sorts after *B*, returns zero if *A* sorts equivalently to *B*, and returns a negative integer if *A* sorts before *B*.

For much greater speed, specify one of the following symbols for *test*:

`'|<|`	Sort in ascending numerical order
`'|>|`	Sort in descending numerical order
`'|str<|`	Sort in ascending string order
`'|str>|`	Sort in descending string order

key Defines the sort key within each array element. Specify `nil` to use the array elements directly as they are. You can specify a path expression, in which case the array elements are assumed to be frames or arrays and the path is applied to each element to find the sort key. Or, you can specify a function that takes one parameter and returns the key.

This example sorts `myArray` in ascending numerical order according to the `timestamp` slot of the entries:

```
Sort(myArray, '|<|, 'timestamp)
```

This example sorts `myArray` in descending string order according to the first and last names concatenated together:

```
Sort(myArray, '|str>|, func (e) e.first && e.last)
```

StableSort

`StableSort(`*array*`, `*test*`, `*key*`)`

Sorts an array, preserving the original relative ordering of equivalent elements.

array The array to modify by sorting.

test Indicates how the array is to be sorted. See the description of the *test* parameter on page 6-36.

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This sort requires working memory, so may not be suitable for extremely large arrays or in low memory conditions.

Sorted Array Functions

This section describes new functions that operate on sorted arrays. These functions are based on binary search algorithms, hence the "B" prefix to the function names.

IMPORTANT

The arrays you pass to these functions must be ordered, otherwise the results are undefined. To sort an array, you can use the functions `Sort`, `InsertionSort`, or `StableSort`. ▲

These sorted array functions each use *test* and *key* parameters to allow them to be adapted to different data structures. Typically, these functions search, or iterate over several items in an array. As each element in an array is examined, the *key* argument is used to extract a value, called the key, from the element. Then that key is treated as specified by the *test* argument.

Here's an explanation of these parameters:

test — Indicates the sort order of the array. Specify one of the following symbols for *test*, to indicate how the array is sorted:

`'|<|`	Sorted in ascending numerical order
`'|>|`	Sorted in descending numerical order
`'|str<|`	Sorted in ascending string order
`'|str>|`	Sorted in descending string order
`'|sym<|`	Sorted in ascending symbol order, based on lexical comparison of symbol name
`'|sym>|`	Sorted in descending symbol order, based on lexical comparison of symbol name

Alternatively, for non-standard sorting situations, you can specify a function object that compares two key values and returns an integer that indicates how they are sorted relative to each other. This function will be called by any of the sorted array functions to determine sorting relationships between elements. The function is passed two parameters, *A* and *B*, and must return a positive integer if *A* sorts after *B*, must return zero if *A* sorts equivalently to *B*, and a must return a negative integer if *A* sorts before *B*. Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key — Defines the key within each array element. Specify `nil` to use the array elements directly as they are. You can specify a path expression, in which case the array elements are assumed to be frames or arrays and the path is applied to each element to find the key. You can also specify a function that takes one parameter (the element) and returns the key.

BDelete

`BDelete(`*array*`, `*item*`, `*test*`, `*key*`, `*count*`)`

Deletes elements from an ordered array.

This function returns the number of elements deleted.

array — The array to be modified.

item — The key value for which to search. Elements with this key are deleted.

test — Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.

key — Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

count	The maximum number of elements to delete. Specify `nil` to indicate that all matching elements are to be deleted.

BDifference

`BDifference(`*array1*`, `*array2*`, `*test*`, `*key*`)`

Returns a new sorted array containing those elements from *array1* that do not have equivalent elements in *array2*.

array1	The first array. This array is not modified.
array2	The second array. This array is not modified.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

BFetch

`BFetch(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The leftmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This function works just like `BFind`, except that `BFind` returns the index of the found item.

BFetchRight

`BFetchRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The rightmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This function works just like `BFindRight`, except that `BFindRight` returns the index of the found item.

BFind

`BFind(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The index of the leftmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This function works just like `BFetch`, except that `BFetch` returns the found item instead of its index.

BFindRight

`BFindRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The index of the rightmost found element is returned, or `nil` is returned if none are found.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

This function works just like `BFetchRight`, except that `BFetchRight` returns the found item instead of its index.

BInsert

`BInsert(`*array*`, `*element*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Inserts an element into the proper position in a sorted array. In the case of equivalent elements, the element is inserted to the left of its equivalent.

array	The array to be modified.
element	The new element to be inserted. Note that the *key* parameter is used to extract its key value.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

uniqueOnly Specify non-`nil` to indicate that the element is not to be inserted if the array already contains an element with an equivalent key value. Specify `'returnElt` to indicate the same thing, and also that this function should return an array element. It returns either the element that was inserted, or if a matching element is found in the array, that element is returned. This is useful when you want to reuse existing objects in order to conserve space or ensure pointer equality.

Specify `nil` to indicate that the element is to be inserted even if the array already contains an element with an equivalent key. In this case, the new element is inserted to the left of the existing equivalent elements.

This function has three possible return values, as follows:

- It can return `nil`, signaling that the element was not inserted.
- It can return an integer, which is the index at which the element was inserted.
- It can return an array element—either the element that was inserted (if it was unique), or an element that already exists in the array, whose key value matches the key value of the element you wanted to insert. This type of return value can occur only if you specify `'returnElt` for *uniqueOnly*.

Here is an example of how you might use this function with *uniqueOnly* set to `'returnElt` to ensure pointer equality:

```
// :GetStr() returns a string input by the user
bodyColor := BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
interiorColor:= BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
if bodyColor = interiorColor then Print("bad idea");
```

If `GetString` returns a string already in `colorList`, this code makes sure that the original string is reused. This is why using the `=` operator to test for equality works. It also allows the duplicate string to be garbage collected, provided there are no remaining references to it.

BInsertRight

`BInsertRight(`*array*`, `*element*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Inserts an element into the proper position in a sorted array. In the case of equivalent elements, the element is inserted to the right of its equivalent. The index at which it was inserted is returned, or `nil` is returned if it was not inserted.

array	The array to be modified.
element	The new element to be inserted. Note that the *key* parameter is used to extract its key value.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.
uniqueOnly	A Boolean value. Specify a non-`nil` value to indicate that the element is not to be inserted if the array already contains an element with an equivalent key value. Specify `nil` to indicate that the element is to be inserted even if the array already contains an element with an equivalent key. In the later case, the new element is inserted to the right of the existing equivalent elements.

BIntersect

`BIntersect(`*array1*`, `*array2*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Returns a new sorted array consisting of the equivalent elements from the two specified arrays.

array1	The first array. This array is not modified.
array2	The second array. This array is not modified.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.

key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.
uniqueOnly	A Boolean value. Specify a non-`nil` value to indicate that elements with duplicate key values are not allowed in the resulting array. Note that this works only if *array1* and *array2* are both free of equivalent elements. Specify `nil` to indicate that elements with duplicate key values are allowed in the resulting array. Note that this guarantees that the resulting array has at least two equivalent elements for every intersecting value, since intersection finds equivalent elements. If equivalent elements are found in the resulting array, they are ordered as follows: equivalent elements from the same source array retain their original ordering, and equivalent elements from *array1* come before those in *array2*.

BMerge

`BMerge(`*array1*`, `*array2*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Merges two ordered arrays into one new ordered array, which is returned.

array1	The first array. This array is not modified.
array2	The second array. This array is not modified.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

uniqueOnly — A Boolean value. Specify a non-`nil` value to indicate that elements with duplicate key values are not allowed in the resulting array. Note that this works only if *array1* and *array2* are both free of equivalent elements.

Specify `nil` to indicate that elements with duplicate key values are allowed in the resulting array.

If equivalent elements are found in the resulting array, they are ordered as follows: equivalent elements from the same source array retain their original ordering, and equivalent elements from *array1* come before those in *array2*.

BSearchLeft

`BSearchLeft(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses binary search to find an element in a sorted array. The index of the smallest and leftmost element that is greater than or equal to *item* is returned. The value `Length(`*array*`)` is returned if *item* is larger than all elements.

array — The array to be searched.

item — The key value for which to search.

test — Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.

key — Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

Here is an example of how this function might be used:

```
// Extract all elements between "F" and "Na"
array := ["Ag","C","F","Fe","Hg","K","N","Na","Ni","Pu","Zn"];
pos1   := Min(Length(array)-1,BSearchLeft(array,"F",'|str<|,nil));
pos2   := Max(0,BSearchRight(array,"Na",'|str<|,nil));
ArrayMunger([],0,nil,array, pos1, pos2-pos1+1);
```

BSearchRight

`BSearchRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses binary search to find an element in a sorted array. The index of the largest and rightmost element that is less than or equal to *item* is returned. The value –1 is returned if all elements are larger than *item*.

array	The array to be searched.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter on page 6-36.
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter on page 6-37.

For an example of how this function might be used, see `BSearchLeft`.

Integer Math Functions

These math functions operate on or return integers. (Some of the floating point functions can also operate on integers.)

Abs

`Abs(`*x*`)`

Returns the absolute value of an integer or real number.

x	An integer or real number.

Ceiling

`Ceiling(`*x*`)`

Returns the smallest integer not less than the specified real number. (Rounds up the real number to an integer.)

x A real number.

Floor

`Floor(`*x*`)`

Returns the largest integer not greater than the specified real number. (Rounds down the real number to an integer.)

x A real number.

Max

`Max(` *a*`,` *b* `)`

Returns the maximum value of the two integers *a* and *b*.

a An integer.

b An integer.

Min

`Min(` *a*`,` *b* `)`

Returns the minimum value of the two integers *a* and *b*.

a An integer.

b An integer.

Random

`Random (`*low*`, `*high*`)`

Returns a random integer in the range between the two integers `low` and `high`. The range is inclusive of the numbers `low` and `high`.

low An integer.

high An integer.

For example:

```
random (0, 100)
#120      72
```

Real

`Real(`*x*`)`

Converts the specified integer to a real number.

x An integer.

SetRandomSeed

`SetRandomSeed (`*seedNumber*`)`

Seeds the random number generator with the number you specify.

seedNumber An integer.

When seeded with the same number, the random number generator (`Random` function) will return the same sequence of random numbers each time you reseed it. Do not use 0 to seed the generator as it will return 0 instead of a random number.

Note

There is only one random number generator on the Newton, so calls by other functions may interfere with your function getting a consistent sequence of values. ◆

Floating Point Math Functions

NewtonScript provides the floating point math functions documented in this section.

The NewtonScript floating point number system is based on standards 754 and 854 adopted by the Institute of Electrical and Electronics Engineers (IEEE). For more details on IEEE-standard arithmetic than are given here, refer to the *PowerPC Numerics* volume of *Inside Macintosh* or to the *Apple Numerics Manual, Second Edition*. These books describe SANE, the standard Apple numeric environment. The NewtonScript environment supports many features of SANE.

NewtonScript floating point numbers (also called *real* numbers) correspond to the double format of the IEEE standards. The number system supports representations for the following values:

- Normal numbers—numbers with approximately 16 decimal digits of precision, ranging from 1.8×10^{308} down to 2.2×10^{-308}.
- Subnormal numbers—numbers ranging from 2.2×10^{-308} down to 4.9×10^{-324}, whose precision diminishes from approximately 16 decimal digits down to less than one digit.
- Signed zeros—the values +0 and –0, which compare equal, but whose behavior differs when, for example, divided into nonzero values.
- Signed infinities—the values `+INF` and `-INF`, which represent results too large to represent or the result of dividing a nonzero numerator by a zero denominator.
- Not-a-Number symbols, or NaNs—values used to represent missing or uninitialized data, or the results of operations, such as $\sqrt{-3}$, which have no meaning in the real number system.

In some application areas, you may find it useful to think of signed zeros and infinities in terms of mathematical *limits*. For example, although +0 and –0 compare equal, it may be the case for a function *f* that $\lim_{x \to 0-} f(x) \neq \lim_{x \to 0+} f(x)$,

and you may find it useful to exploit that fact. Similarly, you may find it useful to interpret $g(\texttt{+INF})$ as $\lim_{y \to \infty} g(y)$.

The functions in this section follow the model of the arithmetic operations set forth in the IEEE standards, namely, they produce results that are exact when the results are exactly representable in the number system, and otherwise they deliver the nearest (or nearly so) representable number to the mathematically correct result. The IEEE standards specify that one or more exceptions be raised when the result of an operation is different from the mathematical result, or when the result is not defined in the real number system. The possible exceptions are

- Inexact—the result is *rounded* or otherwise altered from the mathematical result.
- Underflow—the nonzero result is too tiny to represent except as zero or a subnormal number, and is rounded to less precision than a normal number.
- Overflow—the result is too huge to represent as a normal number.
- Divide by Zero—the quotient of a nonzero value divided by zero produces `+INF` or `-INF`, according to the arguments' signs.
- Invalid—the result is not mathematically defined, as is the case with $0/0$.

See "Managing the Floating Point Environment" on page 6-65 for further discussion of the handling of floating point exceptions.

One feature of the IEEE standards and SANE is the choice of rounding direction for results not exactly representable. In NewtonScript systems, rounding is *always* to the nearest representable number (with ties going to the value whose least significant bit is zero). The IEEE standards also specify rounding to the nearest value toward 0, toward `+INF`, or toward `-INF`. But the standards are written as though rounding direction is determined by a state variable in the floating point environment (see "Managing the Floating Point Environment"), while on the ARM family of processors used by NewtonScript systems, rounding direction is determined on an instruction-by-instruction basis.

Acos

`Acos(`*x*`)`

Returns the inverse cosine in radians of *x*. `Acos` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all values except 1. `Acos` returns values between zero and π.

x An integer or real number.

Acosh

`Acosh(`*x*`)`

Returns the inverse hyperbolic cosine of *x*. `Acosh` raises invalid for $x < 1$. It raises inexact for all values except 1. `Acosh(+INF)` returns `+INF`, but `Acosh` never overflows. Its value at the largest finite real number is approximately 710.

x An integer or real number.

Asin

`Asin(`*x*`)`

Returns the inverse sine in radians of *x*. `Asin` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all values except zero and raises underflow for all finite *x* near zero. `Asin` returns values between $-\pi/2$ and $\pi/2$.

x An integer or real number.

Asinh

`Asinh(`*x*`)`

Returns the inverse hyperbolic sine of *x*. `Asinh` raises inexact for all values except zero. `Asinh(-INF)` returns `-INF` and `Asinh(+INF)` returns `+INF`. `Asinh` raises underflow for *x* near zero.

x An integer or real number.

Atan

`Atan(`*x*`)`

Returns the inverse tangent in radians of *x*. It raises inexact for all values except zero. `Atan(-INF)` returns $-\pi/2$ and `Atan(+INF)` returns $\pi/2$. `Atan` returns values between $-\pi/2$ and $\pi/2$. It raises inexact for all nonzero *x*.

x An integer or real number.

Atan2

`Atan2(`*x*`,`*y*`)`

Returns the inverse tangent in radians of x/y. `Atan2` uses the algebraic signs of *x* and *y* to determine the quadrant of the result. It returns values between $-\pi$ and π. Its special cases are those of `Atan`.

x An integer or real number.

y An integer or real number.

Atanh

`Atanh(`*x*`)`

Returns the inverse hyperbolic of *x*. `Atanh` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all valid arguments except zero and raises underflow near zero.and raises underflow for all finite *x* near zero. `Atanh(-1.0)` returns `-INF` and `Atan(+1.0)`returns `+INF`.

x An integer or real number.

CopySign

`CopySign(`*x*`,`*y*`)`

Returns the value with the magnitude of *x* and sign of *y*.

x An integer or real number.

y An integer or real number.

Note

The order of the parameters for `CopySign` matches the recommendation of the IEEE 754 floating point standard, which is opposite from the SANE `copysign` function. ◆

Cos

`Cos(`*x*`)`

Returns the cosine of the radian value *x*. `Cos` raises inexact for all finite arguments except zero. It is periodic with period 2π. `Cos` raises invalid when *x* is infinite.

x An integer or real number.

Cosh

`Cosh(`*x*`)`

Returns the hyperbolic cosine of *x*. `Cosh` raises inexact for all finite arguments except zero. `Cosh(-INF)` and `Cosh(+INF)` return `+INF`. `Cosh` raises overflow for finite values of large magnitude.

x An integer or real number.

Erf

`Erf(`*x*`)`

Returns $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$, the *error function* of *x*. `Erf` raises inexact for all arguments except zero. It raises underflow for arguments near zero. `Erf(-INF)` returns –1 and `Erf(+INF)` returns 1.

x An integer or real number.

Mathematically, the sum of `Erf(`*x*`)` and `Erfc(`*x*`)` should be 1, though the relationship may not hold when roundoff or underflow affect the results significantly.

Erfc

`Erfc(`*x*`)`

Returns $\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^{\infty} e^{-t^2}\,dt$, the *complementary error function* of *x*. `Erfc` raises inexact for all arguments except zero. `Erfc(-INF)` returns 2 and `Erfc(+INF)` returns +0.

x An integer or real number.

Exp

`Exp(`*x*`)`

Returns e^x, the exponential of the *x*. `Exp` is inexact for all nonzero finite arguments. `Exp(-INF)` returns +0 and `Exp(+INF)` returns `+INF`. `Exp` raises overflow for large, positive, finite *x*, and raises underflow for negative, finite *x* of large magnitude.

x An integer or real number.

Expm1

`Expm1(`*x*`)`

Returns $e^x - 1$, one less than the exponential of *x*. `Expm1` avoids loss of accuracy when *x* is nearly zero, and the difference is nearly zero. `Expm1` is inexact for all nonzero finite arguments. `Expm1(-INF)` returns –1 and `Expm1(+INF)` returns `+INF`. `Expm1` raises overflow for large, positive, finite *x*, and raises underflow for *x* near zero.

x An integer or real number.

Fabs

`Fabs(`*x*`)`

Returns the absolute value of *x*. It never raises an exception.

x An integer or real number.

FDim

`FDim(`*x*`,`*y*`)`

Returns the *positive difference* between its parameters:

If *x* > *y*, `FDim` returns *x* – *y*

- Otherwise, if *x* <= *y*, `FDim` returns +0
- Otherwise, if *x* is a NaN, `FDim` returns *x*.
- Otherwise (*y* is a NaN), `FDim` returns *y*.

x An integer or real number.

y An integer or real number.

FMax

`FMax(`*x*`,`*y*`)`

Returns the maximum of its two parameters. NaN parameters are treated as missing data:

- If one parameter is a NaN and the other is a number, then the number is returned.
- Otherwise, if both are NaNs, then the first parameter is returned.

(This corresponds to the `max` function in FORTRAN.)

x An integer or real number.

y An integer or real number.

FMin

`FMin(`*x*`,`*y*`)`

Returns the minimum of its two parameters. NaN parameters are treated as missing data:

- If one parameter is a NaN and the other is a number, then the number is returned.
- Otherwise, if both are NaNs, then the first parameter is returned.

(This corresponds to the `min` function in FORTRAN.)

x An integer or real number.

y An integer or real number.

Fmod

`Fmod(`*x*`,`*y*`)`

Returns the remainder when *x* is divided by *y* to produce a truncated integral quotient. That is, `Fmod` returns the value *x* `-` *y*`*Trunc(`*x*`/`*y*`)`.

x An integer or real number.

y An integer or real number.

Gamma

`Gamma(`*x*`)`

Returns Γ(*x*), the gamma function applied to *x*. `Gamma` raises inexact for all non-integral *x*. It raises invalid for non-positive integral arguments *z*. `Gamma(`*p*`)` returns (*p*-1)! for positive, integral *p*, with 0! defined to be 1. `Gamma(+INF)` returns `+INF`. `Gamma` can raise overflow.

x An integer or real number.

Hypot

`Hypot(`*x*`,`*y*`)`

Returns the square root of the sum of the squares of *x* and *y*, avoiding the hazards of overflow and underflow when the arguments are large or tiny in magnitude but the result is within range.

x An integer or real number.

y An integer or real number.

IsFinite

`IsFinite(`*x*`)`

Returns `true` if *x* is finite; returns `nil` if *x* is infinite.

x An integer or real number.

IsNaN

`IsNaN(`*x*`)`

Returns `true` if *x* is a NaN; returns `nil` if *x* is a number.

x An integer or real number.

Note
Saying that *x* "is a NaN" and "is not a number" are not the same thing. A NaN is a non-numerical value in a numerical format; on the other hand, a string such as `"foo"` is not a number because it is not a numerical object. ◆

IsNormal

`IsNormal(`*x*`)`

Returns `true` if *x* is a normal number; returns `nil` if *x* is zero, subnormal, infinite, or a NaN.

x An integer or real number.

LessEqualOrGreater

`LessEqualOrGreater(`*x*`, `*y*`)`

Returns `true` if neither *x* nor *y* is a NaN, and therefore the two arguments are ordered; otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

LessOrGreater

`LessOrGreater(`*x*`, `*y*`)`

Returns `true` if either $x < y$ or $x > y$; otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

LGamma

`LGamma(`*x*`)`

Returns the natural logarithm of $\Gamma(x)$, the gamma function applied to *x*. `LGamma` raises inexact for all positive *x*. It raises invalid for negative or zero *x*. `LGamma(+INF)` returns `+INF`.

x An integer or real number.

Log

`Log(`*x*`)`

Returns the natural logarithm of *x*. `Log` raises inexact for positive, finite arguments except 1. `Log(0.0)` returns `-INF` and raises divide by zero. `Log(+INF)` returns `+INF`. `Log` raises invalid for $x < 0$.

x An integer or real number.

Logb

`Logb(`*x*`)`

Returns the integral value *k* such that $1 \leq |x| * 2^{-k} < 2$, when *x* is finite and nonzero. `Logb(0.0)` returns `-INF` and raises divide by zero. `Logb(-INF)` and `Logb(+INF)` return `+INF`.

Log1p

`Log1p(`*x*`)`

Returns the natural logarithm of `1+`*x*. While accurate for all arguments no less than –1, `Log1p` preserves accuracy when *x* is nearly zero—when computing `Log(1.0 +` *x*`)`would suffer from the mere addition of *x* to 1. `Log1p` raises inexact for all finite arguments greater than –1 except 0. It raises invalid for all *x* less than –1 and raises underflow for *x* near zero. `Log1p(-1.0)` returns `-INF` and raises divide by zero. `Log1p(+INF)` returns `+INF`.

x An integer or real number.

Log10

`Log10(`*x*`)`

Returns the logarithm base 10 of *x*. Because of the mathematical relationship $\log_{10}(x) = \log(x)/\log(10)$, `Log10` shares the computational properties of `Log`.

x An integer or real number.

NearbyInt

`NearbyInt(`*x*`)`

Returns *x* rounded to the nearest integral value. `NearbyInt` differs from `Rint` only in that it does not raise the inexact exception.

x An integer or real number.

Note

`NearbyInt` always rounds to nearest. ◆

NextAfterD

`NextAfterD(`*x*`,`*y*`)`

Returns the next representable number after *x* in the direction of *y*.

If *x* and *y* are equal, then the result is *x*. If either argument is a NaN, `NextAfterD` returns one of the NaN arguments. When *x* is finite but the result is infinite, `NextAfterD` raises overflow. When the result is zero or subnormal, `NextAfterD` raises underflow.

x An integer or real number.

y An integer or real number.

Pow

`Pow(`*x*`,`*y*`)`

Returns x^y. When $x < 0$, `Pow` raises invalid unless *y* is an integral value. It can raise inexact, overflow, underflow, and invalid.

x An integer or real number.

y An integer or real number.

RandomX

`RandomX(`*x*`)`

Returns a two-element array, based on the random seed *x*. The first element of the result is a pseudo-random number that is the result of the SANE `randomx` function. The second element is the new seed returned by the `randomx` function. The result is an integral value between 0 and $2^{31} - 1$.

x An integer or real number.

Remainder

`Remainder(`*x*`,`*y*`)`

Returns the *exact* difference $x - n*y$, where *n* is a mathematical integer (as opposed to a NewtonScript integer—*n* may be thousands of bits wide) to x/y in the sense of rounding to nearest. The magnitude of the result is no greater than half the magnitude of *y*. When the result is zero, it has the sign of *x*. `Remainder` raises invalid when *y* is zero or *x* is infinite. It never raises overflow, underflow, or inexact.

x An integer or real number.

y An integer or real number.

RemQuo

`RemQuo(`*x*`,`*y*`)`

Returns a two-element array. The first element is `Remainder(`*x*`, `*y*`)`. The second element is the seven low-order bits of the quotient *x* / *y* rounded to the nearest integer and given the sign of the quotient.

x	An integer or real number.
y	An integer or real number.

Rint

`Rint(`*x*`)`

Is identical to `Nearbyint` except that it raises inexact when its result differs from *x*.

x	An integer or real number.

RintToL

`RintToL(`*x*`)`

Returns an integer obtained by rounding *x* to an integral (real) value and then converting that value to an integer. `RintToL` raises inexact when its result differs in value from *x*. It raises invalid and returns an unspecified value when the rounded value of *x* cannot be represented exactly as an integer object.

x	An integer or real number.

Note

`RintToL` always rounds to nearest. ◆

Round

`Round(`*x*`)`

Returns the integral real number obtained from *x* by adding 1/2 to *x* and truncating the result to the nearest integer toward 0. It raises inexact when the result differs from *x*.

x An integer or real number.

Scalb

`Scalb(`*x*`, `*k*`)`

Returns $x * 2^k$. Scalb avoids explicit computation of 2^k and so avoids the complications of overflow or underflow when 2^k is out of range but the result isn't. `Scalb` can raise overflow, underflow, and inexact. `Scalb` and `Logb` are related by the formula $1 \le$ `Scalb(`*x*`, RintToL(-Logb(`*x*`)))` < 2 for finite, nonzero *x*.

x An integer or real number.

y An integer.

SignBit

`SignBit(`*x*`)`

Returns a nonzero integer if the sign of *x* is negative; otherwise (the sign of *x* is positive), returns the integer 0.

x An integer or real number.

Signum

`Signum(`*x*`)`

Returns the integer value –1 if $x < 0$, 0 if $x = 0$, or 1 if $x > 0$. If *x* is an integer, `Signum` returns an integer; otherwise, if *x* is a real, `Signum` returns a real. If *x* is neither an integer nor a real, `Signum` throws the exception `kFramesErrNotANumber`.

x An integer or real number.

Sin

`Sin(`*x*`)`

Returns the sine of the radian value *x*. `Sin` raises inexact for all finite values except zero. It is periodic with period 2π. `Sin` raises invalid for infinite *x* and raises underflow for *x* near zero.

x An integer or real number.

Sinh

`Sinh(`*x*`)`

Returns the hyperbolic sine of *x*. `Sinh` raises inexact for all finite arguments except zero. `Sinh(-INF)` returns `-INF` and `Sinh(+INF)` returns `+INF`. `Sinh` raises overflow for large finite values and raises underflow near zero.

x An integer or real number.

Sqrt

`Sqrt(`*x*`)`

Returns the square root of *x*. It raises invalid for $x < 0$, and can raise inexact for positive *x*.

x An integer or real number.

Tan

`Tan(`*x*`)`

Returns the tangent of the radian value *x*. `Tan` raises inexact for all finite values except zero. It is periodic with period π. `Tan` raises invalid for infinite *x* and raises underflow for *x* near zero.

x An integer or real number.

Tanh

`Tanh(`*x*`)`

Returns the hyperbolic tangent of *x*. `Tanh` raises inexact for all finite arguments except zero. `Tanh(-INF)` returns –1 and `Tanh(+INF)` returns +1. `Tanh` raises overflow for large finite values and raises underflow near zero.

x An integer or real number.

Trunc

`Trunc(`*x*`)`

Returns the integral real number nearest to but no larger in magnitude than *x*.

x An integer or real number.

Unordered

`Unordered(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy none of $x < y$, $x = y$, or $x > y$ (because one or both of *x* and *y* are NaNs); if neither *x* nor *y* is a NaN, they satisfy one of the three order relations and `Unordered` returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedGreaterOrEqual

`UnorderedGreaterOrEqual(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x \geq y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedLessOrEqual

`UnorderedLessOrEqual(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x \leq y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedOrEqual

`UnorderedOrEqual(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x = y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedOrGreater

`UnorderedOrGreater(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x > y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

UnorderedOrLess

`UnorderedOrLess(`*x*`, `*y*`)`

Returns `true` if *x* and *y* satisfy $x < y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

Managing the Floating Point Environment

The floating point environment is a set of state variables maintained by the Newton system and the underlying processor. The environment contains information about which floating point exceptions have occurred. Floating point exceptions are distinct from NewtonScript exceptions. When floating point exceptions arise (for example, overflow arises when the sum of two huge numbers is too large to represent in the number system), the system raises an exception flag in the environment. Exception flags can be tested, cleared, or raised by functions in this section. Once raised, an exception flag remains raised until you clear it using calls from this section. The predefined constants used to select the floating point exception flags are shown in Table 6-1.

Table 6-1 Floating point exceptions

Constant	Value	Meaning
`fe_Inexact`	`0x010`	inexact
`fe_DivByZero`	`0x002`	divide-by-zero
`fe_Underflow`	`0x008`	underflow
`fe_Overflow`	`0x004`	overflow
`fe_Invalid`	`0x001`	invalid
`fe_All_Except`	`0x01F`	all exceptions

You can refer to multiple exceptions in a single function invocation by forming the bitwise-OR of the predefined constants, using expressions like `Bor(Bor(fe_Invalid, fe_DivByZero), fe_Overflow)`.

Note
The representation of the floating point environment is implementation-dependent. Functions that manipulate the environment and its components do so without exposing their implementation. In particular, the floating point exception flags may or may not be implemented as single bits. ◆

The functions that manage the floating point environment are based on recommended numerical extensions to the ANSI C language. The recommendations for C include functions to test and alter the direction of rounding. Although the direction of rounding is determined by the environment on most systems, Newton systems based on the ARM family of processors determine the rounding direction on an instruction-by-instruction basis, so rounding is not determined by the environment.

You can pass the predefined constant `fe_Dfl_Env` to the functions `FeSetEnv` and `FeUpdateEnv`, which take an environment object as a parameter. `Fe_Dfl_Env` indicates the default environment, in which all exception flags are clear.

FeClearExcept

`FeClearExcept(`*excepts*`)`

Clears the floating point exception flags indicated by *excepts*.

excepts	The integer bitwise-OR of one or more floating point exceptions.

FeGetEnv

`FeGetEnv()`

Returns a data object representing the current floating point environment.

FeGetExcept

`FeGetExcept(`*excepts*`)`

Returns a data object representing the current state of the exception flags indicated by *excepts*.

excepts	The integer bitwise-OR of one or more floating point exceptions.

Note

The representation of the exception flags is unspecified. ◆

FeHoldExcept

`FeHoldExcept()`

Returns a data object representing the current floating point environment, and clears the exception flags.

FeRaiseExcept

`FeRaiseExcept(`*excepts*`)`

Raises the floating point exception flags indicated by *excepts*.

excepts	The integer bitwise-OR of one or more floating point exceptions.

Note

Because floating point exceptions are not tied to the general NewtonScript exception-handling mechanism, raising a flag merely sets an internal variable; raising a flag will not alter the flow of control. ◆

FeSetEnv

`FeSetEnv(`*envObj*`)`

Installs the floating point environment represented by the object *envObj*.

envObj — Either the predefined constant `fe_Dfl_Env` or an object returned by a call to `FeGetEnv` or `FeHoldExcept`.

FeSetExcept

`FeSetExcept(`*flagObj*`,` *excepts*`)`

The parameter *flagObj* is an object containing an implementation-dependent representation of one or more floating point exception flags; *flagObj* must have been set by a previous call to `FeGetExcept`. `FeSetExcept` alters the current environment so that those floating point exception flags indicated by *excepts* match the corresponding values in *flagObj*.

flagObj — An object (returned by a previous call to `FeGetExcept`) containing a representation of one or more floating point exception flags.

excepts — The integer bitwise-OR of one or more floating point exceptions.

This function does not raise exceptions; it just alters the state of the flags.

FeTestExcept

`FeTestExcept(`*excepts*`)`

Returns the bitwise-OR of the floating point exceptions indicated by *excepts* whose flags are rasied in the current environment.

excepts — The integer bitwise-OR of one or more floating point exceptions.

FeUpdateEnv

`FeUpdateEnv(`*envObj*`)`

Saves the state of the current exception flags, installs the environment represented by *envObj*, and then re-raises the saved exceptions.

envObj	Either the predefined constant `fe_Dfl_Env` or an object returned by a call to `FeGetEnv` or `FeHoldExcept`.

You can use `FeUpdateEnv` in conjunction with `FeHoldExcept` to write functions which hide spurious exceptions from their callers:

```
func() begin
   savedEnv := FeHoldExcept(); // clears flags
   result := ...; // ecomputation in which underflow and
                  // divide by zero are benign
   FeClearExcept(BOR(fe_Underflow, fe_DivByZero));
   FeUpdateEnv(savedEnv); // merge old flags with new
   return result
end
```

Financial Function

These functions perform financial calculations.

Annuity

`Annuity(`*r*`, `*n*`)`

Returns the value of the financial formula $\frac{1-(1+r)^{-n}}{r}$. When *r* is the periodic interest rate and *n* the number of periods, *p*`*Annuity(`*r*`, `*n*`)` is the *present value* of a series of *n* periodic payments of size *p*. `Annuity` is robust over the entire range of *r* and *n*, whether financially meaningful or not.

`Annuity` raises invalid for $r < -1$. When $r = -1$:

- `Annuity(-1,` n`)` returns –1 for $n < 0$.
- `Annuity(-1, 0)` returns 0.
- `Annuity(-1,` n`)` returns `+INF` and raises divide by zero for $n > 0$.

Otherwise, $r > -1$. When r is nonzero, `Annuity(`r`, 0)` returns r; otherwise, `Annuity(0,` n`)` returns n. `Annuity` raises inexact in all other cases, and can raise overflow or underflow.

r	An integer or real number.
n	An integer or real number.

Compound

`Compound(`r`,` n`)`

Returns the value of the financial formula $(1 + r)^n$. When r is the periodic interest rate and n the number of periods, P`*Compound(`r`,` n`)` is the *future value* of a principal amount P. `Compound` is robust over the entire range of r and n, whether financially meaningful or not.

`Compound` raises invalid for $r < -1$. When $r = -1$:

- `Compound(-1,` n`)` returns `+INF` and raises divide by zero for $n < 0$.
- `Compound(-1, 0)` returns 1.
- `Compound(-1,` n`)` returns +0 for $n > 0$.

Otherwise, $r > 0$. `Compound(`r`, 0)` returns 1; `Compound(0,` n`)` raises invalid when n is infinite. `Compound` can raise inexact, overflow or underflow.

r	An integer or real number.
n	An integer or real number.

Exception Functions

These functions are used to raise and handle NewtonScript exceptions in an application. For more information about exception handling and how to use these functions, refer to the second half of Chapter 3, "Flow of Control," "Exception Handling" on page 3-13. For a list of system exceptions, see the appendix "Errors" in the *Newton Programmer's Guide*.

The section "Managing the Floating Point Environment" beginning on page 6-65 describes some functions that deal with floating-point exceptions, which are not related to NewtonScript exceptions.

Throw

`Throw(`*name*`,` *data*`)`

Raises an exception and creates an exception frame with the specified name and data.

name	An exception symbol that names the exception being raised.
data	The data for the exception. The possible values for this parameter depend on the composition of *name* and are shown inTable 6-2.

See "Exception Handling" beginning on page 3-13 for more information on `Throw`.

Table 6-2 Exception frame data slot name and contents

Exception symbol	Slot name	Slot contents
contains part with prefix `type.ref`	`data`	a data object, which can be any NewtonScript object
contains part with prefix `evt.ex.msg`	`message`	a message string
any other	`error`	an integer error code

Rethrow

```
Rethrow()
```

Reraises the current exception to allow the next enclosing `Try` statement an opportunity to handle it. `Rethrow` throws the current exception again, passing along the same parameters are were passed with the original call to the `Throw` function. This allows you to pass control from within an exception handler to the next enclosing `Try` statement.

IMPORTANT

You can call the `Rethrow` function only from within the dynamic extent of an `onexception` clause. ▲

CurrentException

```
CurrentException()
```

During exception processing (that is, inside the dynamic extent of an `onexception` block), returns the frame that is associated with the current exception. You can examine the frame returned by `CurrentException` to determine what kind of exception you are handling. For example, you can call the `HasSlot` function to determine if the frame contains a slot named `error`, and take appropriate action thereafter. (The format of the frame depends on the exception, but it always contains a *name* slot with the exception symbol.)

`CurrentException` gives a meaningful response only from within the dynamic extent of an `onexception` clause. Outside the extent of `onexception`, it returns `nil`.

Message Sending Functions

These functions send messages or execute functions.

Apply

`Apply(`*function*`,` *parameterArray*`)`

Calls a function, passing the supplied parameters. The `Apply` function returns the return value of the function it called.

function	The function to call.
parameterArray	An array of parameters to be passed to the function. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

`Apply` respects the environment of the function object it is passed. Using `Apply` is similar to using the NewtonScript `call` statement.

`Apply` is useful when you want to call a function, but don't know until run time the number of parameters it takes. If you do know ahead of time the number of parameters the function takes, then you can use the NewtonScript `call` statement to call the function.

Here's an example of using this function in the Inspector:

```
f:=func(x,y) x*y;
Apply(f,[10,2]);
#50      20
```

The `Apply` call is equivalent to:

```
f(10,2);
```

Perform

`Perform(`*frame*`, `*message*`, `*parameterArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Both parent and proto inheritance are used to search for the method if it does not exist in the frame. If the method is not found, an exception is thrown.

frame	The frame to which to send the message.
message	A symbol naming the message to send.
parameterArray	An array of parameters to be passed along with the message. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

The `Perform` function returns the return value of the message it sent.

Note that the method named by *message* is executed in the context of *frame*, not in the context of the frame from within which `Perform` is called.

The `Perform` function is useful when you want to send a message, but you don't know until run time the name of the message or the number of parameters it takes. If you do know these things ahead of time, then you can just use the standard NewtonScript message sending syntax.

For variations of the `Perform` function, see `PerformIfDefined`, `ProtoPerform`, and `ProtoPerformIfDefined`.

Here's an example of using this function in the Inspector:

```
f:={multiply: func(x,y) x*y};
perform(f, 'multiply, [10,2]);
#50      20
```

Note that

```
f:multiply(10,2)
```

is equivalent to

```
Perform(f, 'multiply,[10,2])
```

PerformIfDefined

`PerformIfDefined(`*receiver*`,`*message*`,`*paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Both parent and proto inheritance are used to search for the method if it does not exist in the frame. If the method is not found, an exception is not thrown.

receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function returns the return value of the message it sent. If the method is not found, this function returns `nil`.

Contrast this function with `Perform` (page 6-74), which is exactly the same, except that `Perform` throws an exception if the method is not found.

Also, contrast this function with `ProtoPerform` and `ProtoPerformIfDefined` (page 6-75), which search only the proto chain for the method.

ProtoPerform

`ProtoPerform(`*receiver*`,`*message*`,`*paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Only proto inheritance is used to search for the method if it does not exist in the frame. If the method is not found, an exception is thrown.

receiver	The frame to which you want the message sent.
message	A symbol that is the name of the message to send to *receiver*.

paramArray An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function returns the return value of the message it sent.

Contrast this function with `Perform` (page 6-74), which is exactly the same, except that `Perform` searches both the parent and proto chains for the method.

Also, contrast this function with `PerformIfDefined` (page 6-75) and `ProtoPerformIfDefined` , which do not throw exceptions if the method is not found.

ProtoPerformIfDefined

`ProtoPerformIfDefined(`*receiver*`,`*message*`,`*paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Only proto inheritance is used to search for the method if it does not exist in the frame. If the method is not found, an exception is not thrown.

receiver The frame to which you want the message sent.

message A symbol that is the name of the message to send to *receiver*.

paramArray An array of parameters to be passed with the *message*. You can specify `nil` if there are no parameters to be passed (this saves allocating an empty array).

This function returns the return value of the message it sent. If the method is not found, this function returns `nil`.

Contrast this function with `PerformIfDefined` (page 6-75), which is exactly the same, except that `PerformIfDefined` searches both the parent and proto chains for the method.

Also, contrast this function with `Perform` (page 6-74) and `ProtoPerform` (page 6-75), which search both the parent and proto chains for the method.

Data Extraction Functions

These functions are used to extract chunks of data out of other objects of various types.

All integers are stuffed and extracted in two's-complement **big-endian** form. In this form, byte 0 is the most significant byte, as found on the Newton and Macintosh. The opposite of this is **little-endian,** where byte 0 is least significant byte, as found on Intel-based computers. For example, the number 0x12345678 is stored as:

big-endian 12 34 56 78

little-endian 78 56 34 12

All Unicode conversions use the Macintosh extended character set for codes greater than or equal to 128.

ExtractByte

`ExtractByte(`*data*`, `*offset*`)`

Returns one signed byte from the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractByte("\u12345678",0);
#3FC      255
```

ExtractBytes

`ExtractBytes(`*data*`, `*offset*`, `*length*`, `*class*`)`

Returns a binary object of class *class* containing *length* bytes of data starting at *offset* within *data*.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.
length	An integer giving the number of bytes to extract.
class	A symbol specifying the class of the return value.

ExtractChar

`ExtractChar(`*data*`, `*offset*`)`

Returns a character object of the character at the given *offset* in the *data*.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

Gets one byte at the specified offset, converts it to Unicode and returns the character it makes from it.

For example:

```
ExtractChar("\uFFFFFFFF",0);
//$\u02C results from a ASCII to UNICODE conversion.
#2C76     $\u02C7
//Note $a is at offset 1 in a Unicode string
ExtractChar("abc",0);
#6        $\00
ExtractChar("abc",1);
#616      $a
```

ExtractLong

`ExtractLong(`*data*`, `*offset*`)`

Returns an integer object of the low 29 bits of an unsigned long at the given offset, right-justified (that is, the low 29 bits of a four-byte value).

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

Reads four bytes at the specified offset, but ignores the high-order bits (first two). Returns a 30 bit signed value

```
ExtractLong("\uFFFFFFFF",0);
#FFFFFFFC -1
ExtractLong("\uC0000007",0);
#1C       7
```

ExtractXLong

`ExtractXLong(`*data*`, `*offset*`)`

Returns an integer object of the high 29 bits of an unsigned long at the given offset, right-justified (that is, the high 29 bits of a four-byte value).

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractXLong("\u0000000F",0);
#4        1
```

ExtractWord

`ExtractWord(`*data*`, `*offset*`)`

Returns an two-byte signed integer object from the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractWord("\uFFFFFFFF",0);
#FFFFFFFC -1
//if you want unsigned use:
band(ExtractWord(-),0xFFFF);
#40004    65535
```

ExtractCString

`ExtractCString(`*data*`, `*offset*`)`

Returns a Unicode string object derived from the null-terminated C-style string at the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

ExtractPString

`ExtractPString(`*data*`, `*offset*`)`

Returns a Unicode string object derived from the Pascal-style string (a length byte followed by text) at the given offset.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

ExtractUniChar

`ExtractUniChar(`*data*`, `*offset*`)`

Gets two bytes at the specified offset and returns the Unicode character represented by those bytes.

data	The data from which the return value is to be extracted.
offset	An integer giving the position in data from which the return value is to be extracted.

For example:

```
ExtractUniChar("abc",0);
#616      $a
```

Data Stuffing Functions

These functions are used to stuff chunks of data into objects of various types.

All integers are stuffed in two's-complement big-endian form. For a discussion of this, see "Data Extraction Functions" on page 6-77.

▲ **WARNING**

It is important that the destination for the data stuffing functions is large enough to hold the data being stuffed. If the destination is not large enough, the NewtonScript heap may become corrupted. Be sure to take into account the offset. Here is a formula you can use:

Length(*destObj*) – *offset* >= *size of stuffed data*

In this formula, *destObj* is the destination object and *offset* is the position within the destination object where the data is to be stuffed. ▲

StuffByte

`StuffByte(`*obj*`, `*offset*`, `*toInsert*`)`

Writes the low order byte of *toInsert,* at the specified *offset* in *obj*.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	The data to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffByte(x,0,-1);
x[0]
#FF006    $\uFF00

x := "\u00000000";
StuffByte(x,0,0xFF);
x[0]
#FF006    $\uFF00
```

StuffChar

`StuffChar(`*obj*`, `*offset*`, `*toInsert*`)`

Stuffs one byte into *obj* at the specified offset.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	A character or integer to be stuffed in *obj*. You pass it a two byte Unicode value as *toInsert*. The function makes a one-byte character from that value and stuffs the one-byte character.

This accepts a character or integer as its third parameter, *toInsert*:

- If *toInsert*: is an integer: writes the low byte of *toInsert*.
- *If toInsert*: is a character: converts from Unicode and writes a byte.

For example:

```
x := "\u00000000";
StuffChar(x,1,Ord($Z));
x[0]
#5A6      $Z

x := "\u00000000";
StuffChar(x,1,-1);
x[0]
#1A6      $\1A

ExtractByte(x,1)
#68       26
ExtractByte(x,0)
#0        0
```

StuffCString

`StuffCString(`*obj*`,` *offset*`,` *aString*`)`

Converts a Newton Unicode string into a null-terminated C-style string and stuffs it at the given offset into a binary object.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
aString	A Unicode string to be stuffed into *obj*.

The string *aString* is converted into ASCII format using Macintosh roman string encoding. It is then stuffed into *obj,* beginning at the byte offset *offset*. It is followed by a null byte terminator.

This function throws an exception if *aString* will not fit into *obj* beginning at the given offest, or if the offset is negative. The length of *obj* will not be altered.

StuffLong

`StuffLong(`*obj*`, `*offset*`, `*toInsert*`)`

Writes four bytes at the specified offset using the 30 bit signed value you pass it as the third parameter, and sign extends it to 32 bytes.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	The data to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffLong(x,0,-1);
x[0]
#FFFF6    $\uFFFF
x[1]
#FFFF6    $\uFFFF
x := "\u00000000";
StuffLong(x,0,0x3FFFFFFA);
x[0]
#FFFF6    $\uFFFF
x[1]
#FFFA6    $\uFFFA
```

StuffPString

`StuffPString(`*obj*`, `*offset*`, `*aString*`)`

Converts a Newton Unicode string into a Pascal-style string (a length byte followed by text) and stuffs it at the given offset into a binary object.

object	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
aString	A Unicode string to be stuffed into *obj*. This string must be no longer than 255 characters.

The string *aString* is converted into ASCII format using Macintosh roman string encoding. Then a length byte followed by the string is stuffed into *obj,* beginning at the byte offset *offset*. The length byte indicates the number of characters in the string.

This function throws an exception if *aString* will not fit into *obj* beginning at the given offest, or if the offset is negative. The length of *obj* will not be altered.

StuffUniChar

`StuffUniChar(`*obj*`, `*offset*`, `*toInsert*`)`

Stuffs the two-byte Unicode encoding for the character indicated by *toInsert* into *obj* at the specified offset.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	A character or integer to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffUniChar(x,0,"\uF00F"[0]);
x[0]
#F00F6    $\uF00F

x := "\u00000000";
StuffUniChar(x,0,0x0AA0);
x[0]
#AA06     $\u0AA0
```

StuffWord

`StuffWord(`*obj*`, `*offset*`, `*toInsert*`)`

Writes the low order two bytes of *toInsert* at the specified offset.

obj	A binary object into which data is to be stuffed.
offset	The position in *obj* at which stuffing is to begin.
toInsert	The data to be stuffed in *obj*.

For example:

```
x := "\u00000000";
StuffWord(x,0,0x3FFF1234);
x[0]
#12346    $\u1234

x := "\u00000000";
StuffWord(x,0,-1);
x[0]
#FFFF6    $\uFFFF
```

Getting and Setting Global Variables and Functions

These functions get, set and test for the existence of global variables and functions.

GetGlobalFn

`GetGlobalFn(`*symbol*`)`

Returns a global function. If the function is not found, `nil` is returned.

symbol	A symbol naming the global function you want to get.

GetGlobalVar

`GetGlobalVar(`*symbol*`)`

Returns the value of a slot in the system globals frame. If the slot is not found, `nil` is returned.

symbol A symbol naming the global variable whose value you want to get.

GlobalFnExists

`GlobalFnExists(`*symbol*`)`

Returns non-`nil` if the global function identified by *symbol* exists, otherwise returns `nil`.

symbol A symbol naming the global function whose existence you want to check.

GlobalVarExists

`GlobalVarExists(`*symbol*`)`

Returns non-`nil` if the global variable identified by *symbol* exists, otherwise returns `nil`.

symbol A symbol naming the global variable whose existence you want to check.

DefGlobalFn

`DefGlobalFn(`*symbol*`, `*function*`)`

Defines a global function. The symbol identifying the function is returned.

symbol A symbol naming the global function you want to define. To avoid naming conflicts with other global functions, you should choose a name that includes your `appSymbol`, which includes the developer signature you have registered with Newton DTS.

function A function object.

Note that the global function is destroyed if the system is reset.

It is very important to remove any global functions created by your application when your application is removed. You can do this with `UnDefGlobalFn` in the application `RemoveScript` function.

IMPORTANT

Do not create global functions unless it is absolutely necessary. Global functions occupy NewtonScript heap space. They can conflict with system global functions and other applications' global functions. In most cases, you can use methods in your application base view instead of global functions. ▲

DefGlobalVar

`DefGlobalVar(`*symbol*`,` *value*`)`

Defines a global variable—that is, a slot in the system globals frame. The value of the variable is returned.

symbol	A symbol naming the global variable you want to define. To avoid naming conflicts with other globals, you should choose a name that includes your `appSymbol`, which includes the developer signature you have registered with Newton DTS.
value	The value you want to assign to the global variable.

The system ensures that the object created exists entirely in internal RAM (it calls `EnsureInternal` on the object identified by *symbol*. Note that the global variable is destroyed if the system is reset.

It is very important to remove any globals created by your application when your application is removed. You can do this with `UnDefGlobalVar` in the application `RemoveScript` function.

IMPORTANT
Do not create global variables unless it is absolutely necessary. Global variables occupy NewtonScript heap space. They can conflict with system globals and other applications' globals. In most cases, you can put any global data that you need in your application base view or in a soup. ▲

UnDefGlobalFn

`UnDefGlobalFn(`*symbol*`)`

Removes a global function you previously defined. This function returns `nil`.

symbol A symbol naming the global function you want to remove.

UnDefGlobalVar

`UnDefGlobalVar(`*symbol*`)`

Removes a global variable you previously defined. This function returns `nil`.

symbol A symbol naming the global variable you want to remove.

Miscellaneous Functions

These are other miscellaneous functions.

BinEqual

`BinEqual(`*a*`, `*b*`)`

a A binary object

b A binary object

Compares two binary objects' data as raw bytes. Returns non-`nil` if they are identical.

BinaryMunger

`BinaryMunger(` *dst*`,` *dstStart*`,` *dstCount*`,` *src*`,` *srcStart*`,` *srcCount* `)`

Replaces bytes in *dst* using bytes from *src* and returns *dst* after munging is complete. This function is destructive to *dst*.

dst	A value to be changed.
dstStart	The starting position in *dst*.
dstCount	The number of bytes to be replaced *in dst*. You can specify `nil` for *dstCount* to go to the end of *dst*.
src	A value. Can be `nil` to simply delete the contents of *dst*.
srcStart	The starting position in the source binary from which to begin taking elements to place into the destination binary.
srcCount	The number of bytes to use from the source binary. You can specify `nil` to go to the end of the source binary.

Bytes are numbered counting from zero.

Chr

`Chr(`*integer*`)`

Converts a decimal integer to its Unicode character equivalent.

integer	An integer.

Here is an example:

```
chr(65)
$A
```

Compile

`Compile(`*string*`)`

Compiles an expression sequence and returns a function that evaluates it.

string	The expression to compile.

Here are two examples. Note that, in the first example, x is a local variable.

```
compile("x:= {a:self.b, b:1234}")
#440F711  <CodeBlock, 0 args #440F711>
f:=compile("2+2")
f();
#440F712 4
```

Note

All characters used in NewtonScript code must be 7-bit ASCII. This usually is no problem, but can create problems with `Compile` in certain situations. Suppose you tried this call:

```
Compile ("blah, blah, blah,  \u0F0F\u")
```

The Unicode character is not a 7-bit character, it is 16 bits. Therefore, you get an error. (The `\u` switch turns on Unicode character mode.) You should do this instead:

```
Compile ("blah, blah, blah,  \\u0F0F\\u")
```

The backslash escape character preceding the `\u` prevents Unicode mode from being turned on for the compile. (The `\u` is read simply as the string `"\u"` instead of the Unicode switch.)

Note, also, that:

```
compile("func()...")
```

returns a function that constructs the function. The environment is captured when the function constructor is executed:

```
f := compile("func()b");
x := {a:f, b:0};
g:=x:a();
#440F713 <CodeBlock, 0 args #440F711>
```

Executing the function construction captures the message environment with x as receiver.

```
g();
#440F714 0
```

So now it can find b. ◆

Ord

`Ord (`*char*`)`

Converts a character to its Unicode decimal integer equivalent.

char A character.

Here is an example:

```
ord($A)
65
```

Summary of Functions and Methods

This section contains a summary of the functions and methods described in this chapter.

Object System Functions

`ClassOf(`*object*`)`
`Clone(`*object*`)`
`DeepClone(`*object*`)`
`GetFunctionArgCount(`*function*`)`
`GetSlot(`*frame*`, `*slotSymbol*`)`
`GetVariable(`*frame*`, `*slotSymbol*`)`
`HasSlot(`*frame*`, `*slotSymbol*`)`
`HasVariable(`*frame*`, `*slotSymbol*`)`
`Intern( `*string*` )`

`IsArray(`*obj*`)`
`IsBinary(`*obj*`)`
`IsCharacter(`*obj*`)`
`IsFrame(`*obj*`)`
`IsFunction(`*obj*`)`
`IsImmediate(`*obj*`)`
`IsInstance(`*obj*`, `*class*`)`
`IsInteger(`*obj*`)`
`IsNumber(`*obj*`)`
`IsReadOnly(`*obj*`)`
`IsReal(`*obj*`)`
`IsString(`*obj*`)`
`IsSubclass(`*class1*`, `*class2*`)`
`IsSymbol(`*obj*`)`
`MakeBinary(`*length*`, `*class*`)`
`Map(`*obj*`, `*function*`)`
`PrimClassOf(`*object*`)`
`RemoveSlot(`*object*`, `*slot*`)`
`ReplaceObject(`*originalObject*`, `*targetObject*`)`
`SetClass(`*object*`, `*classSymbol*`)`
`SetVariable(`*frame*`, `*slotSymbol*`, `*value*`)`
`SymbolCompareLex(`*symbol1*`, `*symbol2*`)`
`TotalClone(`*object*`)`

String Functions

`BeginsWith(`*string*`, `*substr*` )`
`Capitalize(`*string*`)`
`CapitalizeWords(`*string*`)`
`CharPos(`*str*`, `*char*`, `*startpos*`)`
`Downcase(`*string*`)`
`EndsWith(`*string*`, `*substr*`)`
`IsAlphaNumeric(`*char*`)`
`IsWhiteSpace(`*char*`)`
`SPrintObject( `*obj*` )`
`StrCompare(`*a*`, `*b*`)`

StrConcat(*a*, *b*)
StrEqual(*a*, *b*)
StrExactCompare(*a*, *b*)
StrLen(*string*)
StrMunger(*dstString, dstStart, dstCount, srcString, srcStart, srcCount*)
StrPos(*string*, *substr*, *start*)
StrReplace(*string*, *substr*, *replacement*, *count*)
StrTokenize(*str*, *delimiters*)
StyledStrTruncate(*string*, *length*, *font*)
SubStr(*string*, *substr*, *start*)
TrimString(*string*)
Upcase(*string*)

Bitwise Functions

Band(*a*, *b*)
Bor(*a*, *b*)
Bxor(*a*, *b*)
Bnot(*a*)

Array Functions

AddArraySlot(*array*, *value*)
Array(*size*, *initialValue*)
ArrayInsert(*array*, *element*, *position*)
ArrayMunger(*dstArray*, *dstStart*, *dstCount*, *srcArray*, *srcStart*, *srcCount*)
ArrayRemoveCount(*array*, *startIndex*, *count*)
InsertionSort(*array*, *test*, *key*)
Length(*array*)
LFetch(*array*, *item*, *start*, *test*, *key*)
LSearch(*array*, *item*, *start*, *test*, *key*)
NewWeakArray(*length*)
SetAdd(*array*, *value*, *uniqueOnly*)
SetContains(*array*, *item*)
SetDifference(*array1*, *array2*)

SetLength(*array*, *length*)
SetOverlaps(*array1*, *array2*)
SetRemove(*array*, *value*)
SetUnion(*array1*, *array2*, *uniqueFlag*)
Sort(*array*, *test*, *key*)
StableSort(*array*, *test*, *key*)

Sorted Array Functions

BDelete(*array*, *item*, *test*, *key*, *count*)
BDifference(*array1*, *array2*, *test*, *key*)
BFetch(*array*, *item*, *test*, *key*)
BFetchRight(*array*, *item*, *test*, *key*)
BFind(*array*, *item*, *test*, *key*)
BFindRight(*array*, *item*, *test*, *key*)
BInsert(*array*, *element*, *test*, *key*, *uniqueOnly*)
BInsertRight(*array*, *element*, *test*, *key*, *uniqueOnly*)
BIntersect(*array1*, *array2*, *test*, *key*, *uniqueOnly*)
BMerge(*array1*, *array2*, *test*, *key*, *uniqueOnly*)
BSearchLeft(*array*, *item*, *test*, *key*)
BSearchRight(*array*, *item*, *test*, *key*)

Integer Math Functions

Abs(*x*)
Ceiling(*x*)
Floor(*x*)
Max(*a*, *b*)
Min(*a*, *b*)
Real(*x*)
Random(*low*, *high*)
SetRandomSeed (*seedNumber*)

Floating Point Math Functions

Acos(*x*)
Acosh(*x*)
Asin(*x*)
Asinh(*x*)
`Atan(`*x*`)`
`Atan2(`*x*`,`*y*`)`
`Atanh(`*x*`)`
`CopySign(`*x*`,`*y*`)`
`Cos(`*x*`)`
`Cosh(`*x*`)`
`Erf(`*x*`)`
`Erfc(`*x*`)`
`Exp(`*x*`)`
`Expm1(`*x*`)`
`Fabs(`*x*`)`
`FDim(`*x*`,`*y*`)`
`FMax(`*x*`,`*y*`)`
`FMin(`*x*`,`*y*`)`
`Fmod(`*x*`,`*y*`)`
`Gamma(`*x*`)`
`Hypot(`*x*`,`*y*`)`
`IsFinite(`*x*`)`
`IsNaN(`*x*`)`
`IsNormal(`*x*`)`
`LessEqualOrGreater(`*a, b*`)`
`LessOrGreater(`*a, b*`)`
`LGamma(`*x*`)`
`Log(`*x*`)`
`Logb(`*x*`)`
Log1p(*x*)
Log10(*x*)
NearbyInt(*x*)
`NextAfterD(`*x*`,`*y*`)`
Pow(*x*,*y*)
`RandomX(`*x*`)`
`Remainder(`*x*`,`*y*`)`
`RemQuo(`*x*`,`*y*`)`
Rint(*x*)
`RintToL(`*x*`)`
`Round(`*x*`)`
`Scalb(`*x*`,`*y*`)`
`SignBit(`*x*`)`
`Signum(`*x*`)`
`Sin(`*x*`)`
`Sinh(`*x*`)`
`Sqrt(`*x*`)`
`Tan(`*x*`)`
`Tanh(`*x*`)`
`Trunc(`*x*`)`
`Unordered(`*a, b*`)`
`UnorderedGreaterOrEqual(`*a, b*`)`
`UnorderedLessOrEqual(`*a, b*`)`
`UnorderedOrEqual(`*a, b*`)`
`UnorderedOrGreater(`*a, b*`)`
`UnorderedOrLess(`*a, b*`)`

Managing the Floating Point Environment

`FeClearExcept(`*excepts*`)`
`FeGetEnv()`
`FeGetExcept(`*excepts*`)`
`FeHoldExcept()`
`FeRaiseExcept(`*excepts*`)`
`FeSetEnv(`*envObj*`)`
`FeSetExcept(`*flagObj*`, `*excepts*`)`
`FeTestExcept(`*excepts*`)`
`FeUpdateEnv(`*flagObj*`)`

Financial Functions

`Annuity(`*rate*`, `*periods*`)`
`Compound(`*rate*`, `*periods*`)`

Exception Functions

`Throw(`*name*`, `*data*`)`
`Rethrow()`
`CurrentException()`

Message Sending Functions

`Apply(`*function*`, `*parameterArray*`)`
`Perform(`*frame*`, `*message*`, `*parameterArray*`)`
`PerformIfDefined(`*receiver*`,`*message*`,`*paramArray*`)`
`ProtoPerform(`*receiver*`,`*message*`,`*paramArray*`)`
`ProtoPerformIfDefined(`*receiver*`,`*message*`,`*paramArray*`)`

Data Extraction Functions

`ExtractByte(`*data*`, `*offset*`)`
`ExtractBytes(`*data*`, `*offset*`, `*length*`, `*class*`)`
`ExtractChar(`*data*`, `*offset*`)`
`ExtractLong(`*data*`, `*offset*`)`
`ExtractXLong(`*data*`, `*offset*`)`

`ExtractWord(`*data*`, `*offset*`)`
`ExtractCString(`*data*`, `*offset*`)`
`ExtractPString(`*data*`, `*offset*`)`
`ExtractUniChar(`*data*`, `*offset*`)`

Data Stuffing Functions

`StuffByte(`*aString*`, `*offset*`, `*toInsert*`)`
`StuffChar(`*aString*`, `*offset*`, `*toInsert*`)`
`StuffCString(`*obj*`, `*offset*`, `*aString*`)`
`StuffLong(`*aString*`, `*offset*`, `*toInsert*`)`
`StuffPString(`*obj*`, `*offset*`, `*aString*`)`
`StuffUniChar(`*aString*`, `*offset*`, `*toInsert*`)`
`StuffWord(`*aString*`, `*offset*`, `*toInsert*`)`

Getting and Setting Global Variables and Functions

`GetGlobalFn(`*symbol*`)`
`GetGlobalVar(`*symbol*`)`
`GlobalFnExists(`*symbol*`)`
`GlobalVarExists(`*symbol*`)`
`DefGlobalFn(`*symbol*`, `*function*`)`
`DefGlobalVar(`*symbol*`, `*value*`)`
`UnDefGlobalFn(`*symbol*`)`
`UnDefGlobalVar(`*symbol*`)`

Miscellaneous Functions

`BinEqual(`*a*`, `*b*`)`
`BinaryMunger(`*dst*`, `*dstStart*`, `*dstCount*`, `*src*`, `*srcStart*`, `*srcCount*`)`
`Chr(`*integer*`)`
`Compile(`*string*`)`
`Ord(`*char*`)`

Reserved Words

The following words are reserved in NewtonScript. You may not use any of these words as symbols unless you enclose the word in vertical bars, like this: `|self|`.

`and`	`end`	`local`	`self`
`begin`	`exists`	`loop`	`then`
`break`	`for`	`mod`	`to`
`by`	`foreach`	native	`try`
`call`	`func`	`not`	`until`
constant	`global`	`onexception`	`while`
`div`	`if`	`or`	`with`
`do`	`in`	`repeat`	
`else`	`inherited`	`return`	

Special Character Codes

This appendix contains a character code table that has both Macintosh and Unicode (16-bit) character codes for the high 128 characters in the Newton character set (characters 128 through 254). When specifying character constants or strings that contain characters from the high 128 characters, you must use unicode character codes. The Macintosh character codes are provided for convenience if you are used to using them.

Table B-1 Character codes sorted by Macintosh character code

Mac	Unicode	Char
80	00C4	Ä
81	00C5	Ą
82	00C7	Ç
83	00C9	É
84	00D1	Ñ
85	00D6	Ö
86	00DC	Ü
87	00E1	á
88	00E0	à
89	00E2	â
8A	00E4	ä
8B	00E3	ã
8C	00E5	ą

continued

Table B-1 Character codes sorted by Macintosh character code (continued)

Mac	Unicode	Char
8D	00E7	ç
8E	00E9	é
8F	00E8	è
90	00EA	ê
91	00EB	ë
92	00ED	í
93	00EC	ì
94	00EE	î
95	00EF	ï
96	00F1	ñ
97	00F3	ó
98	00F2	ò
99	00F4	ô
9A	00F6	ö
9B	00F5	õ
9C	00FA	ú
9D	00F9	ù
9E	00FB	û
9F	00FC	ü
A0	2020	†
A1	00B0	°
A2	00A2	¢
A3	00A3	£

continued

Table B-1 Character codes sorted by Macintosh character code (continued)

Mac	Unicode	Char
A4	00A7	§
A5	2022	
A6	00B6	¶
A7	00DF	ß
A8	00AE	®
A9	00A9	©
AA	2122	™
AB	00B4	´
AC	00A8	¨
AD	2260	≠
AE	00C6	Æ
AF	00D8	Ø
B0	221E	∞
B1	00B1	±
B2	2264	≤
B3	2265	≥
B4	00A5	¥
B5	00B5	µ
B6	2202	∂
B7	2211	∑
B8	220F	∏
B9	03C0	π
BA	222B	∫

continued

Table B-1 Character codes sorted by Macintosh character code (continued)

Mac	Unicode	Char
BB	00AA	ª
BC	00BA	º
BD	2126	Ω
BE	00E6	æ
BF	00F8	ø
C0	00BF	¿
C1	00A1	¡
C2	00AC	¬
C3	221A	√
C4	0192	*f*
C5	2248	≈
C6	2206	Δ
C7	00AB	«
C8	00BB	»
C9	2026	…
CA	00A0	
CB	00C0	À
CC	00C3	Ã
CD	00D5	Õ
CE	0152	Œ
CF	0153	œ
D0	2013	
D1	2014	

continued

Table B-1 Character codes sorted by Macintosh character code (continued)

Mac	Unicode	Char
D2	201C	“
D3	201D	”
D4	2018	‘
D5	2019	’
D6	00F7	÷
D7	25CA	◊
D8	00FF	ÿ
D9	0178	Ÿ
DA	2044	⁄
DB	00A4	¤
DC	2039	‹
DD	203A	›
DE	FB01	fi
DF	FB02	fl
E0	2021	‡
E1	00B7	·
E2	201A	‚
E3	201E	„
E4	2030	‰
E5	00C2	Â
E6	00CA	Ê
E7	00C1	Á
E8	00CB	Ë

continued

Table B-1 Character codes sorted by Macintosh character code (continued)

Mac	Unicode	Char
E9	00C8	È
EA	00CD	Í
EB	00CE	Î
EC	00CF	Ï
ED	00CC	Ì
EE	00D3	Ó
EF	00D4	Ô
F0	F7FF	
F1	00D2	Ò
F2	00DA	Ú
F3	00DB	Û
F4	00D9	Ù
F5	0131	ı
F6	02C6	ˆ
F7	02DC	˜
F8	00AF	¯
F9	02D8	˘
FA	02D9	˙
FB	02DA	˚
FC	00B8	¸
FD	02DD	˝
FE	02DB	˛
FF	02C7	ˇ

Table B-2 Character codes sorted by Unicode

Mac	Unicode	Char
CA	00A0	
C1	00A1	¡
A2	00A2	¢
A3	00A3	£
DB	00A4	¤
B4	00A5	¥
A4	00A7	§
AC	00A8	¨
A9	00A9	©
BB	00AA	ª
C7	00AB	«
C2	00AC	¬
A8	00AE	®
F8	00AF	¯
A1	00B0	°
B1	00B1	±
AB	00B4	´
B5	00B5	µ
A6	00B6	¶
E1	00B7	·
FC	00B8	¸
BC	00BA	º

continued

Table B-2 Character codes sorted by Unicode (continued)

Mac	Unicode	Char
C8	00BB	»
C0	00BF	¿
CB	00C0	À
E7	00C1	Á
E5	00C2	Â
CC	00C3	Ã
80	00C4	Ä
81	00C5	Å
AE	00C6	Æ
82	00C7	Ç
E9	00C8	È
83	00C9	É
E6	00CA	Ê
E8	00CB	Ë
ED	00CC	Ì
EA	00CD	Í
EB	00CE	Î
EC	00CF	Ï
84	00D1	Ñ
F1	00D2	Ò
EE	00D3	Ó
EF	00D4	Ô
CD	00D5	Õ

continued

Table B-2 Character codes sorted by Unicode (continued)

Mac	Unicode	Char
85	00D6	Ö
AF	00D8	Ø
F4	00D9	Ù
F2	00DA	Ú
F3	00DB	Û
86	00DC	Ü
A7	00DF	ß
88	00E0	à
87	00E1	á
89	00E2	â
8B	00E3	ã
8A	00E4	ä
8C	00E5	ą
BE	00E6	æ
8D	00E7	ç
8F	00E8	è
8E	00E9	é
90	00EA	ê
91	00EB	ë
93	00EC	ì
92	00ED	í
94	00EE	î
95	00EF	ï

continued

Table B-2 Character codes sorted by Unicode (continued)

Mac	Unicode	Char
96	00F1	ñ
98	00F2	ò
97	00F3	ó
99	00F4	ô
9B	00F5	õ
9A	00F6	ö
D6	00F7	÷
BF	00F8	ø
9D	00F9	ù
9C	00FA	ú
9E	00FB	û
9F	00FC	ü
D8	00FF	ÿ
F5	0131	ı
CE	0152	Œ
CF	0153	œ
D9	0178	Ÿ
C4	0192	*f*
F6	02C6	ˆ
FF	02C7	ˇ
F9	02D8	˘
FA	02D9	˙
FB	02DA	˚

continued

Table B-2 Character codes sorted by Unicode (continued)

Mac	Unicode	Char
FE	02DB	˛
F7	02DC	˜
FD	02DD	˝
B9	03C0	π
D0	2013	
D1	2014	
D4	2018	
D5	2019	
E2	201A	‚
D2	201C	
D3	201D	
E3	201E	„
A0	2020	†
E0	2021	‡
A5	2022	
C9	2026	…
E4	2030	‰
DC	2039	‹
DD	203A	›
DA	2044	⁄
AA	2122	™
BD	2126	Ω
B6	2202	∂

continued

Table B-2 Character codes sorted by Unicode (continued)

Mac	Unicode	Char
C6	2206	Δ
B8	220F	∏
B7	2211	∑
C3	221A	√
B0	221E	∞
BA	222B	∫
C5	2248	≈
AD	2260	≠
B2	2264	≤
B3	2265	≥
D7	25CA	◊
F0	F7FF	
DE	FB01	fi
DF	FB02	fl

Class-Based Programming

[*]NewtonScript is often described as an "object-oriented" language. However, even if (or especially if) you have some experience with other object-oriented languages, such as Smalltalk or C++, you may be a bit confused by it. NewtonScript does have many features that will be familiar to you, but it has one important difference: NewtonScript is prototype-based, rather than class-based. That is, rather than dividing the world into classes and instances for the purposes of inheritance, NewtonScript objects inherit directly from other objects.

Don't forget everything you know about class-based programming, though. It is possible, and even desirable, to simulate classes in NewtonScript. Even though the language contains no explicit features to support classes, you can use a simple set of stylistic conventions to gain the familiar advantages of classes when you need them, without losing the flexibility of prototypes.

What Are Classes Good For?

Newton programming puts great emphasis on the view system. The structure of your application is based around the views that make up the user interface. Newton Toolkit reflects this strong view orientation, making it very easy to create views with attached data and methods. However, it's not necessarily appropriate to use the view system alone to organize your program.

Most applications of any complexity use various independent, fairly complicated data structures. A standard programming technique is to implement these structures as abstract data types that encapsulate the functionality. In an object-oriented program, these take the form of classes.

Classes let you divide your program's functionality into manageable pieces. By combining a data structure with the functions that operate on it, classes make the program more understandable and maintainable. A well-written class can be reused in other applications, which saves you effort. There are plenty of reasons why classes are a good idea; you can look in any book on object-oriented programming for more.

You should use the view structure of your application to divide it into parts according to the user interface. It's a good idea to implement some or all of the internal data structures as classes.

Classes: A Brief Reminder

Let's start by reviewing the traditional class-based model of object programming. I use Smalltalk concepts and terminology in this article; C++ folks will need to translate the discussion slightly to fit their frame of reference.

The principal concept in the class-based model is, not surprisingly, the class. A class defines the structure and behavior of a set of objects called the instances of the class. Each instance contains a set of instance variables, which are specified in the class. Instances can respond to messages by executing the methods defined in the class. Every instance has the same instance variables and methods. In addition, the class can define class variables and class methods, which are available to all the instances of the class.

Inheritance is also determined by classes. Each class can have a superclass, from which it inherits variable and method definitions. In some languages, a class can have multiple superclasses, but there's no easy way to simulate that in NewtonScript, so I won't consider that here.

An object is created by sending a message, usually called `New` or something similar, to its class. It may also be created by sending a `Clone` message to an instance of the class. When a message is sent to an instance, the corresponding method is located in the class (or superclasses) and executed. The method can refer directly to instance variables from that particular instance, and to class variables.

Inheritance in NewtonScript

The NewtonScript object model is prototype-based. Frames inherit directly from other frames; there are no classes. A frame may be linked to other frames through its `_proto` and `_parent` slots. These slots define the inheritance path for the frame. When you send a message to a frame, the method that executes can use the slots of that frame (the receiver) as variables. If a variable or method reference cannot be resolved in the receiver, the proto chain is searched. If the desired slot still isn't found, the search moves one step up the parent chain, searching the parent and its proto chain, and so forth.

These rules came about because they are a good fit for the Newton programming environment, which is oriented around the view system. Parent inheritance provides inheritance of variables and messages through the view hierarchy: you can define a variable in a view for all its subviews to access. Proto inheritance allows views to share common templates, and also lets most of the data stay out of RAM.

Even though the inheritance system (and all of NewtonScript) is closely integrated with the view system, it is really just a set of rules that can be applied in whatever way you find useful. You can send messages to any frame, not just a view frame, and non-view frames can take advantage of the inheritance rules as well. As this article will demonstrate, the same rules are suitable for a form of class-based programming.

The Basic Idea

I will now describe the basic class-based NewtonScript technique. Remember that there is no built-in idea of a class or an instance in NewtonScript; this is just a set of conventions for using NewtonScript that will let you create structures similar to those you would create in a class-based language. Thus,

although I will use the terms class, instance, and so forth, they are all just frames being used in specific ways.

The main idea is to use parent inheritance to connect instances with classes. An instance is a frame whose slots make up the instance variables, and whose `_parent` slot points to the class (another frame). The class's slots make up the methods.

As a simple example, consider a class `Stack` that implements a push-down stack. It has the standard operations `Push` and `Pop` that add and remove items, and a predicate `IsEmpty` that determines if there are any items on the stack. The representation is very simple: just an array of items and an integer giving the index of the topmost item.

The class frame looks like this:

```
Stack := {
  New:
       func (maxItems)
         {_parent: self,
          topIndex: -1,
          items: Array(maxItems, NIL)},

Clone:
       func () begin
         local newObj := Clone(self);
         newObj.items := Clone(items);
         newObj
       end,

Push:
       func (item) begin
         topIndex := topIndex + 1;
         items[topIndex] := item;
         self
       end,
```

```
Pop:
       func () begin
         if :IsEmpty() then
           NIL
         else begin
           local item := items[topIndex];
           items[topIndex] := NIL;
           topIndex := topIndex - 1;
           item
         end
       end,

IsEmpty:
       func ()
         topIndex = -1
};
```

The class frame begins with the `New` method. This is a class method that is intended to be used as message of the `Stack` class itself, as in `Stack:New(16)`. It consists simply of a frame constructor that builds an instance frame. An instance always has a `_parent` slot that refers to the class frame; note that because New is a message intended to be sent to the class, it can just use self to get the class pointer. The rest of the slots contain the instance variables: a `topIndex` slot for the index of the topmost item (-1 if the stack is empty) and an `items` slot for the array of items. `New` takes an argument that determines the maximum number of items on the stack, but it would be easy to make this dynamic (if it didn't have to fit in an article like this).

It's usually a good idea to provide a `Clone` method for a class. This lets you make a copy of an object without having to know how deep the copy has to go (such knowledge would violate the encapsulation that is one of the reasons to have a class in the first place). In the case of `Stack`, a simple `Clone` would leave the items array shared between two instances, which would result in confusing and incorrect behavior. A `DeepClone`, on the other hand, would copy the entire class frame along with the instance, because of the pointer in the `_parent` slot. That would actually work in this

case, although it would waste huge amounts of space—watch out for this sort of mistake. The correct way to clone a `Stack` is to clone the instance, then give the clone a clone of the items array, which is what the `Clone` method above does.

After the `New` and `Clone` methods, which are usually present in any class, come the methods particular to this class. The `Push` method increments topIndex and adds the item to the end of the items array. Note that instance variables such as `topIndex` and items are accessed simply by their names, because they are slots of the receiver. The `Pop` method calls the `IsEmpty` method to see if the stack is empty. If so, it returns `nil`; if not, it returns the topmost item and decrements topIndex. It assigns `nil` to the former topmost slot so it won't prevent the item from being garbage collected.

The NewtonScript code you write to use a class is similar to code you would write in a language like Smalltalk. You create an object by sending the `New` message to its class. You use the resulting instance by sending messages to it. Of course, you do all this in NewtonScript syntax, which uses the colon for sending a message.

```
s := Stack:New(16);
s:Push(10);
s:Push(20);
x := s:Pop() + s:Pop();
```

At the end of this code, the value of `x` will be 30.

Practical Issues

Before getting into more advanced topics, here's some practical information about doing class-based programming with current tools. (See the *Newton Toolkit User's Guide*. for information about the Newton Toolkit implementation issues discussed in this section.)

The Newton Toolkit as it exists today doesn't include any features that specifically support class-based programming; for example, the browser only

shows the view hierarchy. Nevertheless, it's not too hard to get classes into your application.

You need to build the class frame itself into your package, and you need to make it accessible from NewtonScript code. You can do both at once by putting the class directly into an evaluate slot of your application's main view. For the above example, you could add a slot called `Stack` to the main view and type the class frame just as it appears (but not including the `Stack` assignment (`:=`) line) into the browser.

If you prefer, you could make the class frame a global compile-time variable by putting it (including the assignment this time) into Project Data. That won't make it available to your run-time code, however; you still have to create the `Stack` slot in the main view, but you can just type and enter–Stack–as its value. You have to put the class in Project Data if you want to use superclasses (more on this later).

Class Variables

You can create "class variables", that is, variables that are shared by all instances of a class, by adding slots to the class frame. This is the same way you add variables to a view to be shared by the subviews, but it's a bit more tricky because the view system does something automatically for views that you have to do manually for classes.

Remember that your class frame, like all other data built at compile time, is in read-only space when your application is running. It's not possible to change the values of the slots; thus, it's impossible to assign a new value to a class variable. The view system gets around this problem by creating a heap-based frame whose _proto slot points to the view and using that frame as the view. The original slots are accessible through proto inheritance, and assignments create and modify slots in the heap-based frame, overriding the initial values in the original. You can use the same trick for your class frame.

For example, let's say you want to have a class variable `x` whose initial value is zero. The class frame, defined in the Project Data file, contains a slot named `x`:

```
TheClass := { ... x: 0 ... }
```

The base view has a slot called `TheClass` whose value is defined simply as `TheClass`. At some point early in your application's execution, perhaps in the `viewSetupFormScript` of the view where your class frame is defined, create a heap-based version of the class and assign it to the variable `TheClass`:

```
viewSetupFormScript:
    func () begin
       ...
       if not TheClass._proto exists then
           TheClass := {_proto: TheClass};
       ...
    end
```

Now you can use and assign the variable `x` in methods of `TheClass`. The instances will inherit it via their `_parent` slot, and the first time `x` is assigned, an `x` slot will be created in the heap-based part of the class, shadowing the initial value of zero. Note that you only want to do this setup once—otherwise you'll end up with a chain of frames, one for each time your application has been opened. Checking for the `_proto` slot, as above, is one way to ensure this; you could also set `TheClass := TheClass._proto` in your main view's `viewQuitScript`.

Superclasses

It's easy to get the close equivalent of a superclass: just give the class a `_proto` slot pointing to its superclass. This requires the class definitions to be in Project Data so their names are available at compile time. For example, if you have a `SortedList` class that should have `Collection` as its superclass:

```
Collection := { ... };
SortedList := {_proto: Collection, ...};
```

Of course, you have to define `Collection` before `SortedList` to do it this way. If you prefer to reverse their definitions for some reason, you can add the `_proto` slot later:

```
SortedList := { ... };
Collection := { ... };
SortedList._proto := Collection;
```

If you override a method in a subclass, you can call the superclass version using the inherited keyword. If you have class variables, note that because assignments take place in the outermost frame in the `_proto` chain (that is, the wrapper you create at initialization time for each class), each class gets its own version of the class variable.

Using Classes to Encapsulate Soup Entries

One use to consider for classes is encapsulating the constraints on soup entries. (Read more about soups and soup entries in the Newton Programmer's *Guide*.)

Normally, entries in a soup are simple data records with no `_parent` or `_proto` slots to inherit behavior. The reason is obvious: if they did, each entry would contain a copy of the inherited frame. (Actually, `_proto` slots are not followed in soup entries anyway, for various reasons.) Thus, soup entries are not normally used in an object-oriented fashion.

Unfortunately, soup entries generally have somewhat complicated requirements, such as a set of required slots, so it would be nice to give them an object interface. You can do this by defining a class as a "wrapper" for a particular kind of soup entry. In addition to a `New` method, it can have class methods to retrieve entries from the soup and create objects from existing entries, and instance methods to delete, undo changes, and save changes to the entry. Each instance has a slot that refers to its soup entry.

Given such a wrapper class, you can treat the soup and its contents as objects, sending messages to entries to make changes and retrieve data. The class is then a central location for the code that implements the requirements for entries in the soup.

ROM Instance Prototypes

If your instances are fairly complicated and have a lot of slots whose values aren't all likely to change, you can save some space by using the same `_proto` trick as classes and views. That is, in your `New` method, create the

instance as a little frame that just refers to the class and an initial instance variable prototype that stays in application space:

```
New: func ()
       {_parent: self,
        _proto: '{ ...initial instance vars... }}
```

Leaving Instances Behind

Because so much is contained in the application, it's very difficult to make instances that can survive card or application removal. The only way to do this is to copy the entire class (and its superclasses, if any) into the heap, which would probably take up too much space to be practical.

Conclusion

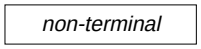
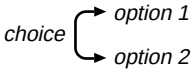
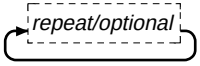
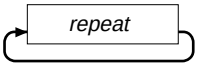
This technique doesn't exactly simulate any existing class-based object system, but it gives you what you need to encapsulate your data types into class-like structures. I find it to be very useful (stores, soups, and cursors all essentially follow this model), and I hope it will help you create better Newton applications. Have fun!

Biography

Walter Smith joined the Newton group in 1988. He is the principal designer and implementor of NewtonScript and the Newton object store. ◆

NewtonScript Syntax Definition

The definitions in this document are presented in two forms, as an extended BNF, and as bubble diagrams, defined as follows:

Bubble Diagram	Extended BNF	Description
	`terminal`	Oval boxes / courier text indicates a word or character that must appear exactly as shown. Ambiguous terminal characters are enclosed in single quotes (‘’).
	*nonterminal*	Rectangular boxes / italics indicate a word that is defined further.
	[]	Dashed lines / brackets indicate that the enclosed item is optional.
	{choose | one}	Forked arrows / a group of words, separated by vertical bars (|) and grouped with curly brackets, indicates an either/or choice.
	[]*	A dashed box with a repeating arrow / an asterik (*) indicates that the preceding item(s), which is enclosed in square brackets, can be repeated zero or more times.
	[]+	A solid box with a repeating arrow / a plus sign (+) indicates that the preceding item(s), which is enclosed in square brackets, can be repeated one or more times.

About the Grammar

The grammar is divided into two parts: the phrasal and lexical grammars.

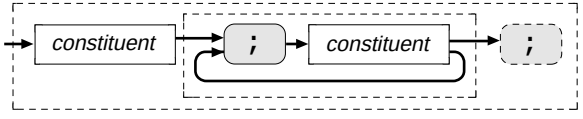
In the phrasal grammar, whitespace is insignificant. `Space`, `tab`, `return`, and `linefeed` characters are considered whitespace. Comments are effectively considered whitespace. Comments consist of the characters between `/*` and `*/` (not nested), and between `//` and a `return` or `linefeed` character.

In the lexical grammar, the nonterminals are characters rather than tokens and whitespace is significant.

Because almost every construct of the language is an expression, many productions ending in expression are ambiguous; the ambiguity is resolved in favor of extending the expression as long as possible. For example, `while true do 2+2` is parsed as `while true do (2+2)` rather than `(while true do 2)+2`. The specific productions affected by this rule are function-constructor, assignment, iteration, if-expression, break-expression, try-expression, initialization-clause, return-expression, and global-function-decl.

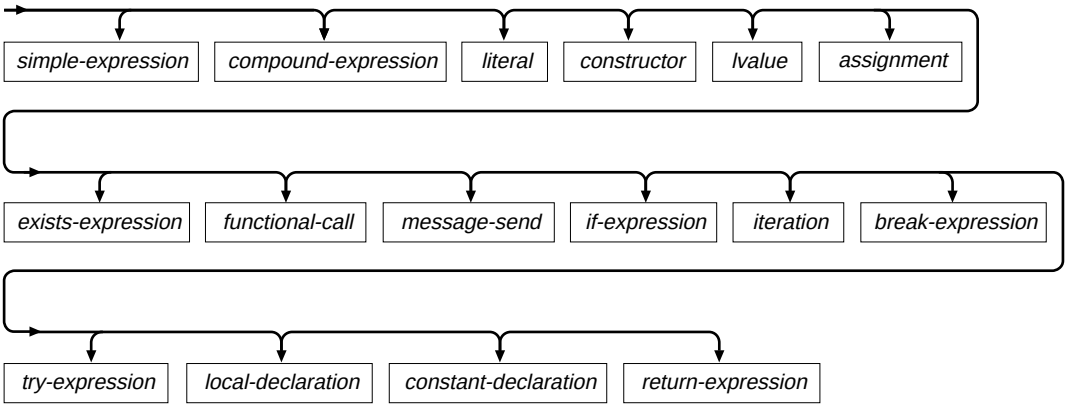
Phrasal Grammar

input:
[*constituent* [`;` *constituent*]* [`;`]]

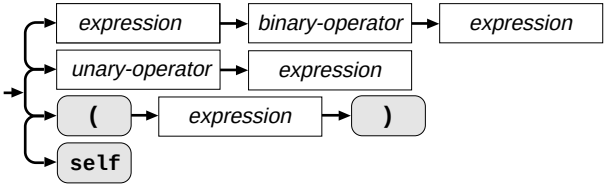


constituent:
{ *expression* | *global-declaration* }

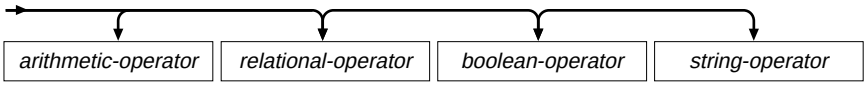
expression:
{ *simple-expression* | *compound-expression* | *literal* | *constructor* | *lvalue* | *assignment* | *exists-expression* | *function-call* | *message-send* | *if-expression* | *iteration* | *break-expression* | *try-expression* | *local-declaration* | *constant-declaration* | *return-expression* }



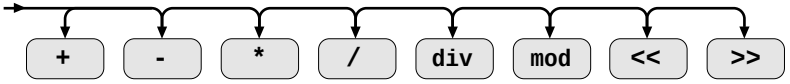
simple-expression:
{ *expression binary-operator expression* | *unary-operator expression* | **(** expression **)** | `self` }



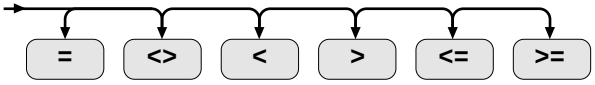
binary-operator:
{ *arithmetic-operator* | *relational-operator* | *boolean-operator* | *string-operator* }



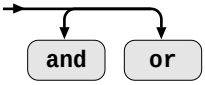
arithmetic-operator:
{ + | - | * | / | div | mod | << | >> }



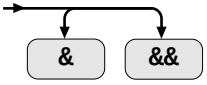
relational-operator:
{ = | <> | < | > | <= | >= }



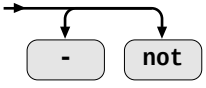
boolean-operator:
{ and | or }



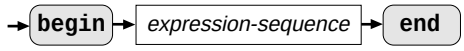
string-operator:
{ & | && }



unary-operator:
{ - | not }

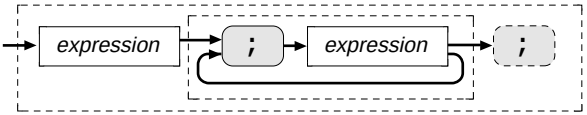


compound-expression:
`begin` *expression-sequence* `end`

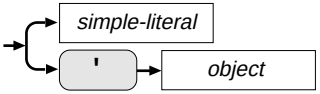


expression-sequence:

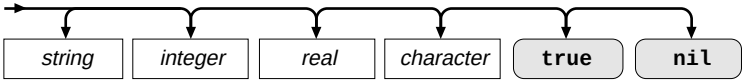
[*expression* [`;` *expression*]* [`;`]]



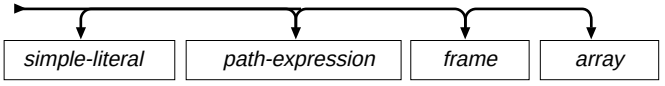
literal:
{ *simple-literal* | `'` *object* }



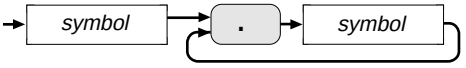
simple-literal:
{ *string* | *integer* | *real* | *character* | `true` | `nil` }



object:
{ *simple-literal* | *path-expression* | *array* | *frame* }



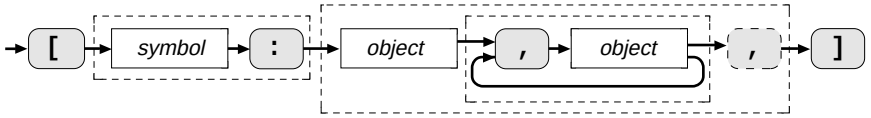
path-expression:
symbol [`.` *symbol*]+



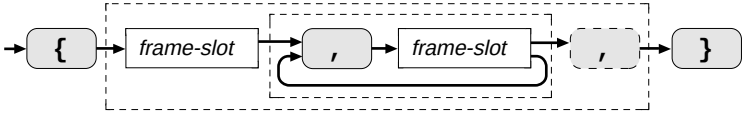
Note

Each dot in ' *symbol* . *symbol* ... is ambiguous: it could be a continuation of the path expression or a slot accessor. NewtonScript uses the first interpretation: `'x.y.z` is one long path expression and not the expression: `('x).y.z`. ◆

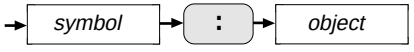
array:
'[' [*symbol* :] [*object* [, *object*]* [,]] ']'



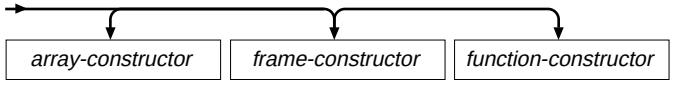
frame:
'{' [*frame-slot* [, *frame-slot*]* [,]] '}'



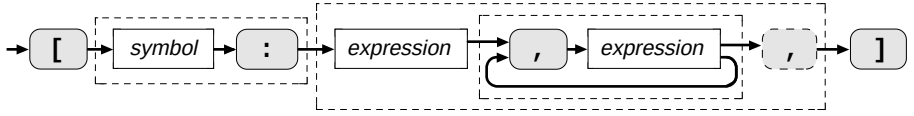
frame-slot:
symbol : *object*



constructor:
{ *array-constructor* | *frame-constructor* | *function-constructor* }



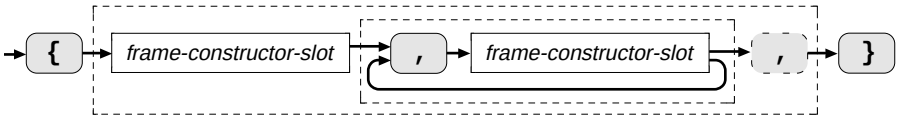
array-constructor:
'[' [*symbol* :] [*expression* [, *expression*]* [,]] ']'



Note

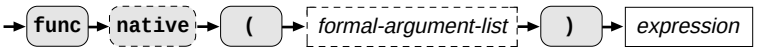
'[' symbol : symbol (… is ambiguous: the first symbol could be a class for the array, or a variable to be used as the receiver for a message send. NewtonScript uses the first interpretation. ◆

frame-constructor:
'{' [*frame-constructor-slot* [, *frame-constructor-slot*]* [,]] '}'

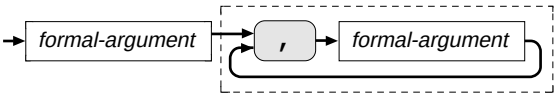


frame-constructor-slot:
symbol : *expression*

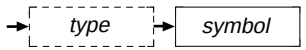
function-constructor:
func [native] ([*formal-argument-list*]) *expression*



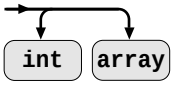
formal-argument-list:
{formal-argument [, formal-argument]*



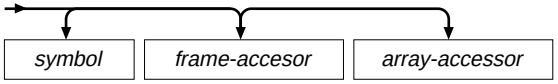
formal-argument:
[[type] symbol



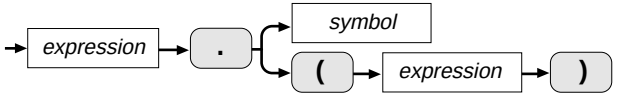
type:
{ int | array }



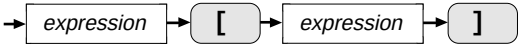
lvalue:
{ *symbol* | *frame-accessor* | *array-accessor* }



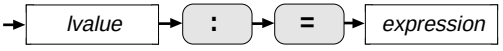
frame-accessor:
expression . { *symbol* | (*expression*) }



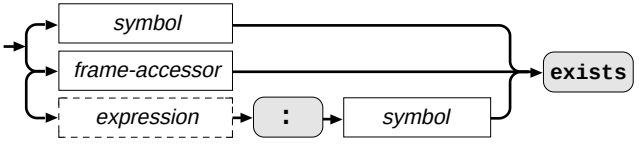
array-accessor:
expression '[' *expression* ']'



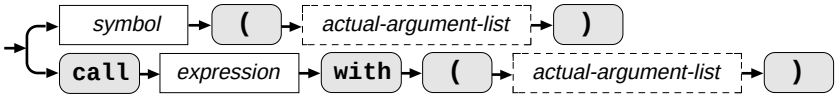
assignment:
lvalue := *expression*



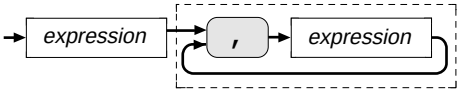
exists-expression:
{ *symbol* | *frame-accessor* | [*expression*] : *symbol* } exists



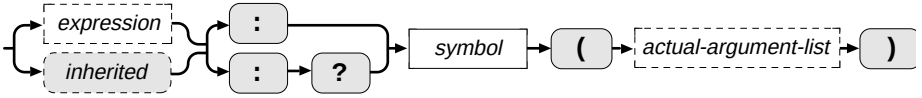
function-call:
{ *symbol* ([*actual-argument-list*]) | `call` *expression* `with` ([*actual-argument-list*]) }



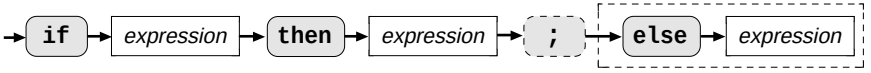
actual-argument-list:
expression [, *expression*]*



message-send:
[{ *expression* | `inherited` }] { : | :? } *symbol* ([*actual-argument-list*])

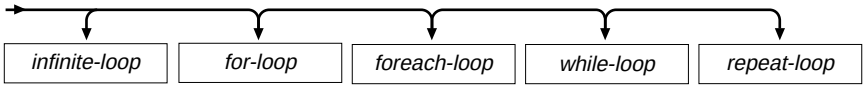


if-expression:
`if` *expression* `then` *expression* [;] [`else` expression]

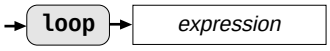


Note
An `else` clause is associated with the most recent unmatched `then` clause. ◆

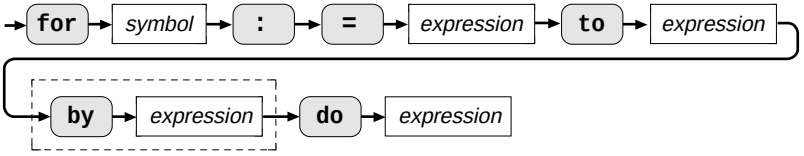
iteration:
{ *infinite-loop* | *for-loop* | *foreach-loop* | *while-loop* | *repeat-loop* }



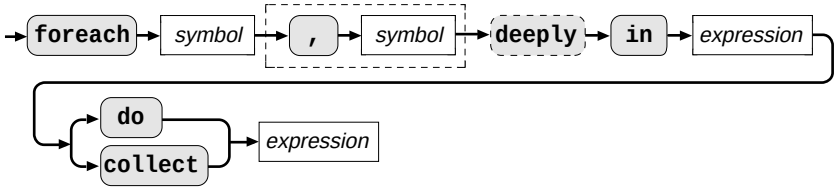
infinite-loop:
`loop` *expression*



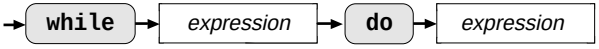
for-loop:
`for` *symbol* `:=` *expression* `to` *expression* [`by` *expression*] `do` *expression*



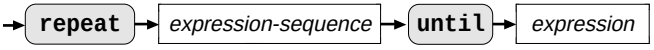
foreach-loop:
`foreach` *symbol* [`,` *symbol*] [`deeply`] `in` *expression* { `do` | `collect` }
expression



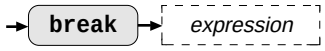
while-loop:
`while` *expression* `do` *expression*



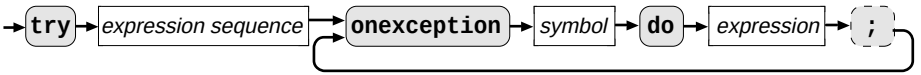
repeat-loop:
`repeat` *expression-sequence* `until` *expression*



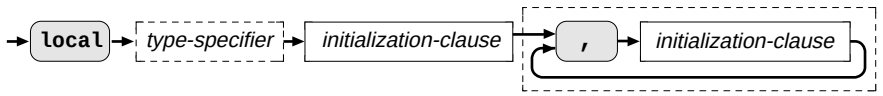
break-expression:
`break` [*expression*]



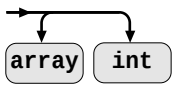
try-expression:
`try` *expression-sequence* [`onexception` *symbol* `do` *expression* [`;`]]+



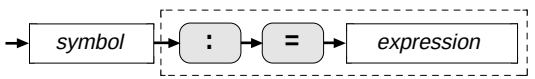
local-declaration:
local [type-specifier] *initalization-clause* [`,` *initalization-clause*]*



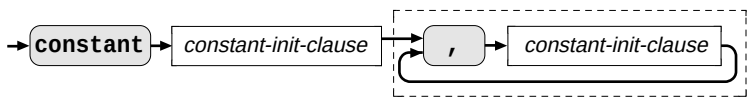
type-specifier:
{ `array` | `int` }



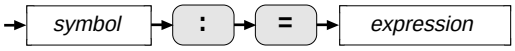
initialization-clause:
symbol [`:=` *expression*]



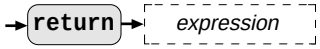
constant-declaration:
`constant` *constant-init-clause* [`,` *constant-init-clause*]*



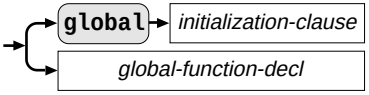
constant-init-clause:
symbol `:=` *expression*



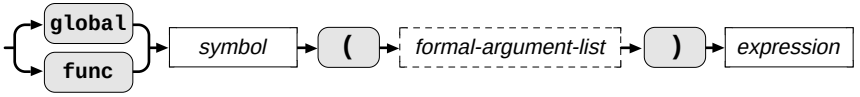
return-expression:
`return` [*expression*]



global-declaration:
{ `global` *initialization-clause* | *global-function-decl* }

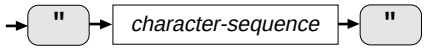


global-function-decl:
{ `global` | `func` } *symbol* `(` [*formal-argument-list*] `)` *expression*

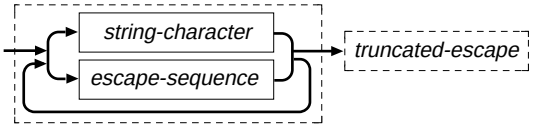


Lexical Grammar

string:
" *character-sequence* "

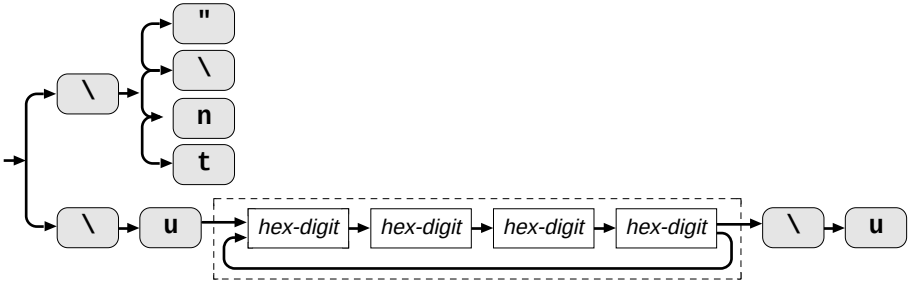


character-sequence:
[{ *string-character* | *escape-sequence* }]* [*truncated-escape*]

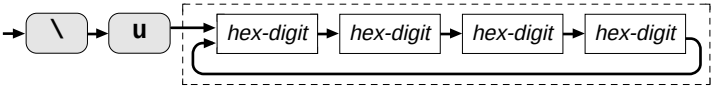


string-character:
<`tab` or any ASCII character with code 32–127 except '"' or '\'>

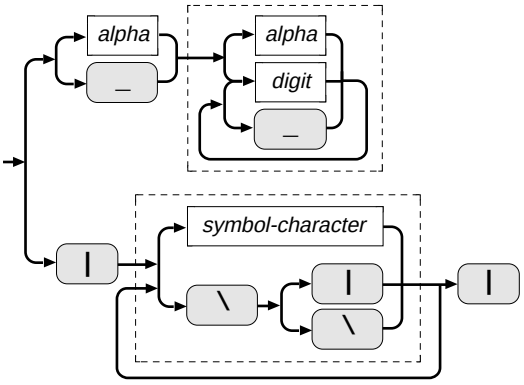
escape-sequence:
{ \ { " | \ | n | t } | \ u [*hex-digit hex-digit hex-digit hex-digit*]* \ u }



truncated-escape:
\ u [*hex-digit hex-digit hex-digit hex-digit*]*



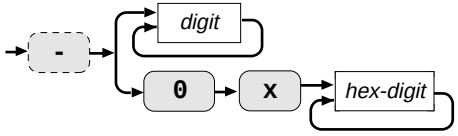
symbol:
{ { *alpha* | _ } [{ *alpha* | *digit* | _ }]* |
'|' [{ *symbol-character* | \ { '|' | \ }]* '|' }



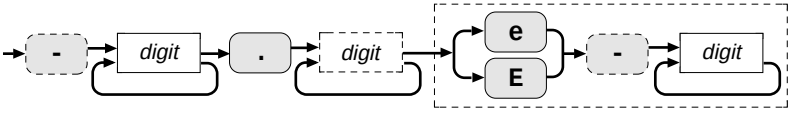
Note
Reserved words are excluded from the nonterminal symbol. ◆

symbol-character:
<any ASCII character with code 32–127 except '|' or '\'>

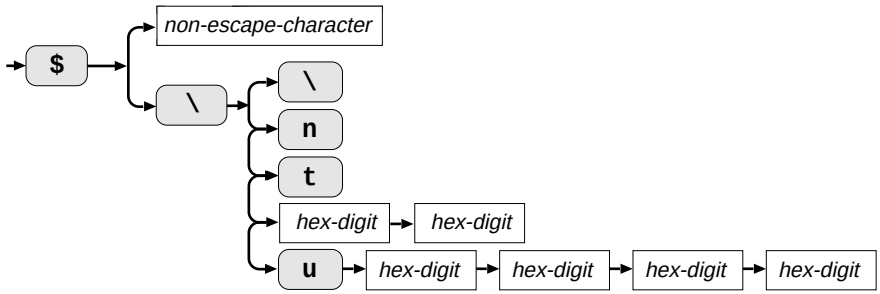
integer:
[-] { [*digit*]+ | `0x` [*hex-digit*]+ }



real:
[-] [*digit*]+ . [*digit*]* [{ e | E } [-] [*digit*]+]



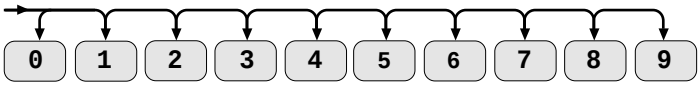
character:
$ { *non-escape-character* | \ { \ | n | t | *hex-digit hex-digit* | u *hex-digit hex-digit hex-digit hex-digit* } }



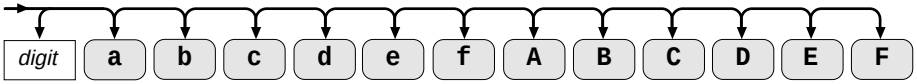
non-escape-character:
<any ASCII character with code 32–127 except '\'>

alpha:
<A–Z and a–z>

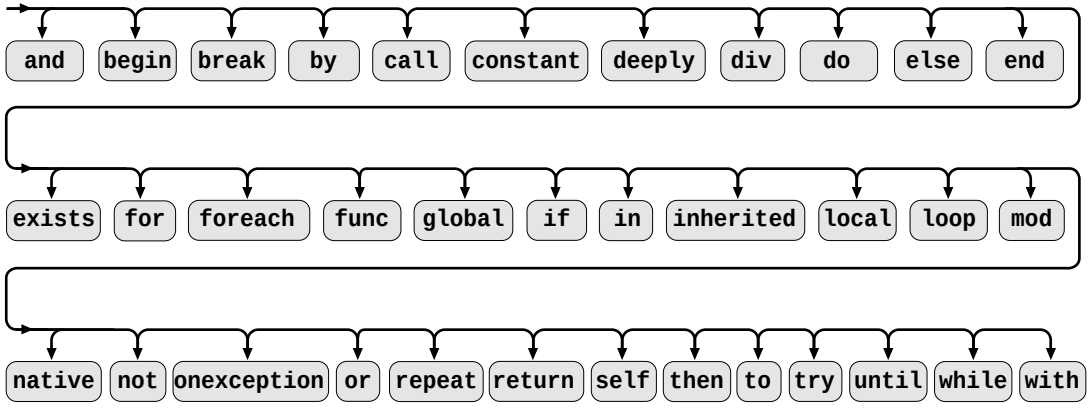
digit:
{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }



hex-digit:
{ *digit* | a | b | c | d | e | f | A | B | C | D | E | F }



reserved-word:
{ and | begin | break | by | call | constant | deeply | div | do | else | end | exists | for | foreach | func | global | if | in | inherited | local | loop | mod | native | not | onexception | or | repeat | return | self | then | to | try | until | while | with }



Operator Precedence

The precedence of operators, from highest to lowest, is shown in Table 2-5 on page 2-39.

Quick Reference Card

The following pages of this appendix contain a quick reference card for the NewtonScript programming language.

NewtonScript Reference Card

Constructs Using Inheritance Lookup	proto	parent
slot	X	X
frame.slot	X	
frame.(*pathExpr*)	X	
GetVariable(*frame*,*slot*)	X	X
GetSlot(*frame*,*slot*)		
frame:*message*() *frame*:?*message*() :*message*()	X	X
inherited:*message*() inherited:?*message*()	X	
symbol exists	X	X
frame.slot exists	X	
frame.(*pathExpr*) exists	X	
frame:*message* exists :*message* exists	X	X
HasVariable(*frame*,*slot*)	X	X
HasSlot(*frame*,*slot*)		

NewtonScript Operator	Description	Evaluation
.	slot access	left-to-right
: :?	message send conditional message send	left-to-right
[]	array element	left-to-right
-	unary minus	left-to-right
<< >>	left shift right shift	left-to-right
* / div mod	multiply float division integer division remainder	left-to-right
+ -	add subtract	left-to-right
& &&	concatenate (string rep of *exprs*) concatenate with 1 space between	left-to-right
exists	variable & slot existence	none
< <= > >= = <>	less than less than or equal greater than greater than or equal equal (pointer equality) not equal (pointer inequality)	left-to-right
not	logical not	left-to-right
and or	logical and short logical or circuit	left-to-right
:=	assignment	right-to-left

ression	Value
in *exprList* end	value of last statment in *exprList*
urn *[expr]*	nil or *expr*
xpr then *expr* *xpr* then *expr1* else *expr2*	value of *expr* or nil value of *expr1* or *expr2*
p *expr*	value of break
var := *expr1* to *expr2* do *expr* *var* := *expr1* to *expr2* by *step* do *expr*	nil or value of break
each *[slot,] val* in *frameOrArray* do *expr* each *[slot,] val* deeply in *frame* do *expr*	nil or value of break (deeply in follows _proto slots)
each *[slot,] val* in *frameOrArray* collect *expr* each *[slot,] val* deeply in *frame* collect *expr*	array of collected *expr* values or value of break (deeply in follows _proto slots)
le *expr* do *expr*	nil or value of break expression
eat *exprList* until *expr*	nil or value of break expression
ak *[expr]*	nil or *expr*
l *functionObject* with (*argList*)	value *functionObject* returns
exprList onexception *exceptionSymbol* do *expr …* t.ex| {name: *exceptionSymbol*, error: *integer*} t.ex.msg| {name: *exceptionSymbol*, message: *string*} t.ex;type.ref| {name: *exceptionSymbol*, data: *datum*}	value of last expr in *exprList* or of the executed onexception Throw(*exSym*, *datum*), CurrentException(), Rethrow()

	Examples	Comments	SPrintObject	ClassOf	PrimClassOf
	42, 0x5BA6, -99	-2^{29} … 2^{29}-1 (-536870911…+536870912)	base 10	Int	Immediate
	1000.02,-3.14,1.0e5, 1.e-12	SANE double, 15-16 digits, exponent: -308…308	1,000.02	real	Binary
n	nil true	don't quote	null string	Weird_Immediate Boolean	Immediate
ter	$a, $7, $\\, $\F0, $\uF7FF	$*xx* for hex, $\u*xxxx* for unicode hex	one char string	Char	Immediate
	"abc", "\n","\t","\"","\\"	abc, newline, tab, double quote, backslash	the string	String	Binary
l	hiho, baz_1, |evt.ex.msg|	254 chars, [a-z, A-Z, 0-9, _], vbars allow any char	symbol name	Symbol	Binary
	[arrayClass: e1,e2,e3]	class optional, trailing comma allowed	null string	*array class* or Array	Array
	{slot1: val1, slot2: val2}	trailing comma allowed	null string	*class slot* or Frame	Frame

Glossary

Array	A sequence of numerically indexed slots (also known as the array elements) that contain objects. The first element is indexed by zero. Like other non-immediate objects, an array can have a user-specified class, and can have its length changed dynamically.
Binary object	A sequence of bytes that can represent any kind of data, can be adjusted in size dynamically, and can have a user-specified class. Examples of binary objects include strings, real numbers, sounds, and bitmaps.
Boolean	A special kind of immediate called `true`. Functions and control structures use `nil` as false and anything else as true. If you don't have anything else use `true`.
Child	A frame that references another frame (its parent) from a `_parent` slot.
Class	A symbol that describes the data referenced by an object. Arrays, frames, and binary objects can have user-defined classes.
Constant	A value that does not change. In NewtonScript the value of the constant is substituted wherever the name of the constant is used as an expression.

Frame	An unordered collection of slots, each of which consists of a name and value pair. The value of a slot can be any type of object, and slots can be added or removed from frames dynamically. A frame can have a user-specified class. Frames can be used like records in Pascal and structs in C, but can also be used as objects which respond to messages.
Function object	Function objects are created by the function constructor: `func(`*args*`)` *funcBody* An executable function object includes values for its lexical and message environment, as well as code. This information is captured when the function constructor is evaluated at run time.
Global	A variable or function that is accessible from any NewtonScript code.
Immediate	A value that is stored directly rather than through an indirect reference to a heap object. Immediates are characters, integers, or Booleans. See also **reference.**
Implementor	The frame in which a method is defined. See also **receiver.**
Inheritance	The mechanism by which attributes (slots or data) and behaviors (methods) are made available to objects. Parent inheritance allows views of dissimilar types to share slots containing data or methods. Prototype inheritance allows a template to base its definition on that of another template or prototype.
Local	A variable whose scope is the function within which it is defined. You must use the `local` keyword to explicitly create a local variable within a function.
Message	A symbol with a set of arguments. A message is sent using the message send syntax, *frame*`:`*messageName*`()`, where the message, *messageName*, is sent to the receiver, *frame*.
Method	A function in a frame slot that is invoked in response to a message.

Object	A typed piece of data that can be an immediate, array, frame, or binary object. In NewtonScript, only frame objects can hold methods and receive messages.
Parent	A frame that is referenced through the `_parent` slot of another frame.
Path expression	An object that encapsulates an access path through a set of arrays or frames.
Proto	A frame that is referenced through another frame's `_proto` slot.
Receiver	The frame that was sent a message. The receiver for the invocation of a function object is accessible through the pseudo-variable `self`.
Reference	A value that indirectly refers to an array, frame, or binary object. See also **immediate.**
Self	A pseudo-variable that is set to the current receiver.
Slot	An element of a frame or array that can hold an immediate or reference.

Index

A

B

C

D

E

F

G

H

I, J, K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro 630 printer. Final page negatives were output directly from the text and graphics files. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITERS
Adrian Yacub, Cheryl Chambers, Christopher Bey

PROJECT LEADER
Christopher Bey

ILLUSTRATOR
Peggy Kunz

EDITORS
Linda Ackerman, David Schneider

PRODUCTION EDITOR
Rex Wolf

PROJECT MANAGER
Gerry Kane

Special thanks to Peter Canning, Bob Ebert, Mike Engber, Kent Sandvik, and Walter Smith.