

Pocket Smalltalk User's Guide

By [Andrew Brault](#)

Updated by N. Chamfort (December 16, 2001)

This brief guide gives an overview of using Pocket Smalltalk to write applications for Palm platform handheld computers.

To use this guide effectively you should be familiar with the Smalltalk programming language, but a brief introduction to Smalltalk is included for beginners.

Contents

1. [Introduction](#)
2. [About Smalltalk](#)
3. [What Makes Pocket Smalltalk Different](#)
4. [Interfacing with the PalmOS](#)
5. [Floating-Point Support](#)
6. [Constants](#)

1. Introduction

Pocket Smalltalk is a Windows 95/NT application for writing Smalltalk programs to run on the Palm platform devices.

You can use the full power of the Smalltalk programming language and its integrated development environment to write programs on your PC, then create an "executable" .PRC file that can be HotSync'd to your PalmPilot for execution and deployment.

Pocket Smalltalk's features include:

- Ability to create standalone executables.
- Tiny virtual machine size: 20k to 25k depending on included features.
- Compact, yet complete, Smalltalk class library.
- Full access to the 800+ PalmOS API functions from within Smalltalk.
- Native widgets — Pocket Smalltalk applications use the built-in PalmOS user interface.
- Ability to access PalmOS databases, including reading and writing of Memo Pad entries.
- Extensible virtual machine.
- Runtime debugger to help pinpoint errors (can be removed from deployed applications to save space).
- Automatic memory management via garbage collection.
- Full Smalltalk Behavior/Metaclass support, #doesNotUnderstand:, proxy classes, #perform:, #isKindOf:, #become: and other reflection methods,
- Extremely compact compiled code size.

Pocket Smalltalk has been used to create several deployed applications and is a practical alternative to other development environments.

2. About Smalltalk

This chapter gives a brief overview of the Smalltalk programming language. There are some excellent tutorials on the World Wide Web for those interested in learning Smalltalk (links given at the end of this page). This chapter is mainly to help you determine whether Smalltalk is the right language for you. Experienced Smalltalk users can skip this chapter.

If you are new to computer programming, don't be intimidated by the somewhat rushed summary given below. Just read it once or twice to get a general feel for things, and don't worry if you don't understand everything you read.

Synopsis

Smalltalk is a simple and powerful object-oriented programming language. Unlike "hybrid" object-oriented languages like Java and C++, Smalltalk is considered to be a "pure" object-oriented language. Smalltalk is said to be "pure" for one main reason: Everything in Smalltalk is an object, whereas in hybrid systems there are things which are not objects (for example, integers in C and Java).

The benefits of the "everything is an object" philosophy are great, and pure languages such as Smalltalk are considered to be more productive and (more importantly) more fun to program in.

Smalltalk is fundamentally tied to automatic dynamic memory management, and as such must be supported by an underlying automatic memory management system. In practical terms this means that a Smalltalk programmer no longer needs to worry about when to free allocated memory. When finished with a dynamically allocated object, the program can simply "walk away" from the object. The object is automatically freed, and its storage space recycled, when there is nothing else referencing it.

Message Sending

All computations in Smalltalk are performed by objects sending messages to one another. A message is a name for a routine to execute, possibly together with some arguments to the routine. The power in this approach is that different objects can respond to the same message in different ways (that is, by executing different routines). Moreover, the sender of a message does not need to know in advance how the receiver will interpret the message. This is the power of object oriented programming.

Smalltalk has a uniquely simple and elegant syntax for expressing message sending. There are three possible forms:

receiver doSomething

This is a "unary" message send. Write the receiver on the left and the message name ('doSomething', in this case) on the right. There are no arguments.

receiver + argument

This is a "binary" message send. As before, the receiver goes on the left, followed by the message name ('+'), followed by exactly one argument. This form of message sending is always used for messages with "funny" names like +. Any message name that is composed of symbols instead of letters and numbers uses this syntax, and all such messages take exactly one argument.

receiver take: 3 of: 'candy'

This is a "keyword" message send. Keywords ending in : (colon) alternate with arguments. The whole thing is considered a single message send. The message name is the concatenation of all the keywords (in this case, 'take:of:'). There are as many arguments as there are keywords, and there can be as many keywords as you want.

Blocks

One of the most useful features of Smalltalk is the ability to define "local functions" in the form of blocks. Blocks have many of the properties of lexical closures and can be used to implement a wide variety of control structures. In fact, Smalltalk has no built-in control structures, but instead implements them in terms of blocks. Blocks are written as one or more Smalltalk statements between square brackets ([and]). Arguments to block functions can be specified by putting the argument names, prefixed by colons, before a vertical bar (|) at the beginning of the block. Here are some examples:

```
5 timesRepeat: [self print: 'Hello world!'].

primes := (1 to: 100) select: [:number | number isPrime].

#('mary' 'had' 'a' 'little' 'lamb')
do: [:string | self print: string]
separatedBy: [self space].

[self shouldContinue]
whileTrue: [self performAction].
```

The important thing to note here is that all control flow constructs are implemented as messages interpreted by various kinds of objects. For example, 'timesRepeat:' is a message understood by integers.

Classes and Inheritance

How do objects know how to respond to messages? The answer reveals another general unifying feature of Smalltalk: every object belongs to a class. A class defines which messages can be understood by members (also called instances) of that class, and it defines how objects respond to those messages.

Since everything in Smalltalk is an object, everything has a class. For example, the number 5 has a class named Integer.

The string 'hello' has a class named String, and so forth. You can define your own classes and create instances of them. You can find out what class any object belongs to by sending it the message 'class'.

Classes themselves are instances of other classes called metaclasses. Metaclass programming is a relatively advanced part of Smalltalk programming, but it can be used to achieve some important effects. The Smalltalk class browser makes both "ordinary class" and metaclass programming quite easy.

When you program in Smalltalk, you spend most of your time creating and modifying class definitions. In the Smalltalk class browser window, you can associate message names (also called selectors) with Smalltalk program code fragments (also called methods). A class, therefore, is a mapping between selectors and methods. When an instance of a class receives a message, it consults its class to see which method it should invoke in response to the message.

A very important ability of Smalltalk classes is that they can inherit from other classes. This reflects the observation that most objects can be viewed as specializations of more general objects. For example a character string like 'hello' can be viewed as a collection of letters (characters). Therefore, the String class inherits many of its abilities from another more general class called Collection. Every class inherits from exactly one other class (this is known as **single inheritance**, in contrast to the **multiple inheritance** of C++, Self, and to a lesser extent Java). Every class therefore inherits abilities from another class, called its superclass. There is one exception to this rule: the class called Object has no superclass. Therefore, every other class ultimately inherits from Object. Inherited abilities are themselves inherited, so superclasses can be "nested" to any depth. For example the Integer class inherits from Number, which in turn inherits from Object.

Summary and Links

This has been a very brief introduction to the Smalltalk language. If you are interested in pursuing further, see the links below for some online Smalltalk tutorials:

- [Dolphin Smalltalk Education Centre](#)
- [IBM Smalltalk Tutorial](#)

3. What Makes Pocket Smalltalk Different

Pocket Smalltalk is in some ways a very different kind of Smalltalk system than what you may be used to. Many of these differences stem from three important characteristics of this system:

- It is a **cross-compiler**, that is, it creates executable programs which run on the PalmPilot rather than on the host PC.
- It is designed to be compact. Some changes from traditional Smalltalk have been made to reduce memory consumption.
- It is **declarative**. Programs can be represented in a linear form free of "floating" references.

Declarative Smalltalk

Pocket Smalltalk is an example of a *declarative* Smalltalk system. The new ANSI Smalltalk standard is moving toward declarative systems. Whereas a traditional Smalltalk program is built by extending the base image, in the process building arbitrary graphs of objects, a Pocket Smalltalk program is completely defined by its classes and methods. As a consequence, a Pocket Smalltalk program can be represented as a simple file in "file-out" format. In contrast, a traditional Smalltalk program cannot be represented declaratively at all, and can only be recreated by reapplying the series of actions used to extend the base image.

Pocket Smalltalk uses a simple and flexible source code management facility called "packages". A package is simply a file in standard Smalltalk file-out format. The IDE knows about packages and provides a user interface for partitioning your classes and methods between packages. In this way, you can separate base system code from your own application code. Packages are described in more detail in later chapters.

Starting Up — the #start method

When your compiled program first starts up, it begins by sending the selector #basicStart to the class Smalltalk, which by default resends #start. You must replace the default #start method with your own in order to get your program running. This is analogous to the "main" routine in C, and is somewhat unique to this version of Smalltalk. Most Smalltalk systems do not have any concept of a well-known entry point, but Pocket Smalltalk programs always begin this way.

Symbols

Symbols are used for three main purposes in Smalltalk:

1. Naming methods (i.e., selectors)
2. Naming classes
3. Serving as unique "tokens"

A symbol is written as #symbolName, where symbolName is any valid Smalltalk identifier.

Since symbol data accounts for a large fraction of the total size of a typical Smalltalk program, Pocket Smalltalk will replace each symbol by a unique SmallInteger at runtime. As a consequence, there is no Symbol class, nor is there a way to distinguish between symbols and integers at runtime. This is almost never a problem in practice though, and if necessary, symbols can be "wrapped" in another class to provide the desired functionality.

This optimization saves a considerable amount of memory space and execution time. The only real disadvantage, aside from losing the distinction between Symbols and SmallIntegers, is that you will be unable to access the characters of a symbol (for example, in order to print the symbol). Again, this should not be a problem since Symbols should not be printed at runtime anyways (Strings should be used for this purpose).

For debugging purposes, you may choose to include the text of each symbol (select the "emit debug info" option in the IDE). Then you may use the expression:

```
Context textOfSymbol: symbol
```

to recover the String containing the symbol's characters. The default MiniDebugger uses this feature to provide a symbolic stack trace when an error occurs.

The System Dictionary

In Pocket Smalltalk, the system dictionary called "Smalltalk" is actually a class. You cannot look up classes dynamically (using #at:, for instance) at runtime as you can with other Smalltalk systems. You also cannot add or remove classes (or methods) at runtime.

Integers

Pocket Smalltalk currently does not provide LargeInteger (arbitrary size integer) arithmetic, due to the large amount of memory space required by such a facility. This may be added later as an add-on package. For now, integers of magnitude -2^{31} to $2^{31}-1$ can be represented in Pocket Smalltalk (i.e. 32 bit signed integers). Integers between -16384 and 16383 are represented directly as SmallIntegers and take no object storage. Integers with larger magnitudes are represented as instances of LongInteger allocated in the heap. Conversions between the integer types occur automatically.

Proxies, nil, and doesNotUnderstand:

When an object is sent a message which it cannot interpret, a Message object representing the message sent is created, and then the message #doesNotUnderstand: is sent to the object with the Message as the argument.

By default, #doesNotUnderstand: just signals an error, but certain classes may want to intercept #doesNotUnderstand: and take some special action.

A class which provides very little behavior of its own, but instead forwards most messages onto another object is called a "proxy". Proxies and other classes with similar needs can subclass from nil instead of from Object (or a subclass of Object), thereby inheriting none of the "basic" methods defined by class Object. They may then use #doesNotUnderstand: to forward messages to another object.

A class inheriting from nil need only implement the one selector: #doesNotUnderstand:. The cross compiler does not allow you to create a subclass of nil which does not implement this selector.

#become:

The primitive operation Object>>#become: swaps the identity of two objects. In Pocket Smalltalk this is an efficient operation. You must be careful with this operation because it can crash the virtual machine if you "become" the receiver of a method into an object with fewer named instance variables. The "become" primitive will fail if you try to convert from a pointerless object (e.g., a string or a byte array) to a pointer object, or vice versa. It will also fail if you try to swap the identities of SmallIntegers, or if either object is a statically allocated object (i.e., a class, metaclass, or literal).

The built-in collection classes implement expansion by means of #become:. This allows a collection to be represented by a single object, rather than two objects as in some Smalltalk implementations.

Global variables

Global variables are handled a bit differently in Pocket Smalltalk than in other Smalltalk systems. Rather than being Associations in the Smalltalk system dictionary, global variables are class variables of Object. Since all classes (except root classes; see above) are subclasses of Object, they all can access these class variables the same way as global variables.

At compile time, references to class (and "global") variables are converted into single instructions. No separate objects are created for global variables; nor can you set the value of a global variable at compile time (it must be done with some kind of initialization code at runtime).

Unlike other Smalltalks, classes are not global variables. For most purposes, you can treat them as such, but the major difference is that you cannot assign to them. Class names must not conflict with class variable names.

`#ifNil:`

A very common idiom in Smalltalk is the following:

```
object isNil ifTrue: [...]
```

This expression can be simplified by defining a new message `#ifNil:` so that you can write:

```
object ifNil: [...]
```

The problem with doing this is that ordinary Smalltalks cannot apply compiler optimizations to this expression, so the above code will execute more slowly and take more space than the usual `isNil ifTrue:` case. Pocket Smalltalk, however, knows how to optimize `ifNil:` and related messages, so you can use them without any penalty. The message forms it recognizes are as follows:

- `ifNil: [...]`
- `ifNotNil: [...]`
- `ifNil: [...] ifNotNil: [...]`
- `ifNotNil: [...] ifNil: [...]`
- `orIfNil: [...]`

The last message deserves some explanation. `orIfNil:` answers the receiver if the receiver is not nil, but if the receiver is nil it answers the result of evaluating the argument block. This can be used to provide "default" values for possibly-nil variables:

```
^name orIfNil: ['anonymous']
```

4. Interfacing with the PalmOS

Pocket Smalltalk provides full support for accessing PalmOS functionality from within Smalltalk programs. Your programs can use PalmOS GUI resources (forms, alerts, menus, etc.) and you have access to the over 800 PalmOS API functions. This chapter describes how to use this functionality.

Using PalmOS Resources

You can instruct Pocket Smalltalk to "link" one or more PalmOS resource databases into the finished "executable" .PRC file. The compiler will combine the resources in each database you specify, along with new resources for the compiled Smalltalk methods and objects, into the final .PRC file. Therefore, you can refer to resources in your Smalltalk program by their resource ID, just as you would when writing in C or another language.

The mechanism for including resource databases in your project will be explained in a later chapter. Briefly, you must use the Constants Browser to create one or more named constants containing the filenames of the resource databases to use.

Calling PalmOS API Functions

PalmOS provides you with a rich library of functions. You can call any PalmOS API function using a special message-send syntax. The receiver must be the word SYSTRAP and the selector is the name of the routine. For example, to get the height of the current font in pixels, you could write:

```
fontHeight := SYSTRAP FntCharHeight.
```

If an API function takes arguments, you write the SYSTRAP line as a keyword message-send. The first keyword is taken to be the function name and the other keywords can be anything you want. For example, to draw a line from the top-left corner of the screen to the bottom-right:

```
SYSTRAP WinDrawLine: 0 y: 0 x: 160 y: 160.
```

Note that the most common SYSTRAP calls are already encapsulated in appropriate methods in the standard Smalltalk class library, so you will only rarely have to deal with SYSTRAP calls directly.

CPointer

Since many PalmOS API functions return pointers or take pointers as arguments, there is a Smalltalk class called CPointer which is the equivalent of a C void pointer. Normally you do not need to know what a CPointer is pointing to, and you can simply pass instances around as "magic cookies". There are some methods in CPointer for dereferencing the pointer which are useful for reading and writing to C structures. You can also allocate and free PalmOS dynamic memory using methods in CPointer.

You must be careful to pass the correct types of arguments to PalmOS API functions. Arguments must either be Smalltalk Integers or CPointers. In particular, boolean arguments must be either 0 or 1, not true or false. If you pass incorrect argument types to an API function the receiver of the method that made the incorrect call is sent `#badAPICall`, which by default signals an error. You can see what parameters an API call expects by using the SYSTRAP Browser.

Some PalmOS API calls expect a pointer to a string, along with a separate length parameter. You can convert a Smalltalk String to a pointer to a C string by sending the String `#copyToHeap`. You must be sure to free the pointer after use by sending `#free` to the CPointer object. As a special feature, you can simply pass a byte-indexable object (such as a Smalltalk String) as the pointer to a string, instead of converting the String to a C-readable string. You still need to pass the length argument. This technique cannot be used with functions that expect 0-terminated strings, as Pocket Smalltalk Strings are not 0-terminated.

5. Floating-Point Support

You can access 64-bit double precision floating-point values from within Smalltalk. To get this ability you must:

- Install the optional package 'double.st'
- Load the included library 'MathLib.prc' into your Pilot
- Link with a VM that supports floating point ('vm-m.prc' or 'vm-md.prc')

Floating-point arithmetic support is only available on PalmOS 2.0 and later. PalmOS 1.0 devices (i.e., the Pilot 1000 and 5000) do not support double precision floating point.

The package 'double.st' adds methods to the built-in Double class to support floating-point operations. It also adds some methods to other number classes to implement Smalltalk's double dispatching arithmetic rules. You can therefore freely intermix Doubles with other numeric types in arithmetic expressions.

Note that you must have the MathLib shared library installed on your Pilot in order to use the floating-point math routines. If the library is not installed, you will get an error message when you try to start your Pilot program. If you distribute a program that requires floating-point support, you should therefore include MathLib.prc in your distribution.

To save space, special operations such as `#cos` (cosine) are implemented only in class Double and not in other Number classes. If you plan to use such functions, you should send `#asDouble` to the receiver first.

6. Constants

Pocket Smalltalk provides a convenient extension to Smalltalk to support named constants. Named constants are similar to pool dictionary entries in other Smalltalks, but with the following differences:

- They have a different syntax: `##constantName` instead of `PoolDictionaryEntry`.
- They are defined centrally via the Constants Browser.
- They can only hold simple literal values: Strings, Arrays, Integers, and so on.
- They can belong to a package and can be saved and loaded along with that package.

Named constants can be used to hold values that would normally be `#include'd` from the PalmOS header files if you were programming in C. Thus, event type IDs, drawing modes, and many other PalmOS constants are available as named constants, and can be used without fear of "bloating" the size of your application.

Constants are created and edited using the Constants Browser. Constants belong to named "categories" for documentation purposes, in a similar manner as methods belong to message categories. Categories for constants can be edited in the upper-left pane of the Constants Browser. The upper-right pane holds the names of the constants in the selected category. By selecting a constant, you can see its name in the status line at the bottom of the window. You can create new constants and rename or delete existing constants using the context menu in the upper-right pane.

When you create a new constant, it is added to the default package. When you save that package, the constants in that package are saved along with the package. You can change the package a constant belongs to by using the upper-right pane's context menu option *Change package...*

Named constants are also used for another very important purpose: to define "system properties". The IDE knows about a few constants and uses their values to determine various attributes of the generated .PRC file. When you start a new project, these constants are initialized with reasonable default values and put in the "uncommitted" package. If you modify one of these constants, you should add it to the package for your application. The constants and their effects are listed here:

```
##applicationTitle
    A String containing the title of your application.
```

```
##creatorID
```

A 4-element String or a 4-element ByteArray containing the Creator ID of your application. Every application must have a unique Creator ID.

##debug

Set to true if debugging information is to be added to the compiled executable; false to omit the debugging information.

##optimization

Set to true to apply optimizations to the compiled code. The code generation process takes longer when this option is set.

##dataStackSize

The number of slots to reserve for the Smalltalk data stack.

##callStackSize

The number of slots to reserve for the Smalltalk call stack. This limits the number of simultaneous nested method calls.

##objectTableSize

The number of slots to reserve for the object table. This limits the total number of simultaneous "live" objects (including literals within methods).

##heapSize

The number of 2-byte words to reserve for the Smalltalk dynamic heap. Be sure to leave enough free dynamic memory for PalmOS to operate effectively.

Last updated: Dec 16, 2001

Palm Powered is a trademark of Palm Inc. Pocket Smalltalk is trademark of Pocket Smalltalk Group. Copyright (c) 1998 - 2001 Pocket Smalltalk Group.