SMALLTALK-Z80

PRIMER

A Smalltalk Object System for the 48K ZX Spectrum

Cassette edition

How to read this book

Read this book with the Spectrum switched on, not with the kettle cooling downstairs and the cassette still in its box. Smalltalk is best learnt by trying small expressions, changing them, and trying again.

There is only one **TYPE THIS** command in the book: the Sinclair BASIC loading command. It must be entered exactly at the BASIC prompt. The Smalltalk examples are not printed as schoolroom copywork. They are things to try in the Workspace, or methods to install in the Browser when the text tells you to do so.

A few later examples alter object memory in powerful ways. They are marked **ADVANCED**. Keep a clean cassette copy and save your image only when you mean to keep the change. The Spectrum is quite obedient, but it is not a nanny.

Smalltalk-Z80 is a live object system for the 48K ZX Spectrum. You will work with objects, classes, methods, messages, blocks, collections, and an image which can be changed while the machine is running.

Foreword: Objects on the Spectrum

This book is about object-oriented programming on the ZX Spectrum.

That phrase is worth taking slowly. It does not mean that the Spectrum has merely learnt a few new commands. It means that we shall organise programmes in a different fashion. Instead of beginning with a list of numbered instructions, we begin with objects. An object has some private information, and it has messages which it understands. A programme is made by arranging objects so that they ask one another to do work.

You already know the Spectrum as an immediate machine. Type a BASIC command, press ENTER, and it answers. That is one of its great pleasures. Smalltalk keeps that pleasure, but changes what is being typed. In the Workspace you do not mostly command the machine line by line. You send a message to an object and look at the answer.

For example, in BASIC a value and the lines which use it may be separated. The number is in one variable, the rule for changing it is elsewhere, and the rule for printing it may be somewhere else again. In a longer programme the reader must remember how these pieces fit together.

In Smalltalk the rule is more disciplined:

```
put the knowledge beside the thing it belongs to
```

A robot object can know whether it is awake. A point object can know its two screen co-ordinates. A collection can know which objects it contains. A string can know how long it is. The screen service can know how to clear the display. Each of these objects is asked by messages.

This is object-oriented programming. The phrase sounds grand, but the first rule is simple:

```
an object receives a message and answers
```

A number receives `negated`. A string receives `size`. A class receives `new`. A collection receives `add:`. The same message may be understood differently by different objects. That is not a trick or a special case. It is the ordinary method by which Smalltalk programmes are built.

This also explains why Smalltalk is mentioned near the languages used for artificial-intelligence work. Lisp and Prolog are already available for the Spectrum, and they show two famous approaches to symbolic programming. Lisp is especially at home with symbolic structures. Prolog is especially at home with facts, rules, and questions. Smalltalk now follows with another approach: build a world of objects, give each object its proper behaviour, and let the world be inspected and changed while it is running.

Smalltalk is therefore not here as a magic brain. It is here as a very good way of modelling things. A pupil, a chess square, a robot, a switch, a question card, a drawing pen, a rule, or a list can all be objects. Each can carry its own state and answer its own messages. That is useful for games, simulations, educational programmes, tools, and many sorts of experimental work.

The Spectrum is a fine machine for this lesson. It has colour graphics, sound, a keyboard with a character of its own, a television display, cassette storage, and BASIC ready at switch-on. Many readers will have no other computer and need no apology for that. This book treats the Spectrum as the machine on the desk, not as a waiting room for some other machine.

Smalltalk-Z80 adds a second style of computing to it. You will use a Workspace for experiments and a Browser for classes and methods. You will make objects, ask their class, add methods, create new classes, save the changed image, and continue from that image later.

The Browser is important because Smalltalk is not a sealed box. The system can show you its own classes and methods. It can be altered from within. When you add a method, the image has learnt a new message. When you add a class, the image has learnt a new kind of object.

A short BASIC programme may ask, "Which variable must I change?" A Smalltalk programme more often asks, "Which object should receive this message?" This question is the beginning of the subject.

The rest of the book builds from that question. First you will load the cassette and try expressions. Then you will learn the syntax of messages, blocks, variables, and cascades. Then you will make classes of your own. Later chapters show collections, errors, the screen, the beeper, weak references, and the more powerful tools of a live object memory.

The cassette recorder may squeal, the television may shimmer, and the keyboard may insist on its own opinions about punctuation. Never mind. Once

the Workspace is on the screen, the essential rule is clear:

```
choose an object, send a message, inspect the answer, then teach the
image something new
```

Part One — Entering the Smalltalk-Z80 World

1. Loading the cassette

Switch on the Spectrum in the usual way. You should see the Sinclair copyright message and the BASIC prompt.

Put the Smalltalk-Z80 cassette into the recorder. Rewind it to the beginning of the programme. Adjust the volume as you normally do for loading Spectrum software.

TYPE THIS in Sinclair BASIC:

```
LOAD ""
```

Press ENTER. Then start the cassette recorder.

The Spectrum will search the tape, find the programme, and load it. The loading sounds and border stripes are part of the ordinary Spectrum loading process. Do not press keys while the programme is loading. A watched border does not load faster, though every Spectrum owner has tested this theory.

When loading has finished, Smalltalk-Z80 starts and enters the Smalltalk-Z80 environment. There are two cassette forms of this edition. The ordinary cassette loads directly. The title-screen cassette loads the Smalltalk machine, the saved object image, and the title screen, then enters the VM without a separate Ready prompt. Type `LOAD ""` once, start the tape, and let the loader do its work.

If loading fails, rewind and try again. Check the tape volume. Smalltalk-Z80 is no different from other cassette software in this respect: a poor load is usually a tape or volume problem, not a Smalltalk problem.

For the curious reader, the BASIC loader first clears memory, then loads the machine-code part of Smalltalk-Z80 and the saved image. On the title-screen cassette it also loads a Spectrum screen picture. You do not type these commands separately; the BASIC loader performs them after your single `LOAD ""`.

2. What appears after loading

Smalltalk-Z80 does not leave you at the Sinclair BASIC prompt. You enter a Smalltalk environment.

The screen is divided into working areas. The top part is used by the Browser, where you can see classes and methods. The lower part is used for editing or for the Workspace, where you try expressions.

At first, learn only these two words:

```
Browser
Workspace
```

The Browser is for looking at the system and adding methods. The Workspace is for trying Smalltalk expressions at once.

A useful command is:

SYMBOL SHIFT + W

This switches between Browser and Workspace.

Use it now. Press **SYMBOL SHIFT + W** and notice that the lower part of the display changes. Press it again and notice that you return.

3. First success in the Workspace

Go to the Workspace with **SYMBOL SHIFT + W**.

Try this on a blank line:

```
2 + 3
```

Then press:

SYMBOL SHIFT + E

SYMBOL SHIFT + E evaluates the line on which the cursor is standing. The system should print the answer.

The Workspace is not a one-line command prompt. You may continue on the following lines:

Again press **SYMBOL SHIFT + .E**. If you move the cursor back to an earlier line and press **SYMBOL SHIFT + E**, that earlier line is evaluated again. This is very useful. You can type an expression, evaluate it, move back to it, alter it, and evaluate it once more.

This is closer to a laboratory notebook than to a school slate. The old lines remain where you can see them, and the cursor chooses which line is to be tried.

The arithmetic is only the surface. What matters is that you have spoken to objects in the Smalltalk world. The number `2` received the message `+ 3` and answered `5`.

Now try a message that affects the real Spectrum display:

```
Screen clear
```

The screen clears. The Workspace header is drawn again after evaluation, so you can continue working.

The keyboard service is useful when you want a programme to pause until you have seen the screen. Try it as a separate line:

```
Spectrum waitForKey
```

The system waits until you press a key and answers the key code. This is not a stray mathematical example. It is the Spectrum keyboard presented as a Smalltalk service. Later, when you draw or sound the beeper, `waitForKey` will let you stop the action until you are ready.

4. Moving around the Browser

Return to the Browser with **SYMBOL SHIFT + W.**

The left list contains classes. A class describes a kind of object. You will soon create classes yourself, but first you should learn to look around.

Use the following keys in the lists:

Key	Alternative	Meaning
Q	cursor up	Move to previous item
A	cursor down	Move to next item
O	cursor left	Move focus left
P	cursor right	Move focus right
W	—	Move one page up
S	—	Move one page down
C	—	Change instance side / class side
N	—	New class or new method
D	—	Delete selected method

On the Spectrum keyboard, the cursor arrows are obtained with **CAPS SHIFT** and the number keys:

Key chord	Cursor
CAPS SHIFT + 5	left
CAPS SHIFT + 6	down
CAPS SHIFT + 7	up
CAPS SHIFT + 8	right

For deleting characters while editing, use:

CAPS SHIFT + 0

Move through the class list until you find `Object`. Then move to the method list. Look at the method names. Do not try to understand them all yet. It is enough to see that the system is made of named classes and named methods.

5. Accepting an edit

Later you will edit methods and class definitions. The important command is:

SYMBOL SHIFT + Q

In the Browser editor this means: accept this edit.

The system asks whether to save the change. Answer:

```
Y
```

or press ENTER for yes. Use:

```
N
```

for no.

Remember this. Many beginners type a method and then forget to accept it. If a new method does not seem to work, first ask: did I press **SYMBOL SHIFT + Q**, and did I answer **Y**?

Part Two — The Image

6. What an image is

Smalltalk-Z80 is not loaded as an empty language. When it starts, it already contains many objects.

It contains class objects such as:

```
Object
SmallInteger
String
Array
OrderedCollection
Dictionary
Screen
Spectrum
```

It contains methods belonging to those classes. It contains special objects such as `true`, `false`, and `nil`. It contains a table of well-known names. It contains enough of a compiler to compile new methods while the system is running.

This whole saved object world is called an **image**.

Do not confuse this with a screen picture. A Smalltalk image is not a drawing. It is the saved state of the object world.

A BASIC programme saved on tape is usually a list of lines. When you load it, the lines return. A Smalltalk image is more ambitious. It is like saving a whole workshop with the tools still on the benches, the labels still on the drawers, and the half-finished model still clamped in place.

When Smalltalk-Z80 starts, the image already knows many things. `SmallInteger` already knows `+`, `-`, comparisons, and the counting loop `to:do:`. `String` already knows `size` and `at:`. `Object` already knows `class`, `isNil`, and the messages used by the Browser. `Screen` already knows how to clear and draw on the display. `Spectrum` already knows how to beep and wait for a key. `Compiler` already knows how to compile a method. The exact contents of the starting image may grow, so learn to browse it rather than memorise a printed inventory.

When you add a method in the Browser, you are not merely adding a line to a

separate programme. You are teaching the image a new method. If the image is saved, that knowledge can be present the next time the system is loaded.

This is a great difference between Smalltalk and ordinary BASIC programming. In BASIC, the language is fixed and your programme is separate. In Smalltalk, the system itself can be extended while you are using it.

7. A practical image story

Here is the matter in practical form.

Suppose the system does not have a message named `notEmpty` for collections. You can add it. After that, collections understand a new message.

Before you add it, this expression may fail:

```
OrderedCollection new notEmpty
```

After you add the method, it answers `false`.

If you save the image, reload it, and try the same expression again, the method can still be there. You have not merely run a programme. You have altered the object world.

Smalltalk programmers often speak as if the system is alive. They do not mean magic. They mean that the system can be inspected, changed, and continued without returning to an empty beginning every time.

8. Saving and loading the image

Smalltalk-Z80 provides image operations through the object named `Smalltalk`.

To save the current image and leave the system:

```
Smalltalk saveImage
```

To load an image, use:

```
Smalltalk loadImage
```

Use these with care. While learning, keep a clean cassette copy and a notebook of the methods you add. Do not save after a dangerous experiment until you

are sure you want to keep the result.

For ordinary safe work, saving the image is very useful. If you add a class such as `Robot` or a method such as `timesRepeat:`, saving the image lets you continue at the next session.

The saved image is not a second copy of the virtual machine. The cassette already contains the machine-code part. `Smalltalk saveImage` saves the object memory and the high work buffers that belong to the living image, so your classes, methods, globals, and Workspace text can be restored by the same VM.

For advanced experiments with object memory, weak references, and `becomeForward:`, reload a fresh image first and do not save unless you understand exactly what happened.

Part Three — Messages, Names, and Syntax

9. Objects receive messages

The simplest Smalltalk expression has an object followed by a message.

Try this:

```
3 negated
```

The object is:

```
3
```

The message is:

```
negated
```

The answer is:

```
-3
```

Try:

```
'HELLO' size
```

The object is the string `'HELLO'`. The message is `size`. The answer is `5`.

Try:

```
Object new
```

The object `Object` receives the message `new`. The answer is a new object.

In Smalltalk books, the object receiving a message is often called the **receiver**. For now, remember the simple phrase: the receiver is the object before the message.

This is more than a new notation. It is the centre of the language. A Smalltalk programme is not mostly made of commands. It is made of objects asking other

objects to do work.

10. The shape of Smalltalk text

Smalltalk text looks unusual at first because it has very little punctuation. This is not because it is vague. It is because almost everything is a message.

In BASIC you might write a special command such as:

```
PRINT 2 + 3
```

In Smalltalk, + is not a special command. It is a message sent to the number 2:

```
2 + 3
```

Read it as:

```
send the message + 3 to the object 2
```

The answer is another object, the number 5.

Most Smalltalk text is made from a few simple pieces:

```
object message
object message argument
object message1; message2; message3
variable := object message
| temporary variables |
[ delayed code ]
```

Those pieces can be combined, but it is best to understand each one separately before trying to read a long line.

A name in Smalltalk is not a box containing a number in the old BASIC sense. A name refers to an object.

```
| a |
a := 42.
a
```

The name a now refers to the SmallInteger object 42.

Later, the same name may refer to another object:

```
| a |
a := 42.
a := 100.
a
```

Now a refers to 100. This does not change the object 42. It only changes the local name a.

This distinction matters more when the object is not a number. If two variables refer to the same robot, and one of them sends wake to the robot, both variables still refer to the same robot, now awake.

11. Characters, strings, and symbols

A string is written between single quotes:

```
'ZX Spectrum'
```

A string understands size:

```
'ZX Spectrum' size
```

Smalltalk collections count from 1, not from 0.

```
'ABC' at: 1
```

answers the first character.

A character constant is written with a dollar sign before the character:

```
$A
```

The dollar sign is only the way we write a character in Smalltalk source. The character itself is the letter A.

A character can answer its code:

```
$A asciiValue
```

A code can be turned back into a character:

```
Character value: 65
```

A symbol is written with #:

```
#name
```

Symbols are often used as names inside the system. You will meet them when creating classes and when storing things in the global table named `Smalltalk`.

Do not worry if symbols seem unusual. At first, think of a symbol as a name object.

There is one important difference between a string and a symbol. A string is usually text for people or for data. A symbol is usually a name used by the system. For example, the class name `#Point` is a symbol, because it is a name being installed in the image.

12. The vertical bar character | and local variables

The character | is called a **vertical bar**. In Smalltalk it is used to declare temporary local variables.

It is easy to mistake it for the digit `1`, the capital letter `I`, or a decorative line. It is none of those. It is punctuation with a precise meaning.

At the start of a method, or at the start of a longer Workspace expression, Smalltalk may see a pair of vertical bars:

```
| total count average |
```

This means:

```
During this method or expression I shall use the temporary names
total, count, and average.
```

The bars are not part of the variable names. The variable names are only:

```
total
count
average
```

You may put one name between the bars:

```
| x |
```

You may put several names between the bars:

```
| x y oldPosition newPosition |
```

You may also have no temporary variables at all. Then you do not write the bars.

A local variable begins with the value `nil` until you assign something else to it. In the runnable examples in this Primer, temporary names are usually kept to one letter because the present cassette compiler accepts only short temporary names in examples intended to be typed directly.

```
| t |
t
```

The result is `nil`, because `t` has been declared but not yet given another object.

The declaration must come before the statements that use those names:

```
| x y |
x := 10.
y := x + 5.
y
```

Do not write this:

```
x := 10.
| x |
x
```

The compiler and the human reader both need to know the temporary names before the statements begin.

Local means local

A temporary variable belongs only to the method or Workspace expression in which it is declared. It is not a new global word in the system dictionary.

If you evaluate:

```
| s |
s := 10.
s
```

then s is known in that expression. It is not automatically known everywhere else afterwards.

This is one reason Smalltalk programmes stay tidy. Temporary names are used where they are needed, then disappear.

Why declare local variables?

You could write:

```
10 + 20 + 30 + 40
```

but this version shows the two halves of the sum before adding them:

```
| t c |
t := 10 + 20.
c := 30 + 40.
t + c
```

The second version is longer, but the names say what is happening. Good Smalltalk often reads like a short explanation of the objects involved.

13. Assignment and statements

A Smalltalk expression may have several statements separated by full stops:

```
| a b |
a := 3.
b := 4.
a + b
```

There are three statements here:

```
a := 3.
b := 4.
a + b
```

The final expression is the answer shown in the Workspace.

You do not need a full stop after the final statement, although it is often harmless. When you are learning, use full stops between statements and do not worry about putting one after the last line.

The assignment sign is:

```
:=
```

Read it as "becomes".

```
a := 3
```

means: let a become a name for the object 3.

Assignment is not equality. These two lines do very different jobs:

```
x := 10
x = 10
```

The first changes what x refers to. The second asks whether the object referred to by x is equal to 10.

The left side of an assignment must be a variable name. The right side may be any expression.

```
| w h |
w := 8.
h := 6.
w + h
```

Here w + h first evaluates the messages that fetch w and h, then sends + to w with h as the argument, producing 14. In a larger Smalltalk installation you would often use longer names such as width, height, and area; in this cassette edition the typed examples keep temporary names short so that they pass through the resident compiler. .

Assignment does not copy an object. It changes a reference.

```
| a b |
a := 10.
b := a.
a := 20.
b
```

The result is still `10`, because `b` was made to refer to the number `10`. Later changing a does not change `b`.

With changeable objects, the distinction is more interesting. If two names refer to the same object, and you send that object a message which changes it, both names still refer to the same changed object.

14. Three kinds of messages and their order

Smalltalk has three message forms.

Unary messages

A **unary message** has only a name and no argument:

```
3 negated
'HELLO' size
Object new
```

Read `3 negated` as:

```
send negated to 3
```

Unary messages bind tightly. This means:

```
3 class name
```

is read as:

```
(3 class) name
```

First ask `3` for its class. Then ask that class for its name.

Binary messages

A **binary message** uses one or two special characters and has one argument:

```
2 + 3
10 - 7
3 < 4
5 = 5
```

Small integers also understand comparison messages such as <, <=, >, >=, and =. These answer booleans, so they are often used with `ifTrue:`, `ifFalse:`, and loop test blocks.

The object on the left receives the message. `2 + 3` means:

```
send + 3 to 2
```

Binary selectors may look like arithmetic signs, but they are still message names.

Keyword messages

A **keyword message** has one or more words ending with colons:

```
Character value: 65
Screen ink: 2
Spectrum beepLength: 110 pitch: 964
```

The last example has two keyword parts. The full message name is:

```
beepLength:pitch:
```

and it has two arguments.

The selector is not just the first word. The selector of:

```
Screen plotX: 10 y: 20
```

is:

```
plotX:y:
```

That is why keyword messages often read like small English phrases.

Message precedence

Message order is simple:

```
1. unary messages
2. binary messages
3. keyword messages
4. within the same kind, read from left to right
```

Thus:

```
'HELLO' size + 1
```

means:

```
('HELLO' size) + 1
```

and answers 6.

This surprises many people who already know school arithmetic. Smalltalk does not have a separate table where multiplication happens before addition. Binary messages are read from left to right.

```
2 + 3 * 4
```

Smalltalk reads this as:

```
(2 + 3) * 4
```

so the answer is 20, not 14.

To get the usual school arithmetic result, use parentheses:

```
2 + (3 * 4)
```

Now the answer is 14.

If you are unsure, use parentheses. A good programme should be clear to a person as well as to the machine.

15. Cascades with ;

A cascade sends several messages to the same receiver.

Suppose you write:

```
Screen clear.
Screen ink: 2.
Screen paper: 7.
Screen border: 1
```

The receiver is Screen every time. A cascade lets you write the receiver once:

```
Screen
  clear;
  ink: 2;
  paper: 7;
  border: 1
```

The semicolon does not mean "new statement". It means:

```
send the next message to the same receiver as before
```

The receiver of all four messages above is Screen.

Without the cascade, the same idea would be:

```
Screen clear.
Screen ink: 2.
Screen paper: 7.
Screen border: 1
```

Cascades are useful when one object is being told to do several things:

```
Screen
  clear;
  plotX: 0 y: 0;
  drawX: 100 y: 0
```

This means:

```
send clear to Screen
send plotX:y: to Screen
send drawX:y: to Screen
```

Cascade is not chaining

A cascade is not the same as ordinary chaining.

This sends `name` to the result of `3 class`:

```
3 class name
```

This sends both `class` and `yourself` to 3:

```
3 class; yourself
```

The second expression is legal, but it is a different idea. After `3 class`, the cascade goes back and sends `yourself` to the original receiver, the number 3.

Remember:

```
a full stop starts a new statement
an ordinary message continues from the result so far
a semicolon goes back to the original receiver of the cascade
```

The value of a cascade is the value of its last message. Usually we use cascades for their effect, not for their final value.

16. A small syntax map

This table is not a replacement for practice, but it is a useful map when a line looks puzzling.

Syntax	Example	Meaning
integer literal	`42`	the SmallInteger object 42
string literal	`'hello'`	a String object
character literal	`$A`	a Character object
symbol literal	`#name`	a Symbol object, often used as a selector or name
variable	`total`	the object currently referred to by `total`
pseudo-variable	`self`	the receiver of the current method
pseudo-variable	`super`	start method lookup in the superclass
pseudo-variable	`nil`	the object meaning "nothing here"
pseudo-variable	`true`	the true Boolean object
pseudo-variable	`false`	the false Boolean object
temporary declaration	`| x y |`	declare local temporary variables
assignment	`x := 10`	make `x` refer to `10`
return	`^ x`	answer from the current method
unary message	`x class`	send `class` to `x`
binary message	`x + 1`	send `+ 1` to `x`
keyword message	`Screen plotX: 1 y: 2`	send `plotX:y:` to `Screen`
cascade	`Screen clear; plotX: 1 y: 2`	send several messages to the same receiver
block	`[ x + 1 ]`	delayed code
block evaluation	`[ x + 1 ] value`	run the delayed code
parentheses	`(2 + 3) * 4`	force grouping

Smalltalk-Z80 does not at present accept the usual Smalltalk comment form written between double quotes. Do not put such comments inside code you type into the Workspace or the Browser. Use plain prose in your notebook or in the Primer margin instead.

Smalltalk has less syntax than many languages. The price is that every mark matters. The reward is that, after a while, most lines can be read as messages flowing between objects.

Part Four — Decisions, Blocks, and Repetition

17. true, false, and nil

Smalltalk has two Boolean objects:

```
true
false
```

They are not numbers. They are objects.

Try:

```
true not
```

and:

```
false not
```

There is also:

```
nil
```

`nil` means no useful object here. It is an object too, and it understands messages such as `isNil`.

Try:

```
nil isNil
```

and:

```
Object new isNil
```

Here is a beautiful Smalltalk point. There is no special BASIC-like `IS NIL` command. The object receives the message `isNil` and answers.

18. Blocks are delayed work

A block is a piece of programme between square brackets.

```
[ 1 + 2 ]
```

A block does not run merely because it is written. It is an object. To run it, send it `value`.

```
[ 1 + 2 ] value
```

The answer is 3.

A block can have an argument:

```
[ :x | x + 1 ] value: 5
```

The answer is 6.

A block can have two arguments:

```
[ :x :y | x + y ] value: 3 value: 4
```

The answer is 7.

Look again at the vertical bar in a block with arguments:

```
[ :x | x + 1 ]
```

Here the bar separates the block arguments from the block body. It is related to the vertical bars used for temporary variables, but the shape is different. In a temporary declaration the bars come as a pair:

```
| x y |
```

In a block argument list, the bar says "the arguments are finished; the work begins now".

Blocks are one of the most important parts of Smalltalk. They let you pass a piece of work as an object. Decisions, repetitions, and collection operations all use blocks.

19. Decisions are messages

In BASIC, you write:

```
IF A=3 THEN PRINT "YES"
```

In Smalltalk, a Boolean receives a message:

```
(3 = 3) ifTrue: [ 10 ]
```

The answer is `10`.

Try:

```
(3 = 4) ifTrue: [ 10 ]
```

The answer is `nil`, because the block was not evaluated.

A two-way decision is written:

```
(3 < 4) ifTrue: [ 111 ] ifFalse: [ 222 ]
```

The object `true` knows to evaluate the first block. The object `false` knows to evaluate the second block. The decision is made by the Boolean object receiving the message.

This is a good example of Smalltalk's consistency. Control structures are not separate magic commands. They are built from objects, messages, and blocks.

20. How ifTrue: and ifFalse: can be methods

It is tempting to think that `ifTrue:` and `ifFalse:` must be special commands, as in BASIC:

```
IF X > 10 THEN PRINT "BIG"
```

Smalltalk does something more radical. The condition produces an object, either `true` or `false`. Then we send that object a message.

```
3 > 2 ifTrue: [ 99 ]
```

Read it slowly:

```
send > 2 to 3
the answer is true
send ifTrue: to true
pass the block [ 99 ] as the argument
```

The important part is this: `true` and `false` are objects. They have behaviour.

Conceptually, the methods look like this.

In class `True`:

```
ifTrue: trueBlock
    ^trueBlock value
```

In class `False`:

```
ifTrue: trueBlock
    ^nil
```

So when the receiver is `true`, the block is evaluated. When the receiver is `false`, the block is ignored.

For the two-way form, the methods are just as simple.

In class `True`:

```
ifTrue: trueBlock ifFalse: falseBlock
    ^trueBlock value
```

In class `False`:

```
ifTrue: trueBlock ifFalse: falseBlock
    ^falseBlock value
```

So this:

```
3 > 2 ifTrue: [ 111 ] ifFalse: [ 222 ]
```

is not a magic statement. It is a message sent to a Boolean object.

Why blocks are necessary

Suppose Smalltalk did not use blocks. Suppose we tried to write this as ordinary immediate work:

```
3 > 2 ifTrue: 99
```

That example is harmless because `99` is only a number. But with real work, the branch would be evaluated too soon. The block prevents that.

```
[ Screen clear ]
```

means:

```
Here is something you may do later, but do not do it yet.
```

Then `true` or `false` chooses whether to send `value` to that block.

The machine may optimise, but the meaning stays Smalltalk

A personal computer must not waste time or memory. The implementation may recognise common conditional forms and compile them efficiently. That does not change the meaning you should learn.

The Smalltalk meaning is:

```
conditionals are messages to Boolean objects, and the branches are
blocks
```

That is why Smalltalk conditionals fit naturally with the rest of the language instead of being a separate command language bolted on the side.

21. Repetition with whileTrue:

Smalltalk-Z80 has `whileTrue:` for repetition.

Try this:

```
| i t |
i := 1.
t := 0.
[i <= 5] whileTrue: [
    t := t + i.
    i := i + 1].
t
```

The answer is 15.

The first block is the test:

```
[i <= 5]
```

The second block is the body:

```
[
    t := t + i.
    i := i + 1]
```

Read the whole expression as:

```
while the test block answers true, evaluate the body block
```

This gives structured repetition without BASIC line numbers and without `GOTO`.

22. Understanding whileTrue: through recursion

This section is not needed for using loops, but it explains an important idea.

Imagine that `whileTrue:` were written in Smalltalk like this:

```
whileTrue: bodyBlock
    self value ifTrue: [
        bodyBlock value.
        self whileTrue: bodyBlock].
    ^nil
```

Here `self` is the test block. First it is evaluated. If it answers true, the body block is evaluated. Then the test block is sent `whileTrue:` again.

That is recursion: a method using itself to continue the work.

In a large system, recursion is a very general idea. It is good for problems that naturally contain smaller versions of themselves. For example, a directory may contain smaller directories, or a rule may ask another rule.

On the Spectrum, deep recursion can use too much stack. Therefore, for ordinary counting loops, use `whileTrue:` or a method such as `timesRepeat:`. Smalltalk-Z80 performs `whileTrue:` specially enough that it is the right way to write ordinary loops.

The important lesson is this: in Smalltalk, even looping can be understood as a message sent to an object, with blocks deciding what is tested and what is repeated.

23. Repetition with whileFalse:

The opposite loop is `whileFalse:`. It repeats while the test block answers `false`.

For example:

```
| i |
i := 1.
[i > 5] whileFalse: [ i := i + 1 ].
i
```

The test `i > 5` is false at first, so the body runs. When `i` becomes greater than `5`, the loop stops.

Most examples in this book use `whileTrue:` because it reads naturally for counting upwards, but `whileFalse:` is part of the same family.

24. The built-in to:do: loop

Smalltalk-Z80 includes a useful counting message for integers:

```
1 to: 5 do: [ :i | Spectrum beep ]
```

Read this as: start with `1`, count up to `5`, and for each value put the value into `i` and evaluate the block.

The block argument `:i` is the current number. Try a safer example first:

```
| t |
t := 0.
1 to: 5 do: [ :i | t := t + i ].
t
```

The answer is `15`, because the block has added `1`, `2`, `3`, `4`, and `5`.

This is close in purpose to BASIC's `FOR` loop, but it is still Smalltalk. The number `1` receives the message `to:do:`. The second argument is a block. The number is not a line-number command; it is an object being asked to count.

The method is deliberately simple. If you browse `SmallInteger` and look at `to:do:`, you will see that it is written using `whileTrue:`. This is a useful lesson: more convenient messages can be built from smaller messages.

25. Boolean shortcuts: not, and:, and or:

You have already used `ifTrue:` and `ifFalse:`. Booleans also understand three small but important messages:

```
true not
false not
```

`not` answers the opposite Boolean.

The messages `and:` and `or:` are more interesting because they use blocks. Try:

```
true and: [ 3 < 4 ]
```

The block is evaluated because the receiver is `true`.

Now try:

```
false and: [ Object new dance ]
```

The answer is `false`. The block is not evaluated, so the unknown message dance is never sent. This is called short-circuit evaluation, although no wires are actually being cut.

Similarly:

```
true or: [ Object new dance ]
```

answers `true` without evaluating the block, because `true or:` already knows enough.

This is another example of Smalltalk's regularity. There is no special `AND` command hidden in the language. `true` and `false` simply have different methods. For the same reason, the less common spelling `ifFalse:ifTrue:` also exists; it is useful when the false case reads more naturally first.

26. Adding timesRepeat: yourself

Smalltalk-Z80 already has `to:do:`, but it does not need to carry every convenience message in the starting image. Let us add a pleasant one ourselves:

```
5 timesRepeat: [ Spectrum beep ]
```

This is a fine early example because it teaches that the system can be extended with a useful new word.

Go to the Browser. Select the class:

```
SmallInteger
```

Create a new instance-side method and type:

```
timesRepeat: b
    1 to: self do: [ :i | b value ].
    ^self
```

Accept it with **SYMBOL SHIFT + Q**, then answer **Y**.

Now try in the Workspace:

```
3 timesRepeat: [ Spectrum beep ]
```

The Spectrum should beep three times.

This is a very important moment. Before this, numbers did not understand `timesRepeat:`. Now they do. You have taught the image a new message.

It is also a good example of object-oriented style. We do not write a special

looping command. We ask the number 3 to repeat a block three times.

The implementation above uses the built-in `to:do:` loop. If you wanted to write it more directly, you could use the same `whileTrue:` idea you saw earlier:

```
timesRepeat: b
    | i |
    i := 1.
    [i <= self] whileTrue: [
        b value.
        i := i + 1].
    ^self
```

Do not install both versions at the same time unless you mean to replace the first with the second. They are shown together here to teach the idea.

Try:

```
5 timesRepeat: [ Screen clear. Spectrum waitForKey ]
```

This clears the screen and waits for a key five times.

Part Five — Classes and Methods

27. What a class is

A class describes a kind of object.

You may think of a class as a plan for making objects and as a book of methods those objects understand.

`SmallInteger` is the class of small integer numbers. `String` is the class of strings. `OrderedCollection` is the class of growable ordered collections. `Screen` and `Spectrum` are classes whose class-side methods give useful services of the machine.

Ask objects about their class:

```
3 class
'ABC' class
Object new class
```

The answer tells you what kind of object you have.

A class is also an object. That is why you can send messages to a class:

```
Object new
Array new: 3
Character value: 65
```

The message `new` asks a class to make an instance.

28. Reading methods in the Browser

In the Browser, select `Object`. Look at the method list.

Find:

```
isNil
```

The method is:

```
isNil
    ^false
```

Now select `UndefinedObject` and find `isNil`.

It is:

```
isNil
    ^true
```

The object `nil` is the special instance of `UndefinedObject`. Therefore:

```
nil isNil
```

answers `true`, but an ordinary object answers `false`.

This is not a special test hidden in the language. It is ordinary message sending. Different objects answer the same message in different ways.

That is one of the simplest examples of polymorphism. You do not need to remember the word yet. Remember the idea: the same message can be understood differently by different kinds of object.

29. Writing a first useful method

If your image does not already have `notEmpty`, add it to `Collection`.

Select class:

```
Collection
```

Add this instance-side method:

```
notEmpty
    ^self isEmpty not
```

Accept it.

Try:

```
OrderedCollection new notEmpty
```

The answer is `false`.

Now try:

```
| c |
c := OrderedCollection new.
c add: 5.
c notEmpty
```

The answer is `true`.

Before this method was added, the system might not know the message `notEmpty`. After accepting the method, the image has learned it.

30. Creating Robot

Now make a new class. We shall use a very small robot object, because it connects better with simulations and AI-style examples than another household switch.

In the Browser class list, press **N** for a new class. Type:

```
Object subclass: #Robot
  instVars: 'awake' classInstVars: ''
```

Accept it.

The class definition uses the compact Browser form `subclass:instVars:classInstVars:`. The image also contains the longer spelling `subclass:instanceVariableNames:` and related forms:

```
variableSubclass:instVars:classInstVars:
variableSubclass:instanceVariableNames:
variableByteSubclass:instVars:classInstVars:
variableByteSubclass:instanceVariableNames:
weakSubclass:instVars:classInstVars:
weakSubclass:instanceVariableNames:
```

Those forms are useful to the system and to advanced experiments; this book uses the clearest Browser form for ordinary class creation.

A robot object needs to know whether it is awake or asleep. The instance variable `awake` will remember that.

Select class `Robot` and add this instance-side method:

```
initialize
    awake := false.
    ^self
```

When `Robot new` makes a new robot, `initialize` sets it to asleep.

Add:

```
wake
    awake := true.
    ^self
```

Add:

```
sleep
    awake := false.
    ^self
```

Add:

```
isAwake
    ^awake
```

Try:

```
Robot new isAwake
```

The answer is `false`.

Try:

```
(Robot new wake) isAwake
```

The answer is `true`.

The robot object has its own memory. It remembers `awake`. The messages that change and inspect that state belong to the robot.

This is object-oriented programming in miniature: the state and the operations that make sense for that state live together.

31. self

Inside a method, the word `self` means the object that received the message.

In:

```
wake
    awake := true.
    ^self
```

`self` is the robot that received `wake`.

Returning `self` is common in methods that change an object. It lets you continue sending messages to the same object.

```
Robot new wake isAwake
```

This works because `wake` answers the robot itself.

32. Class-side methods

A class is an object too. It can have methods of its own.

In the Browser, select `Robot`, then press `C` to switch to the class side.

Add this class-side method:

```
awakeRobot
    ^self new wake
```

Try:

```
Robot awakeRobot isAwake
```

The answer is `true`.

Here `self` is not an ordinary robot. It is the class object `Robot`, because the method is on the class side.

Class-side methods are often used as convenient constructors.

33. Class instance variables

Smalltalk-Z80 supports class instance variables.

An ordinary instance variable belongs to an ordinary object. In `Robot`, the variable `awake` belongs to each robot.

A class instance variable belongs to the class object. There is one value remembered by the class object itself.

Edit the class definition of `Robot` so that it reads:

```
Object subclass: #Robot
  instVars: 'awake' classInstVars: 'Default'
```

Accept the class definition.

On the class side of `Robot`, add:

```
initialize
    Default := self new.
    ^Default
```

Add:

```
default
    ^Default
```

Now try:

```
Robot initialize
Robot default isAwake
```

The answer is `false`.

Try:

```
Robot default wake
Robot default isAwake
```

The answer is `true`.

The class object `Robot` has remembered its default robot.

Smalltalk-Z80 teaches class-side state through class instance variables.

Traditional Smalltalk class variables are not part of this image. Their full behaviour needs careful support, and a half-imitation would teach the wrong lesson. The central rule remains: classes are objects and can have their own state. The messages `classInstanceVariableNames:` and `classInstanceVariables` are the system-level protocol behind this feature; the Browser form is the friendly way to use it.

34. How new, initialize, and basicNew fit together

When you send `new` to a class, two things normally happen.

First, the class makes a fresh blank instance. Second, the new object receives `initialize`.

This is why the `Robot` method:

```
initialize
    awake := false.
    ^self
```

runs when you try:

```
Robot new isAwake
```

The ordinary rule is:

```
Class new = make the object, then initialise it
```

Smalltalk-Z80 also has `basicNew`. It makes the blank object without sending `initialize`. It is meant for system code and for rare advanced work. Beginners should use `new`. `OrderedCollection` also has `initialize: capacity`, used by its class-side `new:` method when a particular starting capacity is wanted.

There is also `new:` for variable-sized objects such as arrays:

```
Array new: 3
```

That asks the class for an array with room for indexed elements.

The distinction matters. If a class relies on `initialize`, then `basicNew` gives you an unfinished object. That can be useful to the system, but it is poor ordinary style.

35. Looking at classes from Smalltalk

A class is not merely a name in the Browser. It is an object that can answer questions.

Try:

```
SmallInteger superclass
```

This asks where `SmallInteger` inherits from.

Try:

```
SmallInteger definition
```

This asks the class to describe its own definition. The exact text may be compact, because the image keeps memory for useful objects rather than for ornamental printing.

Tools also need to ask about methods. The Browser uses messages of this kind:

```
Object selectorAt: 1
```

and:

```
(Object methodAt: 1) sourceString
```

The first asks for a selector from the class's method table. The second asks for the reconstructed source of a compiled method. `methodCount` tells a tool how far such a table extends, but the printed number is deliberately not listed in this book. Do not memorise positions in a method table. They are for tools, and may change as the image changes.

`identityHash` is another useful system message:

```
Object new identityHash
```

It answers a small identity number for an object. It is not the same thing as equality. Two objects may be equal according to =, but identity is about the particular object itself.

These messages are included so that the system can inspect and manage itself. The `Compiler` also has `compileMethod:forClass:`, which is what the Browser

uses when you accept a method. Ordinarily you let the Browser call it for you. Use the Browser for ordinary study, and these messages when you want to understand how the Browser can do its work. The lower-level metaclass messages, such as `basicSubclassOf:named:ivars:`, `subclassOf:named:ivars:format:`, and `basicSubclassOf:named:ivars:format:`, are construction machinery, not beginner vocabulary.

Here is the same idea as a small programme. It creates a method without using the Browser editor:

```
Compiler new compileMethod: 'fortyTwo ^42' forClass: Object
```

Now try the new message:

```
Object new fortyTwo
```

The answer should be 42. The first line taught `Object` a method named `fortyTwo`; the second line sent that new message to a fresh object. This is powerful, so use it carefully. The Browser is still the comfortable way to add ordinary methods, but this shows that method creation is itself available from Smalltalk.

Part Six — Inheritance

36. What inheritance means

Inheritance is one of the chief ideas of object-oriented programming.

Suppose you learn a general rule at school: all pupils must bring their exercise books. Later, a science class has extra rules: bring the science notebook and safety spectacles. The science class does not need to repeat every general school rule. It has the general rules plus its own special rules.

A subclass is similar. It is a more special kind of object. It inherits the methods of its superclass and adds or changes what is special.

`Object` is the most general class. Many other classes inherit from it. If an object understands `class`, `isNil`, or `yourself`, it is often because those methods come from `Object`.

A subclass does not copy all the methods into itself like copying lines from one BASIC programme to another. Instead, when a message is sent, Smalltalk looks for a matching method. If the method is not found in the object's class, the search continues in the superclass. If it is not found there, the search continues upward again.

This is important. Inheritance is not only a way to save typing. It is a way to organise knowledge.

Common behaviour belongs high in the family. Special behaviour belongs lower down.

A `Collection` can know general things about collections. A `SequenceableCollection` can know things about collections with an order. A `String` can then inherit sequence behaviour and add character-string behaviour.

37. Point, ColouredPoint, and the @ message

A point is a pair of coordinates. On the Spectrum screen, a point can describe a place where something is plotted or drawn.

Make a class `Point`:

```
Object subclass: #Point
  instVars: 'x y' classInstVars: ''
```

The instance variables `x` and `y` are the two pieces of private state inside each point.

Add instance-side methods:

```
x
    ^x


y
    ^y
```

These are reader methods. They answer the coordinates.

Now add writer methods:

```
x: v
    x := v.
    ^self


y: v
    y := v.
    ^self
```

These methods change one coordinate and answer the point itself. Answering `self` is useful because it lets a later expression continue working with the same point.

On the class side, add a convenient creation method:

```
x: a y: b
    | p |
    p := self new.
    p x: a.
    p y: b.
    ^p
```

Try:

```
(Point x: 5 y: 8) x
```

```
(Point x: 5 y: 8) y
```

Now add a combined instance-side method. It will be useful for the @ message:

```
x: a y: b
    x := a.
    y := b.
    ^self
```

Smalltalk systems commonly provide a short binary message for making points:

```
10 @ 20
```

Read it as:

```
send the message @ 20 to the number 10
```

The result should be a `Point` whose `x` coordinate is `10` and whose y coordinate is `20`.

This is not special compiler syntax. The `@` is a binary message, just like `+` or `*`. If the image does not yet provide `@`, you can add it yourself.

Add this instance-side method to `SmallInteger`:

```
@ v
    ^Point new x: self y: v
```

Now try:

```
| p |
p := 3 @ 4.
p x
```

and:

```
| p |
p := 3 @ 4.
p y
```

The first answer should be 3, and the second answer should be 4.

The @ message is small, but it teaches several important Smalltalk lessons at once.

First, punctuation-looking things can be messages:

```
3 + 4
3 @ 4
```

Second, you can extend the language by adding methods. You do not need to change the compiler to make @ meaningful. The compiler already knows how to send binary messages. You only teach the receiving object what to do when it receives one.

Third, programmes become nicer to read:

```
| s f |
s := 10 @ 20.
f := 100 @ 20.
Screen clear.
Screen plotX: (s x) y: (s y).
Screen drawX: (f x) y: (f y)
```

If you like cascades, the final three screen messages can also be written as:

```
| s f |
s := 10 @ 20.
f := 100 @ 20.
Screen
  clear;
  plotX: (s x) y: (s y);
  drawX: (f x) y: (f y)
```

The point is not only shorter text. The point is that `start` and `finish` are now objects with meaning, not merely two loose numbers each.

Now make a subclass:

```
Point subclass: #ColouredPoint
  instVars: 'colour' classInstVars: ''
```

Add:

```
colour
    ^colour
```

```
colour: v
    colour := v.
    ^self
```

ColouredPoint inherits x, y, x:, y:, and x:y: from Point. It adds only what is new: colour.

On the class side of ColouredPoint, add:

```
x: a y: b colour: c
    | p |
    p := self new.
    p x: a.
    p y: b.
    p colour: c.
    ^p
```

Try:

```
(ColouredPoint x: 5 y: 8 colour: 2) x
```

The answer is 5.

Try:

```
(ColouredPoint x: 5 y: 8 colour: 2) colour
```

The answer is 2.

The subclass did not have to repeat all the point behaviour. It inherited the common part and added its own part.

38. super

Sometimes a subclass wants to do its own work, but also use the inherited work.

For this, Smalltalk has super.

self means: send the message to the present receiver and begin looking in the receiver's own class.

super means: send the message to the same receiver, but begin looking in the superclass of the method currently running.

That sounds technical, so use this story.

A subclass says: "I shall do my special part, but first let the older rule do its part."

For example, add this method to `Point`:

```
initialize
    x := 0.
    y := 0.
    ^self
```

Now add this method to `ColouredPoint`:

```
initialize
    super initialize.
    colour := 0.
    ^self
```

When a new `ColouredPoint` is made, the inherited point initialization sets `x` and `y`. Then the subclass sets `colour`.

This is better than repeating the `x` and `y` assignments in the subclass. If the superclass initialization later changes, the subclass can still use it.

`super` is not a different object. It is a different starting point for looking up the method.

Use `super` when you want the inherited behaviour plus something extra.

Part Seven — Collections

39. Arrays and ordered collections

A collection holds other objects.

A literal array is written:

```
#(1 2 3)
```

Try:

```
#(1 2 3) size
```

The answer is 3.

A literal array is fixed. It is useful for known objects written directly in the programme.

For a collection that can grow, use `OrderedCollection`:

```
| c |
c := OrderedCollection new.
c add: 10.
c add: 20.
c size
```

The answer is 2.

Read an item:

```
| c |
c := OrderedCollection new.
c add: 10.
c add: 20.
c at: 1
```

The answer is 10.

40. do:, collect:, and select:

A collection can evaluate a block for each element:

```
#(1 2 3) do: [ :e | e ]
```

This is not visually exciting in the Workspace, but it is important. The block is evaluated for each object.

`collect:` builds a new collection by transforming each item:

```
#(1 2 3) collect: [ :e | e + 10 ]
```

`select:` builds a new collection from the items for which the block answers true:

```
#(1 2 3) select: [ :e | e > 1 ]
```

These messages are much more Smalltalk-like than writing many index loops by hand. The collection does the travelling; your block describes what to do.

41. Useful methods to add: first and last

Smalltalk-Z80 is small, so some convenient methods are good exercises.

Add to `SequenceableCollection`:

```
first
    ^self at: 1
```

Add:

```
last
    ^self at: self size
```

Try:

```
'ABC' first
'ABC' last
#(10 20 30) first
#(10 20 30) last
```

You added the methods once to SequenceableCollection, and they work for strings and arrays because both are sequenceable collections. This is inheritance paying you back.

42. species and bulk collection messages

The messages collect: and select: should usually answer a collection of the same general kind as the receiver. Smalltalk uses the message species for that.

Try:

```
#(1 2 3) species
```

For ordinary collections, species usually answers the receiver's class. A dictionary is different: when it collects or selects, it should answer a dictionary, not an ordered collection of loose values. That is why Dictionary has its own species method.

Some collection messages are chiefly for methods and tools rather than for first experiments. Their selectors are from:to:put:, replaceFrom:to:with:startingAt:, and basicIndexOf:startingAt:. An arrayed collection can fill a range:

```
| a |
a := Array new: 3.
a from: 1 to: 3 put: 7.
a at: 2
```

It can copy a range from another collection:

```
| a b |
a := Array new: 3.
a from: 1 to: 3 put: 9.
b := Array new: 3.
b replaceFrom: 1 to: 3 with: a startingAt: 1.
b at: 2
```

It can search from a starting position:

```
#(10 20 30) basicIndexOf: 20 startingAt: 1
```

These messages are not as friendly as add:, first, or collect:. They are the

useful workbench tools underneath higher-level collection behaviour.

43. Dictionary

A dictionary stores values under keys. If an array is like a row of boxes numbered 1, 2, 3, then a dictionary is like a set of labelled drawers.

The global object named `Smalltalk` is a system dictionary. It maps names to important objects. For example, the name `#Object` leads to the class object `Object`.

Try:

```
Smalltalk at: #Object
```

You should get the class `Object`.

You can put your own global name into `Smalltalk`:

```
Smalltalk at: #MyNumber put: 123
```

Then:

```
MyNumber
```

answers `123`.

Do not use many globals in ordinary programmes. Objects with instance variables are usually better. But globals are useful for experiments and for naming important objects in the image.

Part Eight — Errors and Recovery

44. Compile errors

When you accept a method, Smalltalk-Z80 compiles it. If the compiler cannot understand the text, it prints a compile error.

The system is small, so error reports are compact. If a compile error appears, check these things first:

1. Did you type the method header correctly? 2. Did each keyword part have its colon? 3. Did you close every string quote? 4. Did you close every block bracket]? 5. Did you declare temporary variables between vertical bars? 6. Did you accidentally type `Class >> methodName` into the Browser editor? Do not do that. The Browser already knows the class.

When in doubt, make the method shorter. First compile a small method that works. Then add one line at a time.

The compiler reports are deliberately compact and numeric. That saves memory which would otherwise be spent on long messages. Use the number as a signpost, but begin by checking the ordinary mistakes above.

45. Message not understood

If you send a message an object does not understand, Smalltalk-Z80 reports a runtime error. Ordinary missing messages return safely to the Workspace with a runtime error report instead of damaging the caller stack.

For example:

```
Object new dance
```

fails, because a plain new object has no dance method.

This is not mysterious. The object received a message, and the system could not find a matching method in the object's class or its superclasses.

Ask:

1. Did I spell the message correctly? 2. Did I add the method to the right class? 3. Did I accept the method with **SYMBOL SHIFT + Q** and **Y**? 4. Am I sending the message to the right object?

Smalltalk errors are often useful. They tell you where your idea and the system's vocabulary do not yet match.

46. The DNU mechanism

Smalltalk programmers often shorten **does not understand** to **DNU**. The full message is:

```
doesNotUnderstand:
```

The selector is long, but the idea is simple. When normal method lookup fails, the system does not immediately give up. It sends the receiver another message, `doesNotUnderstand:`, with a `Message` object as the argument.

That `Message` object contains the selector that was attempted and the arguments that were supplied. It can answer:

```
selector
arguments
```

The ordinary `Object` version of `doesNotUnderstand:` reports an error. That is the right default. Otherwise misspellings would be silently ignored, which is a splendid way to spend an afternoon finding nothing.

But because DNU is itself a message, you can experiment with it. Make sure you are using a fresh experimental image. In class `Robot`, add:

```
doesNotUnderstand: aMessage
    ^aMessage selector
```

Now try:

```
Robot new dance
```

Instead of an error, the robot answers the symbol #dance. It did not learn to dance. It merely reported which selector it failed to understand.

Now replace the method with:

```
doesNotUnderstand: aMessage
    ^aMessage arguments size
```

Try a message with arguments:

```
Robot new turn: 3 steps: 4
```

The answer is 2, because the missing message carried two arguments.

DNU is powerful because it lets an object decide what to do with messages that are not in its ordinary method dictionary. Larger Smalltalk systems use this idea for proxies, forwarding objects, remote objects, and clever tools. Use it carefully. For beginner programmes, a DNU usually means a spelling error or a method installed on the wrong class.

47. Errors, primitive failures, and protected blocks

`Object` also has the message:

```
error:
```

It signals an error with a message. For example:

```
[ Object new error: 'bad' ] on: Error do: [ :e | e messageText ]
```

The block after `on:do:` handles the error and receives the exception object as `e`. The message `messageText` asks the exception what text it carries. The setter `messageText:` and the class message `signal:` are the pieces used when an exception object is created and raised.

A primitive is a low-level operation supplied by the virtual machine. Many fast or hardware-near messages are primitives: integer arithmetic, object allocation, screen drawing, peeking memory, and so on. If a primitive cannot complete, the method usually sends:

```
primitiveFailed
```

That in turn reports an error. This keeps the normal Smalltalk rule: even low-level failure becomes an object-level error where possible.

There is also `ensure:`. It is used when some clean-up must happen after a block runs:

```
| x |
x := 1.
[ x := 2 ] ensure: [ x := 3 ].
x
```

The answer is 3, because the clean-up block is evaluated after the main block.

`ifCurtailed:` is a still lower-level companion used when a computation is abandoned before its ordinary end. It is mainly for system code and careful advanced experiments. The important beginner lesson is simpler: Smalltalk errors are objects, and blocks can be used to protect work.

Part Nine — The Screen, Sound, and Spectrum Services

48. Screen

The class Screen provides useful class-side messages.

Examples:

```
Screen clear


Screen ink: 2
Screen paper: 7
Screen border: 1


Screen plotX: 10 y: 10
Screen drawX: 20 y: 0
Screen circleX: 100 y: 80 radius: 20
```

The colour and drawing messages are Spectrum operations presented as Smalltalk messages. `circleX:y:radius:` draws a circle, and the colour messages use the lower-level `attribute:value:` service. In this version, line drawing uses the proper Spectrum ROM line-drawing service, so simple combinations such as clearing the screen, plotting a point, and drawing a line are normal examples to try.

This is not a separate kind of programming. It is still object-message-answer.

49. Sound

For an ordinary audible beep, use the Spectrum service:

```
Spectrum beep
```

This gives a clear, easily heard tone and answers `Spectrum`, so it is safe for simple demonstrations.

The lower-level sound message is also available on Spectrum:

```
Spectrum beepLength: 110 pitch: 964
```

These are not musical note numbers. They are the lower-level ROM-style duration and pitch values used by the Spectrum beeper routine; Spectrum beep uses deliberately audible values. Try changes only cautiously, because a very long duration may make you wait for the sound to finish.

50. Spectrum

The class Spectrum gives access to useful services of the machine.

Safe examples:

```
Spectrum beep
Spectrum waitForKey
```

waitForKey waits for a key using the Smalltalk-Z80 keyboard scanner and answers the decoded key code as a small integer.

More powerful examples:

```
Spectrum peek: 100
Spectrum poke: 100 value: 85
```

peek: and poke:value: are powerful, just as PEEK and POKE are powerful in BASIC. A wrong address can disturb the system. Use them with care.

Port input and output are also available:

```
Spectrum in: port
Spectrum out: port value: value
```

These are for programmers who already understand Spectrum hardware. They are not needed for the early lessons.

Part Ten — Garbage Collection and References

51. References: how objects are held

When a variable contains an object, it is better to say that the variable refers to the object.

For example:

```
| r |
r := Robot new.
r wake.
r isAwake
```

The temporary variable r refers to a robot object. The object itself is somewhere in the Smalltalk object memory.

Several variables may refer to the same object:

```
| a b |
a := Robot new.
b := a.
a wake.
b isAwake
```

The answer is true, because a and b refer to the same robot.

This is different from copying a number into two BASIC variables. With objects that can change, references matter.

52. Why garbage collection is needed

When you write:

```
OrderedCollection new
```

an object is made in memory. When you make many objects, memory fills.

In BASIC you may be used to clearing variables by starting again. In Smalltalk, many objects are made and discarded while the image is running. The system must find objects that are no longer reachable and reclaim their

memory.

This is called garbage collection.

An object is still alive if it can be reached from the important parts of the system: global names, active computations, classes, methods, and other live objects. An object that cannot be reached is rubbish. The garbage collector may reclaim its space.

You can ask for garbage collection:

```
Smalltalk garbageCollect
```

Most of the time you need not do this yourself. The system performs collection when allocation needs it.

When garbage collection is active, Smalltalk-Z80 indicates this on the display. Watch the bottom right corner of the screen. A blinking block appears there while garbage collection is running. This is a small but useful sign that the system is working, not frozen.

This version uses mark-and-sweep collection. It marks the objects still reachable, sweeps the memory for unused objects, and rebuilds the free list. It does not perform compacting garbage collection. If an allocation cannot be satisfied, the system tries mark-and-sweep and then retries the allocation. If there is still no suitable free block, it reports a runtime failure rather than moving objects around.

If you see the block blinking, wait. Do not hammer the keyboard.

53. Strong and weak references

Most references are strong references. If a live object strongly refers to another object, that other object is kept alive.

For example, if a live collection contains a robot, the robot remains live:

```
| c |
c := OrderedCollection new.
c add: Robot new.
Smalltalk garbageCollect.
c size
```

The collection still contains the robot because the collection strongly refers to it.

A weak reference is different. It means:

```
I should like to remember this object if it is still alive anyway,
but do not keep it alive merely because I remember it.
```

That sounds peculiar until one meets the problem it solves.

A real example: a cache

Imagine a drawing programme. It may prepare small drawing objects: shapes, pen patterns, character pictures, or computed screen data. Some of these take work to build, so the programme keeps a cache:

```
If I need this shape again, I can reuse the old one instead of
building it again.
```

With strong references, the cache itself keeps every cached object alive. The cache grows. Memory fills. Objects that nobody truly needs any more still cannot be reclaimed, because the cache is holding them by the coat-tails.

With weak references, the cache says:

```
Keep this object only if some real part of the programme is still
using it.
If nobody else needs it, the garbage collector may take it.
```

This is the heart of weak references. They let a programme remember without owning.

Another example: observers

Suppose a model object represents the state of a little robot on the screen. Several tools may want to be told when the robot changes: a drawing tool, a score display, or a log of moves.

The model may keep a list of interested tools. If that list uses strong references, an old tool might remain alive merely because the model still has it in the list. The user may think the tool is finished, but the object world is still keeping it.

With weak references, the list does not prevent the old tool from disappearing. If the tool is still genuinely alive, it can receive updates. If nothing else refers to it, it may be reclaimed.

This is a common real reason for weak references in object systems: observer lists, dependency lists, caches, registries, and other "remember this if it still exists" structures.

Why weak references matter here

A Smalltalk image is meant to live and grow. It contains classes, methods, strings, symbols, compiled methods, live objects, and the user's own work. Accidental retention is costly in such a system. A weak reference gives the programmer a careful way to say:

```
This object is useful if it is still around,
but it is not important enough to keep alive by itself.
```

Weak does not mean a wild pointer

A weak reference is not a machine-code dangling pointer. You do not get a random address to old memory. A proper weak reference is managed by the object memory and the garbage collector.

After collection, the weak reference may answer `nil`, or its slot may be cleared, depending on the exact class used.

The programmer must therefore check it conceptually:

```
cached isNil
```

If the weak reference has gone empty, build the object again or ignore the entry. That is the bargain: memory is not held unnecessarily, but the programme must be prepared for the remembered object to have vanished.

54. A weak box experiment

ADVANCED — use a fresh image. Do not save after this experiment unless you understand it.

This system supports weak references, but the beginner experiment should not pretend that we already have a complete family of variable-sized weak

collection classes. We shall therefore make an ordinary instance with one named instance variable. The instance itself is made with new, like an ordinary object. Its first field is used as a weak reference.

First create a class with one field named item:

```
Object weakSubclass: #WeakBox
  instVars: 'item' classInstVars: ''
```

Install two simple methods on the instance side of WeakBox:

```
item
    ^item


item: anObject
    item := anObject.
    ^self
```

Now make one weak box and one ordinary object. We shall use global names so that the example can be typed line by line in the Workspace.

```
Smalltalk at: #Box put: WeakBox new


Smalltalk at: #StrongThing put: OrderedCollection new
```

Put the ordinary object into the weak field:

```
Box item: StrongThing
```

At this point, the collection is strongly referred to by the global name StrongThing, and weakly referred to by the item field of Box.

Try:

```
Box item
```

You should get the collection object.

Now remove the strong global reference:

```
Smalltalk at: #StrongThing put: nil
```

Request garbage collection:

```
Smalltalk garbageCollect
```

Now try:

```
Box item
```

If no other part of the system still refers to the collection, the field should become `nil`.

This demonstrates the difference between a strong reference and a weak reference. The weak box remembered the collection without keeping it alive.

If the object does not disappear immediately, do not be surprised. In a live system, the Workspace, compiler, or current evaluation may still hold temporary references for a short time. Try the experiment from a fresh image and keep it simple.

Weak references are not needed for most beginner programmes. They are part of the machinery that makes live object systems possible.

Part Eleven — Source, Methods, and Decompilation

55. Why the source is reconstructed

In many BASIC systems, the programme text is the programme. If line 100 says `PRINT A`, that text is stored and listed again.

Smalltalk-Z80 is different. When a method is accepted, the source text is compiled into a compact internal form. This internal form is what the virtual machine executes. Keeping the original source text of every method would take too much memory on a 48K Spectrum.

Therefore, Smalltalk-Z80 reconstructs method source when the Browser needs to show it. This is called decompilation.

The reconstructed source should have the same meaning as the original method, but it may not have exactly the same spacing or parentheses. Some formatting is chosen by the decompiler, not by the original typist.

This is a practical compromise. It allows the Browser to show readable methods without spending precious memory storing all original source strings.

56. What decompilation means for you

When you browse a method, remember:

```
The method you see is reconstructed source.
```

Usually this is exactly what you need. You can read it, learn from it, and replace it.

But if you typed unusual spacing or extra parentheses, do not expect them always to return exactly as typed.

What matters is that the behaviour of the method is preserved. The system is designed to keep enough information to show useful source while leaving room for objects.

If you want to keep notes about a long method, copy those notes into your notebook. The Spectrum memory is for the living image.

Part Twelve — Advanced Live Object Memory

57. Why a live system needs object tools

In a batch language, you may compile a programme, run it, and then throw away all the data. If the programme changes, you start again.

Smalltalk is different. It is a live system. Classes, methods, and objects may already exist while you are changing the system.

Suppose you have made many `Point` objects. Later you decide that `Point` should have another instance variable. What should happen to the old points? In a full Smalltalk system, tools may find old instances and migrate them to the new shape. That means the system must be able to find objects, inspect them, and sometimes replace references.

Smalltalk-Z80 includes some advanced object-memory messages for this reason. They are not ordinary beginner programming tools. They are part of the machinery that makes a live object world possible.

58. someObject and nextObject

ADVANCED — use carefully.

A class can ask for some instance of itself:

```
Point someObject
```

If an instance exists, the system answers one. If none exists, the answer may be `nil`.

An object can ask for the next object after it in object memory:

```
somePoint nextObject
```

The order is not meaningful for your programme. It is an object-memory order, not a sorted list.

A careful object enumeration may look like this:

```
| obj count |
```

```
count := 0.
obj := Point someObject.
[obj notNil] whileTrue: [
    count := count + 1.
    obj := obj nextObject].
count
```

This counts existing points while walking object memory. It is deliberately small: when you enumerate the live object world, do the least work needed.

This is useful for system tools. For example, a tool that migrates old instances after a class change must find the old instances. A browser or inspector in a larger Smalltalk may also need to walk objects.

Do not use this as an ordinary collection. If you want a list of things in your programme, use an `OrderedCollection`. `someObject` and `nextObject` are for looking into the living object memory.

59. become: and becomeForward:

ADVANCED — powerful object-memory operation. Reload a fresh image before experimenting. Do not save after experiments unless you are certain.

Smalltalk systems sometimes need to replace one object with another throughout the image. This is not assignment to one variable. It is a change to references in object memory.

Smalltalk-Z80 provides two messages for this advanced work.

The message:

```
oldObject becomeForward: newObject
```

means: references to `oldObject` should now point forward to `newObject`.

The message:

```
oldObject become: newObject
```

is a two-way exchange: references to the two objects are exchanged. It is even more powerful. Use it only for system work.

A cautious demonstration uses globals:

```
Smalltalk at: #OldThing put: OrderedCollection new
```

```
Smalltalk at: #NewThing put: OrderedCollection new


OldThing becomeForward: NewThing
```

After this, a reference that formerly led to `OldThing` may now lead to `NewThing`.

Why is this useful? Consider class migration. If old objects of one shape must be replaced by new objects of another shape, a live system may build the new object and then make old references lead to the new object. In larger Smalltalk systems, operations like `become:` are important for changing living systems without stopping everything and starting again.

Why is it dangerous? Because it changes object references throughout the image. If used on important system objects, it can confuse or damage the running system.

The rule for beginners is simple:

```
Use becomeForward: only in an advanced experiment on a fresh image.
Do not save the image afterwards unless you know exactly what
changed.
```

60. basicAt: and object structure

Most programmes should use named messages such as `x`, `y`, `at:`, `add:`, and `size`.

Smalltalk-Z80 also has low-level messages:

```
object basicSize
object basicAt: 1
object basicAt: 1 put: value
```

These look directly at an object's fields by number. The full setter selector is `basicAt:put:`. These messages are useful for the system itself and for advanced inspection.

Do not use `basicAt:` as ordinary programming style when a named method would be clearer.

For example, this is good:

```
point x
```

This is poor style for ordinary programmes:

```
point basicAt: 1
```

The first says what you mean. The second depends on the private layout of the object.

Part Thirteen — A Small Project: A Question Card

61. A more interesting object

We shall make a small object suitable for school practice: a question card. It has a question, an answer, and a mark telling whether it has been asked.

Create:

```
Object subclass: #QuestionCard
  instVars: 'question answer asked' classInstVars: ''
```

Add:

```
initialize
    asked := false.
    ^self
```

Add:

```
question: q answer: a
    question := q.
    answer := a.
    asked := false.
    ^self
```

Add:

```
question
    ^question
```

Add:

```
answer
    ^answer
```

Add:

```
markAsked
    asked := true.
    ^self
```

Add:

```
wasAsked
    ^asked
```

On the class side, add:

```
question: q answer: a
    | c |
    c := self new.
    c question: q answer: a.
    ^c
```

Now try:

```
| c |
c := QuestionCard question: 'Capital of France?' answer: 'Paris'.
c question
```

Then:

```
| c |
c := QuestionCard question: 'Capital of France?' answer: 'Paris'.
c wasAsked
```

Then:

```
| c |
c := QuestionCard question: 'Capital of France?' answer: 'Paris'.
c markAsked.
c wasAsked
```

You have made an object that holds state and behaviour. It is not only a record and not only a subroutine. It is both information and messages together.

This small example points toward larger things: quizzes, lessons, expert rules, adventure games, and record systems.

62. A small collection of cards

Try:

```
|c|
c:=OrderedCollection new.
c add:(QuestionCard question:'Q' answer:'A').
c size
```

The answer is 1.

If you added `first`, try:

```
|c|
c:=OrderedCollection new.
c add:(QuestionCard question:'Q' answer:'A').
c first
```

This shows cooperation between objects. The collection holds cards. A card holds a question and an answer. Each object has its own job.

Part Fourteen — What Smalltalk-Z80 Leaves Out

63. Why this section exists

Smalltalk is a large family of techniques. Smalltalk-Z80 brings the central ones to the Spectrum: objects, classes, methods, inheritance, blocks, collections, compilation, browsing, an image, garbage collection, and live object-memory tools.

Large research Smalltalks have more conveniences. They have spacious bit-mapped displays, mice, discs, inspectors, debuggers, printers, and libraries thick enough to make the manual writer nervous.

This section is not an apology for the Spectrum. It is a map. The cassette edition chooses the parts that make the object system clear and useful on this machine. A good pocket knife is not a poor sawmill.

64. Inspector

A full Smalltalk system usually has an inspector. An inspector lets you point at an object and see its instance variables.

For example, an inspector for a `Point` might show:

```
class: Point
x: 5
y: 8
```

This is useful because Smalltalk is a live object world. You often want to look inside one object without writing a special method for it.

Smalltalk-Z80 has low-level tools such as `basicAt:` and a Browser for methods, but not a full comfortable inspector. You can still learn the principle: an object has named parts, and those parts may be examined by suitable messages.

65. Debugger

A full Smalltalk debugger lets you stop inside an error, see the chain of messages that led there, inspect variables, change code, and continue.

This is one of the most famous parts of larger Smalltalk systems. It turns errors into opportunities for exploration.

Smalltalk-Z80 gives brief error reports and has exception machinery, but not a full debugger. A good debugger is not only a few methods; it is a large interactive tool with displays, selections, and a great deal of supporting code.

66. Streams

A stream is an object for reading or writing a sequence one item at a time.

You may think of reading characters from a tape, taking cards one by one from a pile, or printing letters one after another. A stream remembers where it is.

Streams are very useful for files, text building, printing, and parsing. Larger Smalltalk systems use them heavily.

Smalltalk-Z80 does not provide the full stream family in the image. The Primer uses direct strings, arrays, collections, and simple output instead. These are sufficient for the first serious object programmes.

67. Processes and semaphores

Larger Smalltalk systems may have several independent activities called processes. A process can wait while another process runs. Semaphores can be used to signal between them.

This is useful in systems with windows, input devices, background tasks, and complex interactive tools.

Smalltalk-Z80 keeps to a single active computation. That is enough for learning objects, messages, classes, blocks, collections, and the image.

68. Model-view-controller

Some larger Smalltalk systems organise interactive screen programmes using three kinds of object.

A model represents the information. A view represents how it appears on the screen. A controller receives the user's commands.

This organisation is important because it separates what the programme knows from how it is displayed and how the user controls it.

Smalltalk-Z80 has its Browser and Workspace, but it does not provide the full

model-view-controller system. The complete arrangement belongs to larger bit-mapped Smalltalk installations. The Spectrum edition keeps the essential programming lesson: separate the object which knows from the object which shows.

69. Larger numbers and more arithmetic

Smalltalk-Z80 has small integer arithmetic sufficient for many examples. Larger systems often have large integers, fractions, floating-point numbers, and many mathematical messages.

These are useful for scientific and business programming. They are not central to the first object-oriented lessons.

You can add simple arithmetic conveniences yourself where practical. For example, if multiplication is absent in your image, a positive repeated-addition version can be written as an exercise, though it will not be fast.

70. Full class variables

Traditional Smalltalk class variables are variables shared by a class and its subclasses. They are useful, but their exact behaviour requires careful support.

Smalltalk-Z80 uses class instance variables instead. This keeps the rule clean: classes are objects, and class objects may have their own state.

A poor imitation of class variables using global names would be misleading, because two unrelated classes could accidentally share a name. It is better to teach class-side state correctly through class instance variables.

71. Full source preservation

Larger systems can keep original source text, categories, and exact formatting. Smalltalk-Z80 reconstructs method source by decompilation so that object memory is spent on executable methods and objects rather than on duplicate text.

This is why the Browser may show source that is equivalent but not identical in spacing to what was typed.

72. A larger standard library

Full Smalltalk systems have many more classes and methods: more collections, more text handling, more tools, more printing, more files, more numbers, and more user-interface support.

Smalltalk-Z80 includes enough to demonstrate the main ideas and to let the user extend the image. A lean image is also pleasant to explore: there are fewer classes to get lost among, and more reason to add your own.

Part Fifteen — Practical Limits

73. Why limits matter

The Spectrum has 48K of memory. Smalltalk-Z80 contains a virtual machine, image, compiler, Browser, Workspace, classes, methods, and objects inside that space.

Therefore, write small clear methods. This is good style anyway.

74. Useful limits and advice

The exact limits may change between image versions. For the version described by this Primer, the following facts and rules are useful:

Area	Practical limit or advice
Free heap after loading	The exact amount varies between builds. Treat memory as precious, and use the build report rather than this book for the current figure.
Workspace text	The Workspace buffer is about 512 bytes. Keep experiments short. For longer work, make methods in classes.
Method arguments	The runtime compiler has room for up to 8 method arguments. Ordinary beginner methods use far fewer.
Temporary variables	Keep to one or two temporary variables in methods you type in the IDE. Split large methods.
Keyword messages	Keep keyword messages simple. Very long selectors are not a good style on this machine.
Blocks	One- and two-argument blocks are supported, but deeply nested live blocks are a poor use of the reduced heap.
Recursion	Avoid deep recursion. Use `whileTrue:`, `to:do:`, or a method such as `timesRepeat:` for ordinary loops.
Method length	Keep methods short. If a method becomes hard to read, split it.
Strings	Keep literal strings modest. They consume image space.
Collections	Do not build large collections casually. Memory is precious.
Globals	Use few global names. Remove or reset experimental globals when done.
Source display	Browser source is reconstructed by decompilation. Comments and exact spacing are not preserved.
Garbage collection	Mark-and-sweep is available; compacting garbage collection is not present in this version.
Weak references	Use only in advanced experiments.
become: and becomeForward:	Use only on a fresh image and do not save afterwards unless certain.
PEEK/POKE/ports	Use with hardware knowledge. Wrong values may disturb the system.

When something goes wrong, do not panic. Reload a clean image. A Smalltalk system is meant to be explored, but exploration is safer when you know how to return to a good state.

Appendix A — Main keys

Key	Meaning
SYMBOL SHIFT + W	Switch Browser / Workspace
SYMBOL SHIFT + E	Evaluate current Workspace line
SYMBOL SHIFT + Q	Accept Browser edit or evaluate Workspace text, depending on context
Y	Confirm save
N	Refuse save
Q or cursor up	Previous list item
A or cursor down	Next list item
O or cursor left	Move focus left
P or cursor right	Move focus right
W	Page up
S	Page down
C	Toggle instance side / class side
N	New class or method
D	Delete selected method

Appendix B — Common messages

Message	Meaning
`class`	Answer the receiver's class
`new`	Make a new instance
`initialize`	Set up a new object
`yourself`	Answer the receiver itself
`isNil`	Answer whether receiver is nil
`notNil`	Answer whether receiver is not nil
`size`	Answer collection or string size
`at:`	Read indexed item
`at:put:`	Store indexed item
`add:`	Add item to growable collection
`do:`	Evaluate block for each item
`collect:`	Build transformed collection
`select:`	Build selected collection
`whileTrue:`	Repeat while test block is true
`ifTrue:`	Evaluate block if receiver is true
`ifFalse:`	Evaluate block if receiver is false
`value`	Evaluate block or answer object itself
`value:`	Evaluate one-argument block

Appendix C — Supplied classes to inspect

Start with these:

Class	Why inspect it
`Object`	Root behaviour: `class`, `isNil`, `yourself`, errors, `become:` and `becomeForward:`
`UndefinedObject`	Behaviour of `nil`
`True` and `False`	Decisions and Boolean messages
`SmallInteger`	Arithmetic, comparison, and `to:do:` counting
`Character`	Character code conversion
`String`	Text as objects
`Array`	Fixed indexed collections
`OrderedCollection`	Growable collections
`Dictionary`	Key-value storage
`Association`	Key and value pair
`Class`	Class creation
`Compiler`	Runtime method compilation
`CompiledMethod`	Method source reconstruction
`BlockClosure`	Blocks and loops
`Screen`	Display messages
`Spectrum`	Machine service messages: `beep`, `waitForKey`, `peek:`, `poke:value:`

Appendix D — Exercises

Exercise 1: a safer robot

Add to `Robot`:

```
isSleeping
    ^awake not
```

Add:

```
toggle
    awake ifTrue: [self sleep] ifFalse: [self wake].
    ^self
```

Try:

```
| r |
r := Robot new.
r toggle.
r isAwake
```

Exercise 2: second element

Add to `SequenceableCollection`:

```
second
    ^self at: 2
```

Try:

```
'ABC' second
#(10 20 30) second
```

Exercise 3: repeat a screen pause

First check the built-in counting loop:

```
1 to: 3 do: [ :i | Spectrum beep ]
```

After adding `timesRepeat:`, try:

```
3 timesRepeat: [ Screen clear. Spectrum waitForKey ]
```

Exercise 4: question cards

Extend `QuestionCard` with:

```
reset
    asked := false.
    ^self
```

Then test that `markAsked` and `reset` change the same object.

Final Words

You have now met the central rule of Smalltalk-Z80:

```
objects receive messages and answer.
```

A number receives arithmetic messages. A Boolean receives decision messages. A block receives `value`. A collection receives `do:`. A class receives `new`. A screen object receives `clear`. A Spectrum service receives `waitForKey`. Your own objects receive the messages you teach them.

That is the honest A.I. promise of the cassette: Turn your ZX Spectrum into an AI Workstation. The Spectrum has not become a professor in a box. It has become a place where you can build a little world of objects, ask it questions, alter it while it lives, and save it as an image.

The best way to continue is not to type in one enormous listing. Instead, add one clear class, then one clear message, then another. Make a robot, a question card, a point, a counter, a drawing tool, or a tiny rule book. Give each object a proper job.

If future computers are kind, they will give us more memory, quieter storage, and perhaps a pointing device that does not require three hands and a prayer. They will still need good objects.

The Spectrum is ready. Load the cassette, open the Workspace, and send the next message.