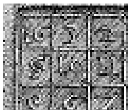
Chapter I: The Inform Language

Language is a cracked kettle on which we beat out tunes for bears to dance to, while all the time we long to move the stars to pity.

— from the letters of Gustave Flaubert (1821–1880)

§1 Routines



If you have already dipped into Chapter II and looked at how some simple games and puzzles are designed, you'll have seen that any really interesting item needs to be given instructions on how to behave, and that these instructions are written in a language of their own. Chapter I is about that language, and its examples are mostly short programs to carry out mundane tasks, undistracted by the lure of adventure.

§*1.1* *Getting started*

Inform turns your description of a game (or other program), called the "source code", into a "story file" which can be played (or run through) using an "interpreter". Interpreter programs are available for very many models of computer, and if you can already play Infocom's games or other people's Inform games on your machine then you already have an interpreter. There are several interpreter programs available, in a sort of friendly rivalry. You should be able to use whichever you prefer and even the oldest, most rickety interpreter will probably not give serious trouble. A good sign to look out for is compliance with the exacting *Z-Machine Standards Document*, agreed by an informal committee of interested parties between 1995 and 1997. At time of writing, the current version is Standard 1.0, dating from June 1997.

Turning source code into a story file is called "compilation", and Inform itself is the compiler. It's also supplied with a whole slew of ready-made source code called the "library" and giving all the rules for adventure games. The story of what the library does and how to work with it occupies most of this manual, but not this chapter.

It isn't practicable to give installation instructions here, because they vary so much from machine to machine, but before you can go any further you'll need to install Inform: try downloading the software for your own machine from `ftp.gmd.de`, which should either install itself or give instructions. A useful test exercise would be to try to create the "Hello World" source code given below, then to compile it with Inform, and finally "play" it on your interpreter. (You can type up source code with any text editor or even with a word-processor, provided you save its documents as "text only".)

Inform can run in a number of slightly different ways, controlled by "switches". The way to set these varies from one installation to another. Note that this chapter assumes that you're running Inform in "Strict mode", controlled by the `-S` switch, which is normally set and ensures that helpful error messages will be printed if a story file you have compiled does something it shouldn't.

§1.2 *Hello World*

Traditionally, all programming language tutorials begin by giving a program which does nothing but print "Hello world" and stop. Here it is in Inform:

```
!  "Hello world" example program
[ Main;
  print "Hello world^";
];
```

The text after the exclamation mark is a "comment", that is, it is text written in the margin by the author to remind himself of what is going on here. Such text means nothing to Inform, which ignores anything on the same line and to the right of an exclamation mark. In addition, any gaps made up of line and page breaks, tab characters and spaces are treated the same and called "white space", so the layout of the source code doesn't much matter. Exactly the same story file would be produced by:

```
    [
       Main   ;
print
          "Hello world^"         ;
      ]
  ;
```

or, at the other extreme, by:

```
[Main;print"Hello world^";];
```

Laying out programs legibly is a matter of personal taste.

△ The exception to the rule about ignoring white space is inside quoted text, where `"Hello     world^"` and `"Hello world^"` are genuinely different pieces of text and are treated as such. Inform treats text inside quotation marks with much more care than its ordinary program material: for instance, an exclamation mark inside quotation marks will not cause the rest of its line to be thrown away as a comment.

Inform regards its source code as a list of things to look at, divided up by semicolons `;`. These things are generally objects, of which more later. In this case there is only one, called `Main`, and it's of a special kind called a "routine".

Every program has to contain a routine called `Main`. When a story file is set running the interpreter follows the first instruction in `Main`, and it carries on line by line from there. This process is called "execution". Once the `Main` routine is finished, the interpreter stops.

These instructions are called "statements", a traditional term in computing albeit an ungrammatical one. In this case there is only one statement:

```
print "Hello world^";
```

Printing is the process of writing text onto the computer screen. This statement prints the two words "Hello world" and then skips the rest of the line (or "prints a new-line"), because the ^ character, in quoted text, means "new-line". For example, the statement

```
print "Blue^Red^Green^";
```

prints up:

```
Blue
Red
Green
```

`print` is one of 28 different statements in the Inform language. Only about 20 of these are commonly used, but the full list is as follows:

```
box         break    continue    do      font    for
give        if       inversion   jump    move    new_line
objectloop  print    print_ret   quit    read    remove
restore     return   rfalse      rtrue   save    spaces
string      style    switch      while
```

§1.3 Routine calls, errors and warnings

The following source code has three routines, `Main`, `Rosencrantz` and `Hamlet`:

```
[ Main;
  print "Hello from Elsinore.^";
  Rosencrantz();
];
[ Rosencrantz;
  print "Greetings from Rosencrantz.^";
];
[ Hamlet;
  print "The rest is silence.^";
];
```

The resulting program prints up

```
Hello from Elsinore.
Greetings from Rosencrantz.
```

but the text "The rest is silence." is never printed. Execution begins at `Main`, and "Hello from Elsinore" is printed; next, the statement `Rosencrantz()` causes the `Rosencrantz` routine to be executed. That continues until it ends with the close-routine marker `]`, whereupon execution goes back to `Main` just after the point where it left off: since there is nothing more to do in `Main`, the interpreter stops. Thus, `Rosencrantz` is executed but `Hamlet` is not.

In fact, when the above source code is compiled, Inform notices that `Hamlet` is never needed and prints out a warning to that effect. The exact text produced by Inform varies from machine to machine, but will be something like this:

```
RISC OS Inform 6.20 (10th December 1998)
line 8: Warning: Routine "Hamlet" declared but not used
Compiled with 1 warning
```

Errors are mistakes in the source which cause Inform to refuse to compile it, but this is only a warning. It alerts the programmer that a mistake may have been made (because presumably the programmer has simply forgotten to put in a statement calling `Hamlet`) but it doesn't prevent the compilation from taking place. Note that the opening line of the routine `Hamlet` occurs on the 8th line of the program above.

There are usually mistakes in a newly-written program and one goes through a cycle of running a first draft through Inform, receiving a batch of error messages, correcting the draft according to these messages, and trying again. A typical error message would occur if, on line 3, we had mistyped `Rosncrantz()` for `Rosencrantz()`. Inform would then have produced:

```
RISC OS Inform 6.20 (10th December 1998)
line 5: Warning: Routine "Rosencrantz" declared but not used
line 8: Warning: Routine "Hamlet" declared but not used
line 3: Error: No such constant as "Rosncrantz"
Compiled with 1 error and 2 warnings (no output)
```

The error message means that on line 3 Inform ran into a name which did not correspond to any known quantity: it's not the name of any routine, in particular. A human reader would immediately realise what was intended, but Inform doesn't, so that it goes on to warn that the routine `Rosencrantz` is never used. Warnings (and errors) are quite often produced as knock-on effects of other mistakes, so it is generally a good idea to worry about fixing errors first and warnings afterward.

Notice that Inform normally doesn't produce the final story file if errors occurred during compilation: this prevents it from producing damaged story files.

§*1.4* *Numbers and other constants*

Inform numbers are normally whole numbers in the range −32,768 to 32,767. (Special programming is needed to represent larger numbers or fractions, as we shall see when parsing phone numbers in Chapter IV.) There are three notations for writing numbers in source code: here is an example of each.

```
-4205
$3f08
$$1000111010110
```

The difference is the radix, or number base, in which they are expressed. The first is in decimal (base 10), the second hexadecimal (base 16, where the digits after 9 are written `a` to `f` or `A` to `F`) and the third binary (base 2). Once Inform has read in a number, it forgets which notation was used: for instance, if the source code is altered so that `$$10110` is replaced by `22`, this makes no difference to the story file produced.

A `print` statement can print numbers as well as text, though it always prints them back in ordinary decimal notation. For example, the program

```
[ Main;
  print "Today’s number is ", $3f08, ".^";
];
```

prints up

```
Today’s number is 16136.
```

since 16,136 in base 10 is the same number as 3f08 in hexadecimal.

Literal quantities written down in the source code are called “constants”. Numbers are one kind; strings of text like `"Today’s number is "` are another. A third kind are characters, given between single quotation marks. For instance, `'x'` means “the letter lower-case x”. A “character” is a single letter or typewriter-symbol.

△ Just as `$3f08` is a fancy way of writing the number 16,136, so `'x'` is a fancy way of writing the number 120. The way characters correspond to numeric values is given by a code called ZSCII, itself quite close to the traditional computing standard called ASCII. 120 means “lower-case x” in ASCII, too, but ZSCII does its own thing with non-English characters like “é”. (You can type accented letters directly into source code: see §1.11.) The available characters and their ZSCII values are laid out in Table 2.

Inform also provides a few constants named for convenience, the most commonly used of which are `true` and `false`. A condition such as “the jewelled box is unlocked” will always have either the value `true` or the value `false`.

△ Once again these are numbers in disguise. `true` is 1, `false` 0.

△ Inform is a language designed with adventure games in mind, where a player regularly types in commands like “unlock the box”, using a fairly limited vocabulary. Writing a word like `'box'` in the source code, in single-quotation marks, adds it to the “dictionary” or vocabulary of the story file to be compiled. This is a kind of constant, too: you can use it for writing code like “if the second word typed by the player is `'box'`, then. . .”.

△ If you need to lodge a single-letter word (say “h”) into the dictionary, you can’t put it in single quotes because then it would look like a character constant (`'h'`). Instead, the best you can write is `'h//'`. (The two slashes are sometimes used to tack a little linguistic information onto the end of a dictionary word, so that in some circumstances, see §29, you might want to write `'pears//p'` to indicate that “pears” must go into the dictionary marked as a plural. In this case the two slashes only serve to clarify that it isn’t a character.)

△ You can put an apostrophe ' into a dictionary word by writing it as `^`: for instance `'helen^s'`.

△△ In two places where dictionary words often appear, the `name` slot of an object definition and in grammar laid out with `Verb` and `Extend`, you're allowed to use single or double quotes interchangeably, and people sometimes do. For clarity's sake, this book tries to stick to using single quotes around dictionary words at all times. The handling of dictionary words is probably the single worst-designed bit of syntax in Inform, but you're past it now.

§*1.5* *Variables*

Unlike a literal number, a variable is able to vary. It is referred to by its name and, like the "memory" key on some pocket calculators, remembers the last value placed in it. For instance, if `oil_left` has been declared as a variable (see below), then the statement

```
print "There are ", oil_left, " gallons remaining.^";
```

would cause the interpreter to print "There are 4 gallons remaining." if `oil_left` happened to be 4, and so on. It's possible for the statement to be executed many times and produce different text at different times.

Inform can only know the named quantity `oil_left` is to be a variable if the source code has "declared" that it is. Each routine can declare its own selection of up to 15 variables on its opening line. For example, in the program

```
[ Main alpha b;
  alpha = 2200;
  b = 201;
  print "Alpha is ", alpha, " while b is ", b, "^";
];
```

the `Main` routine has two variables, `alpha` and `b`.

Going back to the `Main` routine above, the = sign which occurs twice is an example of an "operator": a notation usually made up of the symbols on the non-alphabetic keys on a typewriter and which means something is to be done with or calculated from the items it is written next to. Here = means "set equal to". When the statement `alpha = 2200;` is interpreted, the current value of the variable `alpha` is changed to 2,200. It then keeps that value until another such statement changes it. All variables have the value 0 until they are first set.

The variables alpha and b are called "local variables" because they are local to Main and are its private property. The source code

```
[ Main alpha;
  alpha = 2200;
  Rival();
];
[ Rival;
  print alpha;
];
```

causes an error on the print statement in Rival, since alpha does not exist there. Indeed, Rival could even have defined a variable of its own also called alpha and this would have been an entirely separate variable.

· · · · ·

That's now two kinds of name: routines have names and so have variables. Such names, for instance Rival and alpha, are called "identifiers" and can be up to 32 characters long. They may contain letters of the alphabet, decimal digits or the underscore _ character (often used to impersonate a space). To prevent them looking too much like numbers, though, they cannot start with a decimal digit. The following are examples of legal identifiers:

```
turns_still_to_play     room101     X
```

Inform ignores any difference between upper and lower case letters in such names, so for instance room101 is the same name as Room101.

§1.6 *Arithmetic, assignment and bitwise operators*

The Inform language is rich with operators, as Table 1 shows. This section introduces a first batch of them.

A general mixture of quantities and operators, designed to end up with a single resulting quantity, is called an "expression". For example: the statement

```
seconds = 60*minutes + 3600*hours;
```

sets the variable seconds equal to 60 times the variable minutes plus 3600 times the variable hours. White space is not needed between operators and "operands" (the quantities they operate on, such as `3600` and `hours`). The spaces on either side of the `+` sign were written in just for legibility.

The arithmetic operators are the simplest ones. To begin with, there are `+` (plus), `-` (minus), `*` (times) and `/` (divide by). Dividing one whole number by another usually leaves a remainder: for example, 3 goes into 7 twice, with remainder 1. In Inform notation,

`7/3` evaluates to 2 and `7%3` evaluates to 1

the `%` operator meaning "remainder after division", usually called just "remainder".

△ The basic rule is that `a == (a/b)*b + (a%b)`, so that for instance:

```
13/5 == 2     13/-5 == -2    -13/5 == -2    -13/-5 == 2
13%5 == 3     13%-5 == 3     -13%5 == -3    -13%-5 == -3
```

• WARNING

Dividing by zero, and taking remainder after dividing by zero, are impossible. You must write your program so that it never tries to. It's worth a brief aside here on errors, because dividing by zero offers an example of how and when Inform can help the programmer to spot mistakes. The following source code:

```
[ Main; print 73/0; ];
```

won't compile, because Inform can see that it definitely involves something illegal:

```
line 2: Error: Division of constant by zero
>   print 73/0;
```

However, Inform fails to notice anything amiss when compiling this:

```
[ Main x; x = 0; print 73/x; ];
```

and this source code compiles correctly. When the resulting story file is interpreted, however, the following will be printed:

[** Programming error: tried to divide by zero **]

This is only one of about fifty different programming errors which can turn up when a story file is interpreted. As in this case, they arise when the interpreter has accidentally been asked to do something impossible. The moral is that just because Inform compiles source code without errors, it does not follow that the story file does what the programmer intended.

△ Since an Inform number has to be between −32,768 and 32,767, some arithmetic operations overflow. For instance, multiplying 8 by 5,040 ought to give 40,320, but this is over the top and the answer is instead −25,216. Unlike dividing by zero, causing arithmetic overflows is perfectly legal. Some programmers make deliberate use of overflows, for instance to generate apparently random numbers.

△△ Only *apparently* random, because overflows are perfectly predictable. Inform story files store numbers in sixteen binary digits, so that when a number reaches 2^{16} = 65,536 it clocks back round to zero (much as a car's odometer clocks over from 999999 miles to 000000). Now, since 65,535 is the value that comes before 0, it represents −1, and 65,534 represents −2 and so on. The result is that if you start with zero and keep adding 1 to it, you get $1, 2, 3, \ldots,$ 32,767 and then −32,768, −32,767, −32,766, …, −1 and at last zero again. Here's how to predict an overflowing multiplication, say: first, multiply the two numbers as they stand, then keep adding or subtracting 65,536 from the result until it lies in the range −32,768 to 32,767. For example, 8 multiplied by 5,040 is 40,320, which is too big, but we only need to subtract 65,536 once to bring it into range, and the result is −25,216.

· · · · ·

In a complicated expression the order in which the operators work may affect the result. As most human readers would, Inform works out both of

```
3 + 2 * 6
2 * 6 + 3
```

as 15, because the operator `*` has "precedence" over `+` and so is acted on first. Brackets may be used to overcome this:

```
(3 + 2) * 6
2 * (6 + 3)
```

evaluate to 30 and 18 respectively. Each operator has such a "precedence level". When two operators have the same precedence level (for example, `+` and `-` are of equal precedence) calculation is (almost always) "left associative", that is, carried out left to right. So the notation `a-b-c` means `(a-b)-c` and not `a-(b-c)`. The standard way to write formulae in maths is to give `+` and `-` equal precedence, but lower than that of `*` and `/` (which are also equal). Inform agrees and also pegs `%` equal to `*` and `/`.

The last purely arithmetic operator is "unary minus". This is also written as a minus sign `-` but is not quite the same as subtraction. The expression:

```
-credit
```

means the same thing as `0-credit`. The operator `-` is different from all those mentioned so far because it operates only on one value. It has higher precedence than any of the five other arithmetic operators. For example,

```
    -credit - 5
```

means `(-credit) - 5` and not `-(credit - 5)`.

One way to imagine precedence is to think of it as glue attached to the operator. A higher level means stronger glue. Thus, in

```
    3 + 2 * 6
```

the glue around the `*` is stronger than that around the `+`, so that 2 and 6 belong bound to the `*`.

△ Some languages have a "unary plus" too, but Inform hasn't.

. . . . .

Some operators don't just work out values but actually change the current settings of variables: expressions containing these are called "assignments". One such is "set equals":

```
    alpha = 72
```

sets the variable `alpha` equal to 72. Like `+` and the others, it also comes up with an answer: which is the value it has set, in this case 72.

The other two assignment operators are `++` and `--`, which will be familiar to any C programmer. They are unary operators, and mean "increase (or decrease) the value of this variable by one". If the `++` or `--` goes before the variable, then the increase (or decrease) happens before the value is read off; if after, then after. For instance, if `variable` currently has the value 12 then:

`variable++` evaluates to 12 and leaves `variable` set to 13;
`++variable` evaluates to 13 and leaves `variable` set to 13;
`variable--` evaluates to 12 and leaves `variable` set to 11;
`--variable` evaluates to 11 and leaves `variable` set to 11.

These operators are provided as convenient shorthand forms, since their effect could usually be achieved in other ways. Note that expressions like

```
    500++          (4*alpha)--      34 = beta
```

are quite meaningless: the values of 500 and 34 cannot be altered, and Inform knows no way to adjust `alpha` so as to make `4*alpha` decrease by 1. All three will cause compilation errors.

. . . . .

The "bitwise operators" are provided for manipulating binary numbers on a digit-by-digit basis, something which is only done in programs which are working with low-level data or data which has to be stored very compactly. Inform provides &, bitwise AND, |, bitwise OR and ~, bitwise NOT. For each digit, such an operator works out the value in the answer from the values in the operands. Bitwise NOT acts on a single operand and results in the number whose i-th binary digit is the opposite of that in the operand (a 1 for a 0, a 0 for a 1). Bitwise AND (and OR) acts on two numbers and sets the i-th digit to 1 if both operands have (either operand has) i-th digit set. All Inform numbers are sixteen bits wide. So:

```
$$10111100 & $$01010001  ==  $$0000000000010000
$$10111100 | $$01010001  ==  $$0000000011111101
           ~ $$01010001  ==  $$1111111110101110
```

§1.7 *Arguments and Return Values*

Here is one way to imagine how an Inform routine works: you feed some values into it, it then goes away and works on them, possibly printing some text out or doing other interesting things, and it then returns with a single value which it gives back to you. As far as you're concerned, the transaction consists of turning a group of starting values, called "arguments", into a single "return value":

$$A_1, A_2, A_3, \ldots \longrightarrow \texttt{Routine} \longrightarrow R$$

The number of arguments needed varies with the routine: some, like `Main` and the other routines in this chapter so far, need none at all. (Others need anything up to seven, which is the maximum number allowed by Inform.) On the other hand, every routine without exception produces a return value. Even when it looks as if there isn't one, there is. For example:

```
[ Main;
  Sonnet();
];
[ Sonnet;
  print "When to the sessions of sweet silent thought^";
  print "I summon up remembrance of things past^";
];
```

Main and Sonnet both take no arguments, but they both return a value: as it happens this value is true, in the absence of any instruction to the contrary. (As was mentioned earlier, true is the same as the number 1.) The statement Sonnet(); calls Sonnet but does nothing with the return value, which is just thrown away. But if Main had instead been written like so:

```
[ Main;
  print Sonnet();
];
```

then the output would be

```
When to the sessions of sweet silent thought
I summon up remembrance of things past
1
```

because now the return value, 1, is not thrown away: it is printed out.

You can call it a routine with arguments by writing them in a list, separated by commas, in between the round brackets. For instance, here is a call supplying two arguments:

```
Weather("hurricane", 12);
```

When the routine begins, the value "hurricane" is written into its first local variable, and the value 12 into its second. For example, suppose:

```
[ Weather called force;
  print "I forecast a ", (string) called, " measuring force ",
      force, " on the Beaufort scale.^";
];
```

Leaving the details of the print statement aside for the moment, the call to this routine produces the text:

```
I forecast a hurricane measuring force 12 on the Beaufort scale.
```

The Weather routine finishes when its] end-marker is reached, whereupon it returns true, but any of the following statements will finish a routine the moment they are reached:

rfalse;	which returns false,
rtrue;	which returns true,

`return;`	which also returns `true`,
`return ⟨value⟩;`	which returns ⟨value⟩.

For example, here is a routine to print out the cubes of the numbers 1 to 5:

```
[ Main;
  print Cube(1), " ";
  print Cube(2), " ";
  print Cube(3), " ";
  print Cube(4), " ";
  print Cube(5), "^";
];
[ Cube x;
  return x*x*x;
];
```

When interpreted, the resulting story file prints up the text:

```
1 8 27 64 125
```

△ Any "missing arguments" in a routine call are set equal to zero, so the call `Cube()` is legal and does the same as `Cube(0)`. What you mustn't do is to give too many arguments: `Cube(1,2)` isn't possible because there is no variable to put the 2 into.

△ A hazardous, but legal and sometimes useful practice is for a routine to call itself. This is called recursion. The hazard is that the following mistake can be made, probably in some much better disguised way:

```
[ Disaster; return Disaster(); ];
```

Despite the reassuring presence of the word `return`, execution is tied up forever, unable to finish evaluating the return value. The first call to `Disaster` needs to make a second before it can finish, the second needs to make a third, the third. . . and so on. (Actually, for "forever" read "until the interpreter runs out of stack space and halts", but that's little comfort.)

§1.8 *Conditions:* `if`, `true` *and* `false`

The facilities described so far make Inform about as powerful as the average small programmable calculator. To make it a proper programming language, it needs much greater flexibility of action. This is provided by special statements which control whether or not, and if so how many times or in what order, other statements are executed. The simplest is `if`:

```
if (⟨condition⟩) ⟨statement⟩
```

which executes the ⟨statement⟩ only if the ⟨condition⟩, when it is tested, turns out to be true. For example, when the statement

```
if (alpha == 3) print "Hello";
```

is executed, the word "Hello" is printed only if the variable `alpha` currently has value 3. It's important not to confuse the == (test whether or not equal to) with the = operator (set equal to). But because it's easy to write something plausible like

```
if (alpha = 3) print "Hello";
```

by accident, which always prints "Hello" because the condition evaluates to 3 which is considered non-zero and therefore true (see below), Inform will issue a warning if you try to compile something like this. ('=' used as condition: '==' intended?)

. . . . .

Conditions are always given in brackets. There are 12 different conditions in Inform (see Table 1), six arithmetic and half a dozen to do with objects. Here are the arithmetic ones:

`(a == b)`	a equals `b`
`(a ~= b)`	a doesn't equal `b`
`(a >= b)`	a is greater than or equal to `b`
`(a <= b)`	a is less than or equal to `b`
`(a > b)`	a is greater than `b`
`(a < b)`	a is less than `b`

A useful extension to this set is provided by the special operator `or`, which gives alternative possibilities. For example,

```
if (alpha == 3 or 4) print "Scott";
if (alpha ~= 5 or 7 or 9) print "Amundsen";
```

where two or more values are given with the word `or` between. "Scott" is printed if `alpha` has value either 3 or 4, and "Amundsen" if the value of `alpha` is not 5, is not 7 and is not 9. `or` can be used with any condition, and any number of alternatives can be given. For example

```
    if (player in Forest or Village or Building) ...
```

often makes code much clearer than writing three separate conditions out. Or you might want to use

```
    if (x > 100 or y) ...
```

to test whether `x` is bigger than the minimum of 100 and `y`.

Conditions can also be built up from simpler ones using the three logical operators `&&`, `||` and `~~`, pronounced "and", "or" and "not". For example,

```
    if (alpha == 1 && (beta > 10 || beta < -10)) print "Lewis";
    if (~~(alpha > 6)) print "Clark";
```

"Lewis" is printed if `alpha` equals 1 and `beta` is outside the range −10 to 10; "Clark" is printed if `alpha` is less than or equal to 6.

△ `&&` and `||` work left to right and stop evaluating conditions as soon as the final outcome is known. So for instance `if (A && B) ...` will work out `A` first. If this is `false`, there's no need to work out `B`. This is sometimes called short-cut evaluation and can be convenient when working out conditions like

```
    if (x~=nothing && TreasureDeposited(x)==true) ...
```

where you don't want `TreasureDeposited` to be called with the argument `nothing`.

· · · · ·

Conditions are expressions like any other, except that their values are always either `true` or `false`. You can write a condition as a value, say by copying it into a variable like so:

```
    lower_caves_explored = (Y2_Rock_Room has visited);
```

This kind of variable, storing a logical state, is traditionally called a "flag". Flags are always either `true` or `false`: they are like the red flag over the beach at Lee-on-Solent in Hampshire, which is either flying, meaning that the army is using the firing range, or not flying, when it is safe to walk along the shore. Flags allow you to write natural-looking code like:

```
    if (lower_caves_explored) print "You've already been that way.";
```

The actual test performed by `if (x) ...` is `x~=0`, not that `x==true`, because all non-zero quantities are considered to represent truth whereas only zero represents falsity.

· · · · ·

△ Now that the if statement is available, it's possible to give an example of a recursion that does work:

```
[ GreenBottles n;
  print n, " green bottles, standing on a wall.^";
  if (n == 0) print "(And an awful lot of broken glass.)^";
  else {
      print "And if one green bottle should accidentally fall^";
      print "There'd be ", n-1, " green bottles.^";
      return GreenBottles(n-1);
  }
];
```

Try calling `GreenBottles(10)`. It prints the first verse of the song, then before returning it calls `GreenBottles(9)` to print the rest. So it goes on, until `GreenBottles(0)`. At this point `n` is zero, so the text "(And an awful lot of broken glass.)" is printed for the first and only time. `GreenBottles(0)` then returns back to `GreenBottles(1)`, which returns to `GreenBottles(2)`, which. . . and so on until `GreenBottles(10)` finally returns and the song is completed.

△△ Thus execution reached "ten routines deep" before starting to return back up, and each of these copies of `GreenBottles` had its own private copy of the variable `n`. The limit to how deep you are allowed to go varies from one player's machine to another, but here is a rule of thumb, erring on the low side for safety's sake. (On a standard interpreter it will certainly be safe.) Total up 4 plus the number of its variables for each routine that needs to be running at the same time, and keep the total beneath 1,000. Ten green bottles amounts only to a total of $10 \times 5 = 50$, and as it seems unlikely that anyone will wish to read the lyrics to "two hundred and one green bottles" the above recursion is safe.

§1.9 *Code blocks,* `else` *and* `switch`

A feature of all statements choosing what to do next is that instead of just giving a single ⟨statement⟩, one can give a list of statements grouped together into a unit called a "code block". Such a group begins with an open brace { and ends with a close brace }. For example,

```
if (alpha > 5) {
    v = alpha*alpha;
    print "The square of alpha is ", v, ".^";
}
```

If `alpha` is 3, nothing is printed; if `alpha` is 9,

```
The square of alpha is 81.
```

is printed. (The indentation used in the source code, like all points of source code layout, is a matter of personal taste.†) In some ways, code blocks are like routines, and at first it may seem inconsistent to write routines between `[` and `]` brackets and code blocks between braces `{` and `}`. However, code blocks cannot have private variables of their own and do not return values; and it is possible for execution to break out of code blocks again, or to jump from block to block, which cannot happen with routines.

· · · · ·

An `if` statement can optionally have the form

if (⟨condition⟩) ⟨statement1⟩ else ⟨statement2⟩

whereupon ⟨statement1⟩ is executed if the condition is true, and ⟨statement2⟩ if it is false. For example,

```
if (alpha == 5) print "Five."; else print "Not five.";
```

Note that the condition is only checked once, so that the statement

```
if (alpha == 5) {
    print "Five.";
    alpha = 10;
}
else print "Not five.";
```

cannot ever print both "Five" and then "Not five".

△ The `else` clause has a snag attached: the problem of "hanging elses". In the following, which `if` statement does the `else` attach to?

```
if (alpha == 1) if (beta == 2)
    print "Clearly if alpha=1 and beta=2.^";
else
    print "Ambiguous.^";
```

† Almost everybody indents each code block as shown, but the position of the open brace is a point of schism. This book adopts the "One True Brace Style", as handed down in such sacred texts as the original Unix source code and Kernighan and Ritchie's textbook of C. Other conventions place the open brace vertically above its matching closure, perhaps on its own otherwise blank line of source code.

Without clarifying braces, Inform pairs an `else` to the most recent `if`. (Much as the U.S. Supreme Court rigorously interprets muddled laws like "all animals must be licensed, except cats, other than those under six months old" by applying a so-called Last Antecedent Rule to ambiguous qualifications like this "other than...".) The following version is much better style:

```
if (alpha == 1) {
    if (beta == 2)
        print "Clearly if alpha=1 and beta=2.^";
    else
        print "Clearly if alpha=1 but beta not 2.^";
}
```

· · · · ·

The `if` ... `else` ... construction is ideal for switching execution between two possible "tracks", like railway signals, but it is a nuisance trying to divide between many different outcomes this way. To follow the analogy, the `switch` construction is like a railway turntable.

```
print "The train on platform 1 is going to ";
switch (DestinationOnPlatform(1)) {
    1: print "Dover Priory.";
    2: print "Bristol Parkway.";
    3: print "Edinburgh Waverley.";
    default: print "a siding.";
}
```

The `default` clause is optional but must be placed last if at all: it is executed when the original expression matches none of the other values. Otherwise there's no obligation for these clauses to be given in numerical or any other order. Each possible alternative value must be a constant, so

```
switch (alpha) {
    beta: print "The variables alpha and beta are equal!";
}
```

will produce a compilation error. (But note that in a typical game, the name of an object or a location, such as `First_Court`, is a constant, so there is no problem in quoting this as a switch value.)

Any number of outcomes can be specified, and values can be grouped together in lists separated by commas, or in ranges like `3 to 6`. For example:

```
print "The mission Apollo ", num, " made ";
switch (num) {
    7, 9: print "a test-flight in Earth orbit.";
    8, 10: print "a test-flight in lunar orbit.";
    11, 12, 14 to 17: print "a landing on the Moon.";
    13: print "it back safely after a catastrophic explosion.";
}
```

Each clause is automatically a code block, so a whole run of statements can be given without the need for any braces { and } around them.

△ If you're used to the C language, you might want to note a major difference: Inform doesn't have "case fall-through", with execution running from one case to the next, so there's no need to use `break` statements.

△△ A good `default` clause for the above example would be a little complicated: Apollo 1 was lost in a ground fire, causing a shuffle so that 2 and 3 never happened, while automatic test-flights of the Saturn rocket were unofficially numbered 4 to 6. Apollo 20 was cancelled to free up a heavy launcher for the Skylab station, and 18 and 19 through budget cuts, though the Apollo/Soyuz Test Project (1975) is sometimes unhistorically called Apollo 18. The three Apollo flights to Skylab were called Skylab 2, 3 and 4. All six Mercury capsules were numbered 7, while at time of writing the Space Shuttle mission STS-88 has just landed and the next to launch, in order, are projected to be 96, 93, 99, 103, 101 and 92. NASA has proud traditions.

§*1.10* `while`, `do` ... `until`, `for`, `break`, `continue`

The other four Inform control constructions are all "loops", that is, ways to repeat the execution of a given statement or code block. Discussion of one of them, called `objectloop`, is deferred until §3.4.

The two basic forms of loop are `while` and `do` ... `until`:

```
while (⟨condition⟩) ⟨statement⟩
do ⟨statement⟩ until (⟨condition⟩)
```

The first repeatedly tests the condition and, provided it is still true, executes the statement. If the condition is not even true the first time, the statement is never executed even once. For example:

```
[ SquareRoot n x;
  while (x*x < n) x = x + 1;
  if (x*x == n) return x;
  return x - 1;
];
```

which is a simple if rather inefficient way to find square roots, rounded down to the nearest whole number. If `SquareRoot(200)` is called, then `x` runs up through the values $0, 1, 2, \ldots, 14, 15$, at which point `x*x` is 225: so 14 is returned. If `SquareRoot(0)` is called, the `while` condition never holds at all and so the return value is 0, made by the `if` statement.

The `do` ... `until` loop repeats the given statement until the condition is found to be `true`. Even if the condition is already satisfied, like `(true)`, the statement is always executed the first time through.

· · · · ·

One particular kind of `while` loop is needed so often that there is an abbreviation for it, called `for`. This can produce any loop in the form

```
⟨start⟩
while (⟨condition⟩) {
    ...
    ⟨update⟩
}
```

where ⟨start⟩ and ⟨update⟩ are expressions which actually do something, such as setting a variable. The notation to achieve this is:

```
for (⟨start⟩ : ⟨condition⟩ : ⟨update⟩) ...
```

Note that if the condition is false the very first time, the loop is never executed. For instance, this prints nothing:

```
for (counter=1 : counter<0 : counter++) print "Banana";
```

Any of the three parts of a `for` statement can be omitted. If the condition is missed out, it is assumed always `true`, so that the loop will continue forever, unless escaped by other means (see below).

For example, here is the `while` version of a common kind of loop:

```
counter = 1;
while (counter <= 10) {
    print counter, " ";
    counter++;
}
```

which produces the output "1 2 3 4 5 6 7 8 9 10". (Recall that `counter++` adds 1 to the variable `counter`.) The abbreviated version is:

```
for (counter=1 : counter<=10 : counter++)
    print counter, " ";
```

△ Using commas, several assignments can be joined into one. For instance:

```
i++, score=50, j++
```

is a single expression. This is never useful in ordinary code, where the assignments can be divided up by semicolons in the usual way. But in `for` loops it can be a convenience:

```
for (i=1, j=5: i<=5: i++, j--) print i, " ", j, ", ";
```

produces the output "1 5, 2 4, 3 3, 4 2, 5 1,".

△△ Comma `,` is an operator, and moreover is the one with the lowest precedence level. The result of `a,b` is always `b`, but `a` and `b` are both evaluated and in that order.

. . . . .

On the face of it, the following loops all repeat forever:

```
while (true) ⟨statement⟩
do ⟨statement⟩ until (false)
for (::) ⟨statement⟩
```

But there is always an escape. One way is to `return` from the current routine. Another is to `jump` to a label outside the loop (see below), though if one only wants to escape the current loop then this is seldom good style. It's neatest to use the statement `break`, which means "break out of" the current innermost loop or `switch` statement: it can be read as "finish early". All these ways out are entirely safe, and there is no harm in leaving a loop only half-done.

The other simple statement used inside loops is `continue`. This causes the current iteration to end immediately, but the loop then continues. In particular, inside a `for` loop, `continue` skips the rest of the body of the loop and goes straight to the ⟨update⟩ part. For example,

```
for (i=1: i<=5: i++) {
    if (i==3) continue;
    print i, " ";
}
```

will output "1 2 4 5".

· · · · ·

△ The following routine is a curious example of a loop which, though apparently simple enough, contains a trap for the unwary.

```
[ RunPuzzle n count;
  do {
      print n, " ";
      n = NextNumber(n);
      count++;
  }
  until (n==1);
  print "1^(taking ", count, " steps to reach 1)^";
];
[ NextNumber n;
  if (n%2 == 0) return n/2;      ! If n is even, halve it
  return 3*n + 1;                ! If n is odd, triple and add 1
];
```

The call `RunPuzzle(10)`, for example, results in the output

```
10 5 16 8 4 2 1
(taking 6 steps to reach 1)
```

The definition of `RunPuzzle` assumes that, no matter what the initial value of `n`, enough iteration will end up back at 1. If this did not happen, the interpreter would lock up into an infinite loop, printing numbers forever. The routine is apparently very simple, so it would seem reasonable that by thinking carefully enough about it, we ought to be able to decide whether or not it will ever finish. But this is not so easy as it looks. `RunPuzzle(26)` takes ten steps, but there again, `RunPuzzle(27)` takes 111. Can this routine ever lock up into an infinite loop, or not?

△△ The answer, which caught the author by surprise, is: yes. Because of Inform's limited number range, eventually the numbers reached overflow 32,767 and Inform interprets them as negative – and quickly homes in on the cycle $-1, -2, -1, -2, \ldots$ This first happens to `RunPuzzle(447)`. Using proper arithmetic, unhindered by a limited number range, the answer is unknown. As a student in Hamburg in the 1930s, Lothar Collatz conjectured that every positive `n` eventually reaches 1. Little progress has been made (though it is known to be true if `n` is less than 2^{40}), and even Paul Erdös said of it that "Mathematics is not yet ready for such problems." See Jeffrey Lagarias's bibliography *The* $3x + 1$ *Problem and its Generalisations* (1996).

§1.11 *How text is printed*

Adventure games take a lot of trouble over printing, so Inform is rich in features to make printing elegant output more easy. During story-file interpretation, your text will automatically be broken properly at line-endings. Inform story files are interpreted on screen displays of all shapes and sizes, but as a programmer you can largely ignore this. Moreover, Inform itself will compile a block of text spilling over several source-code lines into normally-spaced prose, so:

```
print "Here in her hairs
       the painter plays the spider, and hath woven
       a golden mesh t'untrap the hearts of men
       faster than gnats in cobwebs";
```

results in the line divisions being replaced by a single space each. The text printed is: "Here in her hairs the painter plays the spider, and hath woven a golden mesh..." and so on. There is one exception to this: if a line finishes with a ^ (new-line) character, then no space is added.

△ You shouldn't type double-spaces after full stops or other punctuation, as this can spoil the look of the final text on screen. In particular, if a line break ends up after the first space of a double-space, the next line will begin with the second space. Since many typists habitually double-space, Inform has a switch `-d1` which contracts "`.  `" to "`. `" wherever it occurs, and then a further setting `-d2` which also contracts "`!  `" to "`! `" and "`?  `" to "`? `". A modest warning, though: this can sometimes make a mess of diagrams. Try examining the 1851 Convention on Telegraphy Morse Code chart in 'Jigsaw', for instance, where the present author inadvertently jumbled the spacing because `-d` treated the Morse dots as full stops.

. . . . .

When a string of text is printed up with `print`, the characters in the string normally appear exactly as given in the source code. However, four characters have special meanings. ^ means "print a new-line". The tilde character ~, meaning "print a quotation mark", is needed since quotation marks otherwise finish strings. Thus,

```
print "~Look,~ says Peter. ~Socks can jump.~^Jane agrees.^";
```

is printed as

> "Look," says Peter. "Socks can jump."
> Jane agrees.

The third remaining special character is `@`, occasionally used for accented characters and other unusual effects, as described below. Finally, \ is reserved for "folding lines", and is no longer needed but is retained so that old programs continue to work.

△ If you want to print an actual ~, ^, `@` or \, you may need one of the following "escape sequences". A double `@` sign followed by a decimal number means the character with the given ZSCII value, so in particular

`@@92`	comes out as "\"	`@@64`	comes out as "@"
`@@94`	comes out as "^"	`@@126`	comes out as "~"

. . . . .

A number of Inform games have been written in European languages other than English, and even English-language games need accented letters from time to time. Inform can be told via command-line switches to assume that quoted text in the source code uses any of the ISO 8859-1 to -9 character sets, which include West and Central European forms, Greek, Arabic, Cyrillic and Hebrew. The default is ISO Latin-1, which means that you should be able to type most standard West European letters straight into the source code.

△ If you can't conveniently type foreign accents on your keyboard, or you want to make a source code file which could be safely taken from a PC to a Macintosh or vice versa, you can instead type accented characters using `@`. (The PC operating system Windows and Mac OS use incompatible character sets, but this incompatibility would only affect your source code. A story file, once compiled, behaves identically on PC and Macintosh regardless of what accented letters it may contain.) Many accented characters can be written as `@`, followed by an accent marker, then the letter on which the accent appears:

`@^`	put a circumflex on the next letter: a e i o u A E I O or U
`@'`	put an acute on the next letter: a e i o u y A E I O U or Y
`` @` ``	put a grave on the next letter: a e i o u A E I O or U
`@:`	put a diaeresis on the next letter: a e i o u A E I O or U
`@c`	put a cedilla on the next letter: c or C
`@~`	put a tilde on the next letter: a n o A N or O
`@\`	put a slash on the next letter: o or O
`@o`	put a ring on the next letter: a or A

A few other letter-forms are available: German ß (`@ss`), ligatures (`@oe`, `@ae`, `@OE`, `@AE`), Icelandic "thorn" `@th` and "eth" `@et`, a pounds-sterling sign (`@LL`), Spanish inverted punctuation (`@!!` and `@??`) and continental European quotation marks (`@<<` and `@>>`): see Table 2. For instance,

```
    print "Les @oeuvres d'@AEsop en fran@ccais, mon @'el@`eve!";
    print "Na@:ive readers of the New Yorker re@:elected Mr Clinton.";
    print "Gau@ss first proved the Fundamental Theorem of Algebra.";
```

Accented characters can also be referred to as constants, like other characters. Just as `'x'` represents the character lower-case-X, so `'@^A'` represents capital-A-circumflex.

△△ It takes a really, really good interpreter with support built in for Unicode and access to the proper fonts to use such a story file, but in principle you can place any Unicode character into text by quoting its Unicode value in hexadecimal. For instance, `@{a9}` produces a copyright sign (Unicode values between `$0000` and `$00ff` are equal to ISO Latin1 values); `@{2657}` is a White bishop chess symbol; `@{274b}` is a "heavy eight teardrop-spoked propeller asterisk"; `@{621}` is the Arabic letter Hamza, and so on for around 30,000 more, including vast sets for Pacific Rim scripts and even for invented ones like Klingon or Tolkien's Elvish. Of course none of these new characters are in the regular ZSCII set, but ZSCII is configurable using Inform's `Zcharacter` directive. See §36 for more.

· · · · ·

△ The remaining usage of `@` is hacky but powerful. Suppose you are trying to implement some scenes from Infocom's spoof of 1930s sci-fi, 'Leather Goddesses of Phobos', where one of the player's companions has to be called Tiffany if the player is female, and Trent if male. The name turns up in numerous messages and it would be tiresome to keep writing

```
if (female_flag) print "Tiffany"; else print "Trent";
```

Instead you can use one of Inform's 32 "printing-variables" `@00` to `@31`. When the text `@14` is printed, for instance, the contents of string 14 are substituted in. The contents of string 14 can be set using the `string` statement, so:

```
if (female_flag) string 14 "Tiffany"; else string 14 "Trent";
...
print "You offer the untangling cream to @14, who whistles.^";
```

The value specified by a `string` statement has to be a literal, constant bit of text. There are *really* hacky ways to get around this, but if you needed to then you'd probably be better off with a different solution anyway.

§1.12 *The* `print` *and* `print_ret` *statements*

The `print` and `print_ret` statements are almost identical. The difference is that the second prints out a final and extra new-line character, and then causes a return from the current routine with the value `true`. Thus, `print_ret` should be read as "print this, print a new-line and then return `true`", and so

```
print_ret "That's enough of that.";
```

is equivalent to

```
    print "That's enough of that.^"; rtrue;
```

As an abbreviation, it can even be shortened to:

```
    "That's enough of that.";
```

Although Inform newcomers are often confused by the fact that this innocently free-standing bit of text actually causes a return from the current routine, it's an abbreviation which pays dividends in adventure-writing situations:

```
    if (fuse_is_lit) { deadflag = true; "The bomb explodes!"; }
    "Nothing happens.";
```

Note that if the source code:

```
    [ Main;
      "Hello, and now for a number...";
      print 21*764;
    ];
```

is compiled, Inform will produce the warning message:

```
    line 3: Warning: This statement can never be reached.
    >   print 21*764;
```

because the bare string on line 2 is printed using `print_ret`: so the text is printed, then a new-line is printed, and then a `return` takes place immediately. As the warning message indicates, there is no way the statement on line 3 can ever be executed.

So what can be printed? The answer is a list of terms, separated by commas. For example,

```
    print "The value is ", value, ".";
```

contains three terms. A term can take the following forms:

⟨a value⟩	printed as a (signed, decimal) number
⟨text in double-quotes⟩	printed as text
(⟨rule⟩) ⟨value⟩	printed according to some special rule

Inform provides a stock of special printing rules built-in, and also allows the programmer to create new ones. The most important rules are:

`(char)`	print out the character which this is the ZSCII code for
`(string)`	print out this string
`(address)`	print out the text of this dictionary word
`(name)`	print out the name of this object (see §3)

Games compiled with the Inform library have several other printing rules built-in (like `print (The) ...`), but as these aren't part of the Inform language as such they will be left until §26.

△ `print (string)` . . . requires a little explanation. Of the following lines, the first two print out "Hello!" but the third prints only a mysterious number:

```
    print (string) "Hello!";
    x = "Hello!"; print (string) x;
    x = "Hello!"; print x;
```

This is because strings are internally represented by mysterious numbers. `print (string)` means "interpret this value as the mysterious number of a string, and print out that string": it is liable to give an error, or in some cases print gibberish, if applied to a value which isn't the mysterious number of any string.

· · · · ·

Any Inform program can define its own printing rules simply by providing a routine whose name is the same as that of the rule. For example, the following pair of routines provides for printing out a value as a four-digit, unsigned hexadecimal number:

```
[ hex x y;
  y = (x & $7f00) / $100;
  if (x<0) y = y + $80;
  x = x & $ff;
  print (hexdigit) y/$10, (hexdigit) y,
        (hexdigit) x/$10, (hexdigit) x;
];
[ hexdigit x;
  x = x % $10;
  switch (x) {
      0 to 9: print x;
      10: print "a";  11: print "b";  12: print "c";
      13: print "d";  14: print "e";  15: print "f";
  }
];
```

You can paste these two routines into any Inform source code to make these new printing rules `hex` and `hexdigit` available to that code. For example, `print (hex) 16339;` will then print up "3fd3", and `print (hex) -2;` will print "fffe". Something to look out for is that if you inadvertently write

```
    print hex(16339);
```

then the text printed will be "3fd31", with a spurious 1 on the end: because you've printed out the return value of the routine `hex`, which was `true`, or in other words 1.

§1.13 Other printing statements

Besides `print` and `print_ret`, several other statements can also be used for printing.

```
new_line
```

prints a new-line, otherwise known as a carriage return (named for the carriage which used to move paper across a typewriter). This is equivalent to

```
print "^"
```

but is a convenient abbreviation. Similarly,

```
spaces ⟨number⟩
```

prints a sequence of the given number of spaces.

```
box ⟨string1⟩ ...⟨stringn⟩
```

displays a reverse-video panel in the centre of the screen, containing each string on its own line. For example, the statement

```
box "Passio domini nostri" "Jesu Christi Secundum" "Joannem";
```

displays the opening line of the libretto to Arvo Pärt's 'St John Passion':

```
Passio domini nostri
Jesu Christi Secundum
Joannem
```

. . . . .

Text is normally displayed in an unembellished but legible type intended to make reading paragraphs comfortable. Its actual appearance will vary from machine to machine running the story file. On most machines, it will be displayed using a "font" which is variably-pitched, so that for example a "w" will be wider on-screen than an "i". Such text is much easier to read, but makes it difficult to print out diagrams. The statement

```
print "+------------+
      ^+   Hello    +
      ^+------------+^";
```

will print something irregular if the letters in "Hello" and the characters "-", "+" and " " (space) do not all have the same width on-screen:

```
+------------+
+  Hello    +
+------------+
```

Because one sometimes does want to print such a diagram, to represent a sketch-map, say, or to print out a table, the statement `font` is provided:

```
font on
font off
```

`font off` switches into a fixed-pitch display style. It is now guaranteed that the interpreter will print all characters at the same width. `font on` restores the usual state of affairs.

In addition, you can choose the type style from a small set of possibilities: roman (the default), boldface, underlined or otherwise emphasized (some interpreters will use italic for this), reverse-colour:

```
style roman
style bold
style underline
style reverse
```

"Reverse colour" would mean, for instance, yellow on blue if the normal text appearance happened to be blue on yellow. An attempt will be made to approximate these effects whatever kind of machine is running the story file.

△ Changes of foreground and background colours, so memorably used in games like Adam Cadre's 'Photopia', can be achieved with care using Z-machine assembly language. See §42.

§1.14 Generating random numbers

Inform provides a small stock of functions ready-defined, but which are used much as other routines are. As most of these concern objects, only one will be give here: `random`, which has two forms:

```
random(N)
```

returns a random number in the range $1, 2, \ldots,$ `N`, each of these outcomes being (in theory) equally likely. `N` should be a positive number, between 1 and 32,767, for this to work properly.

`random(`⟨two or more constant quantities, separated by commas⟩`)`

returns a random choice from the given selection of constant values. Thus,

```
print (string) random("red", "blue", "green", "violet");
```

prints the name of one of these four colours at random. Likewise,

```
print random(13, 17);
```

has a 50% chance of printing "13", and a 50% chance of printing "17".

△ Random numbers are produced inside the interpreter by a "generator" which is not truly random in its behaviour. Instead, the sequence of numbers it produces depend on a value inside the generator called the "seed". Setting this value is called "seeding the random-number generator". Whenever it has the same seed value, the same sequence of `random` values will be harvested. This is called "pseudo-randomness". An interpreter normally goes to some effort to start up with a seed value which is unpredictable: say, the current time of day in milliseconds. Because you just might want to make it predictable, though, you can change the seed value within the story file by calling `random(N)` for a negative number `N` of your own choosing. (This returns zero.) Seeding is sometimes useful to test a game which contains random events or delays. For example if you have release 29 of Infocom's game 'Enchanter', you may be surprised to know that you can type "#random 14" into it, after which its random events will be repeatable and predictable. Internally, 'Enchanter' does this by calling `random(-14)`.

§1.15 *Deprecated ways to jump around*

There are four statements left which control the flow of execution, but one should try to avoid using any of them if, for instance, a `while` loop would do the job more tidily. Please stop reading this section now.

△ Oh very well. 'Deprecated' is too strong a word, anyway, as there are circumstances where jumping is justified and even elegant: to break out of several loops at once, for instance, or to construct a finite state machine. It's the *gratuitous* use of jumping which is unfortunate, as this rapidly decreases the legibility of the source code.

△ The `jump` statement transfers execution to some named place in the same routine. (Some programming languages call this `goto`.) To use `jump` a notation is needed to mark particular places in the source code. Such markers are called "labels". For example, here is a never-ending loop made by hand:

```
[ Main i;
  i=1;
  .Marker;
  print "I have now printed this ", i++, " times.^";
  jump Marker;
];
```

This routine has one label, `Marker`. A statement consisting only of a full stop and then an identifier means "put a label here and call it this". Like local variables, labels belong to the routines defining them and cannot be used by other routines.

△ The `quit` statement ends interpretation of the story file immediately, as if a return had taken place from the `Main` routine. This is a drastic measure, best reserved for error conditions so awful that there is no point carrying on.

△△ An Inform story file has the ability to save a snapshot of its entire state and to restore back to that previous state. This snapshot includes values of variables, the point where code is currently being executed, and so on. Just as we cannot know if the universe is only six thousand years old, as creationists claim, having been endowed by God with a carefully faked fossil record, so an Inform story file cannot know if it has been executing all along or if it was only recently restarted. The statements required are `save` and `restore`:

```
save ⟨label⟩
restore ⟨label⟩
```

This is a rare example of an Inform feature which may depend on the host machine's state of health: for example, if all disc storage is full, then `save` will fail. It should always be assumed that these statements may well fail. A `jump` to the label provided occurs if the operation has been a success. This is irrelevant in the case of a `restore` since, if all has gone well, execution is now resuming from the successful branch of the `save` statement: because that is where execution was when the state was saved.

△△ If you don't mind using assembly language (see §42), you can imitate exceptions, in the sense of programming languages like Java. The opcodes you would need are `@throw` and `@catch`.

• **REFERENCES**
If you need to calculate with integers of any size then the restriction of Inform numbers to the range −32,768 to 32,767 is a nuisance. The function library `"longint.h"`, by Chris Hall and Francis Irving, provides routines to calculate with signed and unsigned

4-byte integers, increasing the range to about ±2,147,000,000. •L. Ross Raszewski's function library `"ictype.h"` is an Inform version of the ANSI C routines `"ctype.h"`, which means that it contains routines to test whether a given character is lower or upper case, or is a punctuation symbol, and so forth. (See also the same author's `"istring.h"`.)

§2 The state of play

§2.1 *Directives construct things*

Every example program so far has consisted only of a sequence of routines, each within beginning and end markers `[` and `]`. Such routines have no way of communicating with each other, and therefore of sharing information with each other, except by calling each other back and forth. This arrangement is not really suited to a large program whose task may be to simulate something complicated, such as the world of an adventure game: instead, some central registry of information is needed, to which all routines can have access. In the author's game 'Curses', centrally-held information ranges from the current score, held in a single variable called `score`, to Madame Sosostris's tarot pack, which uses an array of variables representing the cards on the pack, to a slide-projector held as an "object": a bundle of variables and routines encoding the relevant rules of the game, such as that the whitewashed wall is only lit up when the slide projector is switched on.

Every Inform source program is a list of constructions, made using commands called "directives". These are quite different from the statements inside routines, because directives create something at compilation time, whereas statements are only instructions for the interpreter to follow later, when the story file is being played.

In all there are 38 Inform directives, but most of them are seldom used, or else are just conveniences to help you organise your source code: for instance `Include` means "now include another whole file of source code here", and there are directives for "if I've set some constant at the start of the code, then don't compile this next bit" and so on. The 10 directives that matter are the ones creating data structures, and here they are:

```
[          Array      Attribute   Class      Constant
Extend     Global     Object      Property   Verb
```

The directive written `[`, meaning "construct a routine containing the following statements, up to the next `]`", was the subject of §1. The four directives to do with objects, `Attribute`, `Class`, `Object` and `Property`, will be the subject of

§3. The two directives to do with laying out grammar, `Verb` and `Extend`, are intimately tied up with the needs of adventure games using the Inform library, and are useless for any other purpose, so these are left until §30. That leaves just `Array`, `Constant` and `Global`.

§2.2 *Constants*

The simplest construction you can make is of a `Constant`. The following program, an unsatisfying game of chance, shows a typical usage:

```
Constant MAXIMUM_SCORE = 100;
[ Main;
  print "You have scored ", random(MAXIMUM_SCORE),
      " points out of ", MAXIMUM_SCORE, ".^";
];
```

The maximum score value is used twice in the routine `Main`. The resulting story file is exactly the same as it would have been if the constant definition were not present, and `MAXIMUM_SCORE` were replaced by 100 in both places where it occurs. But the advantage of using `Constant` is that it makes it possible to change this value from 100 to, say, 50 with only a single change to the source code, and it makes the source code more legible.

People often write the names of constants in full capitals, but this is not compulsory. Another convention is that the = sign, which is optional, is often left out if the value is a piece of text rather than a number. If no value is specified for a constant, as in the line

```
Constant BETA_TEST_VERSION;
```

then the constant is created with value 0.

A constant can be used from anywhere in the source code *after* the line on which it is declared. Its value cannot be altered.

§2.3 *Global variables*

The variables in §1 were all "local variables", each owned privately by its own routine, inaccessible to the rest of the program and destroyed as soon as the routine stops. A "global variable" is permanent and its value can be used or altered from every routine.

The directive for declaring a global variable is `Global`. For example:

```
Global score = 36;
```

This creates a variable called `score`, which at the start of the program has the value 36. (If no initial value is given, it starts with the value 0.)

A global variable can be altered or used from anywhere in the source code *after* the line on which it is declared.

§2.4 *Arrays*

An "array" is an indexed collection of variables, holding a set of numbers organised into a sequence. To see why this useful, suppose that a pack of cards is to be simulated. You could define 52 different variables with `Global`, with names like `Ace_of_Hearts`, to hold the position of each card in the pack: but then it would be very tiresome to write a routine to shuffle them around.

Instead, you can declare an array:

```
Array pack_of_cards --> 52;
```

which creates a stock of 52 variables, called the "entries" of the array, and referred to in the source code as

```
pack_of_cards-->0   pack_of_cards-->1   ...   pack_of_cards-->51
```

and the point of this is that you can read or alter the variable for card number `i` by calling it `pack_of_cards-->i`. Here is an example program, in full, for shuffling the pack:

```
Constant SHUFFLES = 100;
Array pack_of_cards --> 52;
[ ExchangeTwo x y z;
  !   Randomly choose two different numbers between 0 and 51:
  while (x==y) {
      x = random(52) - 1; y = random(52) - 1;
  }
  z = pack_of_cards-->x; pack_of_cards-->x = pack_of_cards-->y;
  pack_of_cards-->y = z;
];
[ Card n;
  switch(n%13) {
        0: print "Ace";
        1 to 9: print n%13 + 1;
        10: print "Jack";  11: print "Queen";  12: print "King";
    }
    print " of ";
    switch(n/13) {
        0: print "Hearts"; 1: print "Clubs";
        2: print "Diamonds"; 3: print "Spades";
    }
  ];
  [ Main i;
    !   Create the pack in "factory order":
    for (i=0:i<52:i++) pack_of_cards-->i = i;
    !   Exchange random pairs of cards for a while:
    for (i=1:i<=SHUFFLES:i++) ExchangeTwo();
    print "The pack has been shuffled into the following order:^";
    for (i=0:i<52:i++)
        print (Card) pack_of_cards-->i, "^";
  ];
```

The cards are represented by numbers in the range 0 (the Ace of Hearts) to 51 (the King of Spades). The pack itself has 52 positions, from position 0 (top) to position 51 (bottom). The entry `pack_of_cards-->i` holds the number of the card in position `i`. A new pack as produced by the factory would come with Ace of Hearts on top (card 0 in position 0), running down to the King of Spades on the bottom (card 51 in position 51).

△ A hundred exchanges is only just enough. Redefining `SHUFFLES` as 10,000 takes a lot longer, while redefining it as 10 makes for a highly suspect result. Here is a more efficient method of shuffling (contributed by Dylan Thurston), perfectly random in just 51 exchanges.

```
pack_of_cards-->0 = 0;
for (i=1:i<52:i++) {
    j = random(i+1) - 1;
    pack_of_cards-->i = pack_of_cards-->j; pack_of_cards-->j = i;
}
```

· · · · ·

In the above example, the array entries are all created containing 0. Instead, you can give a list of constant values. For example,

```
Array small_primes --> 2 3 5 7 11 13;
```

is an array with six entries, `small_primes-->0` to `small_primes-->5`, initially holding 2, 3, 5, 7, 11 and 13.

The third way to create an array gives some text as an initial value, occasionally useful because one popular use for arrays is as "strings of characters" or "text buffers". For instance:

```
Array players_name --> "Frank Booth";
```

is equivalent to the directive:

```
Array players_name --> 'F' 'r' 'a' 'n' 'k' ' ' 'B' 'o' 'o' 't' 'h';
```

Literal text like `"Frank Booth"` is a constant, not an array, and you can no more alter its lettering than you could alter the digits of the number `124`. The array `players_name` is quite different: its entries can be altered. But this means it cannot be treated as if it were a string constant, and in particular can't be printed out with `print (string)`. See below for the right way to do this.

• **WARNING**

In the pack of cards example, the entries are indexed `0` to `51`. It's therefore impossible for an interpreter to obey the following statement:

```
pack_of_cards-->52 = 0;
```

because there is no entry `52`. Instead, the following message will be printed when it plays:

> [** Programming error: tried to write to `-->52` in the array "`pack_of_cards`", which has entries 0 up to 51 **]

Such a mistake is sometimes called breaking the bounds of the array.

· · · · ·

The kind of array constructed above is sometimes called a "word array". This is the most useful kind and many game designers never use the other three varieties at all.

△ The first alternative is a "byte array", which is identical except that its entries can only hold numbers in the range 0 to 255, and that it uses the notation `->` instead of `-->`. This is only really useful to economise on memory usage in special circumstances, usually when the entries are known to be characters, because ZSCII character codes are all between 0 and 255. The "Frank Booth" array above could safely have been a byte array.

△ In addition to this, Inform provides arrays which have a little extra structure: they are created with the 0th entry holding the number of entries. A word array with this property is called a `table`; a byte array with this property is a `string`. For example, the table

```
Array continents table 5;
```

has six entries: `continents-->0`, which holds the number 5, and further entries `continents-->1` to `continents-->5`. If the program changed `continents-->0` this would *not* magically change the number of array entries, or indeed the number of continents.

△△ One main reason you might want some arrangement like this is to write a general routine which can be applied to any array. Here is an example using `string` arrays:

```
Array password string "danger";
Array phone_number string "0171-930-9000";
...
print "Please give the password ", (PrintStringArray) password,
    " whenever telephoning Universal Exports at ",
    (PrintStringArray) phone_number, ".";
...
[ PrintStringArray the_array i;
  for (i=1: i<=the_array->0: i++) print (char) the_array->i;
];
```

Such routines should be written with care, as the normal checking of array bounds isn't performed when arrays are accessed in this indirect sort of fashion, so any mistake you make may cause trouble elsewhere and be difficult to diagnose.

△△ With all data structures (i.e., with objects, strings, routines and arrays) Inform calls by reference, not by value. So, for instance:

```
[ DamageStringArray the_array i;
  for (i=1: i<=the_array->0: i++) {
      if (the_array->i == 'a' or 'e' or 'i' or 'o' or 'u')
          the_array->i = random('a', 'e', 'i', 'o', 'u');
      print (char) the_array->i;
  }
];
```

means that the call `DamageStringArray(password_string)` will not just print (say) "dungor" but also alter the one and only copy of `password_string` in the story file.

§2.5 *Reading into arrays from the keyboard*

Surprisingly, perhaps, given that Inform is a language for text adventure games, support for reading from the keyboard is fairly limited. A significant difference of approach between Inform and many other systems for interactive fiction is that mechanisms for parsing textual commands don't come built into the language itself. Instead, game designers use a standard Inform parser program which occupies four and a half thousand lines of Inform code.

Reading single key-presses, perhaps with time-limits, or for that matter reading the mouse position and state (in a Version 6 game) requires the use of Inform assembly language: see §42.

A statement called `read` does however exist for reading in a single line of text and storing it into a byte array:

```
read text_array 0;
```

You must already have set `text_array->0` to the maximum number of characters you will allow to be read. (If this is N, then the array must be defined with at least $N + 3$ entries, the last of which guards against overruns.) The number of characters actually read, not counting the carriage return, will be placed into `text_array->1` and the characters themselves into entries from `text_array->2` onwards. For example, if the player typed "GET IN":

`->0`	1	2	3	4	5	6	7
max	*characters*	*text typed by player, reduced to lower case*					
60	6	`'g'`	`'e'`	`'t'`	`' '`	`'i'`	`'n'`

The following echo chamber demonstrates how to read from this array:

```
Array text_array -> 63;
[ Main c x;
  for (::) {
      print "^> ";
      text_array->0 = 60;
      read text_array 0;
      for (x=0:x<text_array->1:x++) {
          c = text_array->(2+x);
          print (char) c; if (c == 'o') print "h";
      }
  }
];
```

· · · · ·

△ read can go further than simply reading in the text: it can work out where the words start and end, and if they are words registered in the story file's built-in vocabulary, known as the "dictionary". To produce all this information, read needs to be supplied with a second array:

```
    read text_array parse_array;
```

read not only stores the text (just as above) but breaks down the line into a sequence of words, in which commas and full stops count as separate words in their own right. (An example is given in Chapter IV, §30.) In advance of this parse_array->0 must have been set to W, the maximum number of words you want to parse. Any further text will be ignored. parse_array should have at least $4W + 2$ entries, because parse_array->1 is set to the actual number of words parsed, and then a four-entry block is written into the array for each word parsed. Numbering the words as 1, 2, 3, . . ., the number of letters in word n is written into parse_array->(n*4), and the position of the start of the word in text_array. The dictionary value of the word, or zero if it isn't recognised, is stored as parse_array-->(n*2-1). The corresponding parsing array to the previous text array, for the command "GET IN", looks like so:

->0	1	2	3	4	5	6	7	8	9
max	*words*	*first word*				*second word*			
10	2	'get'		2	3	'in'		5	2

In this example both words were recognised. The word "get" began at position ->2 in the text array, and was 3 characters long; the word "in" began at ->5 and was 2 characters long. The following program reads in text and prints back an analysis:

```
    Array text_array -> 63;
    Array parse_array -> 42;
    [ Main w x length position dict;
      w = 'mary'; w = 'had'; w = 'a//'; w = 'little'; w = 'lamb';
      for (::) {
          print "^> ";
          text_array->0 = 60; parse_array->0 = 10;
          read text_array parse_array;
          for (w=1:w<=parse_array->1:w++) {
              print "Word ", w, ": ";
              length = parse_array->(4*w);
              position = parse_array->(4*w + 1);
              dict = parse_array-->(w*2-1);
              for (x=0:x<length:x++)
                  print (char) text_array->(position+x);
```

```
                print " (length ", length, ")";
                if (dict) print " equals '", (address) dict, "'^";
                    else print " is not in the dictionary^";
            }
        }
    ];
```

Note that the pointless-looking first line of `Main` adds five words to the dictionary. The result is:

```
>MARY, hello
Word 1: mary (length 4) equals 'mary'
Word 2: , (length 1) is not in the dictionary
Word 3: hello (length 5) is not in the dictionary
```

△ What goes into the dictionary? The answer is: any of the words given in the `name` of an object (see §3), any of the verbs and prepositions given in grammar by `Verb` and `Extend` directives (see §26), and anything given as a dictionary-word constant. The last is convenient because it means that code like

```
    if (parse_array -->(n*2-1)) == 'purple';
```

does what it looks as if it should. When compiling this line, Inform automatically adds the word "purple" to the story file's dictionary, so that any `read` statement will recognise it.

• REFERENCES

Evin Robertson's function library `"array.h"` provides some simple array-handling utilities. •L. Ross Raszewski's function library `"istring.h"` offers Inform versions of the ANSI C string-handling routines, including `strcmp()`, `strcpy()` and `strcat()`. The further extension `"znsi.h"` allows the printing out of string arrays with special escape sequences like `[B` interpreted as "bold face." (See also the same author's `"ictype.h"`.) •Adam Cadre's function library `"flags.h"` manages an array of boolean values (that is, values which can only be `true` or `false`) so as to use only one-sixteenth as much memory as a conventional array, though at some cost to speed of access.

§3 Objects and classes

Objects make up the substance of the world.
—Ludwig Wittgenstein (1889–1951), *Tractatus*

§3.1 Objects, classes, metaclasses and `nothing`

In Inform, objects are little bundles of routines and variables tied up together. Dividing up the source code into objects is a good way to organise any large, complicated program, and makes particular sense for an adventure game, based as it usually is on simulated items and places. One item in the simulated world corresponds to one "object" in the source code. Each of these pieces of the story file should take responsibility for its own behaviour, so for instance a brass lamp in an adventure game might be coded with an Inform object called `brass_lamp`, containing all of the game rules which affect the lamp. Then again, objects *do* have to interact.

> *In West Pit*
> You are at the bottom of the western pit in the twopit room. There is a large hole in the wall about 25 feet above you.
> There is a tiny little plant in the pit, murmuring "Water, water, . . ."

In this moment from 'Advent', the player is more or less openly invited to water the plant. There might be many ways to bring water to it, or indeed to bring liquids other than water, and the rules for what happens will obviously affect the bottle carrying the water, the water itself and the plant. Where should the rules appear in the source code? Ideally, the `plant` object should know about growing and the `bottle` about being filled up and emptied. Many people feel that the most elegant approach is for the `bottle`, or any other flask, not to interfere with the plant directly but instead to send a "message" to the `plant` to say "you have been watered". It's then easy to add other solutions to the same puzzle: a successful rain dance, for instance, could also result in a "you have been watered" message being sent to `plant`. The whole behaviour of the `plant` could be altered without needing even to look at the rain-dance or the bottle source code. Objects like this can frequently be cut out of the source code for one game and placed into another, still working.

This traffic of messages between objects goes on continuously in Inform-compiled adventure games. When the player tries to pick up a wicker cage, the Inform library sends a message to the cage object asking if it minds being picked up. When the player tries to go north from the Hall of Mists, the library sends a message to an object called `Hall_of_Mists` asking where that would lead, and so on.

· · · · ·

Typical large story files have so many objects ('Curses', for instance, has 550) that it is convenient to group similar objects together into "classes". For instance, 'Advent' began life as a simulation of part of the Mammoth and Flint Ridge cave system of Kentucky, caves which contain many, many dead ends. The Inform source code could become very repetitive without a class like so:

```
Class DeadEndRoom
 with short_name "Dead End",
      description "You have reached a dead end.",
      cant_go "You'll have to go back the way you came.";
```

Leaving the exact syntax for later, this code lays out some common features of dead ends. All kinds of elegant things can be done with classes if you like, or not if you don't.

Objects can belong to several different classes at once, and it is sometimes convenient to be able to check whether or not a given object belongs to a given class. For instance, in adventures compiled with the Inform library, a variable called `location` always holds the player's current position, so it might be useful to do this:

```
if (location ofclass DeadEndRoom) "Perhaps you should go back.";
```

· · · · ·

Items and places in an Inform game always belong to at least one class, whatever you define, because they always belong to the "metaclass" `Object`. ("Meta" from the Greek for "beyond".) As we shall see, all of the objects explicitly written out in Inform source code always belong to `Object`. Though you seldom need to know this, there are three other metaclasses. Classes turn out to be a kind of object in themselves, and belong to `Class`. (So that `Class` belongs to itself. If you enjoy object-oriented programming, this will give you a warm glow inside). And although you almost never need to know or care, routines as in §1 are internally considered as a kind of object, of class `Routine`; while strings in double-quotes are likewise of class `String`. That's all, though. If in doubt, you can always find out what kind of object `obj` is, using the built-in function `metaclass(obj)`. Here are some example values:

```
metaclass("Violin Concerto no. 1") == String
metaclass(Main) == Routine
metaclass(DeadEndRoom) == Class
metaclass(silver_bars) == Object
```

Classes are useful and important, but for most game-designing purposes it's now safe to forget all about metaclasses.

It turns out to be useful to have a constant called `nothing` and meaning "no object at all". It really does mean that: `nothing` *is not an object*. If you try to treat it as one, many programming errors will be printed up when the story file is played.

△ If `X` is not an object at all, then `metaclass(X)` is `nothing`, and in particular `metaclass(nothing)` is `nothing`.

§3.2 *The object tree*

Objects declared in the source code are joined together in an "object tree" which grows through every Inform story file. Adventure games use this to represent which items are contained inside which other items.

It's conventional to think of this as a sort of family tree without marriages. Each object has a parent, a child and a sibling. Such a relation is always either another object in the tree, or else `nothing`, so for instance the parent of an orphan would be `nothing`. Here is a sample object tree:

$$\begin{tikzcd}
\texttt{Meadow} \arrow[d, "\mathit{child}"] & & & & \\
\texttt{Mailbox} \arrow[d, "\mathit{child}"] \arrow[r, "\mathit{sibling}"] & \texttt{Player} \arrow[d, "\mathit{child}"] & & & \\
\texttt{Note} & \texttt{Sceptre} \arrow[r, "\mathit{sibling}"] & \texttt{Bottle} \arrow[r, "\mathit{sibling}"] & \texttt{Torch} \arrow[r, "\mathit{sibling}"] \arrow[d, "\mathit{child}"] & \texttt{Stone} \\
& & & \texttt{Battery} &
\end{tikzcd}$$

The `Mailbox` and `Player` are both children of the `Meadow`, which is their parent, but only the `Mailbox` is *the* child of the `Meadow`. The `Stone` is the sibling of the `Torch`, which is the sibling of the `Bottle`, and so on.

Inform provides special functions for reading off positions in the tree: `parent`, `sibling` do the obvious things, and `child` gives the first child: in addition there's a function called `children` which counts up how many children an object has (note that grandchildren don't count as children). Here are some sample values:

```
parent ( Mailbox )  == Meadow
children ( Player ) == 4
child ( Player )    == Sceptre
child ( Sceptre )   == nothing
sibling ( Torch )   == Stone
```

• **WARNING**

It is incorrect to apply these functions to the value `nothing`, since it is *not an object*. If you write a statement like `print children(x);` when the value of `x` happens to be `nothing`, the interpreter will print up the message:

[** Programming error: tried to find the "children" of nothing **]

You get a similar error message if you try to apply these tree functions to a routine, string or class.

§3.3 *Declaring objects 1: setting up the tree*

Objects are made with the directive `Object`. Here is a portion of source code, with the bulk of the definitions abbreviated to "...":

```
Object Bucket ...
Object -> Starfish ...
Object -> Oyster ...
Object -> -> Pearl ...
Object -> Sand ...
```

The resulting tree looks a little like the source code turned on its side:

$$\begin{tikzcd}
\texttt{Bucket} \arrow[d, "\mathit{child}"] & & \\
\texttt{Starfish} \arrow[r, "\mathit{sibling}"] & \texttt{Oyster} \arrow[r, "\mathit{sibling}"] \arrow[d, "\mathit{child}"] & \texttt{Sand} \\
& \texttt{Pearl} &
\end{tikzcd}$$

The idea is that if no arrows `->` are given in the `Object` definition, then the object has no parent. If one `->` is given, then the object is made a child of the last object defined with no arrows; if two are given, it's made a child of the last object defined with only one arrow; and so on.

An object definition consists of a "head" followed by a "body", itself divided into "segments", though there the similarity with caterpillars ends. The head takes the form:

```
Object ⟨arrows⟩ ⟨identifier⟩ "textual name" ⟨parent⟩
```

(1) The ⟨arrows⟩ are as described above. Note that if one or more arrows are given, that automatically specifies what object this is the child of, so a ⟨parent⟩ cannot be given as well.
(2) The ⟨identifier⟩ is what the object can be called inside the program, in the same way that a variable or a routine has a name.
(3) The `"textual name"` can be given if the object's name ever needs to be printed by the program when it is running.
(4) The ⟨parent⟩ is an object which this new object is to be a child of. This is an alternative to supplying arrows.

All four parts are optional, so that even this bare directive is legal:

```
Object;
```

though it makes a nameless and featureless object which is unlikely to be useful.

§3.4 *Tree statements:* `move`, `remove`, `objectloop`

The positions of objects in the tree are by no means fixed: objects are created in a particular formation but then shuffled around extensively during the story file's execution. (In an adventure game, where the objects represent items and rooms, objects are moved across the tree whenever the player picks something up or moves around.) The statement

```
move ⟨object⟩ to ⟨object⟩
```

moves the first-named object to become a child of the second-named one. All of the first object's own children "move along with it", i.e., remain its own children.

For instance, starting from the tree as shown in the diagram of §3.2 above,

```
move Bottle to Mailbox;
```

results in the tree

$$\begin{tikzcd}
\texttt{Meadow} \arrow[d, "child"] \\
\texttt{Mailbox} \arrow[d, "child"] \arrow[rr, "sibling"] & & \texttt{Player} \arrow[d, "child"] \\
\texttt{Bottle} \arrow[r, "sibling"] & \texttt{Note} & \texttt{Sceptre} \arrow[r, "sibling"] & \texttt{Torch} \arrow[r, "sibling"] \arrow[d, "child"] & \texttt{Stone} \\
& & & \texttt{Battery}
\end{tikzcd}$$

When an object becomes the child of another in this way, it always becomes the "eldest" child: that is, it is the new `child()` of its parent, pushing the previous children over into being its siblings. In the tree above, `Bottle` has displaced `Note` just so.

You can only `move` one object in the tree to another: you can't

```
move Torch to nothing;
```

because `nothing` *is not an object*. Instead, you can detach the `Torch` branch from the tree with

```
remove Torch;
```

and this would result in:

$$\begin{tikzcd}
\texttt{Meadow} \arrow[d, "child"] & & & & \texttt{Torch} \arrow[d, "child"] \\
\texttt{Mailbox} \arrow[d, "child"] \arrow[rr, "sibling"] & & \texttt{Player} \arrow[d, "child"] & & \texttt{Battery} \\
\texttt{Bottle} \arrow[r, "sibling"] & \texttt{Note} & \texttt{Sceptre} \arrow[r, "sibling"] & \texttt{Stone}
\end{tikzcd}$$

The "object tree" is often fragmented like this into many little trees, and is not so much a tree as a forest.

- **WARNING**

It would make no sense to have a circle of objects each containing the next, so if you try to `move Meadow to Note;` then you'll only move the interpreter to print up:

> [** Programming error: tried to move the Meadow to the note, which would make a loop: Meadow in note in mailbox in Meadow **]

· · · · ·

Since objects move around a good deal, it's useful to be able to test where an object currently is, and the condition `in` is provided for this. For example,

```
    Bottle in Mailbox
```

is true if and only if the `Bottle` is one of the *direct* children of the `Mailbox`. (`Bottle in Mailbox` is true, but `Bottle in Meadow` is false.) Note that

```
    X in Y
```

is only an abbreviation for `parent(X)==Y` but it occurs so often that it's worth having. Similarly, `X notin Y` means `parent(X)~=Y`. `X` has to be a bona-fide member of the object tree, but `Y` is allowed to be `nothing`, and testing `X in nothing` reveals whether or not `X` has been removed from the rest of the tree.

· · · · ·

The remaining loop statement left over from §1 is `objectloop`.

    objectloop(⟨variable-name⟩) ⟨statement⟩

runs through the ⟨statement⟩ once for each object in the game, putting each object in turn into the variable. For example,

```
    objectloop(x) print (name) x, "^";
```

prints out a list of the textual names of every object in the game. More powerfully, any condition can be written in the brackets, as long as it begins with a variable name.

```
    objectloop (x in Mailbox) print (name) x, "^";
```

prints the names only of those objects which are direct children of the `Mailbox` object.

The simple case where the condition reads "⟨variable⟩ `in` ⟨object⟩" is handled in a faster and more predictable way than other kinds of `objectloop`: the loop variable is guaranteed to run through the children of the object in sibling order, eldest down to youngest. (This is faster because it doesn't waste time considering every object in the game, only the children.) If the condition is not in this form then no guarantee is made as to the order in which the objects are considered.

• WARNING

When looping over objects with `in`, it's not safe to move these same objects around: this is like trying to cut a branch off an elm tree while sitting on it. Code like this:

```
objectloop(x in Meadow) move x to Sandy_Beach;
```

looks plausible but is not a safe way to move everything in the Meadow, and will instead cause the interpreter to print up

> [** Programming error: objectloop broken because the object mailbox was moved while the loop passed through it **]

Here is a safer way to move the meadow's contents to the beach:

```
while (child(Meadow)) move child(Meadow) to Sandy_Beach;
```

This works because when the Meadow has no more children, its `child` is then `nothing`, which is the same as `false`.

△ But it moves the eldest child first, with the possibly undesirable result that the children arrive in reverse order (Mailbox and then Player, say, become Player and then Mailbox). Here is an alternative, moving the youngest instead of the eldest child each time, which keeps them in the same order:

```
while (child(Meadow)) {
    x = child(Meadow); while (sibling(x)) x = sibling(x);
    move x to Sandy_Beach;
}
```

Keeping children in order can be worth some thought when game designing. For instance, suppose a tractor is to be moved to a farmyard in which there is already a barn. The experienced game designer might do this like so:

```
move tractor to Farmyard;
move barn to Farmyard;
```

Although the barn was in the farmyard already, the second statement wasn't redundant: because a moved object becomes the eldest child, the statement does this:

$$\begin{tikzcd}
\texttt{Farmyard} \arrow[d, "\mathit{child}"] & & & \texttt{Farmyard} \arrow[d, "\mathit{child}"] & \\
\texttt{tractor} \arrow[r, "\mathit{sibling}"] & \texttt{barn} & \Longrightarrow & \texttt{barn} \arrow[r, "\mathit{sibling}"] & \texttt{tractor}
\end{tikzcd}$$

And this is desirable because the ordering of paragraphs in room descriptions tends to follow the ordering of items in the object tree, and the designer wants the barn mentioned before the tractor.

△ An `objectloop` range can be any condition so long as a named local or global variable appears immediately after the open bracket. This means that

```
objectloop (child(x) == nothing) ...
```

isn't allowed, because the first thing after the bracket is `child`, but a dodge to get around this is:

```
objectloop (x && child(x) == nothing) ...
```

The loop variable of an `objectloop` can never equal `false`, because that's the same as `nothing`, which isn't an object.

△△ The `objectloop` statement runs through all objects of metaclass `Object` or `Class`, but skips any `Routine` or `String`.

§3.5 *Declaring objects 2:* `with` *and* `provides`

So far `Objects` are just tokens with names attached which can be shuffled around in a tree. They become interesting when data and routines are attached to them, and this is what the body of an object definition is for. The body contains four different kinds of segments, introduced by the keywords:

```
with    has    class    private
```

These are all optional and can be given in any order.

△ They can even be given more than once: that is, there can be two or more of a given kind, which Inform will combine together as if they had been defined in one go. (This is only likely to be useful for automated Inform-writing programs.)

· · · · ·

The most important segment is `with`, which specifies variables and, as we shall see, routines and even arrays, to be attached to the object. For example,

```
Object magpie "black-striped bird"
  with wingspan, worms_eaten;
```

attaches two variables to the bird, one called `wingspan`, the other called `worms_eaten`. Commas are used to separate them and the object definition as a whole ends with a semicolon, as always. Variables of this kind are called properties, and are referred to in the source code thus:

```
magpie.wingspan
magpie.worms_eaten
```

Properties are just like global variables: any value you can store in a variable can be stored in a property. But note that

```
crested_grebe.wingspan
magpie.wingspan
```

are different and may well have different values, which is why the object whose wingspan it is (the magpie or the grebe) has to be named.

The property `wingspan` is said to be provided by both the `magpie` and `crested_grebe` objects, whereas an object whose `with` segment didn't name `wingspan` would not provide it. The dot `.` operator can only be used to set the value of a property which is provided by the object on the left of the dot: if not a programming error will be printed up when the story file is played.

The presence of a property can be tested using the `provides` condition. For example,

```
objectloop (x provides wingspan) ...
```

executes the code `...` for each object `x` in the program which is defined with a `wingspan` property.

△ Although the provision of a property can be tested, it can't be changed while the program is running. The *value* of `magpie.wingspan` may change, but not the *fact* that the magpie provides a `wingspan`.

△△ Some special properties, known as "common properties", can have their values *read* (but not changed) even for an object which doesn't provide them. All of the properties built into the Inform library are common properties. See §3.14.

· · · · ·

When the magpie is created as above, the initial values of

```
magpie.wingspan
magpie.worms_eaten
```

are both 0. To create the magpie with a given wingspan, we have to specify an initial value, which we do by giving it after the name, e.g.:

```
Object magpie "black-striped bird"
  with wingspan 5, worms_eaten;
```

The story file now begins with `magpie.wingspan` equal to 5, though `magpie.worms_eaten` still equal to 0.

· · · · ·

A property can contain a routine instead of a value. In the definition

```
Object magpie "black-striped bird"
  with name 'magpie' 'bird' 'black-striped' 'black' 'striped',
       wingspan 5,
       flying_strength [;
           return magpie.wingspan + magpie.worms_eaten;
       ],
       worms_eaten;
```

The value of `magpie.flying_strength` is given as a routine, in square brackets as usual. Note that the `Object` continues where it left off after the routine-end marker, `]`. Routines which are written in as property values are called "embedded" and are the way objects receive messages, as we shall see.

△ If, during play, you want to change the way a magpie's flying strength is calculated, you can simply change the value of its property:

```
magpie.flying_strength = ExhaustedBirdFS;
```

where `ExhaustedBirdFS` is the name of a routine to perform the new calculation.

△ Embedded routines are just like ordinary ones, with two exceptions:

(1) An embedded routine has no name of its own, since it is referred to as a property such as `magpie.flying_strength` instead.
(2) If execution reaches the `]` end-marker of an embedded routine, then it returns `false`, not `true` (as a non-embedded routine would).

△△ Properties can be arrays instead of variables. If two or more consecutive values are given for the same property, it becomes an array. Thus,

```
Object magpie "black-striped bird"
  with name 'magpie' 'bird' 'black-striped' 'black' 'striped',
       wingspan 5, worms_eaten;
```

You can't write `magpie.name` because there is no single value: rather, there is an `-->` array (see §2.4). This array must be accessed using two special operators, `.&` and `.#`, for the array and its length, as follows.

```
magpie.&name
```

means "the array held in magpie's `name` property", so that the actual name values are in the entries

```
magpie.&name-->0, magpie.&name-->1, ..., magpie.&name-->4
```

The size of this array can be discovered with

```
magpie.#name
```

which evaluates to the twice the number of entries, in this case, to 10. Twice the number of entries because that is the number of bytes in the array: people fairly often use property arrays as byte arrays to save on memory.

△ `name` is a special property created by Inform, intended to hold dictionary words which can refer to an object.

§3.6 *Declaring objects 3:* `private` *properties*

△ A system is provided for "encapsulating" certain properties so that only the object itself has access to them. These are defined by giving them in a segment of the object declaration called `private`. For instance,

```
Object sentry "sentry"
  private pass_number 16339,
  with challenge [ attempt;
           if (attempt == sentry.pass_number)
               "Approach, friend!";
           "Stand off, stranger.";
       ];
```

provides for two properties: `challenge`, which is public, and `pass_number`, which can be used only by the sentry's own embedded routines.

△△ This makes the `provides` condition slightly more interesting than it appeared in the previous section. The answer to the question of whether or not

```
sentry provides pass_number
```

depends on who's asking: this condition is true if it is tested in one of the sentry's own routines, and elsewhere false. A `private` property is so well hidden that nobody else can even know whether or not it exists.

§3.7 *Declaring objects 4:* `has` *and* `give`

In addition to properties, objects have flags attached, called "attributes". (Recall that flags are a limited form of variable which can only have two values, sometimes called set and clear.) Unlike property names, attribute names have to be declared before use with a directive like:

```
Attribute hungry;
```

Once this declaration is made, every object in the tree has a `hungry` flag attached, which is either `true` or `false` at any given time. The state can be tested with the `has` condition:

```
magpie has hungry
```

is true if and only if the magpie's `hungry` flag is currently set. You can also test if `magpie hasnt hungry`. There's no apostrophe in `hasnt`.

The magpie can now be born `hungry`, using the `has` segment in its declaration:

```
Object magpie "black-striped bird"
  with wingspan, worms_eaten
  has  hungry;
```

The `has` segment contains a list (without commas in between) of the attributes which are initially set: for instance, the steel grate in the Inform example game 'Advent' includes the line

```
  has  static door openable lockable locked;
```

The state of an attribute can be changed during play using the `give` statement:

```
give magpie hungry;
```

sets the magpie's `hungry` attribute, and

```
give magpie ~hungry;
```

clears it again. The `give` statement can take more than one attribute at a time, too:

```
give double_doors_of_the_horizon ~locked openable open;
```

means "clear `locked` and set `openable` and `open`".†

† The refrain from the prelude to Act I of Philip Glass's opera *Akhnaten* is "Open are the double doors of the horizon/ Unlocked are its bolts".

△△ An attribute can also have a tilde `~` placed in front in the `has` part of an object declaration, indicating "this is definitely not held". This is usually what would have happened anyway, except that class inheritance (see below) might have passed on an attribute: if so, this is how to get rid of it again. Suppose there is a whole class of steel grates like the one in 'Advent' mentioned above, providing for a dozen grates scattered through a game, but you also want a loose grate `L` whose lock has been smashed. If `L` belongs to the class, it will start the game with attributes making it locked like the others, because the class sets these automatically: but if you include `has ~lockable ~locked;` in its declaration, these two attributes go away again.

§3.8 *Declaring objects 5:* `class` *inheritance*

A class is a prototype design from which other objects are manufactured. These resulting objects are sometimes called instances or members of the class, and are said to inherit from it.

Classes are useful when a group of objects are to have common features. In the definition of the `magpie` above, a zoologically doubtful formula was laid out for flying strength:

```
flying_strength [;
    return magpie.wingspan + magpie.worms_eaten;
],
```

This formula ought to apply to birds in general, not just to magpies, and in the following definition it does:

```
Attribute flightless;
Class Bird
 with wingspan 7,
      flying_strength [;
          if (self has flightless) return 0;
          return self.wingspan + self.worms_eaten;
      ],
      worms_eaten;
Bird "ostrich" with wingspan 3, has flightless;
Bird "magpie" with wingspan 5;
Bird "crested grebe";
Bird "Great Auk" with wingspan 15;
Bird "early bird" with worms_eaten 1;
```

Facts about birds in general are now located in a class called `Bird`. Every example of a `Bird` automatically provides `wingspan`, a `flying_strength` routine and a count of `worms_eaten`. Notice that the Great Auk is not content with the average avian wingspan of 7, and insists on measuring 15 across. This is an example of inheritance from a class being over-ridden by a definition inside the object. The actual values set up are as follows:

`B`	`B.wingspan`	`B.worms_eaten`
ostrich	3	0
magpie	5	0
crested grebe	7	0
Great Auk	15	0
early bird	7	1

Note also the use of the special value `self` in the definition of `Bird`. It means "whatever bird I am": if the `flying_strength` routine is being run for the ostrich, then `self` means the ostrich, and so on.

The example also demonstrates a general rule: to create something, begin its declaration with the name of the class you want it to belong to: a plain `Object`, a `Class` or now a `Bird`.

△ Sometimes you need to specify that an object belongs to many classes, not just one. You can do this with the `class` segment of the definition, like so:

```
Object "goose that lays the golden eggs"
 class Bird Treasure;
```

This goose belongs to three classes: `Object` of course, as all declared objects do, but also `Bird` and `Treasure`. (It inherits from `Object` first and then `Bird` and then `Treasure`, attribute settings and property values from later-mentioned classes overriding earlier ones, so if these classes should give contradictory instructions then `Treasure` gets the last word.) You can also make class definitions have classes, or rather, pass on membership of other classes:

```
Class BirdOfPrey
class Bird
 with wingspan 15,
      people_eaten;
BirdOfPrey kestrel;
```

makes `kestrel` a member of both `BirdOfPrey` and of `Bird`. Dutiful apostles of object-oriented programming may want to call `BirdOfPrey` a "subclass" of `Bird`. Indeed, they may want to call Inform a "weakly-typed language with multiple-inheritance", or more probably a "shambles".

△△ For certain "additive" common properties, clashes between what classes say and what an instance says are resolved differently: see §5. Inform's built-in property `name` is one of these.

§3.9 *Messages*

Objects communicate with each other by means of messages. A message has a sender, a receiver and some parameter values attached, and it always produces a reply, which is just a single value. For instance,

```
x = plant.pour_over(cold_spring_water);
```

sends the message `pour_over` with a single parameter, `cold_spring_water`, to the object `plant`, and puts the reply value into `x`.

In order to receive this message, `plant` has to provide a `pour_over` property. If it doesn't, then the interpreter will print something like

[** Programming error: the plant (object number 21) has no property pour_over to send message **]

when the story file is played. The `pour_over` property will normally be a routine, perhaps this one:

```
pour_over [ liquid;
    remove liquid;
    switch(liquid) {
        oil: "The plant indignantly shakes the oil off its
              leaves and asks, ~Water?~";
        ...
    }
];
```

Inside such a routine, `self` means the object receiving the message and `sender` means the object which sent it. In a typical Inform game situation, `sender` will often be the object `InformLibrary`, which organises play and sends out many messages to items and places in the game, consulting them about what should happen next. Much of any Inform game designer's time is spent writing properties which receive messages from `InformLibrary`: `before`, `after`, `each_turn` and `n_to` among many others.

You can see all the messages being sent in a game as it runs using the debugging verb "messages": see §7 for details. This is the Inform version of listening in on police-radio traffic.

△ It was assumed above that the receiving property value would be a routine. But this needn't always be true. It can instead be: `nothing`, in which case the reply value is also `nothing` (which is the same as zero and the same as `false`). Or it can be an `Object` or a `Class`, in which case nothing happens and the object or class is sent back as the reply value. Or it can be a string in double-quotes, in which case the string is printed out, then a new-line is printed, and the reply value is `true`.

△ This can be useful. Here is approximately what happens when the Inform library tries to move the player northeast from the current room (the `location`) in an adventure game (leaving out some complications to do with doors):

```
if (location provides ne_to) {
    x = location.ne_to();
    if (x == nothing) "You can't go that way.";
    if (x ofclass Object) move player to x;
} else "You can't go that way.";
```

This neatly deals with all of the following cases:

```
Object Octagonal_Room "Octagonal Room"
  with ...
       ne_to "The way north-east is barred by an invisible wall!",
       w_to Courtyard,
       e_to [;
           if (Amulet has worn) {
               print "A section of the eastern wall suddenly parts
                      before you, allowing you into...^";
               return HiddenShrine;
           }
       ],
       s_to [;
           if (random(5) ~= 1) return Gateway;
           print "The floor unexpectedly gives way, dropping you
                  through an open hole in the plaster...^";
           return random(Maze1, Maze2, Maze3, Maze4);
       ];
```

Noteworthy here is that the `e_to` routine, being an embedded routine, returns `false` which is the same as `nothing` if the `]` end-marker is reached, so if the Amulet isn't being worn then there is no map connection east.

△△ The receiving property can even hold an array of values, in which case the message is sent to each entry in the array in turn. The process stops as soon as one of these entries replies with a value other than `nothing` or `false`. If every entry is tried and they all replied `nothing`, then the reply value sent back is `nothing`. (This is useful to the Inform library because it allows `before` rules to be accumulated from the different classes an object belongs to.)

§3.10 Passing messages up to the superclass

△ It fairly often happens that an instance of a class needs to behave almost, but not quite, as the class would suggest. For instance, suppose the following Treasure class:

```
Class Treasure
 with deposit [;
         if (self provides deposit_points)
             score = score + self.deposit_points;
         else score = score + 5;
         move self to trophy_case;
         "You feel a sense of increased esteem and worth.";
      ];
```

and we want to create an instance called Bat_Idol which flutters away, resisting deposition, but only if the room is dark:

```
Treasure Bat_Idol "jewelled bat idol"
  with deposit [;
          if (location == thedark) {
              remove self;
              "There is a clinking, fluttering sound!";
          }
          ...
       ];
```

In place of ..., what we want is all of the previous source code about depositing treasures. We could just copy it out again, but a much neater trick is to write:

```
self.Treasure::deposit();
```

Instead of sending the message deposit, we send the message Treasure::deposit, which means "what deposit would do if it used the value defined by Treasure". The double-colon :: is called the "superclass operator". (The word "superclass", in this context, is borrowed from the Smalltalk-80 language.)

△△ object.class::property is the value of property which the given object would normally inherit from the given class. (Or it gives an error if the class doesn't provide that property or if the object isn't a member of that class).

△△ It's perfectly legal to write something like x = Treasure::deposit; and then to send Bat_Idol.x();.

§3.11 Creating and deleting objects during play

In an adventure-game setting, object creation is useful for something like a beach full of stones: if the player wants to pick up more and more stones, the game needs to create a new object for each stone brought into play.

Besides that, it is often elegant to grow structures organically. A maze of caves being generated during play, for example, should have new caves gradually added onto the map as and when needed.

The trouble with this is that since resources cannot be infinite, the cave-objects have to come from somewhere, and at last they come no more. The program must be able to cope with this, and it can present the programmer with real difficulties, especially if the conditions that will prevail when the supply runs out are hard to predict.

Inform does allow object creation during play, but it insists that the programmer must specify in advance the maximum resources which will ever be needed. (For example, the maximum number of beach stones which can ever be in play.) This is a nuisance, but means that the resulting story file will always work, or always fail, identically on every machine running it. It won't do one thing on the designer's 256-megabyte Sun workstation in Venice and then quite another on a player's personal organiser in a commuter train in New Jersey.

. . . . .

If you want to create objects, you need to define a class for them and to specify N, the maximum number ever needed at once. Objects can be deleted once created, so if all N are in play then deleting one will allow another to be created.

Suppose the beach is to contain up to fifty pebbles. Then:

```
Class Pebble(50)
 with name 'smooth' 'pebble',
      short_name "smooth pebble from the beach";
```

`Pebble` is an ordinary class in every respect, except that it has the ability to create up to $N = 50$ instances of itself.

Creating and destroying objects is done by sending messages to the class `Pebble` itself, so for instance sending the message `Pebble.create()` will bring another pebble into existence. Classes can receive five different messages, as follows:

`remaining()`
How many more instances of this class can be created?

`create(`⟨parameters⟩`)`
Replies with a newly created object of this class, or else with `nothing` if no more can be created. If given, the parameters are passed on to the object so that it can use them to configure itself (see below).

`destroy(I)`
Destroys the instance `I`, which must previously have been created. You can't destroy an object which you defined by hand in the source code. (This is quite unlike `remove`, which only takes an object out of the tree for a while but keeps it in existence, ready to be moved back again later.)

`recreate(I, `⟨parameters⟩`)`
Re-initialises the instance `I`, as if it had been destroyed and then created again.

`copy(I, J)`
Copies the property and attribute values from `I` to be equal to those of `J`, where both have to be instances of the class. (If a property holds an array, this is copied over as well.)

△ It's rather useful that `recreate` and `copy` can be sent for any instances, not just instances which have previously been created. For example,

```
Plant.copy(Gilded_Branch, Poison_Ivy)
```

copies over all the features of a `Plant` from `Poison_Ivy` to `Gilded_Branch`, but leaves any other properties and attributes of the gilded branch alone. Likewise,

```
Treasure.recreate(Gilded_Branch)
```

only resets the properties to do with `Treasure`, leaving the `Plant` properties alone.

△ If you didn't give a number like `(50)` in the class definition, then you'll find that N is zero. `copy` will work as normal, but `remaining` will return zero and `create` will always return `nothing`. There is nothing to `destroy` and since this isn't a class which can create objects, `recreate` will not work either. (Oh, and don't try to send these messages to the class `Class`: creating and destroying classes is called "programming", and it's far too late when the game is already being played.)

△△ You can even give the number as `(0)`. You might do this either so that `Class Gadget(MAX_GADGETS)` in some library file will work even if the constant `MAX_GADGETS` happens to be zero. Or so that you can at least `recreate` existing members of the class even if you cannot `create` new ones.

. . . . .

The following example shows object creation used in a tiny game, dramatising a remark attributed to Isaac Newton (though it appears only in Brewster's *Memoirs of Newton*).

```
Constant Story "ISAAC NEWTON'S BEACH";
Constant Headline "^An Interactive Metaphor^";
Include "Parser";
Include "VerbLib";
Class Pebble(50)
 with name 'smooth' 'pebble' 'stone' 'pebbles//p' 'stones//p',
      short_name "smooth pebble from the beach",
      plural "smooth pebbles from the beach";
Object Shingle "Shingle"
  with description
          "You seem to be only a boy playing on a sea-shore, and
           diverting yourself in finding a smoother pebble or a
           prettier shell than ordinary, whilst the great ocean of
           truth lies all undiscovered before you.",
  has  light;
Object -> "pebble"
  with name 'smoother' 'pebble' 'stone' 'stones' 'shingle',
       initial "The breakers drain ceaselessly through the shingle,
           spilling momentary rock-pools.",
       before [ new_stone;
           Take:
               new_stone = Pebble.create();
               if (new_stone == nothing)
                  "You look in vain for a stone smoother than
                   the fifty ever-smoother stones you have
                   gathered so far.";
               move new_stone to Shingle;
               <<Take new_stone>>;
       ],
  has  static;
[ Initialise;
  location = Shingle; "^^^^^Welcome to...^";
];
Include "Grammar";
```

In this very small adventure game, if the player types "take a pebble", he will get one: more surprisingly, if he types "take a smoother pebble" he will get another one, and so on until his inventory listing reads "fifty smooth pebbles from the beach”. (See §29 for how to make sure identical objects are described well in Inform adventure games.) Notice that a newly-created object is `in nothing`, that is, is outside the object tree, so it must be moved to `Shingle` in order to come to the player's hand.

· · · · ·

However smooth, one pebble is much like another. More complicated objects sometimes need some setting-up when they are created, and of course in good object-oriented style they ought to do this setting-up for themselves. Here is a class which does:

```
Class Ghost(7)
 with haunting,
     create [;
         self.haunting = random(Library, Ballroom, Summer_House);
         move self to self.haunting;
         if (self in location)
             "^The air ripples as if parted like curtains.";
     ];
```

What happens is that when the program sends the message

```
Ghost.create();
```

the class `Ghost` creates a new ghost `G`, if there aren't already seven, and then sends a further message

```
G.create();
```

This new object `G` chooses its own place to haunt and moves itself into place. Only then does the class `Ghost` reply to the outside program. A class can also give a `destroy` routine to take care of the consequences of destruction, as in the following example:

```
Class Axe(30);
Class Dwarf(7)
 with create [ x;
         x = Axe.create(); if (x ~= nothing) move x to self;
     ],
     destroy [ x;
         objectloop (x in self && x ofclass Axe) Axe.destroy(x);
     ];
```

A new axe is created whenever a new dwarf is created, while stocks last, and when a dwarf is destroyed, any axes it carries are also destroyed.

Finally, you can supply `create` with up to 3 parameters. Here is a case with only one:

```
Class GoldIngot(10)
 with weight, value,
      create [ carats;
          self.value = 10*carats;
          self.weight = 20 + carats;
      ];
```

and now `GoldIngot.create(24)` will create a 24-carat gold ingot.

§3.12 Sending messages to routines and strings

△ §3.9 was about sending messages to `Objects`, and then in §3.11 it turned out that there are five messages which can be sent to a `Class`. That's two of the four metaclasses, and it turns out that you can send messages to a `Routine` and a `String` too.

The only message you can send to a `Routine` is `call`, and all this does is to call it. So if `Explore` is the name of a routine,

```
Explore.call(2, 4);    and    Explore(2, 4);
```

do exactly the same as each other. This looks redundant, except that it allows a little more flexibility: for instance

```
(random(Hello, Goodbye)).call(7);
```

has a 50% chance of calling `Hello(7)` and a 50% chance of calling `Goodbye(7)`. As you might expect, the `call` message replies with whatever value was returned by the routine.

Two different messages can be sent to a `String`. The first is `print`, which is provided because it logically ought to be, rather than because it's useful. So, for example,

```
("You can see an advancing tide of bison!").print();
```

prints out the string, followed by a new-line, and evaluates as `true`.

The second is `print_to_array`. This copies out the text of the string into entries $2, 3, 4, \ldots$ of the supplied byte array, and writes the number of characters as a word into entries 0 and 1. (Make sure that the byte array is large enough to hold the text of the string.) For instance:

```
Array Petals->30;
...
    ("A rose is a rose is a rose").print_to_array(Petals);
```

will leave `Petals-->0` set to 26 and place the letters `'A'`, `' '`, `'r'`, `'o'`, . . ., `'e'` into the entries `Petals->2`, `Petals->3`, . . ., `Petals->27`. For convenience, the reply value of the message `print_to_array` is also 26. You can use this message to find the length of a string, copying the text into some temporary array and only making use of this return value.

§3.13 *Common properties and* `Property`

△△ Many classes, the `Bird` class for example, pass on properties to their members. Properties coming from the class `Object` are called "common properties". Every item and place in an adventure game belongs to class `Object`, so a property inherited from `Object` will be not just common but well-nigh universal. Properties which aren't common are sometimes called "individual".

The Inform library sets up the class `Object` with about fifty common properties. Story files would be huge if all of the objects in a game actually used all of these common properties, so a special rule applies: *you can read a common property for any* `Object`*, but you can only write to it if you've written it into the object's declaration yourself.*

For instance, the library contains the directive

```
Property cant_go "You can't go that way.";
```

This tells Inform to add `cant_go` to the class definition for `Object`. The practical upshot is that you can perform

```
print_ret (string) location.cant_go;
```

whatever the location is, and the resulting text will be "You can't go that way." if the `location` doesn't define a `cant_go` value of its own. On the other hand

```
location.cant_go = "Please stop trying these blank walls.";
```

will only work if `location` actually provides a `cant_go` value of its own, which you can test with the condition `location provides cant_go`.

△△ Using the superclass operator you can read and even alter the default values of common properties at run-time: for instance,

```
    location.Object::cant_go = "Another blank wall. Tsk!";
```

will substitute for "You can't go that way."

△△ The Inform library uses common properties because they're marginally faster to access and marginally cheaper on memory. Only 62 are available, of which the compiler uses 3 and the library a further 47. On the other hand, you can have up to 16,320 individual properties, which in practical terms is as good as saying they are unlimited.

§3.14 Philosophy

△△ "Socialism is all very well in practice, but does it work in theory?" (Stephen Fry). While the chapter is drizzling out into small type, this last section is aimed at those readers who might feel happier with Inform's ideas about classes and objects if only there were some *logic* to it all. Other readers may feel that it's about as relevant to actual Inform programming as socialist dialectic is to shopping for groceries. Here are the rules anyway:

(1) Story files are made up of objects, which may have variables attached of various different kinds, which we shall here call "properties".
(2) Source code contains definitions of both objects and classes. Classes are abstract descriptions of common features which might be held by groups of objects.
(3) Any given object in the program either is, or is not, a member of any given class.
(4) For every object definition in the source code, an object is made in the story file. The definition specifies which classes this object is a member of.
(5) If an object `X` is declared as a member of class `C`, then `X` "inherits" property values as given in the class definition of `C`.

Exact rules of inheritance aren't relevant here, except perhaps to note that one of the things inherited from class `C` might be the membership of some other class, `D`.

(6) For every class definition, an object is made in the story file to represent it, called its "class-object".

For example, suppose we have a class definition like:

```
    Class Shrub
     with species;
```

The class `Shrub` will generate a class-object in the final program, also called `Shrub`. This class-object exists to receive messages like `create` and `destroy` and, more philosophically, to represent the concept of "being a shrub" within the simulated world.

The class-object of a class is not normally a member of that class. The concept of being a shrub is not itself a shrub, and the condition `Shrub ofclass Shrub` is `false`. Individual shrubs provide a property called `species`, but the class-object of `Shrub` does not: the concept of being a shrub has no single species.

(7) Classes which are automatically defined by Inform are called "metaclasses". There are four of these: `Class`, `Object`, `Routine` and `String`.

It follows by rule (6) that every Inform program contains the class-objects of these four, also called `Class`, `Object`, `Routine` and `String`.

(8) Every object is a member of one, and only one, metaclass:
- (8.1) The class-objects are members of `Class`, and no other class.
- (8.2) Routines in the program, including those given as property values, are members of `Routine` and no other class.
- (8.3) Constant strings in the program, including those given as property values, are members of `String`, and of no other class.
- (8.4) The objects defined in the source code are members of `Object`, and possibly also of other classes defined in the source code.

It follows from (8.1) that `Class` is the unique class whose class-object is one of its own members: so `Class ofclass Class` is `true`.

(9) Contrary to rules (5) and (8.1), the class-objects of the four metaclasses do not inherit from `Class`.

(10) Properties inherited from the class-object of the metaclass `Object` are read-only and cannot be set.

. . . . .

To see what the rules entail means knowing the definitions of the four metaclasses. These definitions are never written out in any textual form inside Inform, as it happens, but this is what they would look like if they were. (`Metaclass` is an imaginary directive, as the programmer isn't allowed to create new metaclasses.)

```
Metaclass Object
with name,
     ...;
```

A class from which the common properties defined by `Property` are inherited, albeit (by rule (10), an economy measure) in read-only form.

```
Metaclass Class
with create    [ ...; ... ],
     recreate  [ instance ...; ... ],
     destroy   [ instance; ... ],
     copy      [ instance1 instance2; ... ],
     remaining [; ... ];
```

So class-objects respond only to these five messages and provide no other properties: except that by rule (9), the class-objects `Class`, `Object`, `Routine` and `String` provide no properties at all. The point is that these five messages are concerned with object creation and deletion at run time. But Inform is a compiler and not, like Smalltalk-80 or other highly object-oriented languages, an interpreter. Rule (9) expresses our inability to create the program while it is actually running.

```
Metaclass Routine
with call [ parameters...; ... ];
```

Routines therefore provide only `call`.

```
Metaclass String
with print          [; print_ret (string) self; ],
     print_to_array [ array; ... ];
```

Strings therefore provide only `print` and `print_to_array`.

• **REFERENCES**
L. Ross Raszewski's library extension `"imem.h"` manages genuinely dynamic memory allocation for objects, and is most often used when memory is running terribly short.