The Wayback Machine - https://web.archive.org/web/20210419124201/http://www.firthworks.com/roger/infancy/index.html

InFancy -- using Inform objects

InFancy is another Inform overview site, like InfLight and InfAct. This time around, we're covering the basics of object usage, creation and naming, then showing how to build on those basic concepts in more sophisticated ways.

The material is loosely organised as follows:

Object headers
The **`Object`** directive (or more specifically, its first line).
Object bodies
More on the **`Object`** directive; the **`with`** and **`has`** segments.
Properties and attributes
Objects have 'em: what's the difference?
Names
Puppy-training: teaching an object to respond to its own name.
Listings
What you see in room contents and player inventories.
Descriptions
"So, tell me about yourself..."
Classes
Make the mould once, then send in the clones.
Dynamic objects
Creating an object at run-time, before your very eyes.
The final game
Alright alright, 'game' is overstating it. The code wot we wrote.

Conventions

To clarify where the various example displays come from, a little colour-coding is used:

```
This is a sample of text in an Inform source file.
```

```
This is from a Z-machine interpreter at run-time.
```

Acknowledgements

Thanks to Neil Cerutti for good food, fine wine and general behind-the-scenes support and guidance (ok, I lied about the food and the wine). And of course to Graham Nelson, ever-present in spirit (if not necessarily in body).

And with those few words, off we go!

The Wayback Machine - https://web.archive.org/web/20170701010509/http://www.firthworks.com/roger/infancy/header.html

InFancy -- using Inform objects

Inform is an **object-oriented** language, meaning that you create your game by defining a whole series of self-contained objects. Try not to think too deeply about what this means -- it turns out to be a natural and straightforward way of defining a world in which your player can roam through 'rooms' and interact with 'things'. Once you've got over the hurdle of creating your first handful of objects, you'll quickly become comfortable with the concept (though admittedly the details take longer to master). Trust me: it's worth persisting. To start with, we'll explain the organisation of an object.

"Object" directive

The basic definition of an Inform object begins with the compile-time directive **`Object`** and ends with a semicolon (**`;`**). (On a later page we'll see that an object definition can also begin with the name of an object class, but for now the **`Object`** directive will do nicely.) As with all directives, it's conventional to capitalise the first letter of **`Object`**; this helps us to remember that **directives** control what happens during compilation, whereas **statements** (which we don't capitalise) control what happens when the game runs.

```
Object  ...  ;
```

The content of an object definition -- what goes between the **`Object`** and the **`;`** -- can vary from trivially simple to extremely complex, so we'll creep up on it bit by bit. The first useful realisation is that we can divide the definition into two parts: the **header** (its first line), and the **body** (all the remaining lines).

```
Object  ...    ! this line is the object's header
        ...    !
        ...    ! and these remaining lines are its body
        ;      !
```

(Strictly, the header material isn't constrained to a single line, but in practice it virtually always is.) The header defines two things: the object's name(s), and its relationship (if any) to a parent object. First we'll talk about the pair of names, internal and external, which an object may possess.

Object names

Compare and contrast:

The internal name (*iname*) ...	**Whereas the external name (*xname*) ...**
is optional	is optional
is used at compile-time when one object needs to refer to another object	is used at run-time to tell the player where he is and what he can see
cannot be changed at run-time (obviously)	can be changed at run-time, using the **`short_name`** property (and indeed is also known as the **short** name)
is not normally visible to the player (but see below)	can be output using **`print (name) iname`**
can be up to 32 characters long, comprising letters, digits and underscores, where uppercase and lowercase letters are equivalent, and the first character cannot be a digit	can be up to 256 characters long, with few restrictions, and must be enclosed in quotes "..."
must be unique within the program	need not be unique within the program

Here's the syntax:

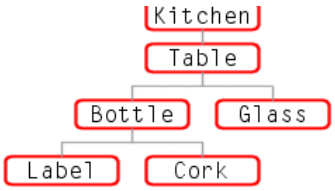
```
Object  iname "xname"
        ... ;
```

If the game needs to mention an object which doesn't have an *xname*, it uses the *iname* in parentheses (...) or, if the object doesn't have an *iname* either, the object's internal number in parentheses. This looks awful -- and can happen unexpectedly even when you thought the object's name would never appear -- so it's good practice to give *every* object a sensible external name. And, since you can't refer within the program to an object which doesn't have an *iname*, you might as well consistently assign those as well. Here's a typical example:

```
Object  kitchen_table "table"
        ... ;
```

Object relationships

In Inform, objects can be related one to another such that any given object has at most one **parent** (superior) object, and any number of **child** (subordinate) objects. This type of relationship model is commonly represented as a tree structure, as in this example:



One of the advantages of this structure is that moving an object within the tree -- changing its parent, if you like -- automatically moves any children of that object. This reflects natural behaviour; if for example the player TAKEs the bottle (so that he, rather than the table, becomes its parent) the label and the cork travel right along too.

While these parent-child relationships change frequently during the course of a game, they need to be established in a known pattern at the game's inception. While this *could* be done dynamically with lots of **`move ... to ... ;`** statements in the game's **`Initialise()`** routine, it's much more efficient to set things up statically at compile-time. Remember, too, that a parent isn't essential: unlike the real world, an Inform object can start out as an orphan, acquiring and losing parents throughout its life.

Relationships are defined in the object headers. There are two possible syntaxes:

```
Object  iname "xname" parent_iname
        ... ;


Object  set_of_arrows iname "xname"
        ... ;
```

The first syntax is easier to explain, so we'll cover it first. Immediately *after* the object's *xname*, you can supply the *iname* of the object's parent. Here's our example tree using this syntax:

```
Object  kitchen "Kitchen"
        ... ;

Object  kitchen_table "table" kitchen
        ... ;

Object  bottle "bottle" kitchen_table
        ... ;

Object  label "label" bottle
        ... ;

Object  cork "cork" bottle
        ... ;

Object  glass "glass tumbler" kitchen_table
        ... ;
```

The **`kitchen`** object has no parent. This is one of the necessary conditions for it being a 'room'; the other is that it is possible during the game for the player to move there, thus becoming a child of the room object. The **`kitchen_table`** specifies that its parent is the **`kitchen`**, while the **`bottle`** and the **`glass`** define the **`kitchen_table`** as *their* parents, and so on. The essential point is: each child object specifies the *iname* of an object which is to be its initial parent; that parent object must already have been defined at some previous location in the program.

The alternative syntax also demands that a object's parent has already been defined, but is more reliant on precise physical ordering. Here's the same tree:

```
Object  kitchen "Kitchen"
        ... ;

Object  -> kitchen_table "table"
        ... ;

Object  -> -> bottle "bottle"
        ... ;

Object  -> -> -> label "label"
        ... ;

Object  -> -> -> cork "cork"
        ... ;

Object  -> -> glass "glass tumbler"
        ... ;
```

The definition of the **`kitchen`** is unchanged. This time, though, the **`kitchen_table`** doesn't explicitly mention the **`kitchen`** as its parent. Instead, it includes a single arrow **`->`** immediately *before* its *iname*: read one arrow as saying "I'm a child of the nearest previous object without an arrow". Next, we find the **`bottle`** with two arrows **`-> ->`**, which reads as "I'm a child of the nearest previous object with a single arrow". You won't now be surprised to learn that the **`label`**'s three arrows make it a child of the two-arrow **`bottle`**; the **`cork`** also has three arrows, and so it too is a child of the **`bottle`**. Finally, the **`glass`** has only two arrows, making it a child of the nearest previous single-arrow object -- the **`kitchen_table`**.

Which syntax you use is a matter for personal preference; they both achieve exactly the same results. And in fact, you might think that mixing the two formats would be a good compromise? Well it might be... as long as you maintain a consistent approach for each top-level (parentless) object. Consider this example:

```
Object  kitchen "Kitchen"
        ... ;

Object  kitchen_table "table" kitchen
        ... ;

Object  bottle "bottle" kitchen_table
        ... ;

Object  -> label "label"
        ... ;

Object  -> cork "cork"
        ... ;

Object  glass "glass tumbler" kitchen_table
        ... ;
```

Does it give the same results as the previous versions? Surprisingly, no: the label and the cork end up as children of the **kitchen** rather than the **bottle**. Seems that the compiler isn't counting literal sequences of **->**, but instead is going by actual depth of object nesting. To achieve the desired structure, you've got to change those **->** back into **-> -> ->**, which rather defeats the object. Bummer, I reckon.

That's covered the object header -- the name and relationship stuff. Next, we'll look at the general organisation of the rest of the object -- its body.

The Wayback Machine - https://web.archive.org/web/20170701010521/http://www.firthworks.com/roger/infancy/body.html

InFancy -- using Inform objects

Unlike the single-line header, the body of an object usually occupies at least three or four lines; thirty or forty lines is common, and three or four hundred isn't unheard-of. But that's advanced stuff: our aspirations at this stage are much more modest -- understand the permitted syntax.

Body segments

Just as we divided the overall object definition into a header and a body, it's useful to subdivide the body into **segments**. There are four possible segments; they're all optional, and often only one or two are needed. The segments can occur in any order; each is introduced by a keyword:

```
Object    header_data
  class   ...    ! inheritance classes
  with    ...    ! public properties
  private ...    ! private properties
  has     ...    ! attributes flags
          ;
```

The term *header_data* represents the name/relationship material that we covered on the previous page; hopefully you now know what that all means. Next, we'll summarise the use of the four segments types.

"class" segment

The **class** keyword introduces a space-separated list of class names from which this object inherits parts of its definition.

```
class class_iname  class_iname ... class_iname
```

This won't make much sense until we've talked about class definitions, so we'll leave the details of the **class** segment until later. To be going on with, here's an example:

```
Object  kitchen_table "table"
  class Furniture
        ... ;
```

"with" segment

The **with** keyword introduces a comma-separated list of the object's (public) property variables.

```
with  property_def, property_def, ... property_def
```

Here, each *property_def* represents a variable which is associated with this object, and therefore consists of a *prop_name* and (optionally) an initial *value*. In fact, you can have:

- no initial value -- the *prop_name* is a scalar variable initialised to zero
- one initial value -- the *prop_name* is a scalar variable initialised to that value
- a space-separated list of initial values -- the *prop_name* is a word array initialized to those values

That is, for each *property_def* in the **with** syntax definition, you can substitute any of:

```
prop_name
  or
prop_name  value
  or
prop_name  value  value ... value
```

Finally, each *value* in this definition can be anything that can be evaluated at compile-time:

```
number
  or
'dictionary_word'
  or
"string"
  or
[; statement; statement; ... statement; ]
```

The next page has more to say about the use of properties, but here's a short example:

```
Object  kitchen_table "table"
  with  name 'battered' 'pine' 'table',
        description "The table is scuffed and stained with indeterminate substances.",
        ... ;
```

"private" segment

The **private** keyword introduces a comma-separated list of the object's internal (private) property variables.

```
private  property_def, property_def, ... property_def
```

The structure of the **private** segment is exactly the same as the **with** segment, and in fact the only difference between the two is that the *property_def* settings in a **with** segment are accessible from other objects, while those in a **private** segment can be accessed only by its own object. It turns out that this distinction is of minimal value in most games, and the majority of authors never feel the need to use **private** properties. Follow their lead, sez I.

"has" segment

The **has** keyword introduces a space-separated list of true/false (on/off, present/absent, set/unset) flags associated with the object.

```
has  attribute_def  attribute_def ... attribute_def
```

Each *attribute_def* is either an *attr_name* (which sets that flag value to true), or **~***attr_name* (which sets the flag to false).

The next page has more to say about the use of attributes, but here's a short example:

```
Object  kitchen_table "table"
  has   static supporter
        ... ;
```

The (almost) complete object syntax

We've mentioned the various elements making up an object definition; now's the time to summarise what we've said so far. The commonly-found and generally useful syntax of an object is something like:

```
Object  set_of_arrows iname "xname" parent_iname
  class class_iname  class_iname ...  class_iname
  with  prop_name value,
        prop_name value,
        ...
        prop_name value
  has   attr_name  attr_name ...  attr_name;
```

Notes

1. The definition starts with **Object** and ends with **;**
2. All four *header_data* items are optional. You can't use both *set_of_arrows* and *parent_iname*
3. The **private** segment (which occurs very rarely) has been omitted from this explanation
4. The **class** segment (which defines inheritance classes), the **with** segment (which defines property variables) and the **has** segment (which defines attribute flags) are all optional, and can occur in any order
5. For each property variable, the *prop_name* is usually followed by a single *value* (a number, a string, a routine, etc) but can be followed by a space-separated list of *values*, or by nothing
6. For each attribute flag, the *attr_name* can be prefixed by a tilde (**~**) to turn off the flag. Since they're off by default, you'll usually do this only to cancel an attribute which was set by the object's **class**
7. In general, spaces are used to separate the various items within an object definition. The only exception is the comma (**,**) which separates one *property_def* from the next. Specifically, you don't need to use commas in the *header_data*, or in the **class** or **has** segments

---

Phew! Spelling out the object syntax is a bit of a hard slog, but it's important to be confident with what's allowed. Next, we talk some more about properties and attributes.

The Wayback Machine - https://web.archive.org/web/20170701010535/http://www.firthworks.com/roger/infancy/panda.html

InFancy -- using Inform objects

Here's a quick roundup of some of the standard object property variables and attribute flags, and how you can use them at run-time.

Library properties and attributes

The Inform Library defines about 48 property variables (those marked "**++**" are **additive**, explained when we cover classes):

add_to_scope	d_to	grammar	n_to	react_after	u_to
after **++**	daemon		name **++**	react_before	
article	describe **++**	in_to	ne_to		w_to
articles	description	initial	number	s_to	when_closed
	door_dir	inside_description	nw_to	se_to	when_off
before **++**	door_to	invent		short_name	when_on
			orders	short_name_indef	when_open
cant_go	e_to	life **++**	out_to	sw_to	with_key
capacity	each_turn **++**	list_together			
			parse_name	time_left	
	found_in		plural	time_out	

The Library also defines about 31 attribute flags:

absent	edible	light	on	scenery	visited
animate	enterable	lockable	open	scored	
		locked	openable	static	workflag
clothing	female			supporter	worn
concealed		male	pluralname	switchable	
container	general	moved	proper		
				talkable	
door		neuter		transparent	

These definitions are heavily inter-related with the actions and verbs supplied with the standard Library: for example, the verbs WEAR/DON and REMOVE/SHED/DISROBE/DOFF invoke the actions **`Wear`** and **`Disrobe`** respectively; these in turn use the object's **`clothing`** and **`worn`** attributes. But that doesn't stop you using Library attributes for your own purposes, if that makes logical sense, and it certainly doesn't prevent you from creating your own definitions.

Using properties and attributes

Compare and contrast:

A property variable ...	**Whereas an attribute flag ...**
is usually a single (16-bit) word, but can be an array of up to 32 such words	is always stored as a single bit
can contain anything: a number, an address of a dictionary word, a (pointer to a) string, a (pointer to a) routine	is either true (on, present, set) or false (off, absent, unset)
may either be a **common** property, pre-declared by the directive **`Property`** ***`prop_name`*****`;`** (as are all of the standard 48), or an **individual** property. A common property applies to *all* objects whereas an individual property, which can be instantiated simply by using a new *prop_name* in the **`with`** segment of an object definition, applies only to that object. Only 62 common properties can be declared, but there is no effective limit on the number of individual properties (so those are what I recommend you to use)	are all common, pre-declared by the directive **`Attribute`** ***`attr_name`*****`;`** (as are all of the standard 31). An attribute applies to *all* objects. Only 48 attributes can be declared
can be tested at run-time by **`if (`*`iname`*` provides `*`prop_name`*`) ...;`** and by statements like **`if (`*`iname.prop_name == value`*`) ...;`** (for an array, by statements like **`if (`*`iname`*`.&`*`prop_name`*`-->0 == `*`value`*`) ...;`**)	can be tested at run-time by **`if (`*`iname`*` has `*`attr_name`*`) ...;`** and **`if (`*`iname`*` hasnt `*`attr_name`*`) ...;`**
can be changed at run-time by ***`iname.prop_name = value`*****`;`** (for an array, by ***`iname.&prop_name-->0 = value`*****`;`**)	can be changed at run-time by **`give `*`iname attr_name`*****`;`** and **`give `*`iname ~attr_name`*****`;`**
the property variable **`number`** is defined by the Library but not used; it is freely available in every object for you to use for your own purposes	the attribute flag **`general`** is defined by the Library but not used; it is freely available in every object for you to use for your own purposes

Notes

1. In all cases, you can use the keyword **`self`** in place of *iname* when an object is referring to its own properties and attributes.
2. The statements about limits on the number of properties and attributes apply to the Inform compiler and the Z-machine; for the new Glulx

compiler and Glulxe virtual machine the limits are much higher.
3. Also, Glulx uses 32-bit words, which means that the construct ***iname.#prop_name*** returning the number of bytes occupied by the *prop_name* array should be treated with caution. Using Inform you must divide this value by two to derive the number of entries in the array, whereas using Glulx you must divide by four.

In the next pages, we'll talk about some of the more commonly-encountered properties and attributes, starting with those that control the object's name and the way that it's addressed.

The Wayback Machine - https://web.archive.org/web/20170701010545/http://www.firthworks.com/roger/infancy/name.html

InFancy -- using Inform objects

Now that we've got the theory out of the way, we can start defining some real objects. First, we'll talk about how you can refer to your objects -- the **name** property and its associates.

"name" property

The **name** property holds a list of dictionary words, given in single quotes '...', which along with the implicit THE and A, may be used when referring to an object. Inform actually allows double quotes "..." also to be used here (elsewhere, something in double quotes is a string) but I recommend that you avoid confusion by sticking to apostrophes in the **name** property. There's one exception to this rule: if you need to give an object a one-character name, you can't use single quotes alone, since one character in apostrophes -- for instance **'X'** -- is an ASCII character constant. Instead, for a one-character name either follow the character with two slashes **'X//'**, use double quotes **"X"**, or employ the decidedly obscure construct **#n$X**. Here's our simple object tree again, with suitable **name** values (and also a handful of appropriate attributes):

```
Object  kitchen "Kitchen"
  with  description "Oddly, there are no exits."
  has   light;

Object  kitchen_table "table" kitchen
  with  name 'battered' 'pine' 'table'
  has   static supporter;

Object  bottle "bottle" kitchen_table
  with  name 'green' 'glass' 'bottle'
  has   transparent;

Object  label "label" bottle
  with  name 'faded' 'label';

Object  cork "cork" bottle
  with  name 'cork';

Object  glass "glass tumbler" kitchen_table
  with  name 'chipped' 'glass' 'tumbler';
```

In this example, the kitchen object (to which we've added a minimal **description** property to avoid run-time errors) doesn't have a **name** property. Inform assumes that you never need to refer to a room by name, and so offers a room's **name** as a suitable place to hold a list of 'irrelevant' words -- ones whose use triggers the response "That's not something you need to refer to in the course of this game". Often, but not always, that's a sensible option... but see also the discussion of **scenic.h** on a later page.

The attributes we're using are mostly self-evident; the kitchen object has **light** (or else we'd be blundering around in the dark), and the kitchen_table object is a **static supporter** (so that it's not takeable, and its children -- the bottle and the glass -- are properly listed in the room description). The oddest attribute is the bottle's **transparent**. This doesn't refer to the fact that it's made of glass, but rather ensures that the bottle's children -- the label and the cork -- are in scope and can be examined (though they don't yet appear in the bottle's description).

"parse_name" property

Given only this basic set of properties and attributes, we can compile and test the room. Even in this minimal state, it all works remarkably well, insofar as the room's contents can be examined and manipulated. You can refer to the only piece of furniture as THE TABLE, THE BATTERED PINE TABLE, PINE, and so on. The most obvious problem is with the word GLASS, used in both the bottle and the tumbler. You can refer to THE GLASS BOTTLE and THE GLASS TUMBLER easily enough, but if you mention simply THE GLASS, Inform will ask which of the two objects is meant. Here, it's fairly clear that a player using THE GLASS would expect to address the tumbler rather than the bottle. To fix this, we need to teach the bottle not to respond to the word GLASS alone. We do this by supplementing or replacing its **name** property with a **parse_name** property. In this example, we replace **name** completely:

```
Object  bottle "bottle" kitchen_table
  with  parse_name [ wd adj_count noun_count;
        if (parser_action == ##TheSame) return -2;
        wd = NextWord();
        while (wd == 'green' or 'glass' or 'wine' or 'corked')
            { wd = NextWord(); adj_count++; }
        while (wd == 'bottle' or 'flask' or 'flagon')
            { wd = NextWord(); noun_count++; }
        if (noun_count > 0) return noun_count + adj_count;
        return 0;
        ]
  has   transparent;
```

The primary job of an object's **parse_name** routine is to count the number of consecutive input words which could apply to that object. In the example code above (shamelessly stolen from Neil Cerutti's excellent Ditch Day Drifter tutorial), zero or more adjectives can precede one or more nouns, so that GREEN GLASS BOTTLE returns a count of 3, GLASS BOTTLE returns 2, BOTTLE returns 1 but GLASS returns 0. Since the bottle now effectively ignores the solitary word GLASS, the tumbler is free to recognise it unambiguously. (You can also write a **parse_name** routine which returns -1, in which case you must additionally supply a **name** property which Inform then uses in the conventional way.)

A **parse_name** routine has a secondary role: helping the parser to choose between apparently identical objects. The test on the first line -- **if (parser_action == ##TheSame)** -- ensures that we deal with that situation separately. Here's a short run-time example of what we've built so far:

```
Kitchen
Oddly, there are no exits.

You can see a table (on which are a bottle and a glass tumbler) here.

>EXAMINE THE TABLE
You see nothing special about the table.

>TAKE THE BOTTLE
Taken.

>EXAMINE IT
You see nothing special about the bottle.

>INV
You are carrying:
  a bottle

>EXAMINE LABEL
You see nothing special about the label.

>TAKE IT
That seems to be a part of the bottle.

>TAKE GLASS
Taken.
```

While the player uses words defined in **name** and **parse_name** properties, the game itself refers to an object by its *xname*. More on that next.

The Wayback Machine - https://web.archive.org/web/20170701010555/http://www.firthworks.com/roger/infancy/list.html

InFancy -- using Inform objects

An object's external name (*xname*) -- the double-quoted text in the header line -- is used whenever Inform needs to tell you of the object's presence. You have considerable control over the way in which the *xname* is used.

"pluralname" and "proper" attributes

When a object is mentioned in a room's description, or in the player's inventory, Inform generally prefixes it with the indefinite article A or AN. Two attributes modify this behavior: **`pluralname`** changes the prefix to SOME (giving, for example "On the table are some scissors."), and **`proper`** removes the prefix altogether (as in "You can also see Aunt Bessie here.").

"article" property

A related technique is to create an **`article`** property -- a string or routine -- which defines an appropriate indefinite article. For example, you could use this to cause messages like "On the table is **a handful of** grain." or "You can also see **your** Aunt Bessie here.".

Note that the **`pluralname`** also affects pronouns, causing its possessor to be addressable as THEM rather than IT. That's fine for some things, for example the scissors, but not for others. You might use **`article "some"`** to achieve "On the table is **some chocolate**." which is still addressable as IT.

There's also an **`articles`** property, but it's useful only in non-English games.

"short_name" property

The **`short_name`** provides a way of extending or replacing an object's *xname* at run-time. We can use it to bring the cork into play, by making the bottle announce itself as a "corked bottle" when appropriate. All it takes is this simple routine:

```
Object  bottle "bottle" kitchen_table
  with  parse_name [ wd adj_count noun_count;
        if (parser_action == ##TheSame) return -2;
        wd = NextWord();
        while (wd == 'green' or 'glass' or 'wine' or 'corked')
            { wd = NextWord(); adj_count++; }
        while (wd == 'bottle' or 'flask' or 'flagon')
            { wd = NextWord(); noun_count++; }
        if (noun_count > 0) return noun_count + adj_count;
        return 0;
        ],
        short_name [;
            if (cork in self) print "corked ";
            rfalse;
            ]
  has   transparent;
```

The routine returns **`false`**, so Inform carries on and prints the original *xname* after the word "corked ". Alternatively, the routine could print a completely new *xname* and then, by returning **`true`**, prevent the original *xname* from being output.

As things stand, we can examine the cork but not take it: Inform treats something inside a **`transparent`** object as being part of that object. Let's fix that, by giving the cork a **`before`** routine:

```
Object  cork "cork" bottle
  with  name 'cork' 'stopper',
        before [;
            Remove, Take, Pull:
                if ((self notin bottle) || (second ~= bottle or nothing))
                    rfalse;
                move self to player;
                "You pull the cork from the bottle.";
            Insert:
                if ((self notin player) || (second ~= bottle)) rfalse;
                move self to bottle;
                "You put the cork in the bottle.";
            ];
```

Finally, while we're dealing with the cork, we can extend the vocabulary by adding CORK and UNCORK verbs, so that UNCORK THE BOTTLE is a synonym for REMOVE THE CORK FROM THE BOTTLE:

```
Object  bottle "bottle" kitchen_table
  with  parse_name [ wd adj_count noun_count;
        if (parser_action == ##TheSame) return -2;
        wd = NextWord();
        while (wd == 'green' or 'glass' or 'wine' or 'corked')
            { wd = NextWord(); adj_count++; }
        while (wd == 'bottle' or 'flask' or 'flagon')
```

```
            { wd = NextWord(); noun_count++; }
        if (noun_count > 0) return noun_count + adj_count;
        return 0;
        ],
        short_name [;
            if (cork in self) print "corked ";
            rfalse;
            ],
        before [;
            Uncork: <<Remove cork self>>;
            Cork:   <<Insert cork self>>;
            ]
  has   transparent;

[ UncorkSub;    "You can't uncork that."; ];
[ CorkSub;      "You can't cork that."; ];

Verb    'uncork'    * noun  -> Uncork;
Verb    'cork'      * noun  -> Cork;
```

Let's see those verbs in action:

```
Kitchen
Oddly, there are no exits.

You can see a table (on which are a bottle and a glass tumbler) here.

>TAKE THE BOTTLE AND GLASS
corked bottle: Taken.
glass tumbler: Taken.

>UNCORK THE BOTTLE
You pull the cork from the bottle.

>INV
You are carrying:
  a cork
  a glass tumbler
  a bottle
```

There's also a **short_name_indef** property, but it's useful only in non-English games.

"invent" property

The **invent** property gives you some control over the way an object is described by the INVENTORY command. An object's **invent** routine is called twice, once before the object's *xname* is printed, and then again afterwards. Returning true from the routine prevents further processing, so either of these examples would have the same effect:

```
Object  cork "cork" bottle
  with  name 'cork' 'stopper',
        ...
        invent [;
            if (inventory_stage == 1) { print "a cork from a wine bottle"; rtrue; }
            ];

Object  cork "wine cork" bottle
  with  name 'wine' 'cork',
        ...
        invent [;
            if (inventory_stage == 2) { print " from a wine bottle"; rtrue; }
            ];
```

Actually, the cork isn't a very interesting example. Instead, we'll try something more ambitious -- putting some wine into the glass. First of all, here's our wine object:

```
Object  wine "red wine" glass
  with  name 'red' 'wine' 'plonk' 'liquid',
        article "some",
        capacity 2;
```

You can see that we're using the standard **article** property, as explained above, and that we've also added a **capacity** property (a standard property normally applied only to containers and supporters) whose use will become apparent in a moment. The wine is a child of the glass, whose definition now looks like this:

```
Object  glass "glass tumbler" kitchen_table
  with  name 'chipped' 'glass' 'tumbler',
        invent [ liq qty;
            liq = ChildWithProp(self,capacity);
            if (liq ~= 0) qty = liq.capacity;
```

```
            if (inventory_stage == 2) switch (qty) {
                2: print " (full)";
                1: print " (partly full)";
                default: ;
                }
            ]
  has   transparent;

[ ChildWithProp obj prop
        k;
        objectloop (k in obj) if (k provides prop) return k;
        return 0;
        ];
```

Lots going on here! This **invent** property is a routine with two local variables. **liq** is set by calling our little **ChildWithProp()** routine which tests whether any of the glass's child objects have a **capacity** property, if so returning that object. Then, **qty** is set to the value of the child object's **capacity** property, or remains at zero. Having obtained that property value from the child, we then use it at the second **inventory_stage** to print "(full)" or "(partly full)" after the glass's *xname*.

You might well ask: why not make the glass an **open container**? Two reasons: you'd then have to write code to prevent commands like PUT THE BOTTLE IN THE GLASS from being obeyed, and because the room description would then look like this:

```
Kitchen
Oddly, there are no exits.

You can see a table (on which are a corked bottle and a glass tumbler
(in which is some red wine)) here.
```

Those nested parentheses look a bit clumsy here, IMHO. You might also ask: why go to the trouble of providing a **ChildWithProp()** routine? why not simplify the glass thus:

```
Object  glass "glass tumbler" kitchen_table
  with  name 'chipped' 'glass' 'tumbler',
        invent [;
            if (inventory_stage == 2 && wine in self) switch (wine.capacity) {
                2: print " (full)";
                1: print " (partly full)";
                default: ;
                }
            ]
  has   transparent;
```

This would work perfectly well; the only disadvantage is that it hard-wires the wine object into the definition of the glass. By using the **ChildWithProp()** routine, we create a universal glass which can also be used for other liquids. (Actually, once we reach classes, we can change the routine to become **ChildOfClass()**, but this'll do to be going on with.)

"short_name" and "parse_name" properties revisited

Now that we've made the inventory listing sensitive to the contents of the glass, let's do the same with its *xname*, by providing a **short_name** property:

```
Object  glass kitchen_table
  with  name 'chipped' 'glass' 'tumbler',
        short_name [ liq qty;
            liq = ChildWithProp(self,capacity);
            if (liq ~= 0) qty = liq.capacity;
            if (qty > 0)
                print "glass of ", (name) liq;
            else
                print "empty glass tumbler";
            rtrue;
            ],
        ...
  has   transparent;
```

And now, instead of always announcing itself as "a glass tumbler", the glass is either "a glass of red wine" or "an empty glass tumbler".

```
Kitchen
Oddly, there are no exits.

You can see a table (on which are a corked bottle and a glass of red wine) here.

>TAKE THE GLASS OF RED WINE
I only understood you as far as wanting to take the glass of red wine.
```

Whoops! If we change the way in which the glass announces itself, we've got to expect that the player will want to use those same words. Time for another **parse_name** routine:

```
Object  glass kitchen_table
  with  parse_name [ liq qty wd container_count contents_count;
            if (parser_action == ##TheSame) return -2;
            liq = ChildWithProp(self,capacity);
            if (liq ~= 0) qty = liq.capacity;
            for (wd=NextWord() : wd~=0 : wd=NextWord()) switch (wd) {
                'chipped','glass','tumbler':
                    container_count++;
                 'of':
                    if (container_count > 0) container_count++;
                'empty','nothing':
                    if (qty == 0) contents_count++;
                 default:
                    if (qty > 0 && IsAWordIn(wd,liq,name)) contents_count++;
                    else jump exitGlass;
                }
            .exitGlass;
            if (container_count > 0) return container_count + contents_count;
            return 0;
            ],
        ;
        ;
        ;
  has   transparent;

[ IsAWordIn wd obj prop
        k l m;
        k = obj.&prop; l = (obj.#prop)/2;
        for (m=0 : m<l : m++) if (wd == k-->m) rtrue;
        rfalse;
        ];
```

Another mass of stuff to take on board. Work thought it slowly, and you'll see that we allow things like THE GLASS and CHIPPED GLASS, also THE EMPTY GLASS and GLASS OF NOTHING (if appropriate), and finally THE GLASS OF RED WINE. This last possibility is thanks to another little routine -- **`IsAWordIn()`** -- which looks in the child object's **`name`** property when parsing. Note that our glass in still independent of its contents: in another context, THE GLASS OF FULL CREAM MILK would be just as acceptable. (By the way, sorry about that **`jump`**; it seems like the cleanest way to exit both the **`switch`** and the **`for`**.)

Compatibility with Glulx

There's one last detail before we move on. In **`IsAWordIn()`** you'll see the statement **`l = (obj.#prop)/2;`** to find the number of bytes occupied by the property array and thence calculate its number of word-length entries. This works fine on Inform, but fails on Glulx, where there are four bytes to a word. Even if you're not using Glulx at the moment you might do so soon, so code defensively:

```
#ifndef WORDSIZE;
    Constant TARGET_ZCODE 0;
    Constant WORDSIZE 2+TARGET_ZCODE;
#endif;

[ IsAWordIn wd obj prop
        k l m;
        k = obj.&prop; l = (obj.#prop)/WORDSIZE;
        for (m=0 : m<l : m++) if (wd == k-->m) rtrue;
        rfalse;
        ];
```

Using Glulx, **`WORDSIZE`** is pre-defined as 4. It's not defined by Inform, so we set it to 2, and everything works properly. Note that by convention we also define **`TARGET_ZCODE`**, even though it's never actually used, and make that little artificial reference to it (setting **`WORDSIZE`** to **`2+TARGET_ZCODE`** rather than simply to **`2`**) to avoid a compiler warning.

That was a pretty heavy session! Next, we'll take things a bit easier, talking about object descriptions.

The Wayback Machine - https://web.archive.org/web/20170701010605/http://www.firthworks.com/roger/infancy/describe.html

InFancy -- using Inform objects

A number of properties control the way in which your game environment -- the rooms and their contents -- is presented to the player.

"description" and "inside_description" properties

The **description** is pretty easy to understand and test: for a room it's what you see upon entry and after LOOK, and for an object it's what you see when you EXAMINE the object. It can contain a routine or a string, as in these two examples:

```
Object  bottle "bottle" kitchen_table
  with
       °
       °
       °
       description [ liq;
           liq = ChildWithProp(self,capacity);
           print "You see an ordinary wine bottle, of green glass,
               with a faded label and a ";
           if (cork in self) print "cork protruding from the ";
           print "slender neck";
           if (liq ~= 0) print ". There is a quantity of liquid inside";
           ".";
           ]
  has   transparent;

Object  label "label" bottle
  with  name 'faded' 'label' 'lable',
        description
           "Though somewhat faded and difficult to make out,
           the label seems to read ~Chat Eau~.";
```

The **inside_description** property comes into play only when the player is inside an **enterable** object.

"initial" and "when_XXX" properties

initial can also contain a routine or a string. Applied to an object, it is invoked (following the description for the room where it appears) until that object has been handled by the player (strictly, until the object's **moved** attribute is set). The effect is to draw the player's attention to the object (though personally I find it does this in a rather obvious and clumsy way). Applied to a room, it is invoked early in the processing cycle, *before* the room's description is printed. Anyway, here it is introducing the table:

```
Object  kitchen_table "table" kitchen
  with  name 'battered' 'scuffed' 'stained' 'pine' 'table',
        initial
           "In the centre of the room stands a battered pine table.",
        description
           "The table is scuffed and stained with indeterminate substances."
  has   static supporter;
```

In a similar vein, the four properties **when_closed**, **when_open**, **when_off** and **when_on** take the place of **initial** for objects which are respectively **container|door**, **container|door open**, **switchable** and **switchable on**.

Note that all of these properties are invoked only when creating a room description -- not when examining the object itself -- and apply only to objects which are immediate children of the room -- on the floor, as it were. Nothing happens for objects on supporters or in containers.

"describe" property

The last of the descriptive properties, **describe** can contain a string, but really requires a routine. If present, it runs before **initial** or **when_XXX**. If it returns false, any other description is then generated; if it returns true, that's all you get. It's a bit of a specialized property, not one for which a natural example is easy to contrive.

scenic.h

If you think about it, the label object doesn't contribute much to the game. Its solitary task is to respond to EXAMINE LABEL commands, and these only come about because the bottle's description happens to mention a label. That is, the bottle is a real object, so we try to give it a convincing description. Unfortunately, in doing so we're likely to imply the existence of sub-objects, in which the player may well take an interest. It's bad game design to respond "You can't see any such thing." when the player tries to EXAMINE something you've just told him about; therefore, most authors find themselves creating a multiplicity of secondary objects whose only role is to handle EXAMINE requests.

The **scenic.h** library package -- obtainable from the Archive or from my home page -- offers a partial solution. It enables you to embed 'scenic' items, which support only examination, within the parent object whose description brought them to the player's attention. For example, we could have coded the label like this:

```
Object  bottle "bottle" kitchen_table
  with
       ...
       description [ liq;
           liq = ChildWithProp(self,capacity);
           print "You see an ordinary wine bottle, of green glass,
               with a faded label and a ";
           if (cork in self) print "cork protruding from the ";
           print "slender neck";
           if (liq ~= 0) print ". There is a quantity of liquid inside";
           ".";
           ]
       scenic
           'faded' 'label' 'lable' 0 "Though somewhat faded and difficult
               to make out, the label seems to read ~Chat Eau~."
  has   transparent;
```

This is a simple case; **scenic.h** is of more value when the alternative is to create a whole handful of secondary objects.

Right, we can't put it off any longer. Time to talk about the **Class** directive and the **class** segment.

The Wayback Machine - https://web.archive.org/web/20170701010616/http://www.firthworks.com/roger/infancy/class.html

InFancy -- using Inform objects

Actually, classes are remarkably straightforward.

"Class" directive

The **`Class`** compile-time directive is almost identical to the **`Object`** directive we studied earlier, except that the *header_data* consists only of a mandatory *iname*; you can't supply a *set_of_arrows*, *xname* or *parent_iname*. In the body, you can use the same four **`class`**, **`with`**, **`private`** and **`has`** segments (and, as for Objects, the **`class`** and **`private`** segments are relatively rare):

```
Class   iname
  with  prop_name value,
        prop_name value,
        ...
        prop_name value
  has   attr_name  attr_name ...  attr_name;
```

The *iname* is the name of your new class of objects, so by convention you should give it an initial capital letter. One of the mostly commonly-encountered user-defined classes is the **`Room`**:

```
Class   Room
  with  description "A bare room."
  has   light;
```

Here we have a prototype -- a default set of properties and attributes -- which is automatically applied to each object of this class, unless over-ridden. Let's see a few examples:

```
Object  cell "Monastery cell"
  class Room;

Object  kitchen "Kitchen"
  class Room
  with  description "Oddly, there are no exits.";

Object  cellar "Gloomy cellar"
  class Room
  with  description "Even with your torch, you see only dust and cobwebs."
  has   ~light;
```

Notice the **`class`** segment; way back, we said we'd defer discussion of this segment until later, and now's the time. **`class`** simply allows us to say that an object is a member of a named (and predefined) class, from which it inherits its behaviour. The cell inherits both the default **`description`** and the **`light`** attribute. The kitchen provides its own **`description`** but inherits the **`light`**. The cellar over-rides both, so that it has a **`description`** which you can't see unless you provide your own **`light`**. Another example:

```
Class   Furniture
  with  before [; Take,Pull,Push,PushDir:
           print_ret (The) self, " is too heavy for that.";
           ]
  has   static supporter;

Object  kitchen_table "table" kitchen
  class Furniture
  with  name 'battered' 'pine' 'table',
        initial
            "In the centre of the room stands a battered pine table.",
        description
            "The table is scuffed and stained with indeterminate substances.";
```

This time, our **`Furniture`** class supplies **`static supporter`** attributes, plus a **`before`** property which generates a more credible response than the standard "That's fixed in place.". These characteristics will be valid for most pieces of furniture, but can again be over-ridden as circumstances dictate. (For example, a marble statue might be defined with **`~supporter`**.)

The basic principle is: if your game has more than one object with the same general characteristics, you should consider basing it on a **`Class`**. The advantages are:

- your individual objects become simpler and more reliable (because some or all of their behaviour need be defined only once in the **`Class`** rather then repeated in each **`Object`**);
- you can use **`if (obj ofclass class_iname) ...`** statements as an easy way of detecting all objects of the same class.

Drinking vessels and their contents might also be fairly useful classes, so let's generalise our glass and wine as **`Vessel`** and **`Liquid`** classes:

```
Class   Vessel
  with  parse_name [ liq qty wd container_count contents_count;
            if (parser_action == ##TheSame) return -2;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            for (wd=NextWord() : wd~=0 : wd=NextWord()) switch (wd) {
                'of':
                    if (container_count > 0) container_count++;
                'empty','nothing':
                    if (qty == 0) contents_count++;
                default:
                    if (IsAWordIn(wd,self,name)) container_count++;
                    else
                        if (qty > 0 && IsAWordIn(wd,liq,name)) contents_count++;
                        else jump exitVessel;
                }
            .exitVessel;
            if (container_count > 0) return container_count + contents_count;
            return 0;
            ],
        before [; Receive:
            print_ret (The) self, " is meant for holding liquids."; ]
  has   transparent;

Class   Liquid
  with  name 'liquid',
        add_liquid [ qty;
            self.capacity = self.capacity + qty;
            if (self.capacity > 0)
                return self.capacity;
            else
                { remove self; return 0; }
            ],
        description [; print_ret "It looks rather like ", (name) self, "."; ],
        article "some",
        before [ container; container = parent(self);
            Take:
                if (container ofclass Vessel) <<Take container>>;
                print_ret "You can't take a puddle of ", (name) self, ".";
            Drink:
                if (container ofclass Vessel) <<Drink container>>;
                print_ret "You can't drink a puddle of ", (name) self, ".";
            ],
        capacity 0;                 ! Liquid currently in the vessel
```

Notice that we've changed the Vessel's **parse_name** slightly to re-use our **IsAWordIn()** routine, so that the dictionary words defining it as a container ('chipped', 'glass' and 'tumbler') are now taken from the object's name property rather than being built in, and that **ChildWithProp()** has become **ChildOfClass()**. Also, the Liquid has an additional **add_liquid** property to handle the additional and removal of quantities of liquid -- this isn't essential, but it makes for cleaner and more self-contained objects. Here's our new glass and wine:

```
Object  glass kitchen_table
  class Vessel
  with  name 'chipped' 'glass' 'tumbler',
        short_name [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            if (qty > 0)
                print "glass of ", (name) liq;
            else
                print "empty glass tumbler";
            rtrue;
            ],
        invent [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            if (inventory_stage == 2) switch (qty) {
                2: print " (full)";
                1: print " (partly full)";
                default: ;
                }
            !!rfalse;
            ],
        description [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            print "The glass tumbler is slightly chipped,
                but still usable with care. It's currently ";
            switch (qty) {
                2: print_ret "full of ", (name) liq, ".";
                1: print_ret "partly full of ", (name) liq, ".";
                default: "empty.";
                }
            ],
        before [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            Drink, Empty:
                if (self notin player)
                print_ret "You need to be holding ", (the) self, ".";
```

```
                switch (qty) {
                    2: liq.add_liquid(-1); print_ret "You take a mouthful of ", (name) liq, ".";
                    1: liq.add_liquid(-1); print_ret "You finish the rest of ", (the) liq, ".";
                    default: print_ret "There's nothing in ", (the) self, ".";
                    }
            ];

Object  wine "red wine" bottle
  class Liquid
  with  name 'red' 'wine' 'plonk',
        capacity 2;
```

You'll see that the glass is only slightly shorter: it still has to handle its own description, inventory and so on. The wine, on the other hand, is much simpler, since almost all of its behaviour is now encapsulated in its **`Liquid`** class. Also, now that we've added a DRINK action, we can actually test it:

```
Kitchen
Oddly, there are no exits.

In the centre of the room stands a battered pine table.

On the table are a corked bottle and a glass of red wine.

>EXAMINE BOTTLE
You see an ordinary wine bottle, of green glass, with a faded label
and a cork protruding from the slender neck.

>EXAMINE GLASS
The glass tumbler is slightly chipped, but still usable with care.
It's currently full of red wine.

>EXAMINE WINE
It looks rather like red wine.

>TAKE IT
Taken.

>INVENTORY
You are carrying:
  a glass of red wine (full)

>DRINK SOME WINE
You take a mouthful of red wine.

>AGAIN
You finish the rest of the red wine.

>INVENTORY
You are carrying:
  an empty glass tumbler
```

Multiple inheritance

Before we reached this page, new objects -- defined by using the **`Object`** directive -- were actually members (or instances) of the **Object** class; that was the common parent from which all objects inherited their default behaviour. (Not that you needed to know this; not that it mattered.) So what happens when we construct a new class -- say **`Room`** -- and then create the kitchen object? Easy: the kitchen is a member of *two* classes, **`Object`** and **`Room`**, and it inherits from both. This idea of multiple inheritance is both important and powerful, because it means that an object can be a member of many classes at once, acquiring some behavioural aspects from each. Those aspects don't need to be large and complex; sometimes just being a member of a class is sufficient to differentiate one collection of objects from the rest. For example, consider these three classes:

```
Class   Small;

Class   Food
  has   edible;

Class   Inflammable
  with  before [; Burn:
            remove self;
            print_ret "You set fire to ", (the) self,
                ", which quickly burns to nothing.";
            ];
```

From these classes, we might defines **`Small`** objects (which can perhaps be put in a pocket, or hidden in a mousehole), **`Food`** objects (which can be eaten), and **`Inflammable`** objects which can be consumed by fire. So here's a chocolate bar, which is all of these:

```
Object  choc_bar "chocolate" kitchen
  class Food Small Inflammable
  with  name 'chocolate' 'bar' 'hershey' 'block',
        article "some",
        description "The bar of chocolate looks yummy!";
```

That is, **`class`** allows us to say that an object is a member of several classes (as well as of the implicit Object class), and inherits its behaviour from all of them. Thus, the choc_bar acquires an **`edible`** attribute from Food, and a **`before`** property from Inflammable (but nothing specific, other than

class membership, from Small). To these it adds its own **name**, **article** and **description** properties... and there's a complete object which can be examined, eaten or burned.

In this simple example, there is no overlap between the properties/attributes of the objects itself, and those of the three component classes. But suppose there had been some commonality? What if we'd written:

```
Class   Small
  with  name 'small' 'tiny' 'little',
        description "It's not very big.";

Class   Food
  with  name 'delicious',
        description "Looks good enough to eat."
  has   edible;

Class   Inflammable
  with  before [; Burn:
            remove self;
            print_ret "You set fire to ", (the) self,
                ", which quickly burns to nothing.";
            ];

Object  choc_bar "chocolate" kitchen
  class Food Small Inflammable
  with  name 'chocolate' 'bar' 'hershey' 'block',
        article "some",
        description "The bar of chocolate looks yummy!";
```

Here there's more than one **name**, and more than one **description**; which one is used? The answer's a bit confusing: *all* of the **names**, but *only one* of the **descriptions** -- the "yummy!" one from the object itself. It's exactly as though we'd written:

```
Class   Small;

Class   Food
  has   edible;

Class   Inflammable
  with  before [; Burn:
            remove self;
            print_ret "You set fire to ", (the) self,
                ", which quickly burns to nothing.";
            ];

Object  choc_bar "chocolate" kitchen
  class Food Small Inflammable
  with  name 'small' 'tiny' 'little' 'delicious'
             'chocolate' 'bar' 'hershey' 'block',
        article "some",
        description "The bar of chocolate looks yummy!";
```

What's happening here? There are two types of property: **additive** and **non-additive**. Additive properties, like **name**, are accumulated from the definitions in the object itself *and* in the class(es) on which it is based. So, all the words defined in the various **name** properties can be used to address the choc_bar. Non-additive properties, like **description**, don't accumulate. An object can have only one **description**, which is taken either from the object itself or, if the object doesn't provide one, from the *first* class in the list. So, the choc_bar is always "yummy!" rather than "not very big" or "good enough to eat".

Most properties are non-additive: the only ones that accumulate are: **after**, **before**, **describe**, **each_turn**, **life** and **name**. And attributes accumulate in a slightly different way: if an individual attribute is set or unset (for example **edible** or **~edible**) in the object itself, that's what prevails. An attribute which *isn't* mentioned in the object itself will be set if it was set by any of its classes; unsetting an attribute in a class definition has no effect.

Replacing the "Object" directive

Inform supports an alternative syntax for defining objects of a user-specified class: you can supply the name of the class in place of the **Object** directive, and then you don't need the explicit class segment. For example, we could have said:

```
Room    cell "Monastery cell";

Room    kitchen "Kitchen"
  with  description "Oddly, there are no exits.";

Room    cellar "Gloomy cellar"
  with  description "Even with your torch, you see only dust and cobwebs."
  has   ~light;
```

Only **one** class name can be used in place of the **Object** directive, so for an object which inherits from more than one class, you still need a class segment. Our chocolate bar could be defined as:

```
Food    choc_bar "chocolate" kitchen
  class Small Inflammable
  with  name 'chocolate' 'bar' 'hershey' 'block',
        article "some",
```

```
    description "The bar of chocolate looks yummy!";
```

It's up to you whether to adopt this syntax: the advantage is that your program is easier to understand if the directive defining each object is a meaningful name rather than the ubiquitous **`Object`**. Be careful with multiple inheritance; remember that non-additive properties *not* defined by the object itself default to the setting from the first listed class -- that's now the one replacing the **`Object`** directive rather than the first one in the class list (in the example, Food).

Next, a bit more on classes, and particularly on how you can create instances of a class at run-time.

The Wayback Machine - https://web.archive.org/web/20210506015748/http://www.firthworks.com/roger/infancy/dynamic.html

InFancy -- using Inform objects

In previous pages we've developed a bottle and glass, placed liquid in the glass, and implemented a DRINK verb. What we need to finish things off is, of course, some way of starting with the liquid in the bottle, so that we can POUR it into the glass. To complicate matters, the bottle can hold more than the glass, so at times we'll have liquid in two places at once. There's any number of ways of tackling this situation: we'll use the technique of **dynamic objects** -- of creating and destroying instances of the Liquid class at run-time. For this to work, we must (a) set up the Liquid class appropriately, and (b) issue the necessary run-time messages. Happily, both of these are easy to do.

Extending the Liquid class

When we left our Liquid class on the previous page, its definition started out like this:

```
Class   Liquid
  with  name 'liquid',
        add_liquid [ qty;
            self.capacity = self.capacity + qty;
            if (self.capacity > 0)
                return self.capacity;
            else
                { remove self; return 0; }
            ],
        description [; print_ret "It looks rather like ", (name) self, "."; ],
        ;
        ;
        ;
```

To make it support dynamic object creation, we've changed it thus:

```
Class   Liquid(1)
  with  name 'liquid' '.spare' '.spare' '.spare' '.spare' '.spare',
        short_name "liquid",
        rename [ sn n0 n1 n2 n3 n4 n5;
            if (sn) self.short_name = sn;
            if (n0) self.&name-->0 = n0;
            if (n1) self.&name-->1 = n1;
            if (n2) self.&name-->2 = n2;
            if (n3) self.&name-->3 = n3;
            if (n4) self.&name-->4 = n4;
            if (n5) self.&name-->5 = n5;
            ],
        add_liquid [ qty;
            self.capacity = self.capacity + qty;
            if (self.capacity > 0)
                return self.capacity;
            else
                { Liquid.destroy(self); return 0; }
            ],
        description [; print_ret "It looks rather like ", (name) self, "."; ],
        ;
        ;
        ;
```

We've made five changes:

1. **added** a number in parentheses after the *iname*, to define the number of dynamic instances of this class which can exist *at any one time* while the game runs. Note that this doesn't include static instances, created at compile-time; there's no limit on those. Also, the phrase "at any one time" is important: our setting of **(1)** permits the sequence **`create() ... destroy() ... create() ... destroy()`**, but prevents **`create() ... create() ... destroy() ... destroy()`**, for which we'd need to have specified **(2)**.
2. **extended** the **`name`** property with five spare entries, for use by **`rename`**. There's nothing special about **`'.spare'`** other than that, since it contains a dot, it's an "untypeable" word, and so will never match anything the player types.
3. **supplied** a **`short_name`** property, also for use by **`rename`**.
4. **created** a **`rename`** property, as a convenient way of changing the **`name`** and **`short_name`** properties.
5. **modified** the **`add_liquid`** property by changing **`remove self`** (which detaches the object from its parent, leaving it floating free and unreferenced but still in existence) to **`Liquid.destroy(self)`** (which completely deletes the object from the game).

Enhancing the bottle object

If we start the game with all of the liquid in the bottle, then it's natural to make the bottle responsible for creating Liquid instances in the glass. When we last saw it, the bottle looked something like this:

```
Object  bottle "bottle" kitchen_table
  with  parse_name [ wd adj_count noun_count;
            if (parser_action == ##TheSame) return -2;
            wd = NextWord();
            while (wd == 'green' or 'glass' or 'wine' or 'corked')
                { wd = NextWord(); adj_count++; }
```

```
            while (wd == 'bottle' or 'flask' or 'flagon')
                { wd = NextWord(); noun_count++; }
            if (noun_count > 0) return noun_count + adj_count;
            return 0;
            ],
        short_name [;
            if (cork in self) print "corked ";
            !!rfalse;
            ],
        description [ liq;
            liq = ChildOfClass(self,Liquid);
            print "You see an ordinary wine bottle, of green glass,
                with a faded label and a ";
            if (cork in self) print "cork protruding from the ";
            print "slender neck";
            if (liq ~= 0) print ". There is a quantity of liquid inside";
            ".";
            ],
        before [;
            Uncork: <<Remove cork self>>;
            Cork:   <<Insert cork self>>;
            ]
  has   transparent;
```

So now let's add the ability to pour its contents into a Vessel:

```
Object  bottle "bottle" kitchen_table
  with
        °
        °
        °
        before [ liq qty liq2;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            Uncork: <<Remove cork self>>;
            Cork:   <<Insert cork self>>;
            EmptyT: if (cork in self)
                        print_ret "You tip ", (the) self,
                            ", but nothing happens.";
                    if (qty == 0) print_ret (The) self, " is empty.";
                    if (~~(second ofclass Vessel))
                        "That would just make a mess.";
                    liq2 = ChildOfClass(second,Liquid);
                    if (liq2 == 0) {    ! no Liquid in Vessel
                        if (qty > second.capacity) qty = second.capacity;
                        liq2 = Liquid.create();
                        if (liq2 == 0) "*** Can't create Liquid object! ***";
                        liq2.rename("red wine",'red','wine','plonk');
                        print "You pour ", (a) liq, " into ", (the) second, ".^";
                        move liq2 to second;
                        }
                    else {              ! already some Liquid in Vessel
                        qty = second.capacity - liq2.add_liquid(0);
                        if (qty == 0)
                            print_ret (The) second, " is already full.";
                        print "You add some more ", (name) liq,
                            " to ", (the) second, ".^";
                        }
                    liq2.add_liquid(qty); liq.add_liquid(-qty);
                    rtrue;
            ],
        capacity 10                     ! Liquid the bottle can contain
  has   transparent;

Liquid  "liquid" bottle
  with  capacity 4;                     ! Liquid currently in the bottle

Verb    'pour'          = 'empty';
```

We've enhanced the bottle's **`before`** property to trap the EmptyT action; basically, that's EMPTY noun INTO noun. There's a lot happening here -- making sure the bottle is uncorked and not empty, that the destination is a Vessel which isn't already full, and so on -- but the most important lines are highlighted. We send a **`create()`** message to the Liquid class, and receive in reply the address of a newly-created instance of the class: a Liquid object. If the reply is zero, a new object couldn't be created, presumably because we're already at our specified limit of concurrent run-time instances. In this example this shouldn't happen, so if it does we just output a debugging message. Assuming that object creation is successful, we then call the new object's **`rename`** property to over-write its default **`short_name`** and **`name`** properties. Finally, more housekeeping: telling the player what's happened, moving the new Liquid object into the Vessel, and adjusting the quantities of Liquid in the bottle and the Vessel.

Once we've primed the bottle by giving it an initial Liquid content, it's time for a test drive:

```
Kitchen
Oddly, there are no exits.

In the centre of the room stands a battered pine table.

On the table are a corked bottle and an empty glass tumbler.

>EXAMINE THE BOTTLE
You see an ordinary wine bottle, of green glass, with a faded label
```

```
and a cork protruding from the slender neck. There is a quantity of
liquid inside.

>UNCORK IT
You pull the cork from the bottle.

>POUR IT INTO THE GLASS
You pour some liquid into the empty glass tumbler.

>TAKE THE GLASS
Taken.

>INV
You are carrying:
  a glass of red wine (full)
  a cork

>DRINK SOME WINE
You take a mouthful of red wine.

>POUR THE BOTTLE INTO THE GLASS
You add some more liquid to the glass of red wine.

>DRINK WINE
You take a mouthful of red wine.

>DRINK WINE
You finish the rest of the red wine.

>EMPTY THE BOTTLE INTO THE GLASS
You pour some liquid into the empty glass tumbler.

>EXAMINE BOTTLE
You see an ordinary wine bottle, of green glass, with a faded label
and a slender neck.

>EXAMINE GLASS
The glass tumbler is slightly chipped, but still usable with care.
It's currently partly full of red wine.

>DRINK WINE
You finish the rest of the red wine.

>INV
You are carrying:
  an empty glass tumbler
  a cork
```

OK, that's about as far as we can take things here; we've reached the point where the complexities of refining the objects' behaviour would outweigh their educational benefit. The classes and objects that we've created provide a simplistic but usable representation of liquid handling, and could potentially be developed into a full-fledged general package; we'll leave that as an exercise for the reader.

Finally, on the past page, you'll find a listing of the complete game in its 'final' state. Enjoy!

The Wayback Machine - https://web.archive.org/web/20170701010657/http://www.firthworks.com/roger/infancy/game.html

InFancy -- using Inform objects

This is the final form of the code, represented here for ease of reference. As ever, we start with a standard preamble:

```
Constant Story "INFANCY";
Constant Headline "^A juvenile INFORM program.^";
Constant MANUAL_PRONOUNS;

Include "Parser";
Include "VerbLib";

! ---------------------------------------------------------------------------- !
```

These are the standalone routines:

```
[ Initialise; location = kitchen; ];

!       Locate the (first) child object of a given Class.
[ ChildOfClass obj class
        k;
        objectloop (k in obj) if (k ofclass class) return k;
        return 0;
        ];

!       Ensure compatibility with glulx, which uses four-byte words.
#ifndef WORDSIZE;
    Constant TARGET_ZCODE 0;
    Constant WORDSIZE 2+TARGET_ZCODE;   ! avoid compiler warning
#endif;

!       Test if a given value occurs in a given property array of an object.
[ IsAWordIn wd obj prop
        k l m;
        k = obj.&prop; l = (obj.#prop)/WORDSIZE;
        for (m=0 : m<l : m++) if (wd == k-->m) rtrue;
        rfalse;
        ];

! ---------------------------------------------------------------------------- !
```

The classes:

```
Class   Room
  with  description "A bare room."
  has   light;

Class   Furniture
  with  before [; Take,Pull,Push,PushDir:
            print_ret (The) self, " is too heavy for that.";
            ]
  has   static supporter;

Class   Vessel
  with  parse_name [ liq qty wd container_count contents_count;
            if (parser_action == ##TheSame) return -2;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            for (wd=NextWord() : wd~=0 : wd=NextWord()) switch (wd) {
                'of':
                    if (container_count > 0) container_count++;
                'empty','nothing':
                    if (qty == 0) contents_count++;
                default:
                    if (IsAWordIn(wd,self,name)) container_count++;
                    else
                        if (qty > 0 && IsAWordIn(wd,liq,name)) contents_count++;
                        else jump exitVessel;
                }
            .exitVessel;
            if (container_count > 0) return container_count + contents_count;
            return 0;
            ],
        invent [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            if (inventory_stage == 2) {
                if (qty == self.capacity) print " (full)";
                else
                    if (qty > 0) print " (partly full)";
                }
            ],
```

```
        description [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            print "It's currently ";
            if (qty == self.capacity) print_ret "full of ", (name) liq, ".";
            else
                if (qty > 0) print_ret "partly full of ", (name) liq, ".";
                else "empty.";
            ],
        before [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            Drink, Empty:
                if (self notin player)
                    print_ret "You need to be holding ", (the) self, ".";
                if (qty > 1)
                    print "You take a mouthful of ", (name) liq, ".^";
                else
                    if (qty == 1)
                        print "You finish the rest of ", (the) liq, ".^";
                    else print_ret "There's nothing in ", (the) self, ".";
                liq.add_liquid(-1);
                rtrue;
            Receive: print_ret (The) self, " is meant for holding liquids.";
            ],
        capacity 0              ! Liquid the vessel can contain
  has   transparent;

Class   Liquid(1)
  with  name 'liquid' '.spare' '.spare' '.spare' '.spare' '.spare',
        short_name "liquid",
        rename [ sn n0 n1 n2 n3 n4 n5;
            if (sn) self.short_name = sn;
            if (n0) self.&name-->0 = n0;
            if (n1) self.&name-->1 = n1;
            if (n2) self.&name-->2 = n2;
            if (n3) self.&name-->3 = n3;
            if (n4) self.&name-->4 = n4;
            if (n5) self.&name-->5 = n5;
            ],
        add_liquid [ qty;
            self.capacity = self.capacity + qty;
            if (self.capacity > 0)
                return self.capacity;
            else
                { Liquid.destroy(self); return 0; }
            ],
        description [; print_ret "It looks rather like ", (name) self, "."; ],
        article "some",
        before [ container; container = parent(self);
            Take:
                if (container ofclass Vessel) <<Take container>>;
                print_ret "You can't take a puddle of ", (name) self, ".";
            Drink:
                if (container ofclass Vessel) <<Drink container>>;
                print_ret "You can't drink a puddle of ", (name) self, ".";
            Empty:
                if (container ofclass Vessel) <<Empty container>>;
                print_ret "You can't do that.";
            EmptyT:
                if (container ofclass Vessel) <<EmptyT container second>>;
                print_ret "You can't do that.";
            ],
        capacity 0;             ! Liquid currently in the vessel

! ---------------------------------------------------------------------------- !
```

The instances of those classes -- the actual game objects:

```
Room    kitchen "Kitchen"
  with  description "Oddly, there are no exits.";

Furniture   kitchen_table "table" kitchen
  with  name 'battered' 'pine' 'table',
        initial
            "In the centre of the room stands a battered pine table.",
        description
            "The table is scuffed and stained with indeterminate substances.";

Object  bottle "bottle" kitchen_table
  with  parse_name [ wd adj_count noun_count;
            if (parser_action == ##TheSame) return -2;
            wd = NextWord();
            while (wd == 'green' or 'glass' or 'wine' or 'corked')
                { wd = NextWord(); adj_count++; }
            while (wd == 'bottle' or 'flask' or 'flagon')
                { wd = NextWord(); noun_count++; }
            if (noun_count > 0) return noun_count + adj_count;
            return 0;
            ],
        short_name [;
```

```
            if (cork in self) print "corked ";
            ],
        description [ liq;
            liq = ChildOfClass(self,Liquid);
            print "You see an ordinary wine bottle, of green glass,
                with a faded label and a ";
            if (cork in self) print "cork protruding from the ";
            print "slender neck";
            if (liq ~= 0) print ". There is a quantity of liquid inside";
            ".";
            ],
        before [ liq qty liq2;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            Uncork: <<Remove cork self>>;
            Cork:   <<Insert cork self>>;
            Drink:  "Please don't drink from the bottle!";
            Empty:  if (cork in self)
                        "Fortunately, the cork prevents you making a mess.";
                    if (qty == 0) print_ret (The) self, " is empty.";
                    liq.rename("red wine",'red','wine','plonk');
                    print "You pour ", (the) liq, " onto the floor.^";
                    move liq to location;
                    rtrue;
            EmptyT: if (cork in self)
                        print_ret "You tip ", (the) self,
                            ", but nothing happens.";
                    if (qty == 0) print_ret (The) self, " is empty.";
                    if (~~(second ofclass Vessel))
                        "That would just make a mess.";
                    liq2 = ChildOfClass(second,Liquid);
                    if (liq2 == 0) {    ! no liquid in Vessel
                        if (qty > second.capacity) qty = second.capacity;
                        liq2 = Liquid.create();
                        if (liq2 == 0) "*** Can't create Liquid object! ***";
                        liq2.rename("red wine",'red','wine','plonk');
                        print "You pour ", (a) liq, " into ", (the) second, ".^";
                        move liq2 to second;
                        }
                    else {              ! already some liquid in Vessel
                        qty = second.capacity - liq2.add_liquid(0);
                        if (qty == 0)
                            print_ret (The) second, " is already full.";
                        print "You add some more ", (name) liq,
                            " to ", (the) second, ".^";
                        }
                    liq2.add_liquid(qty); liq.add_liquid(-qty);
                    rtrue;
            ],
        capacity 10             ! Liquid the bottle can contain
  has   transparent;

Liquid  "liquid" bottle
  with  capacity 4;             ! Liquid currently in the bottle

Object  label "label" bottle
  with  name 'faded' 'label' 'lable',
        description
            "Though somewhat faded and difficult to make out,
            the label seems to read ~Chat Eau~.";

Object  cork "cork" bottle
  with  name 'cork' 'stopper',
        before [;
            Remove, Take, Pull:
                if ((self notin bottle) || (second ~= nothing or bottle))
                    rfalse;
                move self to player;
                "You pull the cork from the bottle.";
            Insert:
                if ((self notin player) || (second ~= bottle)) rfalse;
                move self to bottle;
                "You put the cork in the bottle.";
            ],
        description
            "It's a small cork, of the type associated with wine bottles.";

Vessel  glass kitchen_table
  with  name 'chipped' 'glass' 'tumbler',
        short_name [ liq qty;
            liq = ChildOfClass(self,Liquid);
            if (liq ~= 0) qty = liq.add_liquid(0);
            if (qty > 0)
                print "glass of ", (name) liq;
            else
                print "empty glass tumbler";
            rtrue;
            ],
        description [;
            print "The glass tumbler is slightly chipped,
                but still usable with care. ";
            Self.Vessel::description();
            ],
```

```
        capacity 2;             ! Liquid the glass can contain

! ---------------------------------------------------------------------------- !
```

Finishing up with the standard Grammar and our small extensions to it:

```
Include "Grammar";

[ UncorkSub;    "You can't uncork that."; ];
[ CorkSub;      "You can't cork that."; ];

Verb    'uncork'    * noun          -> Uncork;
Verb    'cork'      * noun          -> Cork;
Verb    'pour'      = 'empty';

! ---------------------------------------------------------------------------- !
```

The techniques that we've talked about here are certainly not the only ways of producing the required behaviour; they're not even necessarily the best ones. As ever, comments are welcome; corrections even more so. I'd hate to be misleading those in most need, so if any of you gurus spot an error, please let me know.