Chapter 4: Items

(Pristine and handled objects • Plural forms • All about appearances • Pristineness of nested objects • Clothing)

Items are objects that can be picked up by the player:

```
#bowl
(name *)        small
```

```
#bowl
(name *) small bowl
(descr *) It's a small bowl.
(item *)  %% This is what allows the player to pick up the bowl.
(* is #on #table)
```

[Copy to clipboard]

Pristine and handled objects

As a general rule, the standard library doesn't call attention to game objects. The story author is expected to mention them as part of the description of their parent object (such as the room).

But there is an important exception: As soon as the player picks up an object (or otherwise moves it from its original location), the responsibility for mentioning that object is transferred to the library. We say that the object has become *handled* or, equivalently, that it is no longer *pristine*.

There's a compelling pragmatic reason for this: When players are able to move objects around, those objects will eventually end up in strange locations, unanticipated by the story author. Players will put the flower pot on top of the bed, or take off their clothes and put them inside the ashtray, and the story author cannot be expected to provide custom text for every combination the player can think of. So, once objects start moving around, the library takes over the initiative in calling attention to them, using bland, default messages. They story author may then choose to override those messages on a case-by-case basis, and we will soon see how that's done in practice.

For now, the important thing to remember is that items (and wearable objects) can move around, and therefore we should only call them out in room descriptions—and in the descriptions of containers and supporters—when they are in their pristine state. Whenever we include a movable object in our story, we have a responsibility to check this. It can be done with an if-statement as in the following example:

```
#bowl
(name *)        small
```

```
#bowl
(name *) small bowl
(descr *) It's a small bowl.
(item *)
(* is #on #table)

(descr #table)
        It's wooden; possibly mahogany.
        (if) (#bowl is pristine) (then)
        A small bowl is placed exactly in its centre.
        (endif)
```

[Copy to clipboard]

Non-movable objects will remain pristine forever, so they can be described without a check.

Here is a complete example game with movable objects:

```
(current player
#player)
```

```
(current player #player)

#room
(name *) tutorial room
(room *)
(#player is #in *)
(look *)  This is a very nondescript room, dominated by a
          wooden table. (notice #table)

#table
(name *) wooden table
```

```
(dict *)    mahogany    %% Add a synonym.
(supporter *)
(* is #in #room)
(descr *) It's wooden; possibly mahogany.
          (if) (#bowl is pristine) (then)
          A small bowl is placed exactly in its centre.
          (endif)
#sapphire
(name *) sapphire
(stone *) %% This is a custom, story-specific trait.
#amethyst
(an *)      %% The indefinite article should be 'an'.
(name *) amethyst
(stone *)
#bowl
(name *) small bowl
(item *)
(container *)
(* is #on #table)
(descr *) It's a small bowl.
%% Some generic properties of stones:
(item (stone $))
(*(stone $) is #in #bowl)
(descr (stone $Obj))
          (The $Obj) looks very pretty.
(dict (stone $))
          precious stone gem
(plural dict (stone $))
          stones
```

[Copy to clipboard]

Try it! You might want to LOOK, X TABLE, LOOK IN BOWL, GET ALL, PUT STONE IN BOWL, PUT STONE ON TABLE, DROP ALL...

Plural forms

(dict $) and (plural dict $) can be used to add synonyms to objects. In the example above, we added both singular and plural synonyms to all objects belonging to the (stone $) category. A command such as GET STONES will result in every stone being picked up, due to the plural form. In contrast, GET STONE triggers a disambiguating question, where the game asks the player whether they meant to pick up the amethyst or the sapphire.

Note that (dict $) may contain adjectives, but (plural dict $) should only contain nouns (in plural form).

All about appearances

You may have noticed a problem with the last example: When the player examines the bowl, there is no mention of the stones within. For an oldschool game, it may be acceptable to expect the player to SEARCH or LOOK IN the bowl in order to find them. But for modern, narrative-driven games, that approach is generally frowned upon. We could mention the stones in the description of the bowl. But there are two stones, so how do we do that? Do we check whether they are both pristine? I.e.:

```
(descr #bowl)
      It's a small
```

```
(descr #bowl)
 It's a small bowl.
 (if)
 (#sapphire is pristine)
 (#amethyst is pristine)
 (then)
 There are two precious stones in it.
 (endif)
```

[Copy to clipboard]

But what if the player only picks up the amethyst, and then puts it back? The sapphire is still in the bowl, but the story

doesn't mention it, and the library only prints a stock message about the amethyst (because it is no longer pristine). Another option is to add lots of special cases:

```
(descr #bowl)
        It's a small
```

```
(descr #bowl)
 It's a small bowl.
 (if)
 (#sapphire is pristine)
 (#amethyst is pristine)
 (then)
 There are two precious stones in it.
 (elseif)
 *($Stone is one of [#sapphire #amethyst])
 ($Stone is pristine)
 (then)
 There's (a $Stone) in it.
 (endif)
```

[Copy to clipboard]

But this doesn't scale well, if there were more than two precious stones in the bowl to begin with. We also have the option to cop out entirely, and tell the library to narrate these objects already from the start:

```
(descr #bowl)
        It's a small
```

```
(descr #bowl)
 It's a small bowl.
(#sapphire is handled)
(#amethyst is handled)
```

[Copy to clipboard]

Remember, handled is the opposite of pristine in this context. Now, when the player first examines the bowl, the game responds:

It's a small bowl.

An amethyst is in the small bowl.

A sapphire is in the small bowl.

But that's decidedly clunky. A somewhat better approach, although still a cop-out, is to print a vague message that encourages the player to look inside the bowl, without mentioning any details about what's inside:

```
(descr #bowl)
        It's a small
```

```
(descr #bowl)
 It's a small bowl.
 (if) ($ is #in #bowl) (then)
 There appears to be something inside.
 (endif)
```

[Copy to clipboard]

But this will backfire, in a sense, if the player takes the amethyst and then puts it back. Examining the bowl would then result in the following output:

It's a small bowl. There appears to be something inside.

An amethyst is in the small bowl.

Here, the library called attention to the amethyst (handled), but not to the sapphire (pristine). The printed text is technically correct, but while the first paragraph encourages the player to look inside the bowl, the second paragraph takes that incentive away, and the player is mislead to believe that there's nothing in the bowl apart from the amethyst.

A better way to handle this situation is to selectively override the *appearance* message that's printed for handled objects by the library. The text “An amethyst is in the small bowl” originates from a predicate called (appearance $Object $Relation $Parent). The first step is to tell the library to refrain from printing such a message about any object that's currently in the bowl:

```
~(appearance $ #in
#bowl) %% Objects
```

~(appearance $ #in #bowl) %% Objects in the bowl have no appearance.
[Copy to clipboard]

Now that we have silenced those particular messages from the standard library, we must provide our own variant in the description of the bowl. But we have to be careful: With the rule above, we turned off automatic descriptions for any object in the bowl, not just the amethyst and the sapphire. So we have to take care of any foreign objects that might end up there too. In some situations, it might be sufficient to drop a vague hint:

```
(descr #bowl)
      It's a small
```

(descr #bowl)
 It's a small bowl.
 (if) ($ is #in #bowl) (then)
 There appears to be something inside.
 (endif)

~(appearance $ #in #bowl)
[Copy to clipboard]

The library provides a predicate, (list objects $Rel $Obj), that prints a neutral sentence along the lines of “In the small bowl are an amethyst and a sapphire”, or nothing at all if there is no object in the specified location. Thus:

```
(descr #bowl)
      It's a small
```

(descr #bowl)
 It's a small bowl.
 (par)
 (list objects #in *)

~(appearance $ #in #bowl)
[Copy to clipboard]

A more advanced technique is to use a multi-query and a collect statement to print a list of all objects currently inside the bowl:

```
(descr #bowl)
      It's a small
```

(descr #bowl)
 It's a small bowl.
 (collect $Obj)
 *($Obj is #in #bowl)
 (into $List)
 You can see (a $List) in it.

~(appearance $ #in #bowl)
[Copy to clipboard]

An entirely different approach is to allow objects to call attention to themselves, but to replace the stock message with a custom one. This is done by overriding (appearence $ $ $) with a rule that prints text:

```
#sapphire
(appearance * $ $)
```

#sapphire
(appearance * $ $)
 (* is handled)
 A gleaming sapphire catches your eye.
[Copy to clipboard]

That rule did not check the location of the sapphire, so it would override the default message also when the sapphire makes an appearance as part of a room description, or in any other place. Without the line (* is handled), the message would also be printed while the object is still pristine.

When the last two parameters are wildcards (as above), they can be omitted:

```
#sapphire
(appearance *)
```

```
#sapphire
(appearance *)
 (* is handled)
 A gleaming sapphire catches your eye.
```
[Copy to clipboard]

In story code, (appearance $ $ $) rules always take precedence over (appearance $) rules. The way this works internally, is that the library queries (appearance $ $ $), but there is only a single rule definition for that predicate in the library: It queries (appearance $), which in turn contains the default code for calling attention to handled objects.

There's one more subtlety to be aware of: Whenever an appearance-rule succeeds, the object in question gets *noticed* by the library. This binds the appropriate pronoun (usually “it”) to the object. Therefore, if the appearance-rule doesn't print a sentence about the object, it should *fail* in order to prevent the noticing. That is why there is a tilde character in front of the rule head in some of the examples above.

Pristineness of nested objects

By definition, objects are pristine until they are moved from their initial location. That initial location could be e.g. a portable container or the player character. Be aware that if the containing object is moved from its initial location, its contents nevertheless remain pristine.

So, for instance, if the player starts out with a wallet containing a receipt, then both the wallet and the receipt are initially pristine, even though they are part of the player's inventory. If the player drops the wallet, it becomes handled, but the receipt inside remains in its original location—the wallet—and is still considered pristine.

It is therefore the story author who should call attention to the receipt, as part of the description of the wallet, until the receipt is no longer pristine:

```
(descr #wallet)
      Imitation
```

```
(descr #wallet)
 Imitation leather. Jammed zipper.
 (if) (#receipt is pristine) (then)
 One pitiful receipt inside.
 (endif)
```
[Copy to clipboard]

To summarize, movable items are more complicated than other objects, because there is a transfer of responsibility for calling attention to them. At first, while they are pristine, the story author should mention them as a natural part of the prose describing nearby objects (e.g. the room). As soon as they are handled, the library takes over, unless the story author explicitly reclaims control over their appearance.

Clothing

Objects—typically animate ones—can wear clothes. Clothes are objects that have the (wearable $) trait, and therefore the (item $) trait by inheritance.

The *outermost* layer of clothing is modelled by the #wornby relation:

```
(#trenchcoat is
#wornby #bob)
```

```
(#trenchcoat is #wornby #bob)
(#shoes is #wornby #bob)
```
[Copy to clipboard]

Clothes may also be worn #under other garments:

```
(#shirt is #under
#trenchcoat)
```

```
(#shirt is #under #trenchcoat)
(#pants is #under #trenchcoat)
(#socks is #under #shoes)
```
[Copy to clipboard]

Use ($ is worn by $) to check whether an object is currently worn by somebody, at any level of nesting.

By default, clothes are see-through, so Bob's socks are visible despite being located #under his shoes. The outer garment can be made opaque to prevent this:

```
(opaque #trenchcoat)
```

(opaque #trenchcoat)
[Copy to clipboard]

Should the player try to remove a piece of clothing that's underneath another, an attempt is made to remove the outer item first. If this fails, the entire action is stopped.

It's possible to indicate that some garments can't be worn together with others. This is done by adding rules to the (wearing $ removes $) predicate:

```
(wearing #glasses
removes
```

(wearing #glasses removes #sunglasses)
(wearing #sunglasses removes #glasses)
[Copy to clipboard]

This might lead to the following exchange:

> wear sunglasses
(first attempting to remove the glasses)
You take off the glasses.

You put on the sunglasses.

For a larger number of mutually exclusive items, it is more convenient to define a trait:

```
(glasses #glasses)
(glasses
```

(glasses #glasses)
(glasses #sunglasses)
(glasses #monocle)
(wearing (glasses $) removes (glasses $))
[Copy to clipboard]

Other articles of clothing would typically be worn *over* others; this is indicated with the (wearing $ covers $) predicate:

```
(wearing
#trenchcoat covers
```

(wearing #trenchcoat covers #shirt/#pants)
(wearing #shoes covers #socks)
[Copy to clipboard]

In the above example, if the player tries to wear the shoes while already wearing the socks, the socks will end up #under the shoes. Later, if the player tries to remove the socks, an attempt is first made to remove the shoes. But we didn't say anything about putting on socks while wearing shoes, so this is allowed. To properly model the socks-shoes relationship, we would also have to define:

```
(wearing #socks
removes #shoes)
```

(wearing #socks removes #shoes)
[Copy to clipboard]

But this combination of constraints—(wearing $A covers $B) and (wearing $B removes $A)—is so common that the library gives us the option to specify both relations in one go:

```
(#socks goes
underneath #shoes)
```

(#socks goes underneath #shoes)
[Copy to clipboard]

Actually, ($ goes underneath $) does more: It is treated as a transitive relation, meaning that if the shirt goes underneath the jacket and the jacket goes underneath the trenchcoat, then the library can figure out that the shirt must go underneath the trenchcoat. Thus the trenchcoat would automatically cover the shirt, and putting on the shirt would involve removing the trenchcoat first.

But for this to work, the library must be able to invoke ($ goes underneath $) in a multi-query, with the second parameter unbound. Therefore, be aware that if the second parameter is a trait, it needs to be prefixed by an asterisk:

```
(#underpants goes
underneath *(pants
```

```
(#underpants goes underneath *(pants $))
((pants $) goes underneath #trenchcoat)
```

[Copy to clipboard]

Advanced technique: Multiple covers

So far in this chapter, we've tacitly assumed that an article of clothing can only ever be worn underneath a single parent. This is inherent in the object-tree model, but it rules out situations such as a spandex one-piece worn underneath a shirt and a pair of trousers at the same time. The library doesn't support such a use case directly, but it can be implemented with the help of before- and after-rules:

```
#onepiece
```

```
#onepiece
(* goes underneath #shirt/#trousers)
(before [remove *])
 (* is worn by #player)
 *($Outer is one of [#shirt #trousers])
 ($Outer is worn by #player)
 (first try [remove $Outer])
(after [remove #shirt/#trousers])
 (* is worn by #player)
 *($Outer is one of [#shirt #trousers])
 ($Outer is worn by #player)
 (now) (* is #under $Outer)
```

[Copy to clipboard]

Onwards to “Chapter 5: Moving around” • Back to the Table of Contents