Chapter 3: Traits

(Custom traits • Linguistic predicates and traits • Full names • Standard traits for categorizing objects)

Traits are single-parameter predicates used by the standard library to categorize objects. For instance, the trait (supporter $) succeeds for an object that the player is allowed to put things on top of. The rule bodies of traits never contain any side effects (such as printing text).

As far as the Dialog compiler is concerned, traits are just ordinary predicates.

We declare that the table is a supporter by ensuring that the query (supporter #table) succeeds. The easiest way to achieve this is with the rule definition:

```
(supporter #table)
```

(supporter #table)
[Copy to clipboard]

Trait predicates are supposed to fail for objects that aren't part of the category. For instance, if the player ever tries to put an object on top of themselves, the library will at some point make a query to (supporter #player) to check whether this is allowed. That query will fail (unless, of course, we also add (supporter #player) to our program).

In the standard library, as a convention, traits are named with a category noun or adjective preceding the object, e.g. (openable $), whereas dynamic per-object flags have the object first, followed by "is" and an adjective, e.g. ($ is open). Most per-object flags also have a negated form, whereas traits do not. You may of course ignore these conventions in your own code, if you wish. The Dialog programming language makes no syntactical difference between traits and dynamic flags, and (now) (openable #door) is under most circumstances a perfectly legal statement, transforming the trait into a dynamic predicate. The purpose of the naming convention is to make it easier to understand library code at a glance.

Custom traits

We can easily define our own traits. For instance, we can invent a trait called (fruit $), to determine whether an object is a fruit. Let's define:

```
(fruit #apple)
(fruit #orange)
```

(fruit #apple)
(fruit #orange)
[Copy to clipboard]

Now, querying (fruit $) for some object will succeed if that object is the apple or the orange, but fail otherwise. Then we could have a generic object description for fruit:

```
(descr $Obj)
(fruit $Obj) It looks
```

(descr $Obj) (fruit $Obj) It looks yummy!
[Copy to clipboard]

When the standard library wants to print the description of an object, it queries (descr $) with the desired object as parameter. Dialog tries each (descr $) rule definition in turn, in program order. When it gets to the fruit-rule above, the rule head matches, binding $Obj to the object in question. Dialog starts to execute the rule body, which immediately makes a query to (fruit $) for the given object. If that query succeeds, execution proceeds with the rest of the rule body, and "It looks yummy!" is printed. If it fails, the rule fails, and Dialog resumes to search through the remaining (descr $) rules in the program.

When a rule body begins with a query, Dialog provides syntactic sugar that allows us to move the query into the rule head. The following line of code is equivalent to the one in the previous example:

```
(descr (fruit $))
It looks yummy!
```

(descr (fruit $)) It looks yummy!
[Copy to clipboard]

It is possible to define *inheritance* relations between traits:

```
(berry #blueberry)
(berry #cherry)
```

```
(berry #blueberry)
(berry #cherry)

(fruit #apple)
(fruit *(berry $))
```
[Copy to clipboard]

That last rule is equivalent to:

```
(fruit $Obj)
*(berry $Obj)
```

```
(fruit $Obj) *(berry $Obj)
```
[Copy to clipboard]

What it means is: $Obj is a fruit given that $Obj is a berry. The asterisk (*) indicates a multi-query. If you haven't gone down the rabbit hole of multi-queries yet, just memorize that inheritance definitions need to have an asterisk in front of the query in the rule body. The asterisk makes it possible to loop over every fruit in the game, for instance like this:

```
(intro)
	Welcome to the
```

```
(intro)
 Welcome to the fruit game! Here you will meet \(and possibly eat\):
 (line)
 (exhaust) {
 *(fruit $Obj)
 (The $Obj).
 (line)
 }
```
[Copy to clipboard]

We can easily add inheritance relations between our own traits and those of the standard library, in either direction. The following line of code adds a rule to the edibility-trait of the standard library, saying that all fruit are edible. This allows the player to EAT THE BLUEBERRY:

```
(edible *(fruit $))
```

```
(edible *(fruit $))
```
[Copy to clipboard]

Here is a complete example with edible fruit and berries:

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

```
#player
(current player *)
(* is #in #room)

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

#table
(name *)		wooden table
(descr *)		It's a sturdy wooden table.
		(if) ($ is #on *) (then)

		There seems to be something on it.
		(endif)
(supporter *)
(* is #in #room)

#blueberry
(name *)		blueberry
(berry *)

#cherry
(name *)		dark red cherry
```

```
(berry *)

#apple
(name *)        green apple
(fruit *)

%% All berries are fruit, and all fruit are edible.
(fruit *(berry $))
(edible *(fruit $))

%% The following are not trait inheritance definitions (descr, dict, and the
%% initial location are not traits), so no asterisk is required.

(descr (fruit $)) Yummy!
(dict (fruit $))  fruit
(dict (berry $))  berry

(*(fruit $) is #on #table)

(intro)
                  Welcome to the fruit game! Here you will meet \(and possibly eat\):
                  (line)
                  (exhaust) {
                  *(fruit $Obj)
                  (The $Obj).
                  (line)
                  }
```

[Copy to clipboard]

Try this game! Try to examine the table, then SEARCH or LOOK ON it, then perhaps EAT BERRY or EAT CHERRY, and see if the description of the table really changes when it's empty.

Did you notice that it wasn't possible to pick up the fruit in this game? They were presumably eaten directly off the table. Objects that can be picked up are called *items*, and we will discuss this trait at length in Chapter 4. But first, we will take a step back and see how the various traits provided by the standard library fit together.

Linguistic predicates and traits

To print the name of an object, most of the time you'll want to use a predicate called (the $). This prints the correct determinate article for the given object, followed by its name. So, given the following object definition:

```
#apple
(name *)        green
```

```
#apple
(name *) green apple
```

[Copy to clipboard]

querying (the #apple) would result in the following text being printed:

the green apple

To print the name of an object together with an indeterminate article, use (a $) instead. Querying (a #apple) results in:

a green apple

If you want the article to start with an uppercase letter, use (The $) or (A $), respectively.

The standard library offers a lot of flexibility when it comes to declaring object names. We have seen the (name $) predicate, which provides the actual noun expression. But a number of *linguistic traits* affect how that name gets printed:

(an $)

Specifies that “an” is the correct indeterminate article for this object name.

(proper $)

Specifies that this is a proper noun, so that neither “a” nor “the” should appear before it.

(plural $)

Specifies that this is a plural noun, so that “some” should be used instead of “a” or “an”. This also changes the verb forms printed by certain predicates (see below).

(pair $)

Inherits all the properties of a plural noun, but also changes the indeterminate article “a” into “a pair of”.

(uncountable $)

Specifies that the indeterminate article “some” should be used, but that the noun behaves like a singular in every other respect.

(singleton $)

Specifies that “the” should be used, even in situations where “a” or “an” are usually called for.

(your $)

Specifies that “your” should be used instead of “a” or “the”.

Some examples:

```
#orange
(an *)
```

```
#orange
(an *)
(name *) orange

#book
(proper *)
(name *) A Clockwork Orange

#bookshelves
(your *)
(plural *)
(name *) bookshelves

#boots
(pair *)
(name *) boots

#water
(uncountable *)
(name *) water

#sun
(singleton *)
(name *) sun
```

[Copy to clipboard]

To use an object name in a sentence, it is often necessary to select a matching verb form. Predicates are available for this, as well as for printing pronouns. To print the correct personal pronoun, for instance, use (it $). This will print the word “it” by default, but if the object has the plural trait, it will print the word “they” instead. And if the object happens to be the current player character, the word “you” is printed. There are several such predicates, corresponding to the rows of the following table. The columns displayed here, corresponding to how the linguistic traits have been set up, are not exhaustive.

Predicate	Singular	Plural	Current player
(a $)	a/an	some	yourself
(A $)	A/An	Some	You
(the $)	the	the	yourself
(The $)	The	The	You
(it $)	it	they	you
(It $)	It	They	You
(its $)	its	their	your
(Its $)	Its	Their	Your
(itself $)	itself	themselves	yourself
(them $)	it	them	you
(that $)	that	those	yourself
(That $)	That	Those	You
(is $)	is	are	are
(isn't $)	isn't	aren't	aren't
(has $)	has	have	have
(does $)	does	do	do
(doesn't $)	doesn't	don't	don't
(s $)	(no space) s		

(es $)	(no space) es		
(it $ is)	it is	they're	you're
(the $ is)	(the $) is	(the $) are	you're
(The $ is)	(The $) is	(The $) are	You're
(That's $)	That's	Those are	You're

The predicates (s $) and (es $) are used for attaching verb endings, e.g. (The $Obj) ponder(s $Obj) (its $Obj) existance.

For each of the objects in the previous example, the expression:

```
You see (a
$Obj). (The $Obj is)
```

You see (a $Obj). (The $Obj is) drawing attention to (itself $Obj).
[Copy to clipboard]

would produce:

- You see an orange. The orange is drawing attention to itself.
- You see A Clockwork Orange. A Clockwork Orange is drawing attention to itself.
- You see your bookshelves. Your bookshelves are drawing attention to themselves.
- You see a pair of boots. The boots are drawing attention to themselves.
- You see some water. The water is drawing attention to itself.
- You see the sun. The sun is drawing attention to itself.

And if $Obj is the current player character, the output is:

- You see yourself. You're drawing attention to yourself.

When the parameter is a list of several objects, such as [#orange #boots #book], that's handled too:

You see an orange, a pair of boots, and A Clockwork Orange. The orange,
the pair of boots, and A Clockwork Orange are drawing attention to
themselves.

There are two additional traits, (male $) and (female $), that modify the pronouns accordingly.

Note: Don't confuse (a $) with (an $)! The former is a predicate for printing the indeterminate article (usually “a”) followed by the name of the object. The latter is a trait, specifying that “an” should be used instead. Thus, somewhere in the standard library, a rule definition for (a $) contains a query to the predicate (an $) in order to determine what article it needs to print.

Full names

Two additional predicates deserve to be mentioned here: Whenever the standard library describes an action (e.g. to narrate an automatic action such as opening a door before walking through it, or as part of a disambiguating question), it prints the names of the involved objects using (the full $) or (a full $). These predicates print the object name using (the $) or (a $), and then, if the query (clarify location of $) succeeds for the object, some additional information pertaining to its location is printed. By default, this flag is enabled for all non-singleton doors, so that the game might ask the player: Did you want to open the door to the north or the door to the east?

Standard traits for categorizing objects

The standard library categorizes objects using a system of traits. Most of these traits model one of the following three different aspects of an object:

- Where it may appear in the object tree.
- Whether the object can be manipulated at all.
- What actions may be carried out on the object.

Traits that determine where an object may appear in the object tree:

Arrows indicate inheritance.

Container objects allow the player to put other objects #in them. *Supporter* objects allow the player to put other objects #on them.

Actor containers are containers that the player is allowed to enter. *Actor supporters* are supporters that the player is allowed to climb on top of.

Rooms are actor containers with no parents in the object tree. They are organized into a map using connections; this will be explained in the chapter on moving around. Some of those connections involve *doors* (physical doors or other

kinds of gatekeepers). Doors are conceptually located in the liminal space between rooms, but for practical reasons they appear as children of rooms in the object tree: When the player enters a room, the library automatically moves adjacent door objects inside the room object.

A less commonly used trait is (seat $), the category of objects that give the player a place to sit down. These can be divided into *on-seats* (that the player may sit on) and *in-seats* (that the player may sit in, such as armchairs). Some behaviour is common to all seats, for instance that when the player tries to go up, this is interpreted as a desire to leave the seat.

Note that e.g. an on-seat is a kind of seat, but an on-seat is also a kind of actor supporter. If you are familiar with object-oriented programming, you may recognize this as a case of *multiple inheritance*. In class-based programming languages, where objects have inherent types that determine what code to execute, multiple inheritance can be problematic. But in languages such as Dialog, where rules are always applied in source code order, this is not the case.

A *vehicle* is an object that moves with the player if the player attempts to go somewhere while the vehicle object is their parent. Usually, vehicles are either actor supporters or actor containers, but this is not enforced. For an example of a situation where a vehicle is neither a container or a supporter, the player might be #heldby some giant non-player character, and directing that character to move around.

The standard library defines twelve *directions* and seven *relations*. The directions are #north, #northwest, #west, #southwest, #south, #southeast, #east, #northeast, #up, #down, #in, and #out. The relations are #in, #on, #wornby, #under, #behind, #heldby, and #partof. Note that #in is both a relation and a direction.

Directions and relations are never part of the object tree. They only appear as predicate parameters, and inside action expressions. Both directions and relations have printed names, (name $), but the relations also have several *name variants*:

Relation	**(name $)**	**(present-name $)**	**(towards-name $)**	**(reverse-name $)**
#in	in	inside of	into	out of
#on	on	on top of	onto	off
#wornby	worn by	worn by	worn by	off
#under	under	under	under	out from under
#behind	behind	behind	behind	out from behind
#heldby	held by	held by	held by	away from
#partof	part of	part of	part of	away from

Traits that determine whether an object can be manipulated at all:

Dialog allows you to model objects that are understood by the parser, but do not really exist in the game world. For instance, a room description might call attention to a cockroach scuttling over the floor and disappearing into a hole in the wall. If the player then tries to do anything to the cockroach (such as EXAMINE it), a response message about the cockroach not being here is preferable to a generic parser error.

Actions involving (not here $) objects generally fail with the message: “(The $Obj) is not here.” Actions that involve the manipulation of (out of reach $) objects fail with the message: “You can't reach (The $Obj).” For (intangible $) objects, the message is: “(The $Obj) is intangible.”

For many actions, the player is allowed to refer to a collection of objects using the word ALL. Objects that are (excluded from all $) are silently omitted from such collections. This also applies to objects marked as (not here $) or (not reachable $), via trait inheritance.

Most objects in the game world cannot be picked up—by default, only items can. The standard response when the player tries to pick up a non-item is: “You can't take (the $Obj).” But if the object is (fine where it is $), that error message is replaced by: “That's fine where it is.”

Topic objects are recognized by the parser in certain grammatical contexts (e.g. ASK BOB ABOUT ... or LOOK UP ... IN THE MANUAL), even if they are currently out of scope.

Traits that determine what actions may be carried out on an object:

Opaque objects hide their contents when closed. *Openable* objects can be opened (unless they are locked) and closed. *Lockable* objects can be locked and unlocked. Lockable objects are openable by inheritance, and openable objects are opaque (and start out closed). But such inheritance relations can be overridden on a per-object basis.

Items can be picked up by the player. Anything that is carried by the player (usually items) can be dropped, or put inside containers or on top of supporters. *Wearable* objects can be worn or removed. Wearable objects are items by inheritance.

Pushable objects can be pushed from room to room. *Switchable* objects can be turned on or off. *Sharp* objects can be used to cut other objects.

Edible objects can be eaten, which causes them to be removed from the object tree. The player may drink *potable* objects, but those are not removed (the player only takes a sip).

Consultable objects can be consulted about various subject matters. That is, objects and other topics can be looked up in them.

Animate objects can be instructed to do things, although they will refuse by default. They can also be talked to, given things, or shown things. Again, the default implementations of those actions merely print a stock response.

Male and *female* are better described as linguistic traits, as their main function is to replace the default pronouns. They are included in the diagram because of their inheritance relation to the animate trait.

Onwards to “Chapter 4: Items” • Back to the Table of Contents