An Inform tutorial

Gareth Rees, April 1995 (slightly updated to Inform 6 by Graham Nelson in November 1996).

1. Introduction

This tutorial aims to give you an introduction to the use of the programming language Inform to write reasonably complex adventure games. I'm going to take a puzzle which is tricky to implement in Inform, and try to take you through the steps needed to turn it into working code.

You'll need to have the following background to appreciate this tutorial:

- You should be familiar with adventure games. I'm going to concentrate on showing how to code objects up in Inform; if you need guidelines on designing puzzles, structuring games, and prose style, then you should read Graham Nelson's essay `The Craft of the Adventure'.
- You can already write programs in some computer language (C would be most helpful, because of the similarity between the syntax of C and that of Inform).
- You have read (even if you haven't fully understood) Graham Nelson's *The Inform Designer's Manual* which describes the Inform language in detail. References to sections in the *Designer's Manual* are given like `(DM 14)'.
- You need to have release 6.01 or later of the Inform compiler and revision 6/1 or later of the Inform library in order to compile and run the examples.

This tutorial will start to write an adventure game based upon the children's novel *Through the Looking Glass* by Lewis Carroll. Carroll died in 1898, so the novel is out of copyright, and I will be free to steal shamelessly from it.

2. Before starting to program

Interactive fiction authors differ on how much planning you should do before you start to write code. Some people like to plunge straight into the coding process with just the vaguest idea in their heads; others prefer to plan in enormous detail before they turn their computer on. Here I'll take the approach of sketching out the solution to a puzzle in advance, and filling in the details when I write the code.

Through the LookingGlass starts out in a room in Alice's house, in which Alice is playing with two kittens, one black and one white. The text mentions an armchair, a ball of worsted, a fireplace, a hearthrug, a chess set (with red and white pieces), a clock, a cat called Dinah, and a lookingglass above the fireplace. It should be possible to involve many of these objects in the puzzle.

The puzzle will be to get through the lookingglass. How can that be made difficult? Perhaps the mantelpiece is too high for Alice to climb up onto without a chair, and the armchair needs to be pushed over to the mantelpiece. But the kittens get in the way of the chair, and Alice wouldn't want to hurt them, would she? So she has to distract the kittens by getting one of them tangled up in the ball of worsted, and giving the other one a chess piece to play with. The chess pieces have become lost, but she can find the red queen under the hearthrug.

That seems like enough complexity for the first room of a game.

3. Getting started: the outline of a game

I won't try to design all those objects at once; instead I'll start out with a minimum of code and slowly add complexity to it. The idea will be that as often as possible I will have a game that I can compile and play to check that I'm proceeding along the right lines. Frequent checking of code is very important, unless you're a very skilful programmer, because adventure games involve very complicated interactions between the states of different objects, and it's very easy to forget something, or fail to check for some case, especially since you probably don't know in very much detail what the library is doing behind your back.

My first attempt is nothing more than a few simple modifications to the simplest possible Inform game (see (DM 7)).

```
Constant Story "THROUGH THE LOOKING GLASS";
Constant Headline "^An Interactive Tutorial^by Gareth Rees^";
Constant DEBUG;
```

The `Story' and `Headline' constants are printed by the library in the opening banner. The `DEBUG' constant gives me access to some debugging commands while testing (that line should be removed or commented out when the game is eventually released).

```
Include "parser";
Include "verblib";
Include "grammar";
```

The Inform library is broken up into three parts: the Parser, the Verb Library, and the Grammar. The reason why it is split into parts is that it may be necessary to set constants or define functions after including one library and before including the other (DM A11).

I need at least one object to be able to compile and run a game, so I define the initial location.

```
Object  Drawing_Room "Drawing room"
 has    light
 with   name "snow",
        description "The gentle sound of snow against the window pane
            suggests that it's cold outside, and you're glad to be here
            in the warmth. The drawing-room is reflected in the large
            looking-glass on the wall above the mantelpiece, and a very
            comfortable room it is too, with a warm hearth, a soft rug,
            and an arm-chair that you can curl up and sleep in.";
```

Note that this room description mentions a number of objects; when I define these objects I'll give them the `concealed' or `scenery' attributes to stop the library from mentioning them twice - I want to avoid output like the following.

```
    ... The drawing-room is reflected in the large looking-glass ...

    You can also see a looking-glass here.
```

The only obligatory piece of code is the `Initialise' function, which sets the location of Alice to the initial room, and prints a welcoming message (traditionally preceded by a handful of newlines because of a bug in the now little-used ITF Infocom interpreter, which doesn't display anything until it starts to scroll the screen, which is after printing 20 lines or so).

```
[ Initialise;
    location = Drawing_Room;
    print "^^^^^It's a cold winter day outside, but in the looking-glass
        house it's summer. All you need to do is pretend there's a way of
        getting through into it somehow...^^";
];
```

This gives me a program, alice1.inf, that can be compiled and tested, although it doesn't do very much yet.

4. Common difficulties

The most complicated parts of the Inform language are to do with which order routines are evaluated during play, and what to return from routines to affect this order. These parts cause more problems to beginning programmers than any of the other features of the language.

4.1 Return codes

A routine inside an object returns 0 (false) by default, and a routine outside of an object returns 1 (true) by default. You can think of this as meaning that there's an invisible `rfalse' statement just before the closing square bracket of a routine inside an object, and at the end of each block of code dealing with an action (and an invisible `rtrue' just before the closing square bracket of a routine outside an object), so that

```
Object  thing
 with   ...
        before [;
         Action1: ...
         Action2: ...
        ];
```

really means

```
Object  thing
```

```
 with   ...
        before [;
         Action1: ... rfalse;
         Action2: ... rfalse;
        ];
```
and
```
[ Routine; ... ];
```
really means
```
[ Routine; ... rtrue; ];
```
You can override this default using the `return', `rtrue' and `rfalse' statements, or by omitting the `print' instruction from a print statement, because
```
    "Text in double-quotes";
```
really means
```
    print "Text in double-quotes"; new_line; rtrue;
```
The use to which the return code of a function is put varies from function to function, but usually when the library calls a function of yours, then you return 0 to mean `apply the usual library rules', and nonzero to mean `do something special' or maybe `don't do anything at all'.

Here are the meanings of the return codes of the most commonly-used routines:

`before'
`life'
: True (1) means that the routine has successfully and completely dealt with the player's input, and the library shouldn't do anything more.
False (0) means that the usual library rules should now apply to the player's input (note that this doesn't necessarily mean that the routine has done nothing).

`after'
: True (1) means that the `after' routine has printed a suitable response to the player's input, and the library shouldn't do anything more.
False (0) means that the usual library message should now be printed (again, this doesn't necessarily mean that the routine has done nothing).

`describe'
: True (1) means that the `describe' routine has printed a suitable description. The library shouldn't add anything else.
False (0) means that the library should print the usual description of the object.

In an object's `before' routine, it's very common to want to execute some code, print some text to tell the player what has happened, and then to return 1 to prevent the library from doing anything else. A good trick in this situation is to omit the `print' keyword from the last print statement. Much of the code in the example game developed here uses this trick.

If in doubt, read the *Designer's Manual*, and look carefully at the example games to see how they use return codes.

4.2 Evaluation order

Here's a summary of what the Inform library does when the player types a command (DM 9).

1. Parse the command, setting up the variables `action', `noun' and `second' appropriately.
2. Call `GamePreRoutine' (if there is one). If this returns nonzero, stop here.
3. Call the `before' routine of the player. If this returns nonzero, stop here.
4. Call the `before' routine of the current location. If this returns nonzero, stop here.
5. Call the `before' routine of the object `noun' (if any). If this returns nonzero, stop here.

6. Carry out the action, if possible. If there was no action to carry out (i.e., the action didn't do anything interesting, for example `jump' or `smell', or the action failed, for example `put apple in banana') then print a failure message and stop here.
7. Call the `after' routine of the player. If this returns nonzero, stop here.
8. Call the `after' routine of the current location. If this returns nonzero, stop here.
9. Call the `after' routine of the object `noun' (if any). If this returns nonzero, stop here.
10. Call `GamePostRoutine' (if there is one). If this returns nonzero, stop here.
11. Print a message describing what happened at stage 6.

It will be seen from this table that `before' routines should be used to change completely the effect of a particular action, and that `after' routines should be used to change the message associated with a particular action, or to cause something else to happen after an action has been successfully completed.

For example (after *Adventure*), a delicate ming vase that shattered when the player dropped it could be coded by using the `after' routine of the vase to look for the `Drop' action. Or (after *Hitchhiker's*) a room with holes in the floor through which any dropped object fell could be coded by using the `after' routine of the room to look for the `Drop' action.

5. Adding objects

Now that I have a room, I can fill it up with objects. I'll start with the simplest objects, and work up to the most complicated (the kittens). I'll try to add the objects in such a way that the code can be tested after each addition (but if two objects refer to each other this won't be possible).

5.1 The red queen: a simple object

The simplest objects just have a `name' and a `description', and the red queen is such an object. The following definition appears after the drawingroom.

```
Object  red_queen "red queen"
 with   name "red" "queen",
        description "She's a fierce little chess piece.";
```

Since she's absent from the game to start with, I don't specify which object is her parent (Inform will ensure that her parent is `nothing').

5.2 The chess board

The chess board is not quite so simple. Because it doesn't appear in the room description, I'll give it an `initial' description. And Alice might try to `put red queen on chess board', so I'll make it a `supporter'.

```
Object  chess_board "chess board" Drawing_Room
 has    supporter
 with   name "chess" "board" "checker" "chequer" "chessboard",
        initial "An abandoned chess board lies on the floor.",
        description "It's left here from the game you were playing just
            now, but the pieces are all missing - the kittens will insist
            on playing with them.";
```

5.3 The hearth: a scenery object

The fireplace is mentioned in the initial room description, so I probably ought to provide a `scenery' object to represent it. The object plays no role in the game; it's just there for decoration.

```
Object  hearth "hearth" Drawing_Room
 has    scenery
 with   name "hearth" "fire" "place" "fireplace",
        description "Looking at the hearth, you wonder if they have a
            hearth in the looking-glass house. You can never tell by
            looking, unless your fire smokes, and then smoke comes up in
            the looking-glass room too - but that may be only pretence,
            just to make it look as if they had a fire.";
```

5.4 The rug: a complex object

Often, the best way to deal with more complicated objects is to consider all the commands that the player might type that ought to make the object behave differently from the default way in which the library handles that action.

In order to do this, it helps to be familiar with the operation of the library. Appendix A7 of the Designer's Manual lists all the actions that the library knows how to deal with, and it's probably worth looking briefly at the source code (in the Verb Library) to see exactly what they all do - in the cases of `Take', `Enter', and `Insert' the algorithm is very complex. You may also need to look at the Grammar file to see which actions are generated by which player commands.

Also, when code fails to do what you intended, you'll often find it worthwhile to look at the library source to see exactly what's going on.

In any case, the rug needs to respond to the following commands.

- `take rug' - so I make it `static', and also provide a `before' routine for the `Take' action to provide a better message than the library's default message (`That's hardly portable').
- `pull rug' and `push rug' - I provide a `before' routine that deals with these.
- `put red queen on rug' - make it a `supporter'.
- `stand on rug' - make it `enterable'.
- `look under rug' - provide a `before' routine for the `LookUnder' action (this routine prevents Alice from looking under the rug while she is standing on it!).

Some other points of interest:

- I call the rug `rug' rather than `hearthrug' because otherwise I will end up with confusion over whether the word `hearth' refers to the fireplace or to the hearthrug.
- The rug has the `concealed' attribute because it appears in the room description.
- The `general' attribute is used to stop Alice finding the red queen under the rug more than once.

```
Object  rug "rug" Drawing_Room
 has    concealed static supporter enterable
        ! general if you've found the red queen under it
 with   name "hearthrug" "hearth-rug" "rug" "indian" "arabian" "beautiful"
            "soft",
        description "It's a beautiful rug, made in some far off country,
            perhaps India or Araby, wherever those might be.",
```

```
        before [;
         Take: "The rug is much too large and heavy for you to carry.";
         Push,Pull: "But a hearth-rug is meant to be next to the hearth!";
         LookUnder:
            if (player in self)
                "You try to lift up a corner of the rug, but fail. After
                a while, you realise that this is because you are
                standing on it. How curious the world is!";
            if (self hasnt general) {
                give self general;
                move red_queen to player;
                "You lift up a corner of the rug and, peering underneath,
                discover the red queen from the chess set.";
            }
        ];
```

5.5 The armchair

My puzzle involves not being able to move the armchair if the kittens are playing near it. But since I'm saving the kittens for last (because they're the most complicated objects) I can't do a full implementation of the armchair. Instead, I'll do the best I can, leaving a comment in the place where I ought to check for the kittens. Here the relevant commands to consider are the following.

- `take armchair' - make it `static'.
- `put chess board on armchair' - make it a `supporter'.
- `enter armchair' - make it `enterable'.
- `push armchair' or `pull armchair' - these (if successful) just toggle the position of the armchair; either it's close enough to the mantelpiece to climb up, or it isn't. I'll use the `general' attribute to indicate which.

```
Object  armchair "arm-chair" Drawing_Room
 has    static concealed supporter enterable
        ! general if its by the mantelpiece
 with   name "arm" "chair" "armchair" "arm-chair",
        description "It's a huge arm-chair, the perfect place for a kitten
            or a little girl to curl up in and doze.",
        before [;
         Push,Pull:
            ! code to check for the kittens
            if (self has general) {
                give self ~general;
                "You push the arm-chair away from the hearth.";
            }
            give self general;
            "You push the arm-chair over to the hearth.";
        ];
```

5.6 The mantelpiece

For the mantelpiece, I need to consider the following commands.

- `climb on mantelpiece' - make it `enterable'. But since its so high up, I should prevent Alice from entering it unless she's standing on the armchair. Also, she can't climb up if she's holding anything in her hands.

- `put red queen on mantelpiece' - make it `supporter'. If it's too high for Alice to climb up onto unless she's on the armchair, it's probably too high to put things on too. So a similar check is needed.
- `take red queen from mantelpiece' - the same discussion applies.

```
Object  mantelpiece "mantelpiece" Drawing_Room
 has    concealed supporter enterable
 with   name "mantel" "mantelpiece",
        description "It's higher off the ground than your head, but it
            looks wide enough and sturdy enough to support you.",
        before [;
         Enter,Climb:
            if (player notin armchair)
                "The mantelpiece is much too high to climb up onto.";
            if (armchair hasnt general)
                "You can't reach the mantelpiece from here.";
            if (children(player) > 0)
                "Your hands are too full.";
         PutOn,LetGo:
            if (player notin self && (player notin armchair ||
                armchair hasnt general))
                "The mantelpiece is so high that you can't reach.";
        ];
```

5.7 The mirror

The mirror presents the following problems.

- `examine mirror' - the amount of the room that Alice can see in the mirror varies, depending on where she is - if she's standing on the floor, she can only see the ceiling in the mirror. If she's on the armchair she can see the whole room in the mirror. If she's on the mantelpiece, then she should be able to see the mirror beginning to melt away.
- `touch mirror', `push mirror', `pull mirror' - Alice shouldn't be able to touch the mirror unless she's standing on the mantelpiece. But if she's standing on the mantelpiece, then her hand should go right through.
- `take mirror' - make it `static'.
- `enter mirror' - if I get around to writing the next part of the game, then this will cause Alice to move to the lookingglass house (using the `PlayerTo' function). For the moment, I just cause her to win the game.

```
Object  mirror "looking-glass" Drawing_Room
 has    static concealed
 with   name "mirror" "looking" "glass" "looking-glass",
        description [;
            if (player in mantelpiece)
                "Strangely, the glass is beginning to melt away, just
                like a bright silvery mist.";
            if (player in armchair)
                "In the looking-glass you can see the drawing-room of the
                looking-glass house. What you can see is very much the
                same as this drawing-room, only all reversed, left for
                right. But you are sure that out of the corners of the
                glass, where you can't see, the looking-glass world is
                quite different from yours.";
            "In the looking-glass you can see the ceiling of the
            drawing-room of the looking-glass house. It looks much the
            same as the ceiling of your drawing-room.";
        ],
```

```
        before [;
            if (action ~= ##Examine && player notin mantelpiece)
                "You can't reach the looking-glass from where you're
                standing.";
         Touch,Pull,Push:
            "Your hand goes right through the silvery mist!";
         Enter:
            ! Really, move Alice to the looking-glass house.
            deadflag = 2;
            "Your hand goes right through the silvery mist, and in
            another moment the rest of you follows, and you are through
            the glass...";
        ];
```

5.8 The ball of worsted

(Worsted is a kind of fine wool used for embroidery.) This object will start out neatly rolled into a ball, but when its given to a kitten, it will get into a tangled state. I'll use the `general' attribute to indicate whether its tangled or not. I want to be able to take care of the player typing `untangle worsted' or `roll up worsted', so I define a new action, and some new grammar to go with it (after the inclusion of the grammar library):

```
Verb "roll" "untangle" "wind"
    * noun -> Untangle
    * "up" noun -> Untangle
    * noun "up" -> Untangle;
```

and write a basic routine to deal with `untangle chess board', and so on.

```
[ UntangleSub; "What curious ideas you have!"; ];
```

Now I can create the ball of worsted; note that like the mirror, its description can vary.

```
Object  worsted "ball of worsted" Drawing_Room
        ! general if its in a tangle
 with   name "ball" "of" "worsted" "fine" "blue" "wool",
        initial "A discarded ball of worsted lies on the floor here.",
        description [;
            if (self has general)
                "It's in a terrible tangle. All that time you spent
                rolling it up, and now look at it!";
            "It's a ball of fine blue wool, all rolled up in preparation
            for some embroidery.";
        ],
        before [;
         Untangle:
            give self ~general;
            "You're as quick as can be at rolling up balls of wool,
            though you say so yourself! Soon it's neat and tidy again.";
        ];
```

6. Testing, part 1

Now I have a program, alice2.inf, which, although incomplete, will compile and run, and allow me to attempt a few actions. I advise you to compile this fragment of a game, and test it yourself to see what problems you can discover (there are quite a few!), before I start to list the ones I found.

1.
```
> throw queen at mirror
You can't reach the looking-glass from where you're standing.
```

This is easy to correct; just add the following to the mirror's `before' routine:

```
         ThrownAt: "You don't want seven years' bad luck, do you?";
```

and change `action ~= ##Examine' to `action ~= ##Examine or ##ThrownAt'.

2.
```
> get on chair
```

```
    You get onto the arm-chair.

    > push chair
    You push the arm-chair over to the hearth.
```

Oops. Alice shouldn't be able to push the chair if she's in it, or on the mantelpiece, or on the rug! Add the following at the start of the armchair's `Push,Pull' action.

```
            if (player notin Drawing_Room)
                "You'll have to get off ", (the) parent(player),
                    " first.";
 3. > climb mantelpiece
    I don't think much is to be achieved by that.
```

The `Climb' action doesn't do the right thing here (see the Verb Library for details), so I have to add my own code to the end of the mantelpiece's `Enter,Climb' action.

```
            move player to mantelpiece;
            "You scramble up onto the mantelpiece.";
 4. > get on mantelpiece
    But you're already on the arm-chair.
```

The solution to the previous problem also solved this problem.

```
 5. > enter chair
    You get onto the arm-chair.

    > look under rug
    You find nothing of interest.
```

Alice shouldn't be able to look under the rug when she's in the armchair or on the mantelpiece. So add the following to the start of the rug's `LookUnder' action.

```
            if (player in mantelpiece || player in armchair)
                "You're unable to reach the rug from here.";
 6. > examine red queen
    She's a fierce little chess piece.

    > drop her
    I'm not sure what "her" refers to.
```

I need to give the queen the `female' attribute.

After making the above changes, Alice can climb onto the mantelpiece, and some more checking reveals a few more bugs.

```
    > enter mantelpiece
    You scramble up onto the mantelpiece.

    > down
    You'll have to get off the mantelpiece first.

    > get on chair
    But you're already on the mantelpiece.

    > get board
    Taken.

    > get off
    You are on your own two feet again.
```

I can take care of `down' and `get off' (and also `exit' and `out') by adding a `before' routine to the drawingroom.

```
        before [;
            if (player notin Mantelpiece) rfalse;
         Exit,Go:
            if (noun == d_obj or out_obj)
                "That's not the way to get down from a mantelpiece!";
        ];
```

And I can introduce code to the armchair's `before' routine allow Alice to climb down from the mantelpiece to the armchair.

```
          Climb,Enter:
             move player to armchair
             "You jump into the warm and comfortable arm-chair.";
```

The final problem (being able to touch objects on the floor while on the mantelpiece) is very tricky to solve. Inform doesn't support the idea of `physically reachable from here', but we can amend the `before' routine of the drawingroom to look like this:

```
         before [;
             if (player notin Mantelpiece) rfalse;
          Exit,Go:
             if (noun == d_obj or out_obj)
                  "That's not the way to get down from a mantelpiece!";
          Examine,Enter,ThrowAt,ThrownAt: ;
          default:
             if (inp1 ~= 1 && noun ~= 0 && Inside(noun,mantelpiece) == 0)
                 "You can't reach ", (the) noun, " from up here.";
             if (inp2 ~= 1 && second ~= 0 && Inside(second,mantelpiece) ==
0)
                 "You can't reach ", (the) second, " from up here.";
         ];
```

The actions `Examine', `Enter', `ThrowAt' and `ThrownAt' are all actions that will work regardless of where the object is, but all remaining actions will only work if the objects are to hand. The code that checks that they really are to hand has to be careful, because `noun' and `second' might not always be objects (they might be numbers, for example). However, the variable `inp1' 1 if `noun' is not an object, and `inp2' is 1 if `second' is not an object (DM 9), so we can check these first.

We also have to be careful not just to use the test `noun in mantelpiece' to see if the noun is at hand, because that would fail if the object were carried by the player. Instead, we use the function `Inside' to carry out this test. The function takes two objects arnd returns 1 if the first object is inside the second, or 0 otherwise.

```
[ Inside x y;
    do {
        x = parent(x);
    } until (x == 0 or y);
    if (x == 0) rfalse;
];
```

These problems arise from having several `enterable' objects in the same room, and trying to restrict what Alice is allowed to do, according to which of the objects she's on. This has turned out to be a much more complicated situation to program than it seemed at first!

7. Adding characters

Plausible characters are the most complicated part of any adventure game; they make the most infernal of devices seem easy to program by comparison.

Truly sophisticated programming of characters will involve you in cuttingedge areas of Artificial Intelligence research, and this is well beyond the scope of this tutorial (but see David Graves' essay `Bringing Characters to Life' for a criticism of existing characters in adventure games and some suggestions for future directions, and the papers of the Carnegie-Mellon Oz Project for some ongoing research). However, you can do fairly well without going to quite such lengths.

We can get a handle on this complexity by dividing up the functionality of a character into three parts.

1. *State*, which is a set of values that describe how the actions of the character vary. We are familiar with some aspects of character state that are handled by the Inform library, such as which room a character is in, but we will have to invent other aspects; for example, how welldisposed a merchant is to the player, how many times the player asked the wizard for help, whether or not the player has bribed the guard, and so on.
2. *Reactions*, which are things the character does in response to interaction from the player, for example when the player talks to the character, attacks him, kisses him, offers things to him, tries to steal things from him, and so on. In Inform these are typically coded in the `life', `before', and `after' routines.
3. *Autonomous actions*, which are things the character does of his own accord, without special prompting from the player. In Inform, these are encoded in the `daemon' and `each_turn' routines.

Obviously, the character's state will affect his actions and reactions, and these in turn will alter his state.

7.1 The kittens

In the case of the two kittens, their state just describes where they are and what they are playing with: each kitten is either being carried by Alice, playing with the red queen, playing with the ball of worsted or playing near the armchair and getting in Alice's way. (I could go for more complicated behaviour, allowing the kittens to be on the armchair, or on the mantelpiece, or on the rug, or under the armchair, and so on, but for simplicity I'll prevent the kittens from getting into these states). These states are going to be represented by numbers, but numbers are hard to remember, so first I'll define:

```
Constant HELD_STATE = 0;     ! Being held
Constant QUEEN_STATE = 1;    ! Playing with the Red Queen
Constant WOOL_STATE = 2;     ! Playing with the worsted
Constant CHAIR_STATE = 3;    ! In the way of the chair
```

The kittens will need to react to being spoken to, being kissed, being picked up, being put on things, being put down and being given objects to play with. I'll give them lots of autonomous actions, but these will just be random messages along the lines of `The white kitten chases its tail.'

I can make the programming much simpler by creating a class called `Kitten' and making each kitten a member. This will give the two kittens exactly the same behaviour, which is probably just about excusable for kittens, but this strategy probably won't convince if you try to apply the same technique to humans!

The first changes to make are those which affect other objects; I'll take the objects in order.

If Alice takes the red queen away from a kitten who's playing with it, then the kitten will change state from QUEEN_STATE to CHAIR_STATE, so add the following to the red queen.

```
after [;
 Take:
    if (white_kitten.state == QUEEN_STATE)
        white_kitten.state = CHAIR_STATE;
    if (black_kitten.state == QUEEN_STATE)
```

```
            black_kitten.state = CHAIR_STATE;
    ];
```
(If there had been many kittens, rather than just two, this rule could have been written like this:
```
        objectloop(x ofclass Kitten)
            if (x.state == QUEEN_STATE)
                x.state = CHAIR_STATE
```
and adding a variable `x', but for just two kittens, it isn't worth it.)

Alice can't push the armchair while the kittens are playing near it, nor while she is carrying a kitten (otherwise she could give one kitten the worsted and pick up the other, and solve the puzzle that way). So add the following to the armchair's `before' routine, where I had previously put a comment (and don't forget to add a local variable `i' to the routine).

```
            if (white_kitten in player || black_kitten in player)
                "Not with a kitten in your arms!";
            if (white_kitten.state == CHAIR_STATE) i = white_kitten;
            else if (black_kitten.state == CHAIR_STATE) i = black_kitten;
            if (i ~= 0)
                "You are about to start moving the chair when you
                 notice that ", (the) i, " is right in the way. It's a
                 good thing you spotted it, or you would have squashed
                 flat the poor little thing.";
```
If Alice takes the ball of worsted away from a kitten that's playing with it, the kitten will change state from WOOL_STATE to CHAIR_STATE, so give the ball an `after' routine.
```
        after [;
         Take:
            if (white_kitten.state == WOOL_STATE)
                white_kitten.state = CHAIR_STATE;
            if (black_kitten.state == WOOL_STATE)
                black_kitten.state = CHAIR_STATE;
        ];
```
I've used the `state' property to describe the kitten's state. This is something I've invented, not a property which has any meaning to the Inform library. Likewise, I'm also naming a property `other_kitten' so that each kitten can work out which kitten is the other (remember that they're going to be using exactly the same code).

Now I can start to code up the kittens. My first attempt at a definition is the following (the kittens start out playing by the armchair, so they are in state CHAIR_STATE).

```
Class   Kitten
 has    animate
 with   state CHAIR_STATE,
        name "kitten" "kitty" "cat",
        description [;
            "What a beautiful kitten ", (the) self, " is. Why,
             it's quite definitely your favourite of the pair, and
             much prettier than that naughty ",
             (name) self.other_kitten, ".";
        ];

Kitten  white_kitten "white kitten" Drawing_Room
 with   name "white",
        this_kittens_turn false,
        other_kitten black_kitten;

Kitten  black_kitten "black kitten" Drawing_Room
 with   name "black",
```

```
        this_kittens_turn true,
        other_kitten white_kitten;
```
(I'll explain why the kittens have `this_kittens_turn' properties below; be patient for a bit. It's another new property I've invented, not one that means anything to the Inform library.)

If you experiment with this definition, you'll discover that you get the following response.

```
    > take kittens
    You can't see any such thing.
```
How to get them to respond to plurals? I'll just remove the `name' property from the kitten class, and copy (with a few modifications) the `parse_name' code for the crown class from Section 25 of the Designer's Manual (essentially, this recognises any string of the words `black' or `white', `kitten', `kitty', `cat', `kittens', and `cats' as referring to a kitten, but if either of the last two words appears then the variable `parser_action' is set to `##PluralFound', which tells the parser that a plural word has been found).
```
        parse_name [ w ok n;
            do {
                ok = 0;
                w = NextWord();
                if (w == 'kittens' or 'cats') {
                    ok = 1; n++; parser_action=##PluralFound;
                }
                if (w == 'kitten' or 'kitty' or 'cat' ||
                    w == ((self.&name)-->0)) {
                    ok = 1; n++;
                }
            } until (ok == 0);
            return n;
        ],
```
Also, the initial description of the kittens (`You can also see a white kitten and a black kitten here') leaves something to be desired. I want the description of the kittens to vary according to what they are doing. So I use a `describe' routine, as follows (note that the text produced by a `describe' routine must start with a newline, otherwise there won't be any space between it and any preceding text).
```
        describe [ i;
            switch (self.state) {
             QUEEN_STATE:
                "^A ", (name) self, " is playing with the red queen.";
             WOOL_STATE:
                "^A ", (name) self, " is playing with a ball of worsted.";
             CHAIR_STATE:
                if (self has general) rtrue;
                if ((self.other_kitten).state == CHAIR_STATE) {
                    i = self.other_kitten;
                    give i general;
                    "^Two kittens, one white and one black, are playing
                    together by the arm-chair.";
                }
                "^A ", (name) self, " is playing by the arm-chair.";
             default: rtrue;
            }
        ],
        daemon [;
            give self ~general;
        ];
```
Why the use of the `general' attribute and the `daemon' routine? Because when the kittens are playing together they are described together. I want to avoid output like the following.
```
    Two kittens, one white and one black, are playing together by the arm-
chair.
```

```
    Two kittens, one white and one black, are playing together by the arm-
chair.
```
(one line from the white kitten, one from the black). So when one kitten outputs this line, it sets the `general' flag on the other so that it knows not to describe itself (by returning 1 from `describe'). The `daemon' routine makes sure that all the `general' flags are cleared at the end of each turn, ready for the next.

Having given the kittens more complicated descriptions, I now run the risk of the following happening.

```
    A discarded ball of worsted lies on the floor here.

    A white kitten is playing with a ball of worsted.
```
So I delete the ball of worsted's `initial' string, and add code to stop it appearing in descriptions if the kitten is playing with it.
```
        describe [;
            if (white_kitten.state ~= WOOL_STATE &&
                black_kitten.state ~= WOOL_STATE)
                "^A discarded ball of worsted lies on the floor here.";
            rtrue;
        ],
```
I write a similar routine for the red queen (not shown).

Next, the kittens' reactions. I have to provide a `before' routine for the `Take' action, because otherwise the library will respond `I don't think the white kitten would care for that.' I also provide special messages for a few other actions, to make sure that the kitten always ends up on the floor of the room (if you dropped a kitten while standing on the chair, the kitten would end up in the chair).

```
        before [;
         Take:
            if (self.other_kitten in player)
                "You can't hold two kittens at once!";
            self.state = HELD_STATE;
            move self to player;
           "You pick up ", (the) self, ". What a beautiful
            creature it is!";
        ],
        after [;
         Drop:
            self.state = CHAIR_STATE;
            move self to Drawing_Room;
            print_ret (The) self, " squirms out of your arms and scampers
                away.";
         Transfer,PutOn,Insert:
            self.state = CHAIR_STATE;
            print (The) self, " jumps off ", (the) parent(self);
            move self to Drawing_Room;
            ", landing lightly on the floor before scampering away.";
        ],
```
The rest of the reactions are covered by the `life' routine.
```
        life [;
         Ask,Answer,Order:
            print_ret (The) self, " twitches its whiskers and looks at
                you with such a clever expression that you are certain it
                understands every word you are saying.";
         Kiss:
```

```
            "You give ", (the) self, " a little kiss on its
             nose, and it looks sweetly and demurely at you.";
          Attack: "You would never do such a beastly thing to such
             a defenceless little animal!";
          Show:
             print_ret (The) self, " bats a paw at ", (the) noun, ".";
          Give,ThrowAt:
             if (noun ~= red_queen or worsted) {
                 if (action == ##ThrowAt) {
                     move noun to Drawing_Room;
                     print "You toss ", (the) noun, " onto the floor, but ",
                         (the) self;
                 }
                 else print (The) self,
                 " just examines ", (the) noun,
                 " with a quizzical expression.";
             }
             print "You toss ", (the) noun, " onto the floor and ", (the)
self;
             if (self in player)
                 print " squirms out of your grasp and";
             move noun to Drawing_Room;
             move self to Drawing_Room;
             print " scampers after it";
             if (noun == worsted) {
                 give worsted general;
                 self.state = WOOL_STATE;
                 print ", quickly turning the neat ball into a tangle";
             }
             else self.state = QUEEN_STATE;
             ".";
          ],
```

Finally, I have the kittens' autonomous actions. It would be too confusing if each kitten output a message every turn. Instead, I make the kittens alternate their messages, and I give each kitten a one in three chance of producing no message at all. I ensure the former by giving each kitten a `this_kittens_turn' property, always true or false, which toggles each turn; and I allow the kitten to do something only if its `this_kittens_turn' is true. Here's the start of the `daemon' routine.

```
          daemon [ i;
              give self ~general;
              self.this_kittens_turn = 1 - self.this_kittens_turn;
              if (self.this_kittens_turn || random(3) == 2) rtrue;
              new_line;
              print (The) self;
              switch (self.state) {
               HELD_STATE:
                  switch(random(5)) {
                   1: " mews plaintively.";
                   2: " purrs quietly to itself.";
                   3: " purrs contentedly to itself.";
                   4: " rubs its ears against you.";
                   5: move self to Drawing_Room;
                      self.state = CHAIR_STATE;
                      " squirms out of your arms and scampers away.";
                  }
```

There are similar lists of random actions for the other states a kitten can be in (not shown). The daemons need to be set going at the start of the game, so add the following to the `Initialise' routine.

```
    StartDaemon(white_kitten);
    StartDaemon(black_kitten);
```

8. Frills

Among the last changes made to a game are the small frills and thrills; extra responses that add little or nothing to the plot, but make it more pleasant to play. If your playtesters are very good, then they'll suggest commands that they tried which ought to produce a more interesting response.

A nice touch would be to provide for the following response (which provides a subtle clue as to the where the red queen might have gone, and what use it would be once you find it).

```
    > take white bishop
    Alas, that chess piece seems to be missing. Those naughty kittens!
```

I could do this with a dummy object whose `before' routine just provides the above message. My first guess was to write the following.

```
Object  chess_pieces "chess pieces" Drawing_Room
 has    scenery
 with   name "white" "red" "pawn" "rook" "castle" "knight" "horse"
            "bishop" "queen" "king",
        before [;
            "Alas, that chess piece seems to be missing. Those naughty
            kittens!";
        ];
```

But I have to be more careful than that; consider the following problems with the above.

```
    > examine white
    Which do you mean, the white kitten or the chess pieces?

    > take red queen
    Which do you mean, the red queen or the chess pieces?
```

I'll solve this problem by writing a `parse_name' routine to parse the exact sequences of words that I want (`white pawn' to `white king', `red pawn' to `red king', and `pawn' to `king'), and no others.

```
Object  chess_pieces "chess pieces" Drawing_Room
 has    scenery
 with   parse_name [ w colour n;
            w = NextWord();
            if (w == 'white' or 'red') {
                n ++;
                colour = w;
                w = NextWord();
            }
            if (w == 'pawn' or 'rook' or 'castle' ||
                w == 'knight' or 'horse' or 'bishop' ||
                w == 'king' || (w == 'queen' &&
                (colour == 'white' || rug hasnt general))) return n + 1;
            return 0;
        ],
        before [;
            "Alas, that chess piece seems to be missing. Those naughty
            kittens!";
        ];
```

Another interesting addition would be the following (there's something like this in the game `Curses').

```
    > look at red queen in the looking-glass
    The looking-glass red queen looks just like the real red queen,
    only all reversed, left for right.
```

I should be careful in implementing this, and consider the following commands.

- `look at hearth in mirror' - you can't see the hearth in the mirror.
- `look at mirror in mirror' - ditto.
- `look at myself in the mirror' - should respond with `the lookingglass Alice', not `the lookingglass yourself'!
- The description of objects in the mirror should change depending on where you are - e.g. you can't see things from the floor, you can see things from the armchair, and things are fuzzy and misty when seen from the mantelpiece.

I'll also need to add some grammar (after the inclusion of the Grammar library file):

```
Extend "look"
    * "at" noun "in" noun -> Reflect;

Extend "examine"
    * noun "in" noun -> Reflect;
```

and an extra action routine.

```
[ ReflectSub;
    if (second ~= mirror) "What a strange idea!";
    if (noun == hearth or mirror || (player notin mantelpiece &&
        player notin armchair))
        "You can't see that in the looking-glass.";
    print "The looking-glass ";
    if (noun == player) print "Alice";
    else PrintShortName(noun);
    if (player in mantelpiece) " looks very misty and blurred.";
    print " looks just like the real ";
    if (noun == player) print "Alice";
    else PrintShortName(noun);
    " only all reversed, left for right.";
];
```

I also need to add `Reflect' to the actions that can be used when the player is on the mantelpiece, and to the actions that can be used on the mirror.

Another response that I could add is the following.

```
    > put red queen on chessboard
    Alone on the chess board, the red queen is monarch of all she
    surveys.
```

All I have to do is add the following to the red queen's `after' routine.

```
       PutOn,Transfer,Insert:
          if (second == chess_board)
              "Alone on the chess board, the red queen is monarch of all
              she surveys.";
```

9. Checking the consistency of the game

Having finished the construction of a reasonably complicated puzzle, it's important to go back and check that it's a fair puzzle; that it allows a reasonable choice of commands; that there are clues to any nonobvious solutions; and that the player need never flounder in want of a task to attempt.

In this example, the task (to get through the mirror) is given at the very start, in the introductory text. The chain of puzzles seems fairly clear, and the only nonobvious puzzle is that you have to give objects to the kittens to distract them. But the description of the chess board and the response to examining nonexistent chess pieces suggests that the kittens like to play with them, and that they get lost in odd places.

In my opinion, the weakest part of the puzzle is figuring out that you have to push the armchair, since the description of the armchair doesn't mention where it is in the room. A good idea might be to change the `description' of the armchair to a routine that says whether it's next to the fireplace or the window (say).

```
        description [;
            print "It's a huge arm-chair, the perfect place for a kitten
                or a little girl to curl up in and doze. It has been
                pushed over to the ";
            if (self has general) "fireplace.";
            "window.";
        ],
```

Having mentioned the window, I also have to add an object to represent it.

```
Object  window "window" Drawing_Room
 has    scenery
 with   name "window" "pane",
        description "Outside the window it's snowing gently, and you're
            glad to be in here in the warmth.",
        before [;
         Open: "You wouldn't want to catch a chill, would you?  Better
            leave the window shut.";
         Search: <<Examine self>>;
        ];
```

The `Search' action is generated by the command `look through the window', so we just cause it to do the same thing as `examine window'.

10. Testing, part 2

With all these changes included, I have a game (well, the first room of a game, anyway), alice3.inf, that's ready to be betatested.

Alphatesting usually refers to testing that goes on while a program is still in development; betatesting is the testing of a supposedly complete program. Because of the enormous number of different states an adventure game can potentially get into, and the number of different user inputs it has to somehow produce sensible responses to, betatesting is very important. (Infocom had three stages of testing: alpha and betatesting were inhouse, and gammatesting involved a large number of outofhouse players.)

Through the Looking Glass is still in betatesting (there is a sense in which no complex enough game ever emerges from this stage!), and I've received the following reports:

Peter Grundy <pgrundy@commerce.otago.ac.nz> suggested:

1. How about adding verbs to stroke, pat or tickle the kittens?

Graham Nelson <graham@gnelson.demon.co.uk> noticed:

2. The command `examine red pawn' works but `examine red pawns' doesn't. Also, I think it would be nice if attempts to examine the black pieces could be met by a polite demurral that the set is in white and red.
3. The command `enter hearth' will occur to inveterate Curses players.

Martin Braun <100106.2673@compuserve.com> noticed:

4. If you put an object on the mantelpiece, climb up to it and type `take all', this object does not respond to the command (though you can pick it up individually).
5. You can't talk to the kittens while on the mantelpiece.
6. On the mantelpiece, the command `go down' produces the rather strange response `But you are already on the mantelpiece', about what I hadn't any doubts.
7. When I discover the red queen under the rug, the game moves the queen to my possessions. I would like the game to tell me that I picked her up instead of only having found her.
8. If one of the kittens plays with the ball and you pick it up again, the game describes it as unrolled. If you then give it to the kittens a second time, the response still says that the kitten is turning the neat ball (which it isn't any more) into a tangle.

   Charles Briscoe-Smith <cpbs@ukc.ac.uk> noticed some of the above, and in addition:

9. Because the kittens are animate, the pronoun `it' never refers to them, despite the fact that the game, following Carroll, always uses `it' to refer to a kitten.
10. The commands `get down', `climb down', `get off mantelpiece', `get down from mantelpiece', `get onto chair', `jump' and `jump onto/into chair' don't work; but if the player types them while on the mantelpiece, the game should move the player to the armchair.

Most of these bugs should be easy to fix, and are left as an exercise for the reader.

However, problem 4 is to some extent a problem with the library. It is a result of the way the library interprets `all' as referring only to objects that aren't contained in other objects in the current room; arguably it should be changed so that rather than referring to objects in the current room, `all' refers to objects with the same parent as the player. Fixing this bug thus involves editing the parser sources - not an easy task - or waiting for the next release of the library, which will, with luck, provide a solution.

To solve problem 9, the best solution is probably to Replace the `ResetVagueWords' library function.

 You can read a transcript of the game being played here.

11. Conclusion

So that's the first room of an adventure game: nearly 500 lines of code (around 200 of which are needed for the kittens). How many more rooms to go before you have a complete game? At least twenty or thirty. Not all will be as complicated as this, but if you plan to have plausible characters in your game then they'll need to be at least as complicated as the kittens, and probably much more so!

I've tried to make it clear that the way an adventure game progresses is by gradual accretion and modification of code, rather than by the `topdown' design of structured programming. Each new object may interact with the old ones in complicated ways, forcing you to modify your old code, and testing tends to indicate lots of small problems that need fixing. The code actually went through considerably more revisions than appear on this page!

I'd be very glad to receive feedback on the ideas and advice in this tutorial - do you think it goes into too much detail, or too little? Were there things you wanted explaining? Could some of the examples have been coded more clearly?

If you want to use the code or the puzzle ideas developed in this tutorial in your own games, feel free. I hereby donate the source code of the `Through the LookingGlass' example games into the public domain.

Doug Atkinson wrote *Alice Through the LookingGlass*, Part Two (using Inform 5.5). Any offers for Part Three?

12. References

- Lewis Carroll, *Through the Looking Glass*.
- David Graves, `Bringing Characters to Life'.
- Graham Nelson, `The Craft of the Adventure'.
- Graham Nelson, *The Inform Designer's Manual*.
- Alma Whitten and Scott Reilly, *The Oz Project*.
