Compiled from the OnyxRing article archive on 4/16/2003 9:15:41 PM

Table Of Contents

Part A
Overview and Setup

Note:

At the time of this writing, the ORLibrary documentation is in an incomplete draft state. That is, portions of this document have not yet been written and those sections that do exist need review and revision. I can say with absolute certainty that typographical and grammatical errors abound.

The decision to release this work-in-progress was made after several requests by ORLibrary users. The library is, even by the most conservative of measures, extensive. The availability of at least some documentation greatly increases -- or so I'm told -- its usability. To that end, I am willingly undergoing the somewhat humbling experience of exposing my draft efforts to the public eye.

As new versions of this document are compiled, they will be made available from the ORLibrary's home at:

www.OnyxRing.com

Introduction

Welcome to the ORLibrary Guide. The purpose of this guide is to explain the function of the ORLibrary, give working examples of what it can accomplish, and demonstrate how it can be leveraged to create more detailed games with less effort.

This text lists the most prevalent entries in the ORLibrary. It attempts to describe each one, gives examples of its use, and discuss how it interacts with other ORLibrary modules.

1. What is the ORLibrary?

The ORLibrary is based upon framework theory discussed in the article "Building a Personal Library" which can be found in "An Inform Developer's Guide" or online at the following URL:

www.OnyxRing.com/informguide.aspx?article=7

and also in PDF format at:

www.OnyxRing.com/downloads/aidg.pdf

It is a collection of objects, routines and enhancements that sits on top of the standard library. All the entries in the library are opt-in, meaning they have no effect whatsoever on the final produced code unless they have been selected for use. Using the ORLibrary

extension framework simply makes them accessible.

2. Newbie ~=Newbie

This guide is intended to be usable by both newbie and advanced coder alike, but the term "newbie" carries a breadth of possible interpretations. First-time Informer's will have trouble getting much use from this text. If you are a newbie that has no idea what a `for` loop is, then this guide is not for you. If you have no idea what an `object` is, then this guide is not for you. If you cannot write a basic program in Inform, then this guide is not for you.

This is NOT a guide on programming in Inform. As such, it will not directly discuss topics such as "What are properties and attributes?" or "What does the Parser.h file do?" or "How do I implement a loop?" Further, it is not a tutorial on using the "standard library," so questions such as "What is the `switchable` attribute?" "How do I change a default message?" and "What is a `before` property?" are also not directly answered. It may be that you can find answers to these questions buried in the text of this guide, but these topics are only touched upon with the assumption that you already know them or at least have a basic understanding of their theory.

Don't despair if you do not have this understanding yet. There are a number of resources at your disposal. The most notable is the Inform Developer's Manual, written by the author of the Inform language, Graham Nelson and downloadable from

ftp://ftp.ifarchive.org/if-archive/infocom/compilers/inform6/manuals/designers_manual_4.pdf

Also very much worth review, and arguably of more use to the absolute newbie than the DM proper, is the Inform Beginner's Guide (IBG) written by Roger Firth and Sonja Kesserich which can be found at

www.inform-fiction.org/manual/IBG.pdf

Additionally, the OnyxRing site hosts the continuously evolving manual "An Inform Developer's Guide" (mentioned above) which you may find useful as several Inform developers have had a hand in its creation.

Setting up for ORLibrary use

Setting up ORLibrary is a trivial matter. It can be setup in virtually any directory and, with the addition of a couple of command-line properties, can be used immediately.

1. Where do I get ORLib?

The ORLibrary finds it's home at the OnyxRing website. The following URL joins to the ORLibrary page:

http://www.onyxring.com/orlibrary.aspx

This page holds the most recent iteration of the ORLibrary. That is, the most recent version of each entry as well as any alpha entries that are currently being tested.

By far, the easiest way to download the ORLibrary is to get the ORLib_complete.zip file, which contains a snapshot of all the non-beta library entries as of the date-stamp indicated in the description.

2. Where do I install ORLib?

The best place to install the ORLibrary is entirely a matter of opinion. It can be anywhere that the Inform compiler can reach, from a networked drive, to your home directory. Some developers simply drop the ORLibrary files in the same directory as the standard library files. This saves them the initial fuss with command-line parameters but only with the priceof code separation. This is not recommended.

Generally, it is a good idea to give the ORLibrary its own directory, possibly on the same directory tier as the standard library files. The important thing to remember is that this directory will need to be added as a parameter (or switch) to the Inform compiler's command-line.

3. ICL Settings

There are a couple of ICL options need to be made in order to have Inform compile in the ORLibrary. Depending on how the your development environment is set up, these might be put in a batch file (or other scripting file), or simply be part of a macro that runs when you press a special button in the IDE of your choosing. Preferably, these may be placed in a common ICL file, but for simplicity the examples given below are given in the context of command-line options.

See the Inform Developer's Manual or the compiler's h1 option for specifics about ICL files.

4a. Telling Inform where ORLibrary is installed

The ICL variable that tells the compiler where the ORLibrary files are installed is generally already used to specify the location of the standard library. It is the variable "include_path" and is most often set on the command line with the familiar "+ command" notation. Additional directories can be added to this variable by separating them with a comma. The following example shows one possibility as it might exist on a Microsoft platform:

```
+include_path=c:\if\inform\library, c:\if\inform\orlib
```

And, of course, here's a Linux variation:

```
+include_path=/usr/inform/library, ~/if/inform/orlib
```

4b. Specifiying the OREnglish LDF

Strictly speaking, the second modification, which includes the OREnglish language definition file, is optional. Many basic library entries can be used without it, but the flexibility of this entry and its immediate transparency when implemented make it a veritable staple in the ORLibrary framework. Some of the library entries depend upon it entirely and will not work without it. For clarity, the examples in this guide assume its inclusion. OREnglish and ORPronouns (a dependency of OREnglish), will be among the first entries touched upon in this guide.

Inclusion of the OREnglish file can be accomplished with the following command line addition:

+language_name=OREnglish

Note: A complete Inform command-line example is shown in "Verifying ORLibrary Installation"

Verifying ORLibrary installation

Verifying that ORLibrary is installed is a simple matter and can be accomplished by checking the compiler's output. We'll compile the OR_Blankgame.inf file to do this.

1. OR_Blankgame

The OR_blankgame.inf file that comes with ORLibrary has several features that make it an ideal starting point for an ORLibrary program. These will be covered more thoroughly in later sections. For now, simply compiling the file with the above-specified ICL settings (as in the following example)...

```
 c:\inform\infrmw32.exe +language_name=OREnglish
include_path=c:\inform\library,c:\inform\ORLibrary OR_blankgame.inf
```

...should render output similar to the following:

```
 PC/Win32 Inform 6.21 (30th April 1999)
 Compiling with the O R L i b r a r y framework
     ****Forcing inclusion of ORBanner (from common include file)****
 Preparing to process extensions REPLACE section...
      Processing library extension ORBanner...
 ------------------------------------------------------------
 Processing the OREnglish LDF...
      ****Forcing inclusion of ORTextFormatting****
      Processing library extension ORTextFormatting...
      ****Forcing inclusion of ORPronoun****
      Processing library extension ORPronoun...
 OREnglish LDF complete.
 ------------------------------------------------------------
 Preparing to process extensions MESSAGE section...
      Processing library extension ORBanner...
      Processing library extension ORPronoun...
      Processing library extension ORTextFormatting...
 Preparing to process extensions CODE section...
      Processing library extension ORBanner...
      Processing library extension ORPronoun...
      Processing library extension ORTextFormatting...
 Preparing to process extensions GRAMMAR section...
      Processing library extension ORBanner...
      Processing library extension ORPronoun...
      Processing library extension ORTextFormatting...
```

If errors occur, or if the results are not similar to the above, then go back and review "Setting Up for ORLibrary Use". Most problems that occur are usually related to directory locations and invalid attempts to signify to the compiler where files are located.

2. Compiler Output Verification

If the ORLibrary is correctly installed and functional, then the compiler output should tip you off immediately. Even without using any ORLibrary entries, the library framework alone

leaves an obvious signature in the output.

2a. Checking for framework reference

If you are referencing the ORLibrary correctly, then right at the top of the compiler output will be the following text:

```
Compiling with the O R L i b r a r y framework
```

Additionally, the compiler will list the steps it goes through during the compilation of the ORLibrary files. This can be seen with lines that are similar to:

```
Preparing to process extensions REPLACE section...
```

The four sections that are referred to in the output will be made clear in the section entitled "Adding your own entries".

2b. Checking for OREnglish

Lines similar to the following signify that a library entry is being compiled:

```
Processing library extension ORPronoun...
```

Since the OREnglish library entry, in particular, is not compiled in the same fashion as other library entries (it is included by the parserm.h file, which is included by the parser.h file) and is, in fact, quite different than the standard ORLibrary modules, the compiler output indicating its presence is highlighted by the inclusion of two lines. The first line appears with the text that indicates compilation of OREnglish is under way. For this reason, making sure that the ICL command was processed and therefore OREnglish language definition file was processed is also easily determined by checking the output. The following lines will appear if it has:

```
------------------------------------------------------------
Processing the OREnglish LDF...
```

The second line appears with text that indicates that the compilation of OREnglish has been completed:

```
OREnglish LDF complete.
-------------------------------------------------------
```

All compiler output generated during the compilation of OREnglish will appear between these two sets of lines. A glance will tell you that including OREnglish also includes two additional library entries. We'll talk about these next...

2c. Automatic Dependencies

In addition to OREnglish, you may notice an unexpected side effect: three unselected library entries are pulled in. It is common practice for library entries to attempt to meet their own dependencies. That is, needed entries will often be pulled into the compilation process automatically.

Strictly speaking, the first, ORBanner, is not really a dependency. The framework itself pulls this in to implement versioning functionality. The additional two library entries that are pulled in are ORTextFormatting and ORPronoun. These are dependencies of OREnglish and are included by that module.

After detecting that a dependency is not referenced, and forcing it to be included, an entry will then volunteer what it has done. This way, the developer is never left unaware of the source of an unexpected entry's inclusion. The following lines in the previous output example indicate this:

```
****Forcing inclusion of ORBanner (from common include file)****
****Forcing inclusion of ORTextFormatting****
****Forcing inclusion of ORPronoun****
```

A quick view of the output will show that these entries were included while processing the OREnglish file.

Philosophies to keep in mind

There are a few common practices that should be kept in mind when coding. These practices are good ideas all of the time, even when not leveraging the ORLibrary, but they are especially useful when you are:

1. Layers (or base classes) by Inheritance

Using inheritance (creating derived classes) gives us the ability to design layers for common changes. This principal is especially helpful when dealing with objects and yields benefits that cannot always be seen at first glance. This is not a guide on OOP techniques but to briefly skim the surface:

- Layers allow for easy, broad changes. Changing code in a single location can affect dozens, or even hundreds, of objects. This provides us with the ability to make sweeping changes in seconds that might otherwise take us weeks.

- Layers allows for a common place to implement similarities. This can help eliminate the need to duplicate code. Certain properties or attributes are, by default used, all the time. Implementing them at the base class layer causes them to be inherited automatically by all derived objects and can shrink the size of the final program.

2. Behavior Granularity by Multiple Inheritance

I have always felt that Multiple Inheritance (creating objects derived from two or more classes at the same time) was an under appreciated technology. It is only by understanding and leveraging MI that a developer can truly weld the full power OOP provides. In Inform, MI is best suited for the design of behaviors, that is, creating a collection of unrelated objects that each do specific and unrelated thing, then applying them each to a single class to create an object that can do several things.

One example of this can be seen in the ORNPC class and its related classes each of which encapsulate different NPC behaviors. By writing NPC behaviors in separate classes we've given the developer a granularity that was not available previously. It is just as easy to write an NPC that walks around and talks as it is an NPC that simply talks, or one that simply walks around. More on this will be discussed in the section on NPCs "Advanced Library Entries".

Part B
Basic ORLibrary Modules

Overview of OR_BlankGame

The OR_Blankgame.inf file, as the name implies, is a template file that can be used as a starting point for games leveraging the ORLibrary. The following is a review of various parts of the file.

1. Constants

With the ORLibrary framework, including registered library entries is as simple as defining the appropriate constant. In order to keep from having to lookup the appropriate constant whenever you want to include a library entry, the blank game template comes with all the current constants already defined, but commented out. All that is required to include a library entry from a file based on the OR_Blankgame.inf template is to locate the appropriate constant, and uncomment it. The following demonstrates a game including the ORDoor library entry:

```
! Specify which extensions to use
!Constant USE_ORProp;
Constant USE_ORDoor;
!Constant USE_ORSeveral;
... more constant definitions to follow this
```

2. Commented segments

There are several commented sections used to indicate where certain types of code should go. Note that this is mostly a matter of preference. Feel free to move these about or eliminate them entirely if you so desire. Honoring the convention does have two added benefits, however: It helps to keep your code organized, and it insures code is placed appropriately with respect to both the ORLibrary includes and the standard library includes.

3. Include locations

The three standard library include files (Parser, Verblib, and Grammar) dictate, by their nature, four sections in any Inform program where blocks of code can be placed. At each of these four locations you will find the following include directive:

```
#Include "OR_Library_Include.h";
```

This file is the common entry point for all framework entries. It expects to be included exactly four times and keeps track of how many times it has been included. This information is then passed along to the library entries, which may or may not choose to provide code for compilation during that pass. All library entries must be registered with this file in order to

be accessible.

4. Additional pieces

Additionally, there are a few places in the template where generic every-game code has been created. These include the Serial and Release constants as well as the Initialize routine. These are no different than they would be in a standard, non-ORLibrary game, and are included simply for convenience.

Building a Two Room Example

We could jump immediately into an overview about the ORLibrary, but it will be much more beneficial if we have a subject for our discussion. Let's begin with a "Hello-World" caliber game using the OR_Blankgame template. Don't bother uncommenting any of the library entries for now. We'll get to those in a moment.

1. Beginning Layers

As was stated previously, it is a good idea to create "base classes" for our objects to derive from. This is a good idea even if you have no code to put in the base class since it provides a place to implement common expansion in future enhancements. The following code, placed in the "Object Templates" section of the blank game template, will create a base class for objects:

```
class MyObject !---the basic object
;
```

And the same can be done for a base room object:

```
class MyRoom !---a basic room object
   class MyObject
   has light
;
```

Notice the definition of the light attribute. This is one (incredibly small) example of the usefulness of base classes. All rooms derived from MyRoom will have the light attribute by default. This is a fairly useful practice, since in the design of a hundred-room game you've saved yourself from having to specify the light attribute 99 times. Even in our very small game we see (some) benefit. Base classes become much more useful as game complexity increases. Keep in mind that if you want to create a dark room that is derived from the MyRoom class you can always unset the light attribute (~light).

2. Two Rooms and an Object

Now, using our base classes, we can create two rooms, joined in an east-west fashion and a simple object in one of the rooms.

```
MyRoom Study "The Study"
  with w_to Laboratory
  ,    initial "Your eyes take a moment to adjust to the
              dimly illuminated room."
  ,    description "This is a cozy room with plush carpeting
```

```
              and walls lined in woodwork. A doorway lies to
              the west."
;
MyRoom Laboratory "The Laboratory"
  with e_to Study
  , description "Drab and bare, the lab is a cold sterile
              environment, not at all comfortable but brightly lit.
              Shards of glass, possibly from broken test tubes litter
              the floor. The study lies to the east."
;
MyObject Candlestick "candlestick" study
  with name 'stick'
, description "The candlestick is copper, or perhaps brass.
              The metal has been polished so as to gleam in the
              room's lighting. It is heavy and hard enough to make
              a formidable weapon."
;
```

3. Positioning the player

With the last step of placing the player in a room, we are finished with our two-room example. Put the following line in the Initialize function:

```
location=Foyer;
```

OREnglish

Even a beginning Inform programmer may notice that the previous example is no different than a plain non-ORLibrary program. What was the point of the exercise? Two things. First, was to demonstrate OREnglish's *transparency*. Second, to provide a place to demonstrate a major feature of OREnglish: The ability to create games in different tenses and person.

1. Transparency: The Same...

Other than the use of the game template and the OREnglish file, there was nothing new introduced in the coding of the previous two-room example. So what's different about the z-file produced? Hardly anything. The very same code given above will compile without the ORLibrary. As the following transcript shows, when you run the game you get the same default messages, that you would get without the ORLibrary:

```
The Laboratory
Drab and bare, the lab is a cold sterile environment, not at all
comfortable but brightly lit. Shards of glass, possibly from
broken test tubes litter the floor. The study lies to the east.

>n
You can't go that way.

>e
Your eyes take a moment to adjust to the dimly illuminated room.

The Study
This is a cozy room with plush carpeting and walls lined in
woodwork. A doorway lies to the west.
You can see a candlestick here.

>get stick
Taken.

>eat stick
The candlestick is plainly inedible.
```

The same code produces the same messages with or without the ORLibrary. No surprises there. Still, if we look at the compiler output we see that the same library entries that we discussed in section 3 were pulled in as "automatic dependencies" again. The OREnglish file was compiled in too. Yet these don't appear to have any affect.

This demonstrates the "transparency" striven for when designing ORLibrary entries. That is, unless instructed to do so, these library entries will mimic the behavior of the standard library. Without forcing us to use it, the OREnglsh module and it's dependencies have given us a great deal of flexibility that we do not have with the standard library alone.

2. Tense & Person: ...Yet different

One of the first things that OREnglish provides us, with the aid of ORPronoun and ORTextFormatting, is the ability to change the person and tense of the narrative. To demonstrate this, add the following command to the Initialize() routine:

```
SetPersonTense(FIRST_PERSON,PAST_TENSE);
```

Now recompile and run. By typing in the same commands as we did previously, we can see that the default messages have indeed changed:

```
The Laboratory
Drab and bare, the lab is a cold sterile environment, not at all
comfortable but brightly lit. Shards of glass, possibly from
broken test tubes litter the floor. The study lies to the east.



>n

I couldn't go that way.

>e
Your eyes take a moment to adjust to the dimly illuminated room.

The Study
This is a cozy room with plush carpeting and walls lined in
woodwork. A doorway lies to the west.
I could see a candlestick there.

>get stick
Taken.

>eat stick
The candlestick was plainly inedible.
```

All library messages have been redesigned to adapt to the current Person/Tense settings. At the time of this writing, additional settings for the SetPersonTense routine include SECOND_PERSON (the default), THIRD_PERSON, and PRESENT_TENSE (also the default). These can be mixed and matched.

The additional setting of FUTURE_TENSE was considered for a time and has been coded with partial support, but is not fully implemented.

3. Creating rudimentary person/tense sensitive Narrative

Of course only the default messages have changed. The room descriptions are still rendered exactly as we typed them, in present tense. Generally this isn't a problem. When a developer designs a game for past tense, the descriptions should reflect this just as they

do when a game is designed in present tense. There are few places however, such as a reusable library entry or a game that allows the person/tense to change dynamically, where it is necessary to create text that transforms itself.

3a. ppf() and fst()

The NarativeTense and NarativePerson variables are set by the SetPersonTense() routine. It is possible (although not recommended) to modify your text by checking these values as in the following example:

```
print "This ";
switch(NarativeTense){
    PAST_TENSE: print "was";
    PRESENT_TENSE: print "is";
}
print " a cozy room with plush carpeting and walls
    lined in woodwork. A doorway ";
switch(NarativeTense){
    PAST_TENSE: print "lay";
    PRESENT_TENSE: print "lies";
}
print " to the west.";
```

The same can be done to achieve person sensitivity by checking the NarativePerson variable. This makes for somewhat clumsy code, however. OREnglish provides two rudimentary routines to help streamline the creation of person and tense sensitive narative: ppf (past, present, future) and fst (first, second, third). We'll touch briefly on these two routines, but be forewarned that due to nuances in the English language, they are not as useful as they may first appear. They do, however, provide the basis for a better, more powerful set of routines for creating morphing text, which will be covered in depth in the section covering "ORPronoun."

For now, they serve our needs adequately:

```
MyRoom Study "The Study"
  with w_to Laboratory
  , initial[; print (ig)fst("My","Your")," eyes ",
         (ig)ppf("took","take")," a moment to adjust
         to the dimly illuminated room.";]
  , description[; print "This ",(string) IS__TX," a
         cozy room with plush carpeting and walls lined
         in woodwork. A doorway ",(ig)ppf("lay","lies"),
         " to the west. ";]
;
MyRoom Laboratory "The Laboratory"
  with e_to Study
  , description [; print "Drab and bare, the lab ",
         (string) IS__TX," a cold sterile environment, not
         at all comfortable but brightly lit. Shards of
         glass, possibly from broken test tubes",
         (ig)ppf("littered","litter")," the floor. The
         study ",(ig)ppf("lay","lies")," to the east. ";]
;
MyObject Candlestick "candlestick" study
```

```
  with name 'stick'
  , description[; print "The candlestick ",(string) IS__TX,
         " copper, or perhaps brass. The metal ",
         (ig)ppf("had","has")," been polished so as to gleam
         in the room's lighting. It ",(string) IS__TX," heavy
         and hard enough to make a formidable weapon. ";]
;
```

Now the descriptions of the objects will be appropriate regardless of the current settings of the game-narrative:

```
The Laboratory
Drab and bare, the lab was a cold sterile environment, not at all
comfortable but brightly lit. Shards of glass, possibly from
broken test tubes littered the floor. The study lay to the east.

>n
I couldn't go that way.

>e
My eyes took a moment to adjust to the dimly illuminated room.

The Study
This was a cozy room with plush carpeting and walls lined in
woodwork. A doorway lay to the west.
I could see a candlestick there.

>get stick
Taken.

>eat stick
The candlestick was plainly inedible.
```

Note the use of the `(ig)` print rule when using the ppf() and fst() routines in a print statement to keep from printing the return value. This, and several other useful print rules are defined in the "ORTextFormatting" entry.

4. Changes made by ChangePlayerTense

In the above example, the reader may have noticed the use of the little-used IS__TX constant. Actually, the ORLibrary has redefined this into a variable and changes it appropriately when ever the ChangePlayerTense() routine is called. The same has been done for the following former-constants:

ARE__TX
IS2__TX
ARE2__TX
FORMER__TX
YOURSELF__TX

Additionally, the CANTGO__TX constant is also changed as well as the accompanying default message that it maps to.

3.5 Revamped OREnglish library functions and defines

There are several routines defined in the standard library's English file that have also been modified to be person/tense sensitive. These being:

CThatorThose
ThatorThose
ItorThem
CTheyreorThats
IsorAre

Also, for the sake of completeness, a lower-case version of the library's CTheyreorThats routine has also been implemented:

TheyreorThats

3.6 NPC Action Support

OREnglish also contains support for NPC actions. When used appropriately, standard library messages such as:

```
You put the candlestick on the table.
```

Are now sensitive to who is performing the action and may read as...

```
The troll puts the candlestick on the table.
```

...if an NPC is performing the action. More on this later in the section "ORNPCVerb".

ORBracketParserMsgs

An increasingly common feature of modern games distinguishes between the telling of the story, and the responses of the parser. Graham Nelson speaks of the different roles of the player, the protagonist and the narrator in his article "A Triangle of Identities" (found, among other places, in the DM4). The parser is, in effect, an unrecognized fourth identity usually separate from the story entirely and more concerned with the mechanics of the game. Messages from the parser seldom add to the story itself and serve a larger role in clarifying commands made by the player. It has become a recent, but well received practice of some authors to enclose parser messages in brackets.

ORBracketParserMsgs depends upon the OREnglish LDF, which was coded to leverage this entry if present. Simply including this module will wrap all parser generated messages in [brackets]. If, for some reason, the developer would like to turn off bracketing in the middle of a game, the global variable BracketedParserMsgs can be set to false to accomplish this. Additionally, the global variables BracketOpen, and BracketClose have been provided to set what text the messages will be wrapped in. For example, if the developer would prefer to wrap the parser messages in double greater-than/less-than characters, then the following lines of code could be added to the Initialise() routine:

```
BracketOpen="<<";
BracketClose=">>";
```

ORBanner

ORBanner is a simple extension to the standard library's banner routine. It identifies the version of the ORLibrary that was compiled in and exposes the global variable/routine ORBannerText which can be defined in the program to add custom text to the banner.

The following exemplifies defining ORBannerText as ether a string or a method.

```
Constant ORBannerText " First time player's should type HELP.";
```

Or

```
[ORBannerText;
  print "This is a BETA version of ",(italics)Story,".";
];
```

ORPronoun

The first module that OREnglish pulls in is the ORPronoun entry. This entry is based upon techniques discussed, or at least touched upon, in the article "Pronouns on Steroids" which can be found in "An Inform Developer's Guide" or online at the following URL:

informguide.aspx?article=11

Review of that article is recommended. Contrary to its name, the ORPronoun module handles more than just pronouns. It also implements routines to facilitate subject-verb agreement in text that is applied to NPCs as well as the player. It also adds fluidity to your narrative by completely automating the use of pronouns.

1. Absolute Pronoun Print Rules

ORProunoun provides several basic print rules to implement the various types of pronouns used in the English language. The names of these print rules are easy to remember since they match the first-person pronouns. These seem more natural than slightly more descriptive names such the (PronounAcc) or (PronounNom) examples from x62 or the DM4. The five lowercase versions of these are (I), (Me), (My), (Mine), and (Myself). Similarly, the five uppercase versions are (CI), (CMe), (CMy), (CMine), and (CMyself).

As you most likely could have guessed, these ten print rules are especially useful when dealing with animate objects (although deal equally well with non-animate objects as well). They utilize the fst() routine discussed previously in order to correctly determine the pronoun when the player is specified, but also check for gender and plurality. Individually, each print rule will produce one of eight possibilities. The (Myself) print rule, for instance, will consult the appropriate variables and choose the correct word(s) from the following list:

myself
ourselves
yourself
yourselves
their selves
itself
herself
himself

2. A Touch More Flexibility With Potential Pronoun print rules
The ten absolute print rules discussed above will always print the appropriate pronoun. However, there are times, especially when dealing with random events, that we do not know for certain that we want to print the pronoun. What is needed is an algorithm for determining if a pronoun will be understood or if the actual name needs to be used for clarity. This algorithm is implemented in the "TheXXX" print rules: (TheI), (TheMe),

(TheMy), (TheMine), and (TheMyself). Additionally, these five print rules have capitalized counterparts: (CTheI), (CTheMe), (CTheMy), (CTheMine), and (CTheMyself). To demonstrate, consider the following code snippet, modified from the examples given in the "Pronouns on Steroids" article mentioned previously. Observe the potential confusion with two male NPCs that is avoided using these new potential print rules rather than the absolute ones:

```
MyRoom foyer "Foyer"
  with description[; print "Of striking elegance, the
         foyer ",(string)IS__TX, " exquisite in every
         detail. ";]
  , each_turn[; print"^";
         if(parent(butler)~=self || parent(maid)~=self) return;
         ClearPronoun();
         if(random(2)==1) print (CTheI) butler,
              " coughs. ";
         if(random(2)==1) print (CTheI) maid,
              " scratches ",(my)maid," chin. ";
         if(random(2)==1) print (CTheI) butler,
              " grimaces at ",(theMe)maid,". ";
         if(random(2)==1) print (CTheI) maid,
              " gives ",(theMe)butler," a wink.";
  ]
;
MyObject -> butler "butler" has animate
  with name 'man'
  , description [; print (CTheI)self,
              " ignores ",(me)player,", preoccupied with
              other things.";
         ]
;
MyObject -> maid "cleaner" has animate
  with name 'maid'
  , description [; print (CTheI)maid," seems to be
              paying more attention to ",(the)self,".";
         ]
;
```

Run the example to see that each turn, any of the following may be output (or other combinations):

```
The cleaner scratches his chin. The butler grimaces at him.
```

```
The butler coughs. He grimaces at the cleaner. The cleaner
gives him a wink.
```

```
The cleaner scratches his chin. The butler grimaces at him.
He gives the butler a wink.
```

Note that the print rules keep track of whom the pronoun is currently referring to so that confusing statements such as "He grimaces at him" are never produced. The rules are, however, smart enough to recognize and adjust for circumstances in which two

objects can both be referenced by pronouns. Suppose the cleaner was female:

```
MyObject -> maid "cleaning lady"
  has animate female
  with name 'maid'
  , description [; print (ig)CIVerb(self,"seemed","seem"),
         " to be paying more attention to ",
         (the)butler,".";]
;
```

The following would then be entirely possible (odds: 1/16):

```
The butler coughs. The cleaning lady scratches her chin. He grimaces at
her. She gives him a wink.
```

3. Noun-Verb Agreement

But for a few small nuances in the English language, all would be well with the fst() and ppf() routines. The nuances have to do with noun-verb agreement. One example of this occurs only when three conditions are true:

1) The narrative is written in present tense.
2) We are speaking about a noun in the third-person.
3) The noun is singular.

When all of the above are true -- as they almost always are in an IF game where NPC actions are described -- then we slightly modify the verb, usually by adding an 's' to the end. For instance:

You **swim** in the water.
The troll **swims** in the water.

Note the modified form of the verb "swim" in the second sentence, which matches the three conditions listed above.

Additionally there are special verbs, the so-called "verbs of being" in particular, that take different forms as well. Several routines have been implemented in the ORPronoun module to address the changing forms of verbs.

3a. vrb()

The first basic routine that we will discuss is the vrb() routine. It is important to note that the vrb() routine is not a print rule. Like the ppf() and fst() routines, when vrb() is called embedded within print statement it is usually preceeded by the (ig) print rule (discussed in "ORTextFormatting").

The vrb() routine was implemented to assist in achieving noun-verb agreement. The first parameter expected is the object that is performing the action. This completes the routines ability to check for the three conditions described above and select the verb appropriately.

The second and third parameters are the past tense and present tense forms of the verb being processed. For most verbs, these three parameters are all that is needed. Consider the following code snip:

```
Object troll "troll" has animate;

 ...

o=random(troll,player);
print (TheI)o," ",(ig)vrb(o, "swam","swim"),
     " in the water.";
```

In the narrative form of the overwhelming majority of IF games - that is, present tense/second person - fifty percent of the time, the output will be:

```
You swim in the water.
```

And the other fifty percent of the time it will be:

```
The troll swims in the water.
```

In the second case, the vrb() routine has added an 's' to the verb in order to make it agree with the noun. This is all that is needed for most verbs. There are many that do not follow this form, however. "Fly," for example, will incorrectly generate the word "flys," which is not correct. The fourth, optional parameter will be used instead if supplied. Changing the above example, to use the verb "fly" looks like this:

```
print (TheI)o," ",(ig)vrb(o, "flew","fly","flies"),
" in the water.";
```

And outputs the following:

You **fly** in the water.

Or:

The troll **flies** in the water.

It is, of course, perfectly legal to specify the third form all the time, however it is generally considered to be wasteful since:

```
print vrb(o,"swam","swim","swims");
```

and

```
print vrb(o,"swam","swim");
```

are equivalent.

For games written in a future tense, there is also a fifth parameter used to accommodate that form of verb as well. Like the fourth parameter, this too is optional and the vrb() routine will attempt to generate the correct form of the verb if it is not specified. The same example that we have been working with so far, when run under a future tense context, will generate a somewhat prophetic statement such as:

The troll **will fly** in the water.

3b. Other Basic Noun-Verb Agreement Routines (am, can, have)

There are other routines that have been included simply because of their usefulness when generating adaptive narrative. The (am) print rule, in particular will appropriately choose and print the appropriate form of the verb "is":

was
am
will
were
are
is

Additionally, because of the frequency of their use, the print rules (can) and (have) have also been implemented in a similar manner.

4. Complete Noun-Verb Agreement Routines (IVerb, IAm, ICan, IHave)

Although there are a number of cases where the above routines are called for alone, it is generally more common (in English anyway) to have a verb immediately follow the noun. For ease, IVerb(), (IAm), (ICan), and (IHave) were created. These routines do exactly what vrb(), (am), (can), and (have) do but also precede the verb output with a call to the (TheI) print rule and a space. Therefore, the following two code snips are equivalent:

```
print (TheI)o," ",(ig)vrb(o, "flew","fly","flies");
```

```
IVerb(o, "flew","fly","flies");
```

A capitalized version of each of these routines has also been created. That is, CIVerb(), (CIAm), (CICan), (CIHave).

Also, because the negative form of "can" is a single word in present tense ("cannot") and two words in past tense ("could not"), two additional rules have also been created: ICant and CICant;

In our maid/butler example, the text is written in present tense. Using these routines, we can more easily make the example print correctly regardless of the narrative's tense and person settings and the correct verb form is correctly chosen:

```
MyRoom foyer "Foyer"
  with description[; print "Of striking elegance,
            the foyer",(string)IS__TX, " exquisite in
            every detail. ";]
  , each_turn[;
            if(parent(butler)~=self || parent(maid)~=self) return;
            print"^";
            ClearPronoun();
            if(random(2)==1) print (ig)CIVerb(butler,
                 "coughed","cough"),". ";
            if(random(2)==1) print (ig)CIVerb(maid,
                 "scratched", "scratch","scratches"),
                 " ",(my)maid," chin. ";
            if(random(2)==1) print (ig)CIVerb(butler,
                 "grimaced","grimace")," at ",
                 (theMe)maid,". ";
            if(random(2)==1) print (ig)CIVerb(maid,
                 "gave","give")," ",(theMe)butler,
                 " a wink.";
      ]
;
MyObject -> butler "butler"
   has animate
   with name 'man'
   , description [; print (ig)CIVerb(butler,
             "ignored","ignore")," ",(me)player,",
             preoccupied with other things.";
    ]
;
MyObject -> maid "cleaning lady"
   has animate female
   with name 'maid'
   , description [; print (ig)CIVerb(self,
             "seemed","seem")," to be paying more
```

```
            attention to ", (the)butler,".";
    ];
```

Wary reader's may notice the (ig) print rule. We will cover that in the section entitled "ORTestFormatting".

ORTextFormatting

The ORTextFormating module contains a collection of print rules that facilitate the formatting of text.

1. (ig) ignore print rule

The ignore print rule (ig) is used to suppress the printing of routine return values. This is primarily useful when embedding routines in print statements. Since all routines have a return value, embedding them in a print statement causes their return value to be printed as well. Consider the first potential line of text from the each_turn routine in the previous example without the use of the (ig) print rule:

```
print CIVerb(butler, "coughed","cough"),". ";
```

This will result in the following output:

```
The butler coughed0.
```

Notice the unwanted zero printed at the end. The (ig) print rule suppresses this.

At this point I feel compelled to give due credit. The (ig) print rule was not so much inspired by, as blatantly stolen from Roger Firth's article on print rules, dubbed with the oh-so-telling name"Print Rules". It is recommended reading and can be found in "An Inform Developer' Guide" or at the following URL:

http://www.onyxring.com/informguide.aspx?article=8

2. Italics, strong, highlight style print rules

The Inform language has several "style" commands that change the way text is formatted to the screen. Use of these style commands is fairly straightforward. The following will print out a line of text with different styles:

```
print "To ";
style bold;
print "boldly";
style roman;
print " go where ";
style underline;
print "no man";
style roman;
print" has gone ";
```

```
style reverse;
print "before...";
style roman;
```

The (italics), (strong), and (highlight) print rules allow formatting to be contained in a single print statement. The above code block is equivalent to the following:

```
print "To ",(strong)"boldly"," go where ",(italics)"no man",
    " has gone ",(highlight)"before...";
```

Both examples produce the same results:

To **boldly** go where *no man* has gone before...

It should also be noted that not all interpreters are equal. The (italics) print rule uses the "underline" style because it appears as italics on specific platforms familiar to the author. Be advised, however, that other interpreters will render text printed with this print rule as underlined text which is, arguably, more appropriate.

3. (arraystring) character array print rule

Although there are many (not me!) that are of the opinion there is no real need to use arrays as text, the desire to is nevertheless present. The (arraystring) print rule has been introduced to help print the contents a character arrays. The following example will populate an array string, and then print it with the print rule.

```
array buffer -> 1000;
@output_stream 3 buffer; !--capture text
print "Hello world!";
@output_stream -3; !--release text
print (arraystring) buffer; !--now print the contents of buffer
```

The ORDynaString module leverages this print rule. Indeed, that module makes the need to use this print rule alone negligible. We will, therefore, let the above example stand on its own.

As far as the examples given so far, these can be downloaded from the following URL:

http://www.onyxring.com/downloads/jimfisher/ORLibDoc1.inf

Part C
Miscellaneous ORLibrary Modules

ORCenter

Center is a flexible routine used to center text on the screen. Its simplest usage is simply:

```
Center("Chapter One");
```

As could probably be guessed, this will simply center the words "Chapter One" on the screen. If the `Center()` routine does not encounter a line-break before reaching the ending of the line, it will word wrap the text and center it anyway. In this manner, unformatted text can also be centered.

In addition to the `text` parameter, there are two parameters that can be used to affect the output of `Center()`. The first is `maxwidth`, which determines the maximum width of each line and inserts line breaks accordingly. If this parameter is not given, then the screen width, as provided by the interpreter, is assumed.

The second optional parameter is the `hilight` parameter. The value, which is defaulted to zero for off can be passed in as either a one or a two. Specifying a value of one centers the text in reverse, prefixing the centered text with non-reversed spaces. A value of two prefixes the text with reversed spaces. Generally this is most useful when printing on the status line.

As a final note, it should be mentioned that the center() routine relies upon mono-spaced fonts to achieve accurate centering calculations and will temporarily assume that mode when centering text.

The following code snip exemplifies the `Center()` method:

```
Center("^^How shall the burial rite be read?^The solemn
    song be sung?^The requiem for the loveliest dead,
    ^That ever died so young? ^^-Edgar Allan Poe ~A Paean~");
```

ORTransition

The Transition() routine generates a transitional effect. It is most often used to separate passages of stand-alone text, such as chapters, by printing separating text ("---"), prompting the user to press the space bar, and then clearing the screen. This is the default behavior of the routine when called with no parameters; however, this can be configured to some degree by first and second parameters, `text` and `skiperase`, respectively.

The following tutorial example, added to the `initialize()` routine, demonstrates the transition between an introduction and the actual game substance:

```
 Center("^^Prologue^");
 print "It had been a game, a murder-mystery party where the
     players paired off and investigated a mock killing.
     Fun, I supposed, if something hadn't gone wrong. Or
     right. One of the players, it seemed, had a problem
     with another. Ms. Corpus had been murdered for real
     by her game-partner during the course of the mock
     investigation. ^^It looked as though the killing had
     been planned all along, and the game was just a ruse
     to get the right people together. ",(italics)"But who
     had been her partner?","^^That was our job, to find
     out. Both me, and that ",(strong)"creep",", Inspector
     Snobby, had been assigned to the same case, though we
     were working independently as we had numerous times
     before.^^Historically, my track record competing
     against Snobby wasn't too great.",(italics)" But this
     time",", I swore with conviction, ",(italics)"I would
     solve the case first!";
 Transition();
 Center("^^^^Part the First^^");
 print "Snobby and I had arrived at exactly the same moment, as we
     often did. We were escorted into the Living Room and
     each given a list of the game's players. The suspects,
     we were told, were still wandering free in the house. I
     would have to question everyone and see who Corpus'
     partner had been. Snobby, I knew, would be doing the
     same.^^";
Transition("* * *",true);
```

Note the use of the `(italics)` and `(bold)` print rules covered in the ORTextFormatting section.

ORFirstThoughts

One of the considerations that must be made when writing interactive fiction concerns the coalescing of player and character knowledge bases. The character of a story will usually have some knowledge or opinion about something that needs to be communicated to the player. The ORFirstThoughts entry provides one possible mechanism for accomplishing this.

Objects derived from the ORFirstThoughts class make use of descriptions in two logical parts: The character's subjective thoughts when (s)he first examines the object and the actual non-subjective description of the object.

In practice the subjective portion of a description can both precede and follow the actual descriptive text. Think of it as a description sandwich where the meat of a description is wrapped in subjective bread. Each of these three description pieces are placed into one of three properties:

firsttime
descrip
firsttimePost

When an object (or room) is first examined, the text in the Descrip property is output between the text of the two Firsttime properties. By default, subsequent examinations do not include the Firsttime text.

For players that want to re-read the character's initial first impressions, the verb REEXAMINE has been implemented and variations of EXAMINE and LOOK point to this (e.g. EXAMINE BOOK AGAIN, LOOK AGAIN AT MIRROR).

On a slightly technical note, the ORFirstThoughts class makes use of the description property. Defining this property effectively disables the ORFirstThoughts functionality for any given object.

This is a good example for again mentioning the usefulness of base classes. One simple addition to the MyObject class will affect every object in the example as it stands thus far:

```
class MyObject !the basic object that you can pick up manipulate etc...
   class ORFirstThoughts
;
```

Now we can implement some of the thoughts for the character as he sees the object or room in question. For the sake of clarity, we'll eliminate the morphing text enhancements that we made in previous sections and write the text in first-person, past-tense

```
MyRoom Study "The Study"
```

```
  with w_to Laboratory
  , initial "My eyes took a moment to adjust to the dimly
        illuminated room."
  , firsttime "The study reminded me of my grandfather. I
        am not certain why; possibly because of the manner
        in which it was decorated. "
  , descrip "It was a cozy room with plush carpeting and walls
        lined in woodwork. An iron door lay to the west."
  , firsttimepost "It was just the sort of room that my
        grandfather would have as his own. He had always
        loved woodworking. I could almost imagine him sitting
        here, puffing on his pipe."
;
MyRoom Laboratory "The Laboratory"
  with e_to Study
  , firsttime "It was a strange room to have in a house, a
        laboratory. I wondered what sort of experiments were
        conducted in the room as I looked around sizing it up. "
  , descrip "Drab and bare, the lab was a cold sterile
        environment, not at all comfortable but brightly lit.
        Shards of glass, possibly from broken test tubes littered
        the floor. The iron door to the study lay to the east."
;
MyObject Candlestick "candlestick" study
  with name 'stick'
  , descrip "The candlestick was copper, or perhaps brass. The metal
        had been polished so as to gleam in the room's lighting. It
        was heavy and hard enough to make a formidable weapon."
  , firsttimepost "I hefted it, calculating the force it would exert
        across the back of someone's head. It could definitely serve
        as a murder weapon."
;
MyRoom foyer "Foyer"
  with descrip "Of striking elegance, the foyer was exquisite in
        every detail. The front door on the southern wall was
        closed and locked, preventing suspects from leaving, but
        they wandered freely throughout the rest of the house. To
        the north I could see a living room; to the east, a long
        hallway. "
  , firsttime "I looked around the entrance room and
        smirked. This house was like something out of a movie.
        Extravagance was the defining trait of this crime scene. "
  , firsttimepost "I wandered in amazement why the suspects
        hadn't been confined for the duration of the investigation. "
;
```

ORDoor and ORDoorInit

The ORDoor class offers no new functionality over and above the standard door covered in section 13 of the DM4. It does, however, greatly reduce the amount of code needed to create one of these doors. The following is an example of a basic traditional door which we could put between the Laboratory and the Study in our tutorial (Don't do this):

```
!--A sample door without the ORDoor entry
Object SteelDoor "iron door"
  has openable door static scenery
  with name 'metal'
  , door_dir [;
     if (parent(actor) == Study)
          return w_to;
     else
          return e_to;
    ]
  , door_to [;
     if (parent(actor) == Study)
          return Laboratory;
     else
          return Study;
    ]
  , found_in Study Laboratory
;
```

The `door_dir` and `door_to` properties implement a fairly straightforward algorithm. Specifically, they use the current location of the actor (or player if the door is not sensitive to NPC actions) to decide which of two possible values to return. The ORDoor class implements a generic algorithm to do this so there is no longer a need to write specialized code for these properties.

Additionally, use of the ORDoorInit module (which automatically includes the ORDoor module) will interrogate the world map and populate the `found_in` property during game initialization. Therefore, by including the ORDoorInit module, the same door as above can be created in one line.

```
ORDoor SteelDoor "iron door" with name 'metal';
```

All that remains is to point the map directions to our door:

```
MyRoom Study "The Study"
  with w_to SteelDoor
```

```
. . .

MyRoom Laboratory "The Laboratory"
  with e_to SteelDoor
. . .
```

ORRecogName

The name property is the mechanism by which the Inform parser identifies which objects the player is referring to. The separation between the name and the display name has often been a source of grumbling by those who program in Inform. Many developers are annoyed at having to specify the words in the display name in the name property as well:

```
Object DirtyBook "old black leather bound book"
  with name 'old' 'black' 'leather' 'bound' 'book'
;
```

The ORRecogName library entry eliminates the need to specify the name property at all. The words of the display name will be used to identify an object and the name property can be used to specify synonyms alone.

There is no code needed to make the ORRecogName entry to work. Simply include it and it will begin working. In the examples give so far, notice that without this entry, the `Candlestick` object can only be reference by the word "stick". and the `SteelDoor` object can can only be referenced by the word "metal". After adding this entry, these objects can also be addressed by any of the words contained within their display name. That is, commands such as EXAMINE CANDLESTICK or OPEN DOOR will now work.

ORLookRoom

This module adds the functionality of examining a neighboring room. That is, a player presented with the description of a room:

```
You are in the Dining Room. To the North is the Laboratory.
```

will have the ability to use any of the following:

```
>look north
>look at laboratory
>examine laboratory
```

The description of a neighboring room is determined by the following logic:

1) If the current room has a direction-description property defined, then that is used. The following are direction-description properties:

n_look
s_look
e_look
w_look
nw_look
sw_look
ne_look
se_look
u_look
d_look

2) Failing the above, the room being looked at is searched for the 'remote_description' property. If found, then that is use.

3) Alternately, the room being looked at is checked for the property 'describe_as_if_present' to be set to true. If so, then the normal room descriptions are called as though the character were actually present in the room.

4) Finally, if the all of the above fails, then the room being looked at is checked for the property 'describe_as_if_present' to be set to true. If so, then the normal room descriptions are called as though the character were actually present in the room.

Additionally, for completeness, the commands EXAMINE ROOM and LOOK AROUND have been implemented along with minor variations.

Note that determining the description of a remote location is moderately complex and an effort is made to determine if the room can actually be seen. That is, if the room is directly adjacent to the current location, or connected via a door that is either open or transparent. An additional property `visible_from` can be defined for a location to list other rooms from which it can be seen, even if it is not directly connected to them.

If the player attempts to look in a direction for which scenery text cannot be determined, then the default error message is printed. It is possible to redefine the text for this message in the traditional fashion using the LibraryMessages object, but it is often preferred to override this with room that is specific to certian locations. In these cases, the `cant_look` property may be defined.

The following code snip exemplifies this entry:

```
object mountain "mountain"
   has light
   with name 'mountain'
   ,    description "You are high on a mountain.
           Far below, to the south, lies a field."
   ,    remote_description "Far above rises the
           imposing mountain.  Snow capping the top."
   ,    w_to field
   ,    d_to field
;
object field "field"
   has light
   with name 'field'
   ,    description "It is bright and cheery here. A
           mountain lies off to the east, and a small
           open house lays to the west."
   ,    w_look "The house is aged house is open and
           inviting."
   ,    w_to house
   ,    e_to mountain
   ,    u_to mountain
;
object house "house"
   has light
   with name 'house'
   ,    description "It is a dark and empty house. The
           doorway to the west is open, revealing a
           field."
   ,    e_to field
   ,    e_look "The field to the east is bright and cheery.
           A mountain rises up above the horizon."
   ,    cant_look "You can't see through walls. Only
           through the open doorway to the east can
           you see anything."
;
```

In the above example, the field and the house are visible from each other by virtue of the `w_look` and `e_look` properties. The descriptions specific to the room they are in and so it is acceptable to put directions in them ("...the field to the east...") The mountain location

defines the `remote_description` property. Since the text of this property will be printed from any location that can see the mountain, it is slightly more generalized.

Note that while the mountain is automatically visible from the field (since the locations are connected) it had to be *made* visible from the non-adjacent location (the house) with the visible_from property.

ORSupporterContainer

One of the areas in which Inform falls short of the proverbial mark is in regard to the concept of "parent objects". In particular, an object can either be a supporter and have things put upon it (like a table) or be a container and have things put in it (like a bowl), but not both. This is limiting because in true life, nearly any container that can be opened and closed can also be a supporter (like a jewelry box or a refrigerator).

The ORSupporterContainer library entry implements fairly wide-spread changes to the standard library to allow the container and supporter attributes to exist simultaneously on an object. A new attribute -- "contained" -- is given to objects that are "in" a container. Absence of this attribute means that the child object is "on" its supporter parent. No specialized classes need to be leveraged to achieve this new behavior, simply include the module and it will work.

To demonstrate this module in action, let's create a refrigerator and place some items both inside and on top of it:

```
class MyBook
   with short_name "leather bound book"
   ,    plural "leather bound books"
   ,    description "It was a single book, bound in leather."
   ,    name 'book' 'books//p'
;
object fridge "refrigerator" house
   has static openable supporter container
   with description "It is a small, dorm-sized refrigerator that
            seemed to double as a end table."
   ,    name 'fridge' 'refrigerator' 'table' 'dorm-sized'
;
object ->-> apple "apple"
   has edible contained
   with description "Yum! The apple looks delicious."
   ,    name 'apple'
;
MyBook ->-> book1;
MyBook ->-> book2;
MyBook ->-> book3;
MyBook ->-> book4;
```

Notice in the sample transcript how the distinction is made between what is contained in the refrigerator and what in on top of the refrigerator:

```
house
It is a dark and empty house. The doorway to the
west is open, revealing a field.

You can see a refrigerator (closed and on which
are four leather bound books) here.

>open fridge
You open the refrigerator, revealing an apple.

>search fridge
```

```
On the refrigerator are four leather bound books.
In the refrigerator is an apple.
```

ORSeveral

This module allows multiple objects of the same class to be described with an adjective rather than an exact number. For instance, it may be preferable that the description of 22 gold coins read as "several gold coins."

Two properties and one attribute have been introduced to implement this behavior:

plural_many - Set this property to the actual text to be displayed instead of the number. If this property is not defined for the class of the multiple objects, then the default library behavior takes precedence

many_number - Set this property to the number which must be surpassed in order for the new behavior to occur. If this property is not defined for the class of the multiple objects then the default is used (3).

specific_number This attribute is given to a parent object (or container) to force the old behavior. This is often given to the player so that 22 gold coins reads as such in the player's inventory (instead of "several gold coins")

To demonstrate this module, we will add the `plural_many` property to the MyBook class created previously:

```
class MyBook
  class MyObject
  with short_name "leather bound book"
  , plural "leather bound books"
  , plural_many "several"
  , descrip "This is a single book, bound in leather."
;
```

We'll leave `many_number` set to the default of 3, since that seems a reasonable point to begin counting books. Having done this, SEARCHing the refrigerator will now render the description as "several leather bound books" instead of listing the exact number of "four".

There is one place, that we would like to continue to express the exact number: our inventory. To accomplish this, we need to give the player object the `specific_number` attribute in the Initialize() routine:

```
give player specific_number;
```

Now there will never be any doubt as to exactly how many books the player is holding.

OORExamWithContents

Usually, to view the contents of a container, LOOK IN must be used. The OORExamWithContents entry is a simple module that causes the contents of a container to be described when it is examined. It takes no additional code to implement. Using the previous example to demonstrate this module, try examining the refrigerator without the `OORExamWithContents` module. Notice that the books sitting on top of it are mentioned in the initial description of the house, but not when the player issues the EXAMINE FRIDGE command. In order to see its contents the command EXAMINE FRIDGE must be used. Now uncomment the USE_OORExamWithContents constant and recompile. Notice that EXAMINE FRIDGE will now display the contents.

ORUniqueMultiMessage

Several identical messages are often generated when a player performs an action that affects multiple objects. Consider the following normal transcript:

```
>TAKE ALL BOOKS FROM FRIDGE
leather bound book: Removed.
leather bound book: Removed.
leather bound book: Removed.
leather bound book: Removed.
```

The ORUniqueMultiMessage module will analyze the responses made by each of the commands and attempt to consolidate them (provided they occur sequentially).

By leveraging this entry, the above example would appear as:

```
>TAKE ALL BOOKS FROM FRIDGE
4 leather bound books: Removed.
```

No additional code need be written to make this functionality occur. Simply include this module and consolidated messages will occur.

ORDistinctRead

Generally, the commands Read and Examine are treated the same and are mapped to the same verb which simply displays an object's description. This module adds a behavior to make a distinction between reading and just looking at an object.

To do this, ORDistinctRead relies upon the property `read_value`. If present, then the value of this property will be displayed in response to the READ command. If not, then a message is printed explaining that the object "cannot be read, only examined" and the normal description is printed.

The following code snip demonstrates this entry:

```
Object ->-> newspaper "newspaper"
  with name 'paper' 'newspaper' 'headline'
  , description "A wrinkled newspaper with an imposing headline."
  , read_value "In bold black letters the headline reads, ~Martians Invade!~"
;
```

ORProp

This class eases implementation of a generic object which does not need to be referenced in the game. Sample use:

```
ORProp -> "butterflies" has pluralname with name 'butterflies';
```

This method has two distinct advantages over specifying words in the room's name property. Specifically, it allows the developer to specify the pluralname attribute thereby generating a plurality sensitive message such as "those don't need to be referenced in this game" rather than the standard singular version "that doesn't need to be ..."

The second advantage is that it works in conjunction with packages that utilize the name property for rooms (Such as ORLookRoom).

Some of the functionality of the ORProp class can also be overridden. For instance, many developers prefer to provide a description for scenery objects, even though it cannot be manipulated in anyway. ORProp will allow the EXAMINE command if a description is provided. Also, ORProp defines a property called `message` that will override the default "...that is not significant..." message. This is particularly useful when creating objects that are viewable but unreachable.

The following code snips exemplify possible uses of ORProp:

```
ORProp -> "glass shards"
   has pluralname
   with name 'glass' 'shards'
;
ORProp -> "carpeting"
  with name 'carpet'
  ,   description "It is red, and quite nice."
;
ORProp -> "actress"
  with name 'actress'
  ,   description "She looks quite pretty up there on stage."
  ,   message  "You can't reach her.  She's up on stage."
;
```

ORGibberish

Some modules are fun, but have little practical use. Still, working under the belief that the existence of such a module creates its own need, we've introduced ORGibberish, which generates random, human pronounceable words of varied length.

The ORGibberish object defines a method called MakeWord(). MakeWord() takes a single number as a parameter which specifies the number of requested vowel sounds. It then generates a random word, with the requested number of vowels, based upon rules derived from phonetic principals of pronunciation and then prints it.

Of what possible use could this module be? Random passwords, the names of monsters, and spell incantations come to mind.

It is important to say here that most entries in the ORLibrary do not create objects, but rather define classes for the developer to create. There are a few notable exceptions of which ORGibberish is one and ORNPC is another.

Use of the ORGibberish object is basic. Although it can be leveraged in many more powerful ways, as we will see in the ORDynaString example. For now, the following will demonstrate:

```
[Initialise;
    print "^";
    ORGibberish.MakeWord(2);
];
```

Now run the program to see the randomly generated word. The output will likely not be even remotely similar to this:

```
squeesle
```

For another example of using the ORGibberish object, refer to the ORDynaString section.

ORDynaString

There is little need for dynamic string generation in Inform. Hundreds of games have been written without the dynamic strings ever being needed or even desired. Still, there are a very few occasions where building a string dynamically is more elegant than alternative methods. (See the ORInsultCompli_KT entry for one such example of this.)

ORDynaString is a wrapper around the @output_stream functionality. That is, it is turned on to print text to an array, and then turned off. The resulting string, stored in an array pointed to by the "buf" property (by default, ORDynaBuf) can then be printed using the (dynastring) print rule. For interchangeability, the (dynastring) print rule also works equally well with strings.

The following routines are exposed from the ORDynaString class:

- buf - points to an array which will store the target string.
- capture() - redirect all printed text to the array pointed to by buf.
- release() - stop redirection text to array and redirect back to the screen.
- upper(id) - takes the character stored in the array at position id (1-based index) and makes it uppercase.
- lower (id) - takes the character stored in the array at position id and makes it lowercase.
- upper_all() - makes the entire string upper case.
- lower_all() - makes the entire string lower case.
- get_char(id) - returns the character at position id.
- set_char(id,val) - sets the character at position id to the value val.
- length() - returns the length of the string.
- substring(start,count) - prints a portion of the total string.

There are several obscure uses for the ORDynaString class. One fo these uses is the ability to maintain a copy of randomized text. The following example is an elaborate implementation of ORDynaString. Specifically, it is an office desk toy that displays a different word/definition every time it is shaken. For amusement, we'll leverage the ORGibberish entry to generate random, human-pronounceable words and piece together a random definition to accompany them. We will also leverage the read_value property defined in the ORDistinctRead module.

```
ORDynaString rword;
ORDynaString rtext;
array word_buf -> 30;

Object word_cube "dictionary cube" glasstable
  with name 'word' 'definition' 'toy' 'office' 'electric'
  , description "It was an electric office-toy in the shape
         of a cube. It displayed a changing dictionary
```

```
        definition of little-known words which could be
        read at any time."
  , before[;
        Shake: self.Shuffle();
             print (ig)CIVerb(actor,"shook","shake")," the
             cube and the words ", (ig)ppf("rearranged",
             "rearrange")," themselves.";
             rtrue;
  ]
  , Shuffle[;
        give self general;  !----we have shuffled at least once
        rword.buf=word_buf; !---assign different array
                            !---so dynas don't overwrite
                            !---each other
        rword.capture(); !----start capturing word text
        ORGibberish.MakeWord(2); !---make gibberish
        rword.release(); !---done with new word.
        rtext.capture(); !---now capture formatted text
        print "On the cube appeared the following
                  word/definition pair:^^";
        rword.upper(1); !--capitalize the word
        print"~",(dynastring)rword," - ";
        rword.lower (1); !--uncapitolize
        !---generate and print a random definition...
        noun=random(true,false);
        if(noun) print "(N) The"; else print "(V) To";
        print " ",(dynastring)rword," is ";
        if(noun) print "a"; else print "to";
        if(noun){
             print (string)random("n abysmal"
                  ,"n abominable"," brainless"," dreadful"
                  ," horrendous"," miserable"," stupid"
                  ," fabulous"," marvelous","n amazing"
                  ," popular"," delightful"
                  ," valuable")," ";
             print (string)random("barrel of"
                  ,"apology for","pile of","heap of"
                  ,"sack of","glob of","paragon of"
                  ,"modal of","example of", "hunk of"
                  , "bundle of","manifestation of")," ";
             print (string)random("monkey filth"
                  ,"vomit","gunk","garbage","fertilizer"
                  ,"lizard snot","disirablity" ,"precision"
                  ,"accomplishment","pig phlegm"
                  ,"maggot fodder");
        }
        else{
             print " ";
             print (string)random("cry to"
                  ,"yell at","smell","examine closely"
                  ,"mix up","spit upon","pee against"
                  ,"insult","rub down","worship","flatter"
                  ,"propose to","forget about")," a";
             print (string)random(" stranded"
                  ," dirty"," neighbor's favorite"
                  ," free"," smelly"," disproportionate"
                  ," slightly used","n insignificant"
                  ,"n extrordianary"," stolen"
                  ," recently washed"," spoiled")," ";
             print(string)random("daughter","idol"
                  ,"donkey","tree","mailbox","monkey"
                  ,"wife","brother","lunch","coat"
                  ,"lawyer");
        }
        print ".~^^Followed by the words ~Shake cube to change
             definition.~";
        rtext.release(); !----done capturing text
  ]
  , read_value [;
        if(self hasnt general) self.shuffle();
        print(dynastring)rtext;
  ]
;
```

Now all that remains is to actually define the verb shake and provide a default message:

```
Verb "shake" * held -> Shake;
[ShakeSub;print (ig)CIVerb(actor,"shook","shake")," ",(the)noun,
  ", and ",(ig)vrb(actor,"felt","feel")," better, less tense.";];
```

ORDynaMap

This module dynamically creates the ties between certain rooms based upon movements by the player, similar to an effect in the "mars" scene of Adam Cadre's "Photopia".

The use is fairly straightforward: Simply create an instance of the ORDynaMap class with the found_in property containing a list of the rooms that are to be arranged, in the order that they are to appear. The first entry in the list should already be accessable by the player.

As for the rooms that are to be created, they should initially provide all directions. The cant_go property on the ORDynaMap object, if defined, will be propagated down to the rearranged rooms which also provide cant_go, but that do not define it.

To simplify the creation of the rooms that can be rearranged by the ORDynaMap object, a simple class is provided that defines all directions and the cant_go property as zero. Note the following example which allows the player to explore in virtually any direction from the water's edge:

```
ORDynaMap
  with cant_go "The trees are too dense to travel
               in that direction."
  ,   found_in wateredge sparseforest forestmidst
                  forestclearing treasuretrove
;
ORDynaMapRoom wateredge "Edge of water/forest"
  with s_to "Your boat is that way, but you can't
               go home until the treasure is found."
  , description "This is a small strip of land separating
               the southern inlet of ocean from the vast forest
               in all other directions."
;
ORDynaMapRoom sparseforest "Sparse Forest"
  with description "This is the start of a vast forest.
               The trees here are thin and sparse."
;
ORDynaMapRoom forestmidst "Midst of Forest"
  with description "Smack dab in the middle of the forest.
               Trees are everywhere."
;
ORDynaMapRoom forestclearing "Forest Clearing"
  with description "Suddenly clear of trees. You feel yourself
               draw closer to the target."
;
ORDynaMapRoom treasuretrove "Pile of Treasure"
  with description "At last! The long sought-after pile of
               treasure. You are victorious."
;
```

ORCantGoOdd

CantGoOdd is a modification to the regular GoSub routine which causes it to ignore the cant_go property if the direction is up, down, in, or out. This seems to be one of the more common oddities in Inform adventures. For example...

```
You are in a small forest, surrounded by
trees. Only to the north is there a passage.

>s
Trees block travel in that direction.

>in
Trees block travel in that direction.

>down
Trees block travel in that direction.

>up
Trees block travel in that direction.
```

Alternatively, a new property has been defined "cant_go_odd" which will be used instead with these directions. A default library message has also been created for generic cant_go responses in these odd directions. (Go #13)

ORDipensor

This is an object that is designed to dispense seemingly endless quantities of a specific object. It is useful for things like a pot of gold coins, where the pot may contain numerous gold coins, but the developer does not want to actually create all coins at once. The class of the items that the derived object dispenses must be defined as the property "itemclass". When the Dispensor object is initialised (automatically), a number of objects of the itemclass type are created. This number is stored in the "initial_count" property. When the items are removed, more are generated to replace them; when the items are returned to the dispensor, they are destroyed.

The following properties are of particular interest when dealing with the ORDispensor class:

- itemclass - Set this property to the class of the array that will be dispensed.
- conceal_dispensing_items - Set this to true if the dispensing items will have the concealed attribute while they are in the object. This is particularly useful when its dispensing item defines the object represented by this class. For example, a pond is only a pond by virtual of the fact that it has water in it. In this case, a description of a pond full of water should show it as empty despite being filled with water, so this property would be set to true.
- accept_alternate_itmes This property is set to disallow insertion of items that are not of the same class as the itemclass.
- Initial_count Set this to the number of instances of itemclass that should initially be visible.
- receive_item_msg outputs the message when an item is inserted into the object.
- cannot_receive_item_msg outputs the message when an item cannot be inserted into the object.

As an example, let us create a candy dish:

```
class candy(40)
  has edible
  with short_name "piece of dry candy"
  , name 'candies//p' 'candy' 'peppermint'
  , description "It is an unwrapped peppermint. "
  , plural "dry candies"
  , before[;
        eat: candy.destroy(self);
             "You eat the piece of candy."
           ]
;
ORDispensor CandyDish "dish" coffeetable
  has open
  with name 'crystal'
  , itemclass candy
  , accept_alternate_items false
  , initial_count 10
  , description "The dish is made of crystal. "
;
```

ORReferByContents

The ORReferByContents object provides a specialized parse_name routine that allows a container to referred to by the objects it contains. For instance, when given a jar of marbles, it is perfectly natural for the player to try to "PUT THE MARBLES ON THE TABLE" when "PUT JAR ON TABLE" is what was intended. This is even more likely for other types of contained substances, such as a bottle of water.

By default, any object that inherits from the ORReferByContents class will be considered by the parser for any verb. This works well for verbs like Drop, Take or Throw. There are also several actions where considering the container may not be the appropriate solution, such as Eat. To cause the container to not inherit its container's names when certain verbs are used, the "ignore_actions" property list is provided.

Alternatively, for developers may find that more verbs are disqualified than are selected. In these cases, the "refer_actions" property list is provided.

Below is a glass bottle which things can be placed in. Note that the ORReferByContents does not automatically imply the container attribute since it could just as easily have been a tray.

```
ORReferByContents glassbottle "glass bottle" coffeetable
  has transparent open container
  with description "It was just a glass bottle, capable of
         containing all kinds of things."
  , ignore_actions ##Eat ##Drink
;
```

ORLiquid

Liquids have been touted as being some of the most troublesome objects to implement. This is primarily because they do not act as regular objects. They follow different rules. Water cannot be held in same manner as a book, so rules like take and drop do not apply to liquids. To put liquid on the floor is to have seep into the carpet and effectively disappear. Different verbs apply to liquids such as pour. A quantity of liquid can almost always be divided in half. There are also non-liquid substances that follow some of the same basic rules as liquid, such as sand.

Perhaps one of the most troublesome features of a liquid substance is the fact that they are always treated neither as singular nor plural. You do not say that you have "a water" as you would have "a coin." Neither do you say that the "water are in the bowl" as you would say that the "coins are." For non liquid substances that can be handled in collections (like coins), there is both a plural and a singular form (e.g. "coin" and "coins"). This is not the case with liquid like substances. Although there may be several water objects defined in a game, to refer to each object alone requires a form of measurement that plurality and singularity can be applied to, such as "ounce(s)" or "handful(s)" or "measure(s)".

The ORLiquid module was designed to ease the difficulty in implementing liquids in a game. Two classes are defined in the module: ORLiquid and ORLiquidSource which is an implementation of ORDispensor. Additionally, several liquid specific verbs have been defined, like scoop and pour and a few others have been extended.

Two routines have been created to determine if an object can contain liquids of various types: CanContainDryLiquid(), and CanContainWetLiquid().

Also some attributes have been created to help define the world model:

- wetliquid - to be applied to ORLiquid objects if they are to indeed act as liquid as opposed to the so called dry liquids, such as sand or dirt.
- wet - given to objects that have been dipped in wet liquids.
- water_tight given to containers so signify that they can, indeed contain wet liquids.

Additionally, because liquids can not be poured into just any container, a property called "liquid_measures" must be provided by any container that can contain liquid. This property defines what quantity, or how many measures, of liquid can be contained in the object. Note that this property can also be defined for other objects that are not containers as well. These objects, when placed within a liquid carrying container, reduce the amount of space available for liquid. For example, a glass with a liquid_measures property of five, that contains three ice cubes, each with a liquid_measures of one, will have room for two measures of liquid.
For our tutorial, let's take the bottle that we created and make it capable of holding five measures of water:

```
MyObject glassbottle "glass bottle" coffeetable
  class ORReferByContents
  has transparent open container water_tight
  with descrip "It was just a glass bottle, capable of
        containing all kinds of things."
  , ignore_actions ##Eat ##Drink
  , liquid_measures 5
;
```

Now we can define two liquid substances, one dry, and the other wet:

```
class sand(20)
  class ORLiquid
  with short_name "sand"
  , plural "handfuls of sand"
;
class water(20)
  class ORLiquid
  has wetliquid
  with short_name "water"
  , plural "ounces of water"
;
```

The sand and water will be dispensed by the beach and the ocean respectively. These two are placed in the most reasonable location, the beach:

```
ORLiquidSource beachsand "beach" beach
  has supporter ~container
  with itemclass sand
  , conceal_dispensing_items true
;
ORLiquidSource ocean "ocean" beach
  with itemclass water
  , conceal_dispensing_items true
;
```

As an added touch we can create a before rule to the beach location which will translate drop actions into puton actons, so that things dropped end up in the sand. Here is the beach location again in its entirety:

```
MyRoom Beach "Beach"
  with s_to darkdoor
  , w_to glassdoor
  , e_to "No, I didn't have time to go for a swim."
  , n_to "The trees blocked completely any travel in that
        direction. No clues would be found there."
  , remote_description "Through the door I could see a
        beautiful beach with water lapping up on the ocean
        shore line."
  , descrip "Beautiful and peaceful. My shoes were half
        buried in the sand. Water lapped continuously upon
        the shore merely an arm's reach away. "
  , firsttime "The scenery rendered my thoughts of
        house murder all but forgotten. "
  , before[;
  drop: <<PutOn noun beachsand>>;
```

```
]
;
```

Note how the beachsand object and the beach room share the same name. Since one is a room, and is only referenced by the specialized scope rules defined in the ORLookRoom module, these will not conflict.

ORDefer2ndReference

The example from the previous section acts a little unpredictably. As it stands, we have an object that inherits from `ORReferByContents` (the glass bottle) which, can be referred to by its contents.

This allows for some strange command configurations that confuse the parser, but still make sense to people. For example, fill the glass bottle with sand and then type the command: POUR BOTTLE INTO SAND. This will result in the parser believing that the word SAND refers to the bottle and interpret that as POUR BOTTLE INTO BOTTLE. Sure, we could add ##Pour to our bottle's `ignore_actions` property, but that would prevent us from being able to type POUR SAND INTO WATER.

What we really need is a way of skipping consideration of the bottle if it was already the first noun. This is done via the ORDefer2ndReference module. In effect, when the parser is choosing an object for the variable `second`, it will defer to a different object rather that encourage the same object to be referred to twice.

No code is needed for this module. Just include it for our example and pour sand into sand all day long.

ORSuppressTakeAll

There's been a fair amount of attention paid to the suppression of TAKE ALL in games. The DM4 discusses a technique for suppression of scenery objects using TAKE ALL in section 33. Roger Firth demonstrates disabling TAKE ALL completely at the following URL:

http://www.firthworks.com/roger/informfaq/aa.html#7

This module implements a similar implementation. Simply include this module and DM4's method of ignoring scenery objects from take all will be in effect. However, if the developer would rather suppress TAKE ALL entirely, then this behavior can be turned on by setting the `take_completely` property of the SuppressTakeAll object (probably best done in `Initialise`):

```
SuppressTakeAll.take_completely=true;
```

Note that this technique differs from Roger Firth's in that it still allows for qualified lists. For example, "TAKE ALL APPLES" will still function as expected.

Because TAKE ALL can take the form of the REMOVE command (e.g.: TAKE ALL FROM BOX),this command can also be suppressed in the same manner as TAKE. Similar to the "take_completely" propery, so can the the unqualified REMOVE ALL command be completely suppressed with the "remove_completely" property:

```
SuppressTakeAll.remove_completely=true;
```

Again, qualified REMOVEs will continue to work (e.g.: REMOVE ALL MARBLES FROM BOX).

ORBetterChoice

ORBetterChoice uses the ORLibrary's extensible entry point functionality (OREntryPoints) to implement a ChooseObjects routine. This routine simply adds a little more intelligence to the Parser's ability to decide between objects. For example, with ORBetterChoice referenced, the parser will give less priority to objects that are already carried during a TAKE action. Likewise, edible objects will be preferred to non-edible ones during the EAT command.

ORExits

The `ORExits` module implements the typical EXITS verb as well as a standalone routine which does the same thing (`DescribeExits`). This functionality actually walks through and tests each of the `Compass` directions, finally generating the appropriate text when all directions have been determined.

There are times that a developer may want suppress or change this functionality for specific rooms (perhaps choosing to not describe a hidden exit). The default implementation can be overridden on a room-by-room basis, by defining the property `exits_text` on the room as either a string or a routine.

ORMenu

This is a full screen menu help system (that works for both Z Code and GLULX).

The following is an example of how a menu system might be implemented:

```
ORMenu toplevel_m "Help System";
ORMenu -> "About this game"
  with text "In this game, you play a...^^^...Feel free to send
             any bug reports or thoughts on the game to:
             ^^Jim@@64OnyxRing.com^^I hope you enjoy the
             game.^^^-Jim Fisher";
ORMenu -> "Author's Disclaimer" with text "I've
             never...^^^...It's only a story.";
ORMenu -> "Hints";
ORMenu ->-> "Avoiding the White Demon"
  with text[; print (italics)"~The white demon keeps coming
              into my castle and grabbing me!  How do I avoid him?~",
              "^^You ...^^^...best spent examining the world around
              you.";];
ORMenu ->-> "Who is this Bauefred, guy?"
              with text [; print (italics)"~This Baufred fellow
              keeps...","^^...later in the game.";];
ORMenu -> "About NPCs" with text "Blah, blah, blah....
              Put NPC Text here.";
ORMenu -> "Special Commands" with text "There are various
              commands that are atypical to other IF games, or at least
              are infrequent enough to warrant mentioning.  These can be
              divided into two groups:";
ORMenu ->-> "Commands specific to this game";
ORMenu ->->-> "Mamberflop"
  with text[; print "Mamberflop is a spell which...";]
;
ORMenu ->->-> "Ducalolly" with text[; print "The
             Ducalolly spell can...";]
;
ORMenu ->-> "Commands derived from the ORLibrary"
  with text "The ORLibrary offers several modules which add new verbs
             for the player. "
;
ORMenu ->->-> "Name"
  with text "The player has the ability to ~name~ an object and then
             refer to it by that name with a command such as:^^
             >NAME ORDERLY ~BOB~ ^^or^^ >REFER TO THE KNIFE AS A SWORD"
;
ORMenu ->->-> "Talk"
  with text "A generic addition to the ASK/Tell/Paradigm, the player has
             the ability to ~talk~ to another NPC...";
ORMenu -> "About the ~ORLibrary~"
  with text "The ORLibrary, or the ~OnyxRing~ Library is a framework of
             extensions to the standard library.  It contains...^^
             ...in finding out more about the ORLibrary framework,
             point your browser to: www.OnyxRing.com."
;
ORMenu -> "About the Author"
  with text "Jim Fisher, a software engineer by profession for close
             to a decade, has programmed in a dozen or more languages
             including Assembler, C/C++, C#, Delphi, Java, LISP, Perl,
             and SQL.  He has..."
;
```

Calling the menu programmatically can be accomplished with a call to the global routine `DoORMenu()`:

```
DoORMenu(toplevel_m);
```

Note that if a menu object is NOT specified to DoORMenu, then the routine checks the variable ORMenuContext for a menu object. Changing this variable basied upon game state allows context sensitive menus to be implemented. If this variable does not define an ORMenu instance then the routine attempts to "guess" and find a top level menu object on its own.

ORHideDirectionWalls

One of the most experience-breaking commands in an Inform game is X WALL. Note the affect in the below transcript:

```
You are outside the house.

>X WALL
Which do you mean, the north wall, the south wall, the east
wall, the west wall, the northeast wall, the northwest wall,
the southeast wall or the southwest wall?
```

These walls, defined in the `Compass` object, are needed to support commands like GO and make the directions referable. They do, however, shatter mimesis. By simply including this module, these " wall" objects become no longer referable by the word "WALL". Note in the following transcript how this does not completely satisfy the problem presented by these directions:

```
You are outside the house.

>X WALL
You do not see anything like that.

>X NORTH
There is nothing special about the north wall

>X SOUTH
There is nothing special about the south wall

>X EAST
There is nothing special about the east wall
```

Including the `ORLookRoom` module will provides a mechanism for looking in specific directions and eliminates this remaining detail.

ORMiniMenu

A menu system (that works for both Z Code and GLULX). This menu does not clear the screen so is ideal for CYOA style decisions that affect game play. It additionally allows for a parent/child relationship between menu items which enables the user to traverse a tree of options to locate a selection.

The following is an example of how a menu system might be implemented:

```
ORMiniMenu toplevel_m "Choose from the following";
ORMiniMenu -> "Do stuff with sticks.";
ORMiniMenu ->-> takesticks "Pick up sticks.";
ORMiniMenu ->-> burnsticks "Set sticks on fire.";
ORMiniMenu ->-> kicksticks "Kick sticks.";
ORMiniMenu -> "Look at stuff";
ORMiniMenu ->-> ash "Look closely at the fire ash.";
ORMiniMenu ->-> river "Look in the river.";
ORMiniMenu ->-> hole "Look into the hole.";
ORMiniMenu ->-> fish "Look at the fish.";
ORMiniMenu ->-> dog "Look around the tree.";
```

Calling the menu programmatically can be accomplished with a call to the "show" property which also returns the user's selection (if any):

```
result=toplevel_m.show();
```

The result is the actual menuobject. It is determined when a menu option that has no children is chosen. The user's "choice" is also stored in the result parameter of the parent menu item (that is, the menu item actually called):

```
print (name)toplevel_m.result;
```

The above code will print the user's selection (provided there was a selection. A zero result indicates no user selection (the user backed out of the menu rather than make a choice). Obviously the result should be checked for a zero value before attempting to use it.

The `menupos` property of the called menu object will determine the position of the menu at either the top of the screen or the bottom of the screen. The following are valid values for this property:

```
    ORMENU_BOTTOM
```

```
ORMENU_TOP
```

Note that `ORMenu_Bottom` does not work on all interpreters. Generally it works on Frotz-based and GLK-based interpreters, but not Zip-based.

ORNameable

To create a nameable object, simple derive it from ORNameable:

```
object golem "Golem"
  has   animate
  class ORNameable
  with description "The creature looks dirty and pathetic."
  ,   name 'golem'
;
```

And during game play you can name the golem with a command similar to:

```
>name golem "bob"
```

ORPathMaker

The ORPathMaker object will determine a path between two rooms. Two starting objects are taken by the `determine_path` routine. These can be rooms or objects or characters. Passage through doors is resolved, however if the doors require keys, then the starting object (first passed in) must be an object that contains the needed keys. This allows an NPC with the appropriate key to successfully take a path that an NPC without the appropriate key could not.

Sample use:

```
ORPathMaker.determine_path(player,bathroom);
ORPathMaker.determine_path(bathroom,bedroom);
```

The return values for this routine are:

-1 : There is no connecting path between the two given objects.
-2 : Ran out of workspace while trying to calculate path. Define the constant PATHDEPTH with
value.
-3 : Although there WAS a path correctly determined, there was NOT enough space in the path
room obj : The first connected room that must be traveled to in order to finally reach the destination.

Additionally, once the path has been correctly determined, the size property contains the size and the path property contains the final path.

There is a limited amount of scratch workspace used to determine a correct path. If there is not enough space, the path will not be findable. The constant `PATHDEPTH` is used to allocate this scratch space. By default it is set to 30, however it can be defined at a greater value in the main program's source.

ORPrefixSuffix

When creating an object with a period in the name -- like "Mrs. Robinson" -- the player often has issues. For instance...

```
>EXAMINE MRS. ROBINSON
```

...will cause the parser to stop parsing the input at the period and treat it as two separate commands:

```
>EXAMINE MRS
```

and

```
>ROBINSON
```

Thus, assuming there is only one object which the word "Mrs" can refer to, you will get the description followed by the "That's not a verb I recognize" error message. The parser fails entirely if the prefix can refer to multiple objects ("Mrs. Robinson" standing next to "Mrs. Baker").

This module scans the input and removes periods that follow general prefixes, such as "Mr" "Mrs" "Dr" "Col", etc...

ORReverseDirection

This is a short, simple routine, not at all worthy of its own module but used by multiple library entries and so is placed in its own module for the sake of sharing.

ORReverseDirection simply takes a direction property or a direction object as a parameter (such as `nw_to`) and returns the reverse direction (in this case, `se_to`). It is used by several ORLibrary modules, such as ORNPCVerb, ORNPC_movement, and ORDynaMap.

For clarity, it should be reiterated that the ORReverseDirection routine also resolves direction objects such that passing in `s_obj` as a parameter will yield `n_obj` as a result.

ORSpecializedExit

This module extends the standard behavior for `exit` so as to allow a noun. A peculiarity of the standard library is that it simply doesn't like a noun specified with the `exit` verb. It is natural for a player who has just issued the command: GET ON TABLE or try to: GET OFF TABLE. Strangely, without modification, the standard library only supports EXIT, or GET OFF. With this module, commands such as...

```
>EXIT BALLOON
```

or

```
>GET OFF TABLE
```

or

```
>GET DOWN FROM THE CABINET
```

...or many other variations will work.

ORStartingTurn

This module was created after a review of Matthew Auger's submitted bug report for standard library (Issue L61011 : "Move count starts at 1") which can be reviewed at:

http://www.inform-fiction.org/patches/L61011.html

This module implements Matthew's solution but also leverages a constant `STARTING_TURN` which is defaulted to 0 if not defined. (Why would you want to make it anything else? Who knows...)

ORAdjective

The DM4 Exercise 75 gives an example of a ParseNoun routine which distinguishes between nouns and adjectives. This module implements a modified version of that routine, but leverages the ORParseNoun modules so that other ParseNoun-like routines can co-exist without conflict. Additionally, this version of ParseNoun will also work with the ORRecogName module.

The usefulness of adjectives is apparent when dealing with similar objects. Consider the following three items:

```
object -> glass "glass" with name 'glass';
object -> marble "glass marble" with name 'glass' 'marble';
object -> marble_table "marble table" with name 'marble' 'table';
```

Notice how the parser is unable to distinguish the player's obvious intentions when referencing the marble:

```
You can see a glass, a glass marble and a marble table here.

>get marble
Which do you mean, the glass marble or the marble table?
```

In order to distinguish between the marble and the table, the adjective must be specified. It becomes even worse when trying to refer to the glass. Since it has no additional adjectives, it cannot be referenced at all:

```
>get glass
Which do you mean, the glass or the glass marble?
```

In order to make the glass uniquely referable, the designer must manufacture another word (perhaps 'drinking'). The introduction of adjectives when including the ORAdjectives module, is accomplished through the `adjective` property. The following code snip demonstrates:

```
object -> glass "glass" with name 'glass';
object -> marble "glass marble"
   with name 'marble'
   ,   adjective 'glass' 'red'
;
object -> marble_table "marble table"
   with name 'marble' 'table'
   ,   adjective 'marble'
```

```
;
```

Adjectives will now be used solely to distinguish between similar objects (say, a wooden marble and a glass marble) and will no longer confuse objects with similar adjectives:

```
>get marble
Taken.

>get glass
Taken.
```

As the example now sits, notice how an adjective cannot be used alone to refer to an object, but can be used in conjunction with a noun to disambiguate:

```
>get red
You cannot see any such thing.

>get red marble
Taken.
```

Although this is arguably more correct, many players tend to frown upon this. A balance between the DM4's Exercise 75 and the standard library's default behavior can be achieved by adding the following line to the `initialize()` routine:

```
ORAdjective.mode=PREFER_NOUN;
```

Unlike the default mode of `REQUIRE_NOUN`, this mode allows objects to be referred to by adjectives alone, but always prefers objects that have been referenced by a noun (`name`) value.

ORMagic

Of particular use in stories of the fantasy genre, the ORMagic module implements a system for casting magic spells. The most frequently used features of ORMagic will be covered in this section; however there are additional features, such as "persistent" spells that continue their effect long after their initial casting, which we will avoid detailing in favor of simplicity.

1. The SPELLS and CAST Verbs

By including the ORMagic module -- that is, by defining the USE_ORMagic constant -- the SPELLS verb and the CAST verb are immediately available. Of course, not having defined any spells makes them fairly useless, but they are there regardless:

```
Cave
This is a small, empty cave with no exits. You have no idea
how you got in here in the first place.

You can see a rock here.

>SPELLS
You have no magic at your disposal.

>CAST MURM
I am uncertain what you want to cast.
```

2. Creating a Spell

The actual creation of a spell is a relatively painless experience. Below is an example spell called "MURM" which does little more than assign an attribute:

```
attribute murmuring;
ORMagic Murm "Murm"
  with name 'murm'
  ,    knownby selfobj
  ,    cast[;
          if(second==0)
               "The MURM spell must be cast upon something.";
          if(second has murmuring)
               print_ret (The)second," is already murmuring.";
          give second murmuring;
          print_ret "You cast the spell upon ",(the)second,
               ". It begins to spew forth a continuous
               muffled murmuring noise.";
];
```

There are two properties of particular noteworthiness. The first is the `knownby`property which lists all characters that know and are able to cast the spell -- in this case, only the player. The second is the `cast` routine which is called when a character uses the CAST verb. Notice that it is the responsibility of the spell itself to determine if casting is possible and to ensure that the object of the spell has been specified. See how the presence of the spell

changes the previous transcript:

```
>SPELLS
You have the following spells at your disposal:
    Murm

>CAST MURM
The MURM spell must be cast upon something.

>CAST MURM AT ROCK
You cast the spell upon the rock. It begins to spew
forth a continuous muffled murmuring noise.
```

3. Additional Notes

See that our SPELLS verb listed the spell by its name. A description of what the spell does will also be displayed if it is defined in the generic `text`property:

```
ORMagic Murm "Murm"
  with name 'murm'
  ,  text "Cause objects to murmur."
  ...
```

This makes the result of the SPELLS command a little more informative:

```
>SPELLS
You have the following spells at your disposal:
   Murm (Cause objects to murmur.)
```

As a final note, remember that the CAST verb is really optional. The name of our newly crafted spell can be used as though it were a verb:

```
>MURM ROCK
The rock is already murmuring.
```

ORWAE_Formatting

The standard library's `WriteAfterEntry` routine interrogates objects and prints qualifying text such as "(providing light)" or "(closed, empty and providing light)". The `WriteAfterEntry` routine is anything but straightforward and trying to fathom the rules that govern these snippets of qualifying text can be a tedious task. What follows is a brief and simplistic overview:

Without modification, the standard library uses a total of 17 library messages to qualify various combinations of six object states. Three of these object states are described only when an object is part of the player's inventory, and not when it is depicted in a room description. The other three are described in either case. The following table details these:

Message	When Is It Displayed?
worn	Inventory only
open	Inventory only
locked	Inventory only
providing light	Inventory or Room description
closed	Inventory or Room description
empty	Inventory or Room description

Since the messages are implemented individually, it takes seventeen to cover the possible combinations, even given that some are mutually exclusive. Adding additional states is cumbersome and often means adding four or five additional messages, not to mention hacking the library to test for the appropriate settings.

This module implements an extensible framework to ease the burden of defining new qualifiers as well as redefining or removing the existing qualifiers.

1. ORWAE_FullFormatting and ORWAE_PartFormatting

Two routine lists (`ORRoutineList`) have been created to manage the full and partial listings (that is, the inventory and room description listings) of game objects. It is the routines that are registered with these lists that determine if an object meets the required criteria and prints the appropriate text if so. The following example creates two new qualifiers. One (sticky) will only be listed when part of the player's inventory while the other (muddy) will be described regardless:

```
attribute sticky;
attribute muddy;

[Initialise;
    ...
    !--add muddy to inventory description
    ORWAE_FullFormatting.add_routine(wae_muddy);
```

```
    !--add muddy to unheld description
    ORWAE_PartFormatting.add_routine(wae_muddy);
    !--add sticky to inventory description only
    ORWAE_FullFormatting.add_routine(wae_sticky);
];
[wae_muddy obj suppress;
    if(obj has muddy && suppress==false) print "covered in mud";
    if(obj has muddy ) rtrue; rfalse;
];
[wae_sticky obj suppress;
    if(obj has sticky && suppress==false) print "sticky";
    if(obj has sticky) rtrue; rfalse;
];
```

Now for the sake of example, let's create a couple of objects that have attributes which will be described in parentheses:

```
object box "box" cave
  has openable ~open container lockable locked muddy
  with description "Just a box."
  ,   name 'box'
;
object gumball "gumball" cave
  has sticky
  with description "A gumball, recently chewed."
  ,   name 'gumball' 'gum' 'ball'
;
```

Note in the following transcript, how our new qualifiers blend seamlessly with the traditional ones:

```
Cave
This is a small, empty cave with no exits. You have no idea
how you got in here in the first place.

You can see a box (closed and covered in mud) and a gumball here.

>TAKE ALL
box: Taken.
gumball: Taken.

>I
You are carrying:
  a gumball (which is sticky)
  a box (closed, locked and covered in mud)
```

2. Routine specifics

The qualifier-printing routines that we created above (`wae_sticky` and `wae_muddy`) take two parameters. The first is the object to be tested; the second is a flag which determines whether or not the text should be suppressed. In either case, the routine should always return `true` or `false` to indicate that the object either does or does not meet the necessary criteria for printing the text.

Six routines have also been created and automatically registered with the appropriate routine lists to handle the traditional library defined qualifiers. These are listed below. Remember that the first three have only been registered with the `ORWAE_FullFormatting` object by default.

_WAE_worn
_WAE_opencontainer
_WAE_lockedcontainer
_WAE_light
_WAE_closedcontainer
_WAE_emptycontainer

These routines can optionally be removed by using the `ORRoutineList` object's remove_routine method. For instance, the "(which is empty)" qualifiers can be eliminated from a game by adding the following lines to the initialise routine:

```
ORWAE_FullFormatting.remove_routine(_wae_emptycontainer);
ORWAE_PartFormatting.remove_routine(_wae_emptycontainer);
```

3. Freed Messages

Recall that the standard library defines 17 library messages to handle the possible combinations of the six different qualifiers. Since the combination of these qualifiers is handled programmatically within ORWAE_Formatting, messages ListMiscellany #7-17 are no longer used and are free to be redefined.

ORNumberedContainer

This is an object that is designed to emulate a collection of openable, stationary containers (like lockers).

The following properties are of particular interest:

- singular_name the name of a singular compartment (e.g. "locker").
- start_numb the minimum number assigned to the compartments.
- end_numb the maximum number assigned to the compartments.
- numbered_description this routine will print pertinent information about which compartments are open and what contents are available. It is useful to call this routine after the description.
- contained_obj & contained_in - these property lists hold the contents of each locker. There should be enough elements in these arrays to hold all potential contents. Both of these arrays should be the same size as there is a one to one relationship between the items in these lists.
- open_state - a list of the compartments that are currently open. If more compartments are provide than there is room in this list, then the potential exists for the open to fail. In this case the property "cannot_open_more_msg" is run which outputs an appropriate message.
- cannot_open_more_msg output when the player has tried to open more compartments than there is room for.
- add_item_to call to add an object to a specific container.
- remove_item call to remove an object.

As a note, the display name, or short_name as may be the case, is displayed when referencing the object as a whole and should be plural in form (e.g. "lockers"). The singular_name property is used when referring to a single construct so should be singular in form.
Also, although it is fine to redefine the array properties of this object, it might prove to be easier to define an array table and assign it upon initialization.

For instance:

```
array lockerstate table 1000;

[Initialise;
  numcont.open_state=lockerstate; !----allow 1000 lockers
                                  !---to be open at once.
];
```

For our tutorial, lets create this as an eccentric object in the hall. First, we'll add a description of it to the hall:

```
MyRoom Hall "Hall"
  with w_to foyer
  , e_to staircase
  , firsttime "A flag when off in my head as I stepped
         into the hall. Something was strange here...
         but what? I considered it for a moment before
         it hit me. The tiling on the southern wall was
         something other than it first appeared. In fact,
         as I looked closely, each tile appeared to be
         a door to a small cubby hole. Like little
         mailboxes... ^^"
  , descrip "This was a long hallway connecting the
         western entryway with the small landing at the
         base of the staircase to the east. The southern
         wall was lined with numerous tiny cubby doors,
         hidden in the tiling. "
;
```

Now we can actually create the object in the hall:

```
MyObject tilecubby "tile cubbies" hall
  class ORNumberedContainer
  with name 'cubby' 'door' 'doors'
  , singular_name "cubby"
  , end_numb 9999
  , descrip[;
         print "Almost invisible, the tiles along the southern
              wall were actually tiny doors to little cubby
              spaces with numbers etched into them. There were
              thousands. ";
         self.numbered_description();
         print " ";
  ]
  , firsttimepost [;
         print "I looked at the tiled cubbies
              in wonder. ",(italics)"What kind of eccentric
              people would do something like this? ";
  ]
;
```

For a final bit of fun, let's create a revolver...

```
Global gun_position;
MyObject gun "revolver"
  with name 'gun'
  , descrip "The gun was sleek and heavy; powerful and deadly. "
  , firsttimepost "I fell immediately in love with the gun,
         having a liking for that sort of thing. "
;
```

...and hide it in a random cubby. The initialise() routine will serve us well for that:

```
gun_position=random(tilecubby.end_numb);
tilecubby.add_item_to(gun,gun_position);
```

Part D
NPC Oriented Modules

Introduction to ORLibrary NPCs

Non-player characters (NPCs) represent, potentially, the most complex objects in a work of interactive fiction. Since they typically represent other people, NPCs should, in theory, be able to do all of the things that the player character can do. They might be able to hold a conversation with another character. They might be able to perform a chain of actions such as finding a key and then unlocking a door. Massive amounts of code have been written to shape NPCs in this fashion.

Inform's standard library supports NPCs to a limited degree. It defines the `animate` attribute to indicate something is alive, and leverages this attribute to modify the behavior of player commands. In this way the player can try to TALK to an animate NPCs, but not a book. Likewise, trying to TAKE an NPC will generate an appropriate error message. Support for NPCs is further supplied by the `life` and `orders` properties which provide hooks for the developer to code reactions to player commands.

Conspicuously missing from the standard library's support of NPCs is the ability to do anything. Creating an NPC that can initiate an action requires designing a daemon. Actions are also strangely player-oriented. For example, the TAKE command allows only the player to take things. For an NPC to pick up an item, the designer must actually code a routine that moves an item into the NPC's inventory. This is complicated by determining scope for the NPC and taking into account the possibility of the item having the static or scenery attributes. Perhaps the item is within a closed, transparent case so that it is visible, but untouchable. What if an object's before rule prevents the take from succeeding? Even were all of these checks in place for NPCs, the standard library's default messages for actions are all coded to assume the player character is performing the action.

The ORLibrary's support for NPCs addresses all of these inefficiencies in the standard library. All standard action routines have been revised to support NPCs and a method has been put into place to cause an NPC to perform actions, even new verbs designed by the developer. The default messages have been revised as well. Additionally, the ORLibrary's NPC support provides a framework which can be leveraged to support goal-oriented NPCs that act of their own accord.

Conversations between NPCs and the player character as well as conversations between just NPCs are supported. Enhancements to the Ask/Tell model of conversation and an implementation of conversation menus is also in the ORLibrary. Entire topic webs can be created where a conversation with an NPC flows naturally from one subject to another and branches in various directions.

Because not every NPC needs to do all of these things, NPC abilities have been packaged as opt-in components. This keeps a small footprint for simple NPCs and allows more complex NPCs to be granted additional abilities with a single addition to the "class" line. The complexity of an NPC can therefore be limited and more easily managed.

What follows is a guide to leveraging the various ORLibrary modules to create powerful NPCs. This includes the ORNPC component modules as well as the ORKnowledgeTopic modules. It should be noted that the internals of the NPC and Knowledge modules are extremely complex and these articles will not detail their every nuance. They will, however, cover the most-used features and give examples geared towards getting developers up and running quickly.

ORNPC

As was mentioned previously, the standard library implements only minimal NPC support. Nevertheless, this support is the required foundation upon which more significant NPCs can be based. ORNPC builds upon this foundation to provide a more versatile and sophisticated base from which to construct NPCs.

Below is an example butler derived from the ORNPC class:

```
object -> butler "butler"
   class ORNPC
   with name 'man'
   , description "The butler ignores you, preoccupied with other things."
;
```

A little experimentation will show that the butler acts no differently than a bare-bones object with the standard library's `animate` attribute. So what is the use of the ORNPC class? The power behind ORNPC is not what it makes NPCs do, but rather the hooks it provides to enable NPCs to act on their own...

1. ORNPCControl

The core of NPC activity is powered by the ORNPCControl object. This object wraps the daemon which imbues life into an NPC. Each turn, the daemon calls an additive property routine called `heartbeat` which is defined in every ORNPC object. After `heartbeat` returns, a last chance to do something is provided by calling the `heartbeat_post` property if it is defined. The order in which NPCs act is determined by a property called `priority`. Lower priority NPCs act first.

Because all ORNPC objects are raised to life in this fashion, by a single daemon, it is an easy matter to halt the actions of all NPCs at once. A property in the ORNPCControl object named `pause` can be set to true to accomplish this. Likewise, it can be set to false to restart all NPC activity. This daemon is started automatically, and the call to `StartDaemon()` never actually needs to be implemented by the developer. Since starting and stopping all NPCs is not commonly done, the ORNPCControl object is often not referenced at all; however there are a few occasions, such as the middle of a verb command, that it is convient to tell whether an NPC is acting of its own accord or following an order. During the NPC action phase, the `ORNPCControl` property `npcs_acting` is set to true.

It should be noted that the daemon does more than simply call the "heartbeat" routine. It also takes some steps to ensure that the world is as it should be for the NPC, just as it is for the player. This is accomplished by taking steps such as making the NPC the current actor and calling a replaced version of MoveFloatingObjects().

2. Heartbeat

`Heartbeat` is the entry point for all NPC activity. Leveraging this property gives us a great deal of versatility. Many authors have found that the addition of even very small statements to remind the player of our NPCs presence can have a profound effect and go a long way toward making it appear more lifelike. Dan Schmidt does this with the tool man character in "For a Change" and leaves small, lasting reminders of his presence. Emily Short, also does this in her game "Metamorphosis" despite the fact that the only NPC in the game serves little more purpose than scenery. Sentences such as "The toolman jingles in the breeze" or "The gondolier lifts his head and sighs softly" fire at arbitrary times and add to the illusion of life. We can accomplish this easily with the heartbeat property:

```
object -> butler "butler"
   class ORNPC
   with name 'man'
   , description "The butler ignores you, preoccupied with other things."
   , heartbeat[;
     if(random(5)~=1) rfalse; !--only do this 1 in 5 turns (average)
     if(PlayerCanWitness()==false) rfalse;
     switch(random(6)){
          1: "The butler clears his throat.";
          2: "The butler gives a small sigh.";
          3: "The butler looks around as though searching for something.";
          4: "The butler scratches his chin absent mindedly.";
          5: "The butler hums a tuneless melody.";
          6: "The butler figits.";
     }
     rfalse;
   ]
;
```

As shown above, it is a relatively straightforward matter to extend an NPC using this property, but it is worth mentioning that `heartbeat`, like `life` and `before`, is an additive property. Provided newer versions do not interrupt it (by returning true), the default implementation of `heartbeat` will select a registered action for the NPC, and perform it. We'll cover registered actions in a moment, but before we do lets talk briefly about a routine called above that we've not yet discussed, the CanPlayerWitness() routine.

3. CanPlayerWitness

Unlike the `each_turn` property, the `heartbeat` routine is fired for an NPC regardless of the player's current location. This enables NPCs to act in ways that meet their own agendas even with the player is no where to be seen. It is necessary for this reason to verify that the player is indeed able to see or hear the NPC before text is output. The CanPlayerWitness() routine can be called for this. It returns true or false if the player can witness the actor's actions. Additionally, another object can be passed in if it is not the actor that we are checking (such as a CB radio that the actor is speaking through).

In the above example, the `heartbeat` routine simply exits if the player cannot observe the

actions of the NPC, but care should be taken to ensure that this routine only filters the text output and does not filter out changes the NPC may make to the game world. In short, the PlayerCanWitness() routine should determine if the player is notified when an NPC picks up an object, but not whether the NPC actually does pick up the object.

4. The DoNothing_msg Property

When the NPC has considered all potential actions, it is entirely possible that there is nothing to do. At this point the `DoNothing_msg` property will be run or printed if it is defined.

5. Registered NPC Actions

It was mentioned previously that the default implementation of the additive `heartbeat` property will search for "registered actions" to perform. For the purposes of this discussion, "actions" are essentially member properties that have been registered with a call to the `register_action()` routine.

An action property takes the single parameter `can_perform`. When `can_perform`; is passed in as true, then the NPC is attempting to make a decision about what action to perform. At this point, the action property should check and make sure that the action is possible (such as making sure there is an empty chair nearby if the action is to sit down) and return true if it is. When the `can_perform` parameter is set to `false`, then the action is actually being attempted and the appropriate code should be run.

Since some actions take more than one turn to complete, ORNPC implements a property called `continued_action` which an action routine can set to ensure it will be called again next turn. When continued_action is set to point at an action property, the normal steps in determining which action to take are circumvented. The action being referenced will continued to be called from the `heartbeat` routine until the `can_perform` test returns false.

Alone, the ORNPC class does not implement any actions, but this framework is the basis for many of the NPC component classes. The `ORNPC_movement` and `ORNPC_converse` modules are examples of NPC actions.

6. Component Class Order

Recall that more advanced NPCs can be assembled from the NPC component classes. As you would expect, using a component class means simply referencing it on the NPC's class line; but due to the nature of Inform's implementation of multiple inheritance, the required order in which these component classes appear is unintuitive. For example, ORNPC should always appear last in this list, while component classes are added to the front of the list. To complicate matters, several component classes require placement between two other components.

In order to simplify this, each NPC component has been assigned an NPC Component Priority value (NPComPri) which is listed in the header of each source file. Higher priority components should appear further left in the class list, while lower components should appear further to the right. The below list summarizes the values for common NPC components:

ORNPC	0
ORNPC_AskTellLearn	10
ORNPC_Converse	15
ORNPC_DoVerb	5
ORNPC_Interact	10
ORNPC_MapKnown	15
ORNPC_Moods	5
ORNPC_Movement	10

ORKnowledgeTopic

The ORKnowledgeTopic class is an advanced implementation of a topic-based conversation/information system. It aids in separating information from conversation so that two NPCs could potentially present the same information in two separate ways. It implements a topic memory, so that knowledge can be learned and taught to other NPCs. It records who has said what so that NPCs can respond differently when told or asked about the same information multiple times. Also, ORKnowledgeTopic can be used to create inanimate knowledge sources, such as books that can be consulted. It further implements and unifies the entire ASK/TELL/ANSWER/CONSULT paradigm.

Examples for all ORKnowledgeTopic functionality would be a very large and go beyond this guide's intent for a quick introduction. In the future, expect additional documentation to surface describing more customized implementations of ORKnowledgeTopic. For now, we'll scratch the surface and cover the most common uses for this class.

1. Basic ORKnowledgeTopic usage

To have a conversation, we need to have someone to talk to. Below, find three NPCs all derived from a base class which provides a central location to make changes:

```
class npc_c
   class ORNPC
;
npc_c dragon "dragon" with name 'dragon';
npc_c soldier "soldier" with name 'soldier';
npc_c halfling "halfling" with name 'halfling';
```

Now we can create a simple knowledge topic that we can talk about:

```
ORKnowledgeTopic hello_halfling_t "hello"
    with knownby selfobj
    ,    name 'hello'
    ,    TopicInformation "~Hello, halfling,~ you say.
                  ^^The halfling nods his head shyly."
;
```

We have now completely implemented a basic knowledge topic which we can successfully tell to an NPC:

```
>TELL HALFLING HELLO
"Hello, halfling," you say.
The halfling nods his head shyly.
```

Note that it is the knowledge topics themselves that keep track of who or what "knows"

them. In this example, the player can only say "hello" because he/she is listed in the object's `knownby` property list. It is entirely possible for two or more characters to "know" the same topic.

It should be mentioned here that knowledge topics can also have parents which are knowledge topics. When this is the case, a knowledge topic is also known by the same characters that know its parent. This enables "knowledge groupings" and offers a more flexible way to manage knowledge bases.

2. Common Knowledge

In addition to topic grouping by parent, there is another alternative. An object has been defined with the name CommonKnowledge, and any knowledge topics placed as children of the CommonKnowledge object are automatically known by every character in the game. That is, being a child topic of the CommonKnowledge object is equivalent to having the player and every NPC in the game listed in the KnownBy list.

3. Context & IsInContext

The hello_halfling_t topic given in the previous example has a significant drawback. It is completely possible to speak this topic to the wrong character:

```
>TELL DRAGON HELLO
"Hello, halfling," you say.

The halfling nods his head shyly.
```

Obviously the text is inappropriate. One possible remedy to this behavior would be to turn the `TopicInformation` property into a routine and modify the text based upon who is being addressed, however this is not a real possibility for most topics.

The `IsInContext` routine is used to determine whether or not a topic can be told to a given character. The default implementation checks to see if the character in question is listed in the context property. If so, then the routine returns true. Additionally, if the context property list is empty, then it is assumed that the topic is in context for everyone and true is returned again. For most topics, this is all that is needed, but it is possible to override `IsInContext` to check other criteria, like gender for example. For our current example, the default implementation works just fine and we can specify the context property.

```
ORKnowledgeTopic hello_halfling_t "hello"
   with knownby selfobj
   ,    name 'hello'
   ,    TopicInformation "~Hello, halfling,~ you say.
                ^^The halfling nods his head shyly."
   ,    context halfling
```

```
;
```

The `context` property can also be used to resolve ambiguity. Consider the following:

```
ORKnowledgeTopic hello_dragon_t "hello"
   with knownby selfobj
   ,    name 'hello'
   ,    TopicInformation "~Hello, Dragon,~ you say.
                ^^The blows smoke in your face in
                acknowledgment."
   ,    context dragon
;
ORKnowledgeTopic hello_soldier_t "hello"
   with knownby selfobj
   ,    name 'hello'
   ,    TopicInformation "~Hello, Soldier,~ you say.
                ~Greetings!~ the soldier replies."
   ,    context soldier
;
```

Without limiting the context, the parser would ask for clarification and the developer would be forced to supply additional words in the name property for the player to specify.

4. Asking for Information

Although TELLing an character about something is good functionality to have, ASKing an NPC about something is a far more useful thing to do in a story. The topic's query property, if provided at all, determines how a question would be phrased to solicit the topic's information.

Consider the following new topic known by the halfling:

```
ORKnowledgeTopic "dragon"
  with knownby halfling
  , query "~So, do you think you could out run this
                dragon?~ you ask."
  , TopicInformation "The halfling eyes the dragon for a
                moment. ~My uncle was eaten by a dragon,~
                he says. ~Were the dragon to try and chase
                me, I would probably meet the same fate.~"
;
```

Which enables the following transcript snippet:

```
>ask halfling about dragon
"So, do you think you could out run this dragon?" you ask.


There was no reply.
```

As was stated previously, the lone ORNPC class does little beyond the minimal

functionality of the standard library. The three example NPCs given above cannot do anything on their own. They can be talked to, but cannot really answer the questions to which they know the answers. This ability can be added to a character with the `ORNPC_AskTellLearn` component module.

Wary reader may notice the absence of the context property in the above example. To provide clarity as to why it is not needed here, it may be worth the effort to restate what is likely obvious: While the knownby property defines who *knows* the topic, the context property defines who the topic can be *told to*. Since we are ASKing about a topic that the halfling knows rather than TELLing something that the player character knows, limiting the context is not necessary. It *will be*, however, when the `ORNPC_Converse` component makes our NPCs wily enough to ask questions on their own.

5. ASK/TELL Enhancements

There are a couple of additional enhancements to the standard library's ASK/TELL implementation which should be mentioned now.

The first is the implementation of two abbreviations for ASK and TELL. These are A and T respectively. These were first seen (by this author at least) in Emily Short's game "Galatea" and have been commented upon by the IF community with varying degrees of favor.

The second is the implementation of a mechanism to try to determine with whom the player is intending to speak.

These two features make it possible to issue the command:

```
>a dragon
```

...and have it translate into:

```
>ask halfling about dragon
```

ORNPC_AskTellLearn

The `ORNPC_AskTellLearn` module is an NPC component class. That is, it is a class which encapsulates a specific behavior that can be used in conjunction with other classes to piece together an effective NPC. As we discussed in previous sections, basic NPCs do not come with the ability to relate knowledge topics, even though they may "know" the information. Adding the `ORNPC_AskTellLearn` class to an NPC's class list rectifies this and makes speaking to the NPC a more natural experience. In particular, the NPC will now be able to answer questions when asked and respond when told something. This is what was termed the "Librarian" model of conversation inthe theory article NPC Conversations: Ask/Tell Theory.

As is the case with several of the NPC and KnowledgeTopic classes, `ORNPC_AskTellLearn` contains a substantial amount of functionality that will not be documented here. Instead, we will cover the most commonly used aspects of this NPC component. To begin, recall from the previous section that the halfling was unable to respond when asked about the dragon. Simply adding this component to the front of the class list of the npc_c class will change this:

```
class npc_c
   class ORNPC_AskTellLearn ORNPC
;
```

In addition to the KnowledgeTopic provided in the previous section, we'll go ahead and provide another example for easier reference:

```
ORKnowledgeTopic "dragon"
  with knownby soldier
  , query "~You think you're faster than this hungry dragon?~
                you ask."
  , name 'dragon'
  , TopicInformation "The soldier looks up at the dragon.
                ~No,~ he concedes. Then a broad smile breaks
                out over his face. ~But I don't have to be,~
                he says looking down at his companion. ~I only
                have to be faster than the halfling."
;
```

Observe the new verbal ability of our NPCs:

```
>ASK HALFLING ABOUT DRAGON
"So, do you think you could out run this dragon?" you ask.


The halfling eyes the dragon for a moment. "My uncle was
eaten by a dragon," he says. "Were the dragon to try and
chase me, I would probably meet the same fate."
```

```
>ASK SOLDIER ABOUT HALFLING
"You think you're faster than this hungry dragon?" you ask.


The soldier looks up at the dragon. "No," he concedes. Then
a broad smile breaks out over his face. "But I don't have
to be," he says looking down at his companion. "I only have
to be faster than the halfling."

>ASK DRAGON ABOUT HALFLING
"I'm afraid I don't know anything about that," the dragon
says.
```

Notice the default message printed when an NPC doesn't know how to answer. The additional property `IDunno_msg` can be defined to override this message for specific NPCs. The below code demonstrates:

```
npc_c "dragon"
   with name 'dragon'
   ,   IDunno_msg "~Geroumph?~ sounds the dragon quizzically."
;
```

ORNPCVerb

The ORNPCVerb module redefines the majority of the standard library's verbs and enables them to be performed by NPCs as well as the player. Like the OREnglish "language definition file," this module exhibits no apparent difference when simply included in the compilation process. In truth, the ORNPCVerb module is part of a triad of modules that work together to bring the NPCs of the game world under the same umbrella of rules that the player is subjected to.

To put the components of this simulation in to a nutshell: ORNPCVerb makes the standard library's verbs NPC compatible. OREnglish makes the default messages NPC compatible. ORNPC_doverb empowers NPC with the ability to issue verb-defined actions.

Since this module is best exemplified when used with the `ORNPC_doverb` module, further covering will continue in that section.

ORNPC_doverb

Like ORNPC_AskTellLearn, ORNPC_doverb is a component class. It leverages the ORNPCVerb module and can be used to give an NPC the ability to perform actions in much the same way as the player does.

In a traditional Inform program, NPC actions must be coded directly. In order for an NPC to pick up an object and eat it, code like the following might be written and called from the NPC's "orders" property when an Eat command has been issued to him:

```
!--example of traditional NPC code
[NPCEat npc obj;
  print (The)npc," eats ",(the)obj,".";
  remove obj;
];
```

The problem with this approach is that it doesn't enforce any of the rules that are enforced for the player character. A slightly better implementation would be:

```
!--example of traditional NPC code
[NPCEat npc obj;
  if(parent(obj)~=npc){
       if(obj has static or scenery) print_ret (The)obj " cannot be
               picked up.";
       print (The)npc" picks up ",(the)obj,".";
       move obj to npc;
  }
  if(obj hasn't edible) print_ret (The)obj " cannot be eaten.";
  print (The)npc," eats ",(the)obj,".";
  remove obj;
];
```

Despite the extended code, this still isn't a complete implementation. No `before` or `after` routines are called to enable, for example, the rules a magic elixir would run. Nor are the `react_before` or `react_after` routines run which may interfere with the actions completely.

The previously discussed module ORNPCVerb modifies the standard library's verbs and a few other routines to enable them to work with the current actor, and therefore an NPC, rather than the player. The previously discussed module OREnglish modifies the default messages to support NPCs performing actions instead of the player. These two modules are requirements for this module, which actually enables NPCs to act.

When derived from the ORNPC_doverb class, an NPC's actions are initiated by the `DoVerb` property routine. For example, the following single line could accomplish what the above code does and still honor any rules like `before` and `after`:

```
npc.DoVerb(##EAT,obj);
```

This module also adds additional support to NPCs for following orders, preventing the developer from having to code an `orders` routine to handle individual verbs with a call to `DoVerb`. The `follow_orders` property can contain a list of verbs that the NPC will follow. Alternatively, setting the `will_follow_all_orders` property to `true` will turn this support on for every verb available to the player. When this option is in use, a list of actions can be defined in the `ignore_orders` property to suppress certain verbs. For verbs that are not handled, the default message is issued, however orders can also be handled and specialized messages issued in the traditional manner via the `orders` property.

Below is a new version of our base NPC class. Note the inclusion of `ORNPC_doverb` on the `class` line to include this component in the class hierarchy. Also notice that `will_follow_all_orders` is turned on and the EXAMINE verb has been listed as an exception to this rule:

```
Class npc_c
  class ORNPC_doverb ORNPC_asktelllearn ORNPC
  with will_follow_all_orders true
  , ignore_orders ##Examine
;
```

In addition, let's add a couple of objects to our game world that the NPCs can interact with (note that the location of these objects is game dependant and therefore left to the reader to define):

```
object table "table" has static supporter with name 'table';
object steak "steak" table has edible with name 'steak';
```

Now we can start bossing people around (except for examining) and see how the game rules bind our NPCs just as they do the player:

```
>dragon, get steak
Taken.

>dragon, eat steak
The dragon eats the steak.

>dragon, get table
The table is fixed in place.

>dragon, get halfling

The dragon does not suppose the halfling would care for that.
```

ORNPC_Converse

A cousin to `ORNPC_AskTellLearn`, `ORNPC_Converse` gives an NPC the ability to initiate conversations. It is an NPC componant which can be used to implement the "Rambling Idiot" form of conversing, described in the article NPC Conversations: Ask/Tell Theory.

1. The TALK verb

A new verb called TALK has been implemented. TALK is very similar to the TELL verb except that it requires no subject be specified. Instead, TALK selects a random topic which is passed off to the TELL command.

Since the ability to TALK was created as a standard verb, it is also usable by the player as well as NPCs. Note that an NPCs ability to use this verb (and others) is provided by the `ORNPC_DoVerb` component which is required by this module.

2. The Initiatable and Repeatable properties

It should be noted that some effort is made to select an appropriate topic. For instance, only topics that the NPC knows about are considered. Additionally, the topic must have an `initiatable` property set to `true` and must be in context for the character being spoken to (see the section on ORKnowledgeTopic for more information about the `IsInContext` property). Finally, either the NPC must not have already talked to the player about the topic, or the topic must supply the `repeatable` property with a `true` value to indicate that this topic can be said multiple times.

To demonstrate this module, we'll continue our example with the halfling, soldier, and dragon by first modifying our `npc_c base class to derived from ORNPC_Converse` as well as `ORNPC_DoVerb` and then creating some sample topics for them to volunteer:

```
class npc_c class ORNPC_Converse ORNPC_DoVerb ORNPC;

...

ORKnowledgeTopic "size"
  with knownby halfling
  , context dragon
  , initiatable true
  , TopicInformation "~It's just my opinion,~ says the halfling to
              the dragon. ~But you don't look as though you would be
              satisfied with a halfling. We have very little meat on us. Now a
              muscle-bound human on the other hand, I would think he would
              make quite a tasty snack.~"
;
ORKnowledgeTopic "catching"
  with knownby halfling
  , context dragon
  , initiatable true
  , TopicInformation "~You look to be the sort of dragon that likes to
              work for his prey,~ the halfling says to the dragon. ~I would think
              catching a well-conditioned soldier would be much more satisfying
              than slow, skinny little halfling like myself.~"
```

```
;
ORKnowledgeTopic  "halfling meat"
  with knownby soldier
  , context dragon
  , initiatable true
  , TopicInformation "~Don't be fooled by our apparent size difference,~ the
              soldier tells the dragon. ~Halfling meat is renowned for its flavor.~"
;
```

Observe how the dialog plays out:

```
>Z
Time passes.

"It's just my opinion, " says the halfling to the dragon.
"But you don't look as though you would be satisfied with a
halfling. We have very little meat on us. Now a muscle-bound human on
the other hand, I would think he would make quite a tasty snack."

"Don't be fooled by our apparent size difference," the soldier
tells the dragon. "Halfling meat is renowned for its flavor."

> Z
Time passes.

"You look to be the sort of dragon that likes to work for his
prey," the halfling says to the dragon. "I would think catching
a well-conditioned soldier would be much more satisfying than slow, skinny
little halfling like myself."
```

3. Prolonged Topics

Support is included with this module for topics that last more than one turn (e.g.: ORKnowledgeScript). When a knowledge topic is selected, it is placed in the `current_subject` property. When it has been discussed to completion, that is, when the `HasBeenSpokenOfBy` routine returns `true`, it is removed and the `Current_subject` property is set to zero. While `Current_subject` is not zero, the `ORNPC_converse` action takes precedence over all other potential NPC actions.
A point of clarification should be made here: Although this seems to be related to the previously mentioned `repeatable`property, it is not. If the last topic talked about returns from from the `HasBeenSpokenOfBy`property, it will continue to be talked about regardless of the `repeatable` property.

4. The Can_Converse Property

Although an NPC may derive from this class, setting the `can_converse` property to `false` effectively disables this action.

ORNPC_movement

The ORNPC_movement module implements what was previously referred to as an "NPC action." As the name describes, this module gives the NPC the ability to initiate acts related to movement. Movement can be defined and configured in a number of ways, but before we discuss these, let's create a few simple rooms to serve as a context for our examples, and place the player in one:

```
object foyer "Foyer" has light
  with description "This is a clean, simple entry hall.
           ^^Exits lead north, east, and northeast."
  ,   n_to diningroom
  ,   e_to study
  ,   ne_to atrium
;
object diningroom "Dining Room" has light
  with description "This is really more of a breakfast nook.
           ^^Exits lead south, east, and southeast."
  ,   s_to foyer
  ,   e_to kitchen
  ,   se_to atrium
;
object kitchen "Kitchen" has light
  with description "The place where food is prepared.
           ^^Exits lead south, west, and southwest."
  ,   s_to study
  ,   w_to diningroom
  ,   sw_to atrium
;
object study "Study" has light
  with description "A library without books.
           ^^Exits lead north, east, and northeast."
  ,   n_to kitchen
  ,   e_to foyer
  ,   ne_to atrium
;
object atrium "Atrium" has light
  with description "Light filters down from above.
           ^^Exits lead northeast, northwest, southeast, and southwest."
  ,   nw_to diningroom
  ,   ne_to kitchen
  ,   se_to foyer
  ,   sw_to study
;
[Initialise; location=foyer; ];
```

1. NPC Wandering

By default, when an NPC has made the decision to move, it will wander. That is, the NPC will walk around in any available direction from which he did not just come, and turn around to retrace his steps only when there are no other exits. Additionally, wandering NPCs will open closed doors as they pass through, possibly unlocking them if they hold the key. What follows is a definition of a character capable of moving around and a single NPC derived from it. Take note that the NPC has been set up to obey the player's orders by virtue of the `will_follow_all_orders property`. See the section on NPC_DoVerb for more information on this.

```
Class npc_c class ORNPC_movement ORNPC_doverb ORNPC
    with will_follow_all_orders true
;
npc_c maid "maid" foyer has female with name 'maid', can_move false;
```

2. The HALT and UNHALT Orders and the Can_Move Property

Notice the `can_move` property in the maid. An NPC's ability to wander around can be turned off and on programmatically by flipping the `can_move` property to `true` or `false`. As we'll see in the transcript of the next section, this can also be done with the HALT and UNHALT orders.

3. The FOLLOW Verb

When dealing with autonomous NPCs, there are times that the player may want to follow an NPC around. To avoid the tediousness of scanning through text to find out what direction the NPC went, the FOLLOW verb has been implemented. Note, that the FOLLOW verb only works on NPCs that have just left and are therefore in an adjacent location. The following transcript demonstrates:

```
Foyer
This is a clean, simple entry hall.

Exits lead north, east, and northeast.

You can see a maid here.

>MAID, UNHAULT
The maid resumes a more mobile posture.

The maid wanders away to the east.

>FOLLOW MAID

Study
A library without books.

Exits lead north, east, and northeast.

You can see a maid here.

The maid wanders away to the northeast.
```

4. The FOLLOW Order and the Start_Following Property Routine

In addition to wandering around, an NPC can be made to follow another object's movements. This could either be another NPC, or perhaps an object that the NPC carries. The `start_following` property routine is used to specify the followed focus in code, but NPCs can also be ordered to follow an object with the FOLLOW order.

Although using FOLLOW as an order is similar to using FOLLOW as a command, it differs in that the behavior is persistent. That is, the NPC will continue to follow that object in

subsequent turns until the `stop_following` property routine has been called, or the STOP FOLLOWING order is issued. To exemplify this, let's add another NPC:

```
npc_c butler "butler" foyer with name 'butler';
```

Notice that the butler does not have the `can_move` property turned off. This means that he will immediately start wandering around unless we catch him with the FOLLOW order first:

```
Foyer
This is a clean, simple entry hall.

Exits lead north, east, and northeast.

You can see a maid and a butler here.

>BUTLER, FOLLOW ME
The butler looks at you and then nods agreeably.

>MAID, FOLLOW BUTLER

The maid looks at the butler and then nods agreeably.

>N
Dining Room
This is really more of a breakfast nook.

Exits lead south, east, and southeast.

The butler enters from the south (following you).

The maid enters from the south (following the butler).
```

5. Using the Tracker Property in Conjunction with the ORPathMaker Module

An important point of restate what was specified above in section 3: FOLLOW is usually limited to adjacent rooms only. If, in the previous example, the butler were to be transported to a room not adjacent to the maid's location (in this case, the study) then the maid would not be able to FOLLOW him. Instead she would fall back to wandering mode until she stumbled across him again at random.

Including the ORPathMaker module offers another capability which can be turned on by setting the `tracker` property to `true`. This gives the NPC the ability to "track," what they are following as a bloodhound might do. Distance is limited only by the constraints of the `PATHDEPTH constant`(see the description of the ORPathMaker module for details).

Keep in mind that the ORPathMaker module is not an auto-dependency. It must be include for this functionality to work; setting the `tracker` property alone will accomplish nothing.

6. NPCs Following a Path

As an alternative to wandering, a `path` can be defined for an NPC to follow in the `path`

property list. An element of this list can either be a direction object or a neighboring room. If a neighboring room is specified, then the direction in which the room lies is determined and an attempt is made by the NPC to walk that way.

There are occasions where an NPC's `path` can be made impossible to follow. Assuming the defined `path` is valid, these occasions generally occur when an NPC's location has been changed by an external means. For instance, transporting an NPC to another location would do this, perhaps making the NPC "lost". When an NPC is "lost", it will wander until it reaches a location that it "recognizes" (is in its `path`), and resume following its `path` from there. It is important to note, that an NPC's ability to gracefully recover when diverted from its `path` is stunted by the use of direction objects in the `path` list rather than room objects. The NPC's ability to recognize its location is limited to the rooms in the `path` list. Perhaps the most versatile method is to use a combination of both.

By default, when the end of the defined `path` is reached, it is looped back to the beginning and started over. It is possible, however, to have an NPC retrace his steps and follow the `path` in reverse. This behavior can be implemented by setting the `reverse_at_path_end` property to true.

The following code demonstrates the same path set up for the butler and the maid, one using direction objects, the other using rooms. Note that the maid and butler will walk together for the first four turns, but because of the `reverse_at_path_end` property, the butler will then turn around and walk back the way he came.

```
npc_c butler "butler" foyer
   with name 'butler'
   ,   path n_obj e_obj s_obj w_obj
   ,   reverse_at_path_end true
;
npc_c maid "maid" foyer
   has female
   with name 'maid'
   ,   path diningroom kitchen study foyer
;
```

ORNPC_moods

Typically, when we code an NPC to conditionally take actions, it is based upon world elements. We could, for example, cause the maid to pick up anything near her that is out of place. While this is a necessary technique for determining whether or not an NPC can or will take an action, there is an additional factor that influences people which has thus far not been addressed: their state of mind.

The maid might choose to not pick up an object if someone she does not like drops it, or not at all if she is simply in a bad mood. The ORNPC_moods class does not directly affect the NPC's behaviors; rather it provides a "mood system" for other modules to utilize. For NPCs that derive from this class, a state of mind is maintained and mechanisms for adjusting the NPCs frame of mind are provided. Additionally, these NPCs can remember how they feel about someone or something so that slapping an NPC, running away, and returning later does not result in instant forgiveness.

1. Agitating and Cheering Up an NPC

This class adds two routines to an NPC. The first, `Agitate`, adds a little irritation to an NPC's mindset when called. The second, `CheerUp` makes an NPC a little less irritable and a little more disposed toward happiness.

Both of these routines take a parameter named `silent`. When passed as true, the adjustment to the NPC's attitude is performed without feedback. This is useful when more than one call is made (when an event is really irritating). If `silent` is not true, then the appropriate message contained in either the property `cheerup_msg` or the property `agitate_msg` is printed.

2. The Mental State of an NPC

An NPC's attitude can be evaluated with a call to the property routine `MentalState`. This routine will return one of five predefined constants:

```
NPCLivid
NPCAngry
NPCNormal
NPCHappy
NPCEcstatic
```

It is important to understand that these values are a form of "mindset notation," a simplified representation, of the NPC's current mood. Aggravating an NPC in a "normal"

state of mind will not necessarily make him "angry," but aggravate him enough and he will eventually become angry and finally cross the threshold into lividness.

3. Changing the standard of mindset

The "mindset notation" discussed previously is defined in a property routine called `FrameOfMind`. It is this routine that translates a broad-ranged numerical value into one of the five notation constants. `FrameOfMind` can be overridden to redefine the ranges at which an NPC is considered "Happy", or "Angry".

4. Recalling and Recording Impressions

As we interact with the people around us, we remember our past experiences of them. We harbor grudges and play favorites based upon our likes and dislikes of the people we know, however it is seldom that we like someone by a logical choice. More often than not, we remember how we felt when last we interacted with them. To the point, we generally like or dislike people based upon how they make us feel.

Two routines are provided by NPCs that inherit from the `ORNPC_moods` class to implement this behavior. The first, `Record_Impression` takes an object as a parameter and will associate the way the NPC currently feels with that object.

As time passes, the NPC's mood may change, but his last impressions of an object, that is, how he felt at the time the impressions were made, can always be retrieved with a call to the routine `Recall_Impression`.

Both of these routines deal with a snapshot of the NPC's mindset and the result of `Recall_Impression` can be passed through the `FrameOfMind` routine to transform it into mindset notation.

5. Effect by Recollections

Our impressions of things do more than determine our likes and dislikes. They also affect our current frame of mind. When coming across a person he dislikes, even if currently in a good mood, a person's frame of mind will likely be adversely affected.

The property routine `affect_from` takes an object as a parameter. This routine looks for the object in the NPC's likes and dislikes and makes adjustments to the NPC's mindset accordingly.

ORNPC_Map_Known

This NPC componant class modifies the behavior of `ORNPC_movement` so it is required that NPCs which derive from `ORNPC_Map_Known` to also derive from `ORNPC_movement`.

One of the features of the `ORNPC_movement` class allows NPCs to follow a path. There are restrictions to this, such as the rooms in the path need to be contiguous. If the NPC ever arrives in a room which is not adjacent to the next location listed in his path, then the NPC can become "lost" and wander around until he stubles across a location that he "recognizes" (appears in his path).

This module removes this restriction. Allowing rooms in the NPC's `path` property to be seperated by several other rooms. It leverages the `ORPathMaker`module to generate the shortest path between the NPC's current location and target location thereby enabling the NPC to arrive at his path and never wander around lost (assuming the destination is actually reachable and falls within the limitations of the `ORPathMaker` object.)

Other than adding the `ORNPC_MapKnown` class to the NPC's inheritence chain (and prerequisits) no additional code need be implemented to make this functionality occur.

Part E
Other Stuff

Using the ORLibrary Entries Without the Framework

The library files in the ORLib section of the OnyxRing website have been designed to be implemented via the ORLibrary framework, specifically the OR_Library_Include.h file and a primary source file modeled after OR_BlankGame.inf. But the ORLibrary framework doesn't suite everyone's preferences. It is entirely possible to pull a library entry out and use it without committing yourself to the whole concept of the framework. What follows is a description of how.

In the description of each library entry, a constant is described that needs to be included in the game file proper. Generally it is the name of the extension prefixed with the text "USE_". To use the ORDoor library entry, the text

```
Constant USE_ORDoor;
```

should appear at the top of the game. This is true whether you use the framework or not.

Next there are four places where you should use the include directive to import the library entry file (yes, the same file in four places):

- 1) Before inclusion of Parser
- 2) Before inclusion of VerbLib
- 3) Immediately after the inclusion of VerbLib
- 4) Immediately after the inclusion of Grammar

Alone, however, these includes will not work. Each location corresponds to a section of the library file that needs to be compiled. Which section is to be compiled is indicated to the compiler by the existence of four constants:

- 1) REPLACEPOINT
- 2) MESSAGEPOINT
- 3) CODEPOINT
- 4) GRAMMARPOINT

Each of these four constants has a priority that circumvents the existence of lower prioritized constants so they cannot simply all be defined at once. Instead, each must precede the appropriate inclusion of the library file. Putting it all together gives us the following:

```
!...  code that belongs before PARSER

Constant USE_ORDoor;
Constant REPLACEPOINT;
#include "ORDoor.h";
#Include "Parser";

!...  code that belongs between PARSER and VERBLIB

Constant MESSAGEPOINT;
#include "ORDoor.h";
#Include "VerbLib";
Constant CODEPOINT;#include "ORDoor.h";

!... code that belongs between VERBLIB and GRAMMAR
```

```
#Include "Grammar";
Constant GRAMMARPOINT;
#include "ORDoor.h";

... code that follows GRAMMAR
```

It should be mentioned here that an increasing number of ORLibrary modules have dependencies upon other modules and attempt to meet these dependencies automatically by using framework techniques. The OREnglish LDF is one such example. As a rule, it is **really** less troublesome to simply utilize the framework (by leveraging the OR_LibraryInclude.h file) when more than a couple of modules are going to be utilized.

That being said, for those who are simply against this idea (for whatever reason) the dependencies must be included manually. By convention, a list of all dependencies that a library entry tries to include can be found in the "AutoDep" section of the module's comment header.

Part F
Quick Start

How do I quickly setup and use the ORLibrary?

I've been compiling Inform programs for a while, and just want to quickly use the ORLibrary without juggling directories. What's the easiest way?

The easiest way to **setup** the ORLibrary is simply to dump all the ORLibrary files into the same directory as the standard library and include the following text on your compiling command line:

```
+language_name=OREnglish
```

That's it. The ORLibrary is now setup and ready to be used.

The easiest way to **use** the ORLibrary is to leverage the OR_BlankGame.inf template as a starting point for your game. It already contains the required four includes at the appropriate locations and provides a list of commented-out references to most library entries. Simply uncomment the ones you would like to use and they will automatically be included.

Return to the ORLib "How Do I" index.

How can I avoid specifying an object's name?

Modules used: ORRecogName.h
Auto Dependencies: None

I have a simple wooden cube that I would like to refer to. It doesn't make sense to me why I have to specify the "name" property when I've already named it. I just want to do the following:

```
object -> wcube1 "wooden cube";
```

How can I make this work?

The ORRecogName entry does this automatically. Simply include (or uncomment) the USE_ORRecogName constant in your source file and the cube can be referenced by any of the words printed by in it's textual name. The "name" property now needs only to be specified for synonyms and plural words (with the //p indicator).

Return to the ORLib "How Do I" index.

"How can I generate the map from the player's moves?"

Modules: ORDynaMap
Auto Dependancies: None

I'm planning a scene in a game where the player needs to wander around for a while before reaching his destination. I don't want to create a large map because it could take the player too long to find the destination, but I also don't want him to "stumble" upon the end in just a single move.

I played a game* once where a map of the Martian landscape was generated by the player's choice in movements, giving the illusion that the map was bigger than it was. I wanted the player to wandering through about four rooms altogether. How easily can this be done?

This can be done very easily with the ORDynaMap module. Just follow three simple steps:

1) Add (or uncomment) the USE_ORDynaMap constant.

2) Create the "wandering rooms" which will be generated. It is easiest to derive these from the ORDynaMapRoom class:

```
ORDynaMapRoom forest_start "forest clearing"
  with description "The beginning of a vast forest."
  ,    s_to "Don't go that way. You just came from there.";

ORDynaMapRoom forest1 "forest"
  with description "You are surrounded by trees.";

ORDynaMapRoom forest2 "forest"
  with description "No place to go but toward the trees.";

ORDynaMapRoom forest3 "forest"
  with description "You are definitely lost in the woods.";

ORDynaMapRoom forest_end "forest clearing"
  with description "At last! You've found the forest, hidden
              by all the trees!";
```

3) Create an instance of the ORDynaMap class and specify the rooms in the "found_in" clause in the order that they will appear:

```
ORDynaMap found_in forest_start forest1
             forest2 forest3 forest_end;
```

That's it. obviously the player needs to have access to forest_start. From there, any direction not already defined will lead to the forest1 room and the map will be altered accordingly.

*The game was "Photopia" by Adam Cadre.

Return to the ORLib "How Do I" index.

How can I easily setup a door?

Modules used: ORDoorInit.h
Auto Dependencies: ORDoor.h, ORObjectInitialise.h

I'd like to implement several doors in my game, and have been looking at sample code. Yuck! Having to specify the door_dir, door_to, and found_in properties is a lot of complexity that I would like to avoid. What would really be nice is to just create a door and point to it from the room's direction property. Is this possible?

Sure. The easiest way to implement a door is to define (or uncomment) the USE_ORDOORINIT constant in your source file. The following code will now work:

```
!---Two rooms with a door between them...
object east_room "East Room" has light
  with description "The eastern room. A door lies to the west. "
  ,     w_to door1;
object west_room "West Room" has light
  with description "The western room. A door lies to the east. "
  ,     e_to door1;
!--- And now the door...
ORDoor door1 "door" with name 'door';
```

Return to the ORLib "How Do I" index.

How can I make an NPC follow the PC?

Modules: ORNPC_movement

Auto Dependancies: ORNPC, ORNPCVerb, ORNPC_doverb, ORObjectInitialise, ORLibraryMessages

I've got an NPC that I want to make follow something else around. Is there an easy way to do this?

Yes. The ORNPC_movement class supports this functionality implicitly, using "NPC verbs". Simply define (or uncomment) the USE_ORNPC_Movement constant and all needed library entries will be pulled in.

At this point I'd like to mention that NPCs are especially complex and a considerable amount of work has been put in them. The actual ORLib documentation should be consulted for any real understanding of the ORNPC component classes, but for a quick answer to the question, the following NPC will follow the player character around:

```
object bob "Bob" east_room has proper
  class ORNPC_movement ORNPC_doverb ORNPC
  with name 'bob'
  , follow_object selfobj;
```

The reference to the "selfobj" object can be changed to any object in the game, including carriable items and obviously "Bob's" location is game dependant.

Return to the ORLib "How Do I" index.

1. How do I say something to an NPC?

Q: I've created a small game with a single NPC and a single room. I'd like to talk to the NPC. How can I accomplish this?

A: What you've described so far probably looks something like this:

```
#Include "OR_Library_Include"; #Include "Parser"; #Include "OR_Library_Include";
#Include "VerbLib";
#Include "OR_Library_Include";
Object theroom "room"
has light
with description "This is just a room.";
object bob "Bob" theroom
has animate proper
with name 'bob';
[Initialise; location = theroom; ];
#Include "Grammar"; #Include "OR_Library_Include";
end;
```

The above example is a pointless exercise as far as the ORLibrary is concerned. Since no modules are specified, the included file "OR_Library_Include" has no functional effect. The resulting program runs identically to a plain vanilla Inform program.

The ability to talk to NPCs is provided by the ORKnowledgeTopic module. With very little work (only three additional lines of code after referencing the module) the player can begin to speak to other characters immediately. For clarity, the new lines have each been indented:

```
       Constant USE_ORKnowledgeTopic;
#Include "OR_Library_Include"; #Include "Parser"; #Include "OR_Library_Include";
#Include "VerbLib";
#Include "OR_Library_Include";
Object theroom "room"
has light
with description "This is just a room.";
object bob "Bob" theroom
has animate proper
with name 'bob';
[Initialise; location = theroom; ];
       ORKnowledgeTopic with name 'hello'
       , knownby selfobj
       , topicinformation "~Hello there,~ you say.";
#Include "Grammar"; #Include "OR_Library_Include";
end;
```

The code of interest details the ORKnowledgeTopic instance. Other than the traditional "name" property, only two properties are really needed for this basic conversation topic to work:

The first is the "knownby" property which lists all the characters that know this topic and can therefore speak about it. Only the player object, that is "selfobj," knows this topic.

The "TopicInformation" property stores the actual text that is printed when the topic is spoken.

A short transcript shows that you can now talk to Bob:

```
room
This is just a room.
You can see Bob here.


>tell bob hello
"Hello there," you say.
```