The Dream Maker

Designer's Guide to Worlds BYOND

2nd Edition

Dantom

Cover illustration by AZA, Original illustrations by Dan.

ISBN 1-890661-17-1

Inquiries regarding this book may be directed via email
to dantom@dantom.com. Comments and corrections are
welcome!

Printed Internationally

http://www.dantom.com
http://www.byond.com

Table of Contents

Original Foreword 7

New Foreword 8

Acknowledgments 9

  Additional Acknowledgements 9

Preface 11

Meet the Dream Maker 13

Navigating The Code Tree 25

  1. Formatting Code 26

  2. Compilation Errors 27

  3. Paths in the Tree 28

  4. Code Comments 29

Objects in the Tree 31

  1. Object Properties 33

    1.1 Location 35

    1.2 Additional Mob Properties 36

  2. Assigning Variables 37

    2.1 Constant Values 38

  3. Putting It All Together 40

Verbs 41

1. Defining a Verb 42

2. Setting Verb Properties 42

3. Verb Accessibility 44

 3.1 Explicit versus Implicit Source 46

 3.2 Default Accessibility 47

 3.3 Possible Access Settings 48

4. Overriding Verbs 50

5. Friendly Arguments 51

 5.1 Parameter Input Types 51

6. Generating Output 55

 6.1 Variables in Text 56

7. Flexibility in Choice of the Source 57

8. A Choice of Arguments 59

9. Default Arguments 60

10. *anything* input type 61

Original Foreword

And now for a little history.

Way back in 1994, when the Internet was a harmless baby, "surfing" referred to an outdoor recreation, and "Yahoo!" was an expression of glee, Dan approached me with the following proposition: Let's build a game. Back then, in the midst of our fruitful college years, we were devoting much of our free time to attempting to destroy quantum mechanics, but despite numerous attempts had not yet succeeded. Frankly, it was wearing me down. This sounded like the perfect diversion, so I prodded on: What do you have in mind, fellow scholar, slayer of the quantum fallacy? Little did I know that this seemingly innocuous inquiry would lead me down a path so full of ideas and inspirations that once trapped, I would never again escape into the safe haven of the world for which my degree was intended. Unless you, too, want to succumb to the same fate, I advise you to stop reading now!

Dan wanted to build not just any game, but an *online*, *graphical* game. At the time, this struck me as quite revolutionary. I had only first experienced text-based online games a few years before, and recalled my amazement upon first interacting with another person whom I had never before met. Actually, I recalled my stupidity, for that first encounter involved me making a fool of myself by mistaking this fellow player as a computer-controlled being--and being perplexed by its intelligence! To combine this interaction in a graphical setting would be tremendous indeed.

Over the next few months, the workings of a system began to fall into place. Soon we had little sprites moving around on screen, bumping into walls and other players, players on other machines even! But we realized too, that we had our work cut out for us. Just over the horizon, huge companies were amassing scores of programmers and artists to shape what would become the online gaming revolution. We were only two inspired souls--and neither of us could draw a straight line!

It was at this point that we made what would be the biggest decision of this project, one that would shape our lives for many years to come. Why not let other people write the game? Instead of forcing the players to conform to our system, why not let them build their own system, why not let them *build their own net dream*? We knew that if we could provide the tools to make this process exciting and enjoyable, the netizens, in all their collective creativity, would do the rest. They would do it better than we could, and it would be *a lot more fun* than letting corporations rule the field alone.

It is up to you to decide whether we have been successful in this goal or not. Dan will now take the helm and guide you through the inner workings of this Dream Maker. May your journey be safe, and your dreams bountiful!

New Foreword

Well, it has been a while, but here is the second edition of The Dream Maker (or the 'Blue Book'). It has major updates from the previous book to encompass all of the changes from BYOND 2.0 to BYOND 4.0, including the new icon procedures, better explanations of old techniques and newer explanations of all the new procedures and methods available. There is a brand new section that puts particular emphasis on Design, too.

I'm not trying to bore you now, so here's The Dream Maker, Second Edition.

Acknowledgments

The development of the BYOND system has been driven largely by the innovative community of users who have tested it and made insightful comments during its beta-testing phase. BYOND continues to grow to this day, and will likely do so as long as the people are there to push it to new and wonderful directions.

Special thanks go out to the old group of developers, the class of '2000, so to speak: Nick "AbyssDragon" Cash, Erron "Dragon" Flaherty, Jeremy "Spuzzum" Gibson, Julio Monteiro, James Murphy, Joanna "Zilal" Panosky, Mike Schmid, Gabriel Schuyler, and Chris "Manifacae" Sivak.

And an extra debt of gratitude is reserved for Ron "Deadron" Hayden and Guy Tellefsen for not only being excellent developers, but for helping edit this manuscript. Any errors which remain are claimed by Dantom and may be attributed to endemic feature creep.

Tom of Dantom
June 9, 2000
Irvine, California

Additional Acknowledgements

Special thanks go out to the current BYOND community, which has been growing since the year 2000 to encompass almost 6,000 users online at any one time. While many of the old developers are no longer active and have moved on to bigger and greater things, some have stayed, and many more have come to take their places. Without the community BYOND would have been driven into the ground by now – thanks guys.

Special gratitude is given to the current developers of the BYOND platform – “Lummox JR”, “Tom” and any others that I may have missed – they’ve been improving and upgrading BYOND since the year 2000 are currently gearing up to move BYOND into the isometric field!

AZA of BYOND
September 21, 2009
Berkshire, United Kingdom

Preface

Creation of the *next* moment is of far greater significance than was creation of the first.
--Some Wise Sage

So you want to play god. It happens to a certain fraction of us--the desire to create a world. It could be a fantastical place, a land of adventure and mystery, or it could be a secluded island, a secret hideout, or an outpost on Mars--who knows.

To conjure such an illusionary realm, one must know the right incantations, powerful spells of arcane origin that weave the thread of reality into a tapestry of your own design. Indeed, language is the engine of illusion. And illusion is but another glimpse of reality. Thus reality is language. Nothing more, nothing less.

Let me begin again from another angle. A computer programmer arranges letters to form words of obscure meaning, organizes these into phrases unintelligible to the common mind, and formulates from this an algorithm--a sort of ephemeral spirit who, in the blink of an eye, can do the work of days, or perhaps some mischief if its master has made the slightest error. In short, a computer programmer is a magician, a person whose very words are power.

And every programmer starts out with the desire to create games. I should say, every programmer *with a soul*. There must be a few ghoulish creatures, pale shadows of humanity, who are born with the desire to write statistical analysis software for the census bureau. But I imagine that deep down inside, even these outwardly unfeeling corpses feel the urge to slip in a tic-tac-toe board, activated by punching the first 30 digits of pi into the data entry screen.

So what is it about computer games that is so attractive to a programmer? It's not necessarily the desire to *play* the games. I personally very rarely feel the urge! But if it's not that, what is it?

I propose that it is the same desire mentioned initially--the desire to play god. A game, after all, is a sort of artificial world, a place run entirely according to an invented set of rules. And rules are nothing but the minions of sorcerers and programmers--that is, words.

Language again! We keep coming back to it. (Or it keeps coming back to us?) Perhaps it is time to deal with that subject. It is, after all, the reason I am writing (as well as the reason I am able to write).

It has been my passion to discover a language suitable for the creation of worlds. I say *discover* rather than create, because a language does not come into being at the whim of a mere mortal but instead chooses to reveal itself at the appropriate time and place. One must only provide a suitable home for it.

A language is not a solitary creature, being inherently social by nature. With this in mind, I sought out a company of adventurers, sharp of mind and willing to embark on the arduous quest. We called ourselves Dantom.

It soon became apparent that even our dedicated band of explorers would not provide enough company for this guest, the language of worlds. Its dwelling place, we decided, must reside in the ethers themselves--a meeting place for thousands upon millions of minds. So we began to construct such a place, called BYOND.

Having done so, like minds began to arrive, attracted by the bold and tantalizing statement: Build Your Own Net Dream. And with the growing host of receptive people at hand, a presence began to take shape. Certain words, when uttered in the right context, seemed to cause a stirring, a certain fluttering at the edge of perception. What started as a drip soon became a trickle and then a roaring flood of understanding. We were speaking the language of creation! The quarry so painstakingly sought had come to dwell among us.

This language of worlds within worlds, of illusion become reality, is called DM, the Dream Maker.

Chapter 1

Meet the Dream Maker

The first step to mastery in the lands of sleep is the realization, without waking, that one dreams. In the day worlds, mastery begins by forgetting, without dreaming, that one is awake.
--DoD

DM is a programming language for the creation of multi-user worlds. By `world' I mean a virtual multi-media environment where people assume personae through which they interact with one another and computer-controlled objects. This could take the form of a competitive game, a role-playing adventure, a discussion board, or something we haven't even imagined.

Frequently, the terminology of a role-playing game is most suitable: humans are PCs (playing characters) and computer-controlled personalities are NPCs (non-playing characters). The virtual embodiment of a player is often called an *avatar*. The game rules are written in DM and faithfully carried out by the computer. These define what actions players may instruct their avatar to perform, what effect these will have in the game, and any other events that may happen as time progresses.

To understand the mechanics of the system fully, it is helpful to know a few simple terms. Computer programs that operate over a network are often divided into two parts: a client and a server. In this case, the client is the program that players use to enter commands and see what happens as a result. In other words, the client handles input and output. The server is the program that runs the game, carrying out the rules defined in the DM language. The game designer writes these rules in a third program called the compiler. This reads the DM program (known as the *source code* by programmers), checks it for grammatical errors, and generates a more compact, computer friendly, file known as the *byte code* or *binary*. It is this file which the server reads to see how to run the game.

So there are three main programs: the client, server, and compiler. We call these Dream Seeker, Dream Daemon, and Dream Maker, respectively. (The word *daemon* is just another (more fantastical) word for server.) As a whole, we refer to this collection of software as BYOND, which stands for Build Your Own Net Dream--an apt description of its purpose and also of how far it has wandered *beyond* our original plans. But that is another story!

Every introduction to a programming language must begin with the same example. Call it destiny, inevitability, or pure chance; it is rather uncanny that the name of the universal introductory example is *hello world*. Spooky, no? That's exactly what happens in this example--we say hello to the world.

In DM you write it like this:

```
mob
   Login()
      world << "Hello, world!"
```

If you have any prior programming experience, the last line probably looks vaguely sensible. It outputs the text inside the double quotes to the whole world. But what on earth is a *mob* and why is each line indented like stair-steps? All in good time. For now, simply understand that the player's avatar in the game is a *mob*. When a player logs in, the server is instructed to output the message "Hello, world" to everybody.

Compile and run this program (see figure 1.1). If all goes according to plan, you should see the words, "Hello, world" magically appear on Dream Seeker's output screen. Voila! You have created your first BYOND world.

Now you know the basic steps for designing worlds. You write some DM code, compile it, and run it. But this world didn't have anything for the player to do. That's next.

Figure 1.1: Hello World!

This first world serves not only as an introduction to the DM language, but to the Dream Maker editor/compiler as well. Fortunately, the system has been designed to be quite simple to use, and with just a few steps you should be on your way to BYOND programming wizardry!

Dream Maker refers to the collection of files comprising the project as the world *environment*. When you make a new project, you create a single environment file, which has the name "`[worldname].dme`". This file may contain source code, but in general it will only be comprised of automatically generated references to other files in the project. This is best seen by example, so let's stop talking, and get coding!

1. Create the "hello" project by selecting **New Environment...** from the **File** menu. This prompts for a directory in which your project will be stored. Enter "`hello`" as the desired directory. This creates a new directory called "`hello`", which now contains the `hello.dme` environment file. Notice that `hello.dme` also appears in the file tree displayed on the left side of the screen. All files in the environment directory are listed there.
2. Let's put the code for this project in a separate file. Select **New File...** from the **File** menu. Choose **Code File** for the type, and enter "`hello`" for the name. This creates a file called "`hello.dm`" in the environment directory, and a corresponding listing in the file tree. The checkbox next to it indicates that the code in `hello.dm` will be included in this project.
3. The `hello.dm` file is now ready for editing. Type the following code. (Make sure the first line is not indented, the second is indented once, and the third is indented twice. It is easiest to use tabs for the purpose.)

```
mob
   Login()
      world << "Hello, world!"
```

4. Compile the code by selecting **Compile** from the **Build** menu. If there are problems, they will appear in the output box at the bottom of the screen. But unless you indented incorrectly, all should proceed smoothly.
5. Run the compiled world by selecting **Run** from the **Build** menu. This launches Dream Seeker, which should subsequently welcome the world!

Consider the Hello World example again. The DM code says that when a player logs in, a message should be displayed. We can do a similar thing for other types of actions. For example, if the player types a command, a message could be displayed.

In DM, commands are called *verbs*. A verb is defined in the following example:

```
mob
   verb
      smile()
         world << "[usr] grins."
```

Notice the funny stair-step indentation again! That will be explained in a little bit. For now, read this example from top to bottom. Once again we are defining a property of a *mob* (the player's avatar). In this case we are adding a *verb*, an action that the player can instruct the mob to perform. The name of the verb is *smile*. The final line displays a message when the mob smiles. Notice the `[usr]` in the message; as you may have guessed, that is not displayed literally but is replaced by the name of the user, the player who initiates the command.

Run this example. Once you have logged in, try typing *smile* and pressing enter. You should see the grinning message with your login name substituted into it. Amazing! Fantastic! But playing god is a serious business. Don't let anyone catch you grinning.

For variety, you could add some more verbs. Here are a few:

```
mob
   verb
      smile()
         world << "[usr] grins."
      giggle()
         world << "[usr] giggles."
      cry()
         world << "[usr] cries \his heart out."
```

Now the stair-step pattern has been broken because all three verbs *smile*, *giggle*, and *cry* are at the same level of indentation. In DM, indentation at the beginning of a line serves to group things together. Here, *smile*, *giggle*, and *cry* are all grouped together as verbs belonging to *mob*. Each of these verbs has it's own contents indented beneath it.

Notice the use of `\his` in the *cry* verb. This macro is replaced by the appropriate possessive pronoun. It could be *his*, *her*, *its*, or *their*, depending on the gender. DM provides a few other useful macros like this one to make life easy.

So far nothing has been said (because you never asked) about the empty parentheses after the verb names in the above examples. They were in the first example after *Login* too. These are the mark of a procedure definition. The verbs and *Login* are all examples of procedures, which are a sequence of instructions to be carried out. In the examples so far each procedure consisted of only one line--an instruction to display some text. They can, of course, become much more complicated than that.

There are two general categories of procedures: those that show up as player commands and those that do not. These are called *verbs* and *procs* respectively. By that definition, *Login* in the *Hello World* example was a proc.

The parentheses after a procedure name are more than decorative. They can be used to define procedure parameters. This allows for providing additional information to the procedure. The information is stored in a variable, that is, a piece of memory with a name. To confuse matters, a programmer will often call such variables, which serve as the parameters to procedures, *arguments*. Why? Well, just for the sake of argument.

Here is an example of a verb that takes a parameter--in this case a short message to be broadcast to the world.

```
mob
   verb
      say(msg as text)
         world << "[usr] says, [msg]"
```

In these few short lines are the bare bones of a chat world. Users can log in and start gabbing using the *say* verb. Try it out. Your session might look something like the following:

```
say "hello world!"
Dan says, hello world!
```

The main point of interest in the DM code is inside the parentheses where the parameter msg is defined. It could have been called anything; the variable name is arbitrary. The statement as text indicates that a short message supplied by the user will be stored in the variable. This message is then inserted into the final output at the position marked by the expression[msg].

So far I have casually introduced mobs, verbs, procs, and arguments. Now it is time for a formal (tediously exciting) description of the DM syntax. It may take several multi-clausal sentences to get through this, so don't hold your breath.

DM code is structured like a tree. The top of the tree is called the root. The various types of virtual objects (mobs being one) branch off of the root and may in turn give rise to additional strains that branch down from them.

If you haven't noticed, the code tree terminology is upside down. Of course, so is the file-system on your hard-drive, and every other informational tree in existence. It is quite possible that the vast majority of computer scientists have never actually seen a real tree. The sheer weight of their ignorance keeps the jargon from flipping right side up, and we are stuck with trees having a root at the top and leaf nodules at the bottom. Or it might just be standard obfuscation. That's why I do it.

It is time for an example. One particularly interesting type of virtual object is a *turf*. It is a building block used to make graphical maps that players can walk around on. Suppose we wanted to make a maze. That would require two types of turfs: floors and walls. Here's how you would define them:

```
turf
   floor
   wall
```

All we did was branch two new types of objects off of the basic turf. One is called floor and the other wall. The terminology of a family tree is often used to describe the relationship of the various objects defined here. Turf is the parent of floor and wall. The two children are siblings of each other. A child inherits all the properties of its parent and adds some of its own to distinguish it from its siblings. Both floor and wall are turfs because they are derived from the turf object type.

To make a maze, we need to specify a few properties of floors and walls: what they look like and whether you can walk through them. While we're at it, the appearance of a player should be defined too. This is how it is done:

```
turf
   floor
      icon = 'floor.dmi'
   wall
      icon = 'wall.dmi'
      density = 1
mob
   icon = 'player.dmi'
```

Several assignments have been made. These take the form of a variable on the left-hand side and a value on the right. In the case of the icons, the value is the name of an icon file inside single quotes. In the case of density, the value should be 1 or 0 to indicate if it is dense or not. A dense turf will not allow other dense objects (like mobs) to walk through them.

Figure 1.2: The Amazing Mapper

For most programs, adding graphic support is a massive chore. The facilities in Dream Maker, however, make this task quite simple. For our example, we'll just draw a couple of icons and put them on a map.

1. Create a project called `maze` through the **New Environment...** option.
2. Create the main `maze.dm` file, and enter the following code:

```
turf
   floor
      icon = 'floor.dmi'
   wall
      icon = 'wall.dmi'
      density = 1
mob
   icon = 'player.dmi'
```

3. Build the `floor.dmi` icon. Do this by selecting **New File...** and choosing **Icon File** for the type. This will bring up a new window, with the option to build pixmaps (static, directionless, icons) and movies (animated or directional icons). Choose **New Pixmap...** from the **Graphic** menu and show off your artistic flair by drawing a picture of a floor. Repeat this step for the `wall.dmi` and `player.dmi` icons.
4. With the three icons in place, the project should compile. Test this by selecting the **Compile** option. If it is successful, you should be able to see your icons in the object tree tab on the left-hand side of the screen. This tree illustrates the objects defined in your world.
5. You can run the world now, but you won't see any icons because you haven't put any objects on the map yet. To build a map, again select **New File...** and choose **Map File**. You can name it whatever you'd like; the `.dmp` extension will be appended, indicating that this file is a map.
6. Now for the fun part! Create the map by selecting objects from the tree and placing them with the various functions. For example, to add a row of walls, select the **Add** function on the map pane, click on the `wall` tile in the tree (it's underneath `turf`), and draw them on the map by left-clicking the mouse. You can remove tiles by right-clicking. The map editor has considerable functionality; you can learn about it by reading the included documentation.
7. Compile the new, graphical project and run it with Dream Seeker. If all goes well, you should see your creation on screen. Your avatar can roam the floors and bump into the walls. Not too bad for a couple minutes of work!

The reason we did not have to set the density of the floor to 0 is that the default density of a turf is 0. Since the floor is derived from a turf, it inherits all the default properties of one. This sort of inheritance of characteristics is one of the important elements of object-oriented languages like DM. Ultimately, it is just a compact way of describing closely related things.

Before you test this example, you will need to design the icons and the maze itself. Fortunately, this process is a natural part of Dream Maker's functionality (see figure 1.2).

When you are done making the map, you can compile and test the world. When you log in, you should be able to walk around in the maze you designed by using the arrow keys. Amazing!

Of course there are always small details that one doesn't think about until after the fact. For example, where is the starting point in the maze? We never specified, so players are just dumped onto the map in the first available spot. Here is one way to do it:

```
turf
   floor
      icon = 'floor.dmi'
   start
      icon = 'start.dmi'
   wall
      icon = 'wall.dmi'
      density = 1
mob
   icon = 'player.dmi'
   Login()
      loc = locate(/turf/start)
```

You will have to make a new icon for the `start` turf and then edit the map to mark the starting position with it.

The code that makes the initial placement of the mob is in the *Login* proc. It sets the location of the mob (*loc*) to the position of the start turf. This is done by using the *locate()*instruction--one of the many built-in procedures in DM (see figure 1.3). It computes the position of an object type (in this case, the `start` turf).

Notice how the object type `/turf/start` is specified. This notation is called a *type path* because of the way you specify the path (starting from the root) down to the specific type of object you want.

Now suppose you forgot to put a start turf on the map. What would happen? The *locate()* instruction would fail and the player would not get placed on the map and therefore wouldn't even be able to see the maze after logging in. A total disaster! Wouldn't it be nice to fall back on the default behavior of at least putting the mob somewhere on the map? In other words, we have to somehow run the default *Login* proc as well as the one we defined, just in case there is no start turf. Here is how to do it:

```
mob
   Login()
      loc = locate(/turf/start)
      ..()
```

The final line does the job. It invokes a procedure with a strange name: just two dots. That is the name DM uses for the default procedure, more generally known as the parent or super procedure. In the case of *Login*, the default proc checks to see if the mob is somewhere already. If not, it finds a vacant spot on the map, which is just what we wanted.

Now you can begin to see the general flavor of DM programming. There are a number of events (*Login* being one) which are handled by procedures. When necessary, you can override the default procedure with one of your own to make things work exactly how you want.

This is another important component of object oriented programming. Each type of object can respond to events differently. The way in which they respond is inherited from their parents by default, but can be redefined and augmented as needed.

This introduction has just scratched the surface of DM. You should begin to see some interesting possibilities. At the same time, you should have a lot of unanswered questions. Keep both of those in mind; they will be your guide through the more detailed exploration of the language that follows.

Figure 1.3: Help is on the way!

No programming environment is complete without a comprehensive, accessible reference. Dream Maker provides this in the form of a searchable index of topics and built-in properties. You may access this by selecting **Help On...** in the menu, or by hitting the **F1** key. If the cursor is positioned on a word (such as "*locate*"), help will be found for that topic.

Chapter 2

Navigating The Code Tree

The real nature of this cosmic tree cannot be known here, nor its beginning, nor end, nor foundation.
--Bhagavat Gita

The previous chapter was a quick introduction to give you a taste for DM programming. This and the next few chapters will cover the same basic ideas in greater detail.

A DM program begins at the root of the tree and descends along multiple branches. Each branching point (or node) is given a name to distinguish it from the other branches at the same level. Names are case-sensitive (so apple and Apple are different). They may be any length and may contain any combination of letters, numbers, and underscores as long as they do not start with a number.

Consider the following code:

```
turf
   trap
      pit
      quicksand
      glue
```

Several types of traps are defined (though no instructions have been included to actually make them work). Here, each type of object is on a line by itself and indentation is used to define the relationships between them. The three siblings pit, quicksand, and glue are all children of trap, which is in turn derived from turf, the map terrain object.

1. Formatting Code

DM provides some flexibility in the way code is formatted. For example, blank lines may be inserted between other lines without effect. This may help code from getting too dense and unreadable.

To compress code that is overly spread out, a semicolon may be used in place of a newline. In this way, several children may reside on the same line. To put a parent and child on the same line, a slash is used. It is equivalent to a newline followed by an additional level of indentation.

The following code is equivalent to the previous example:

```
turf/trap
   pit; quicksand; glue
```

In addition, braces may be used to mark the beginning and ending of a node's children. C, C++, and Java programmers may feel more at home using this style. With the compiler checking both braces and indentation, it is hard for simple mistakes to slip through unnoticed. Sometimes it is the simple spelling and typesetting errors which are the hardest to see.

Here is yet another encoding of the same objects, this time using braces:

```
turf/trap
{
   pit
   quicksand
   glue
}
```

You may use either tabs or spaces in any number to indent your code. The only requirement is that you be consistent. Each block of code must use the same type of indentation throughout. In general, DM provides enough freedom to format your code the way you like without so much freedom that mistakes are likely to slip through unnoticed.

2. Compilation Errors

While we are on the subject of mistakes, you may as well make one now on purpose so you know what is going on later when it's not on purpose. Remove one of the braces from the above code and try compiling it. You should get a compilation error. If you double-click on it, the cursor will jump to the line in the code where the compiler ran into trouble. You can correct the problem and then try recompiling.

Often, you will need to think less like a human and more like a machine to see what is wrong. Forget about what the code is trying to do and focus more on its form: the grammar, the spelling, and little fussy details that only a computer would care about.

The more frequently you compile your code, the less trouble you will have in locating problems. Also realize that if there are many compilation errors, some of the later ones may just be confusion caused by earlier ones. Try fixing the first few and recompiling if the rest don't make any sense.

3. Paths in the Tree

You have already seen how to use a slash to make code more compact. It is used to separate a parent and a child node, for example `turf/trap`. This notation is known as a *path* because it tells the compiler how to get from the current position in the tree to some other point by enumerating the branches to take along the way. If a given branch doesn't exist, it will be created.

Paths have several uses. Sometimes indentation can get so deep that it becomes hard to read. You can use paths instead to get deep down inside the tree without indenting so much to get there. Another time to use a path is when you want to branch off of existing objects from somewhere else in the code.

The following example adds some variation to the basic pit that was already defined.

```
turf/trap/pit
   tar
   lava
   bottomless
```

You could place this code at the bottom (or even the top) of the same file or in another file. (You will see how to use multiple files in chapter 19.)

Finally, there are a few rare cases in which you may want to use an absolute path--that is, a path starting with a slash. This allows you to derive something from the root even if you are not currently at the root. Now does not happen to be one of those rare cases; using an absolute path would only make the code more confusing.

```
turf {
   trap {pit; /turf/trap/quicksand}
   /turf/trap/glue
}
```

If you guessed that this is yet another encoding of the same three traps, you have passed obfuscation level one.

4. Code Comments

When you start writing code as confusing and tangled as the last example, it would be a good idea to leave a few clues behind. Otherwise you may find it incomprehensible the next time you visit, a rather embarrassing situation. There are two ways to write comments in the code. One is for multi-line comments and the other is for single line ones. Example:

```
/*
  This is a multi-line comment.
  You can put whatever you want inside of it.
  The compiler just skips right over.
  Some of you may recognize it as a C style comment.
*/

//This is a single-line comment.
//Some people know of it as the C++ style comment.
```

Comments can occur anywhere in the code--at the end of lines, on lines by themselves, or wherever. They are often used to make statements of intent or purpose. It frequently takes only a very short note to make code much easier to read.

Chapter 3

Objects in the Tree

As mind is the witness and reality of all dream-objects, so soul is the only reality in the diversities of this universe.

--Bhagavat Gita

There are four basic object types. Each has its own special properties, as well as those that they all share. The basic objects are *mob*, *obj*, *turf*, and *area*. There are other objects as well, but these four are the ones which are visible to players. We call them *atomic* objects.

The simplest difference between them is the order in which they appear on the map. Areas are drawn in the first layer. The icon of an area is often simply a solid background color. Turfs are drawn on top of areas; these usually represent some type of terrain like grass, roads, or walls. Objs are drawn next, and might stand for items such as swords or cookies. Mobs are drawn on top of everything else. They normally represent players or computer-controlled creatures. (The term *mob* stands for *mobile object*. It is also suggestive of *monster*, which is a common role they play.)

On the map, the mobs and objs are said to be *contained* by the turf. That in turn is contained by the area. It is also possible for mobs and objs to contain things. For example, a chest obj might contain a bunch of treasure items; and a player's mob could contain all the player's possessions.

Notice how I keep using words like `might,' `could,' or `normally.' That is because DM gives you, the designer, a great deal of flexibility. Many of the basic object properties were defined with a particular purpose in mind. That doesn't mean you have to use them that way. The meaning of the game objects is up to you.

Figure 3.4: Atomic Objects from Lowest to Highest Drawing Layer

area

turf

obj

mob

1. Object Properties

First let us look at the properties that all of the objects have in common. We have already seen that they each have a name and an icon. These are variables. (There are also some procs, but the discussion of those takes place later in chapter 7.) Here is a list of each variable and a description of its purpose.

- **name**

This is the name of the object, which by default is the same as the type (i.e. node) name with any underscores replaced by spaces.

- **gender**

The grammatical gender of the object may be set using this variable. The possible values are `"neuter"`, `"male"`, `"female"`, and `"plural"`. The default is `"neuter"`.

- **desc**

This is a description of the object. It often appears in the "stat" panels when the player examines the object. Controlling the content of those panels will be discussed in section 7.3.

- **suffix**

This is some text commonly displayed after the name of the object in the "stat" panels. For example, this could indicate the status of equipment items: `"(weapon in hand)"`, `"(worn on body)"`, and so on.

- **text**

This is a single character used to represent the object on a non-graphical map. If you have ever played rogue or any of its derivatives, you will know what this means.

- **icon**

This is the icon file used to graphically represent the object.

- **icon_state**

Icon files may contain several alternate representations for an object. For example, a door could be open or closed. This variable is the name of the currently active state.

- **dir**

This is the direction the object is facing. Some icons may be directional, meaning that they look different depending on which way the object is pointed. This is most often used for mobs, which change direction as they move around.

- **overlays**

This is a list of icons or object types which appear on top of the object's main icon.

- **underlays**

This is a list of icons or object types which appear underneath the object's main icon.

- **invisibility**

This determines the object's level of invisibility. The corresponding mob variable see_invisible controls the maximum level of invisibility that the mob may see. It is 0 to 101 to control the visibility of the object.

- **luminosity**

This is 0 to 6 to indicate how far the object emits light. Only areas are luminous by default, which has the effect of casting light on everything else in the area.

- **opacity**

This is 1 or 0 depending on whether the object blocks light. An opaque object will block the view of objects behind it.

- **density**

This is 1 or 0 to indicate whether the object fills up the space it occupies. Only mobs are dense by default. Normally, no two dense objects may occupy the same position (but you will see how to circumvent that in section 7.1).

- **contents**

This is a list of all the objects directly inside of an object. The term often used in the case of mobs is *inventory*. You will learn more about this and lists in general in chapter 10.

- **verbs**

This is a list of the verbs (that is, commands) associated with the object. These will be discussed in chapter 4.

- **type**

This is the type path of the object. For example, it might be `/turf` or `/turf/trap`. You could look at this value in a procedure in order to find out what type of object you are dealing with.

1.1 Location

The following location variables apply to all object types except areas, which never exist inside other objects. These variables are only used in proc code, which will be discussed in chapter 6.

- **loc**

This indicates the container of an object. In other words, if object A contains object B, B's loc will be equal to A, and B will exist in A's contents list.

- **x, y, z**

These indicate the position of an object on the map. Valid coordinates start at (1,1,1). The x and y coordinates represent east/west and north/south positioning, respectively. The z coordinate specifies the map level.

1.2 Additional Mob Properties

In addition to these commonly held variables, mobs add a few of their own.

- **key**

This is the login name of the player. By default, when a new mob is created for a player, the mob's name is set equal to this. Every key is unique--even when stripped of punctuation and ignoring case. That makes it a good way to keep track of people.

- **ckey**

This is the player's key in canonical form (stripped of punctuation and converted to lowercase). This is useful when saving information about the player for future use. More will be said about doing that in chapter 12.

- **client**

This is the player's client object (if any). The client object will be described in chapter 9.

- **sight**

This value controls special visual powers of a mob, permitting sight of invisible or obscured objects. It can be one or more of the following numerical constants added together: SEEINVIS, SEEMOBS, SEEOBJS, SEETURFS, BLIND.

(Normally, one uses the bitwise OR operator **|** to combine sight values. However, you can use a plain old **+** as long as you don't include the same value more than once.)

- **group**

This is a list of one's mob friends. It serves the very practical purpose of avoiding traffic problems. When a friend tries to move past another, the two switch places. Otherwise it can be rather annoying to continually bump into each other. This variable would be manipulated in the proc code.

2. Assigning Variables

By assigning a few variables, one can create a wide variety of objects. This is done in the object definition. For example, here are a few different combinations of opacity and density.

```
turf
   floor
      icon = 'floor.dmi'

   wall
      icon = 'wall.dmi'
      density = 1
      opacity = 1

      secret_door
         name = "wall"
         density = 0

   window
      icon = 'window.dmi'
      density = 1
```

This example defines four turfs: `floor`, `wall`, `secret_door`, and `window`. One can walk and see across the floor but not the wall. The secret door is just like a wall, except one can walk through it. To round out the set, the window is transparent but not traversable.

Notice how we had to explicitly set the name of the secret door to `wall` to prevent the default `secret door` from taking effect. You would also need to do this if you wanted the name to contain a character that is not allowed in a node name.

2.1 Constant Values

This example illustrates three basic types of values: numbers, text strings, and resource files. These are called constant values.

2.1.1 Numbers

Numerical constants may be positive or negative, integer or floating point, and can make use of scientific notation. For example 3.15e7 or 31500000 is approximately the number of seconds in a year. The maximum possible value is 3.4e38 and the smallest is 1.4e-45. (The numerical limits are those of a single-precision IEEE floating point value, in case you were wondering.)

2.1.2 Text

Text constants are often simply called *strings* by programmers because they consist of a string of characters. They begin and end with a double quote. There are several special *text macros* that can be used inside the text. For example, a name is assumed to be a proper noun if it is capitalized. You can override that by using the `\improper` text macro.

Like all text macros, `\improper` begins with a backslash. The space after it serves merely as a separator and is ignored. A complete description of this and other text macros will be given in section 11.3.

```
obj/CPU
   name = "\improper CPU"
```

In this particular example, the purpose of using `\improper` is to modify how the name of the object is treated in output text. As an improper noun, it would produce sentences like "You insert the CPU." rather than "You insert CPU." You will see exactly how to construct sentences like that later.

2.1.3 Resource Files

Resource files, such as icons or sounds are specified inside single quotes. For instance, to access an icon file located at `C:\myworld\man.dmi`, you would enter the value `'man.dmi'`. You can make use of sub-directories within your project to organize things as you wish.

2.2 Constant Macros

If you use a particular value in several places, you might want to define a macro for it, rather than repeatedly entering the same value. This is also useful if you want to be able to easily change the value in the future. Rather than hunt for it in your code, you can simply change the definition. To easily identify them in the code, it is standard practice to capitalize macros. Example:

```
#define MASTER_KEY "Dan"  //The all-powerful super-user!

mob/DM
   key = MASTER_KEY
```

This example defines a special mob type for use by the DM (Dungeon Master, Dream Maker, Dan Maestro, or whatever self-serving title you desire). No code has been included to this effect, but one could give the DM all sorts of awesome powers to manage the game. By defining the `MASTER_KEY` macro at the beginning of the code, it is easy to notice and change at a later date (say if someone else takes over).

When players log in, before creating a mob for them, a search is first made to find an existing mob or type of mob with the corresponding key. If one is found, that mob becomes the player's avatar. Otherwise, a new mob is created for the purpose and assigned the player's key. Therefore, a player who logs out and comes back later would normally re-inhabit the same mob.

3. Putting It All Together

As a demonstration of all four basic object types, consider the following example:

```
area/dark
   luminosity = 0

obj/torch
   icon = 'torch.dmi'
   luminosity = 3

turf
   floor
      icon = 'floor.dmi'
   wall
      icon = 'wall.dmi'
      density = 1

mob/DM
   key = "Dan"
   density = 0  //I can walk through walls!
```

Make the necessary icons and a map. Spread the dark area out over part of the map to see what effect it has when you walk around in it. If you change the name of the DM's key to match your own, you should be able to walk through walls.

If you play around with this code a bit, you will undoubtedly run into things that you are not sure how to do yet. For example, what if you want the DM to be able to turn off the special ability of walking through walls at will? How about a command to create a torch with the wave of a hand? Such actions require the stronger magic of verb procedures. Read on!

Chapter 4

Verbs

It might be only a dream after all, part and parcel of this magic house of dreams.

--L.M. Montgomery, Anne's House of Dreams

Players have several ways to interact with their world. They can move around with the arrow keys, select objects with the mouse, and type commands (or select them from a menu). In DM, the commands you define for players to use are called verbs. They form a sort of player language.

1. Defining a Verb

Verbs are always attached to some object. This is done by putting the verb inside the object definition. This object is called the *source* of the verb. The simplest case, is a verb attached to the player's mob.

```
mob
   verb
      intangible()
         density = 0   //now we can walk through walls!
```

This example defines a verb called `intangible`. A player who has executed this command can walk through other dense objects like walls.

As you can see, the definition goes inside a node called *verb* and is followed by parentheses. The name of the verb itself (like any node) follows the same rules as an object type name. It is case sensitive, consists of letters, digits, and underscores, and must not start with a digit.

2. Setting Verb Properties

Just as with object types, you can override the actual name of the verb (as seen by the user) to overcome the limitations of the node name. This is done with the *set* command. Example:

```
obj/lamp/verb/Break()
   set name = "break"
   luminosity = 0
```

The reason the verb node could not be directly specified in lowercase is that there is a reserved word *break* that prevents this. Capitalization is a simple way to avoid conflicts with reserved words because they never begin with a capital letter. (If you are curious about the meaning of *break* hang on until chapter 6.)

Notice the condensed style using slashes in place of stair-step indentation. The use of the *set* command distinguishes between an assignment of the object's name and an assignment of the verb's name. Also note that the name assignment is said to take place at *compile-time* rather than *run-time*. Otherwise, like the assignment of luminosity, it would not happen until the verb was executed by the player.

There are other properties of a verb that can be configured using the *set* command. They are presented in the following list.

- **name**

As you have already seen, this is the name of the verb as seen by the user. It defaults to the node name with any underscores replaced by spaces.

- **desc**

A description of the verb. Players can see this by typing the verb name and pressing **F1**. Another way is to hold the mouse over it in the verb panel. The syntax is described for you, by default, but if you wish to override the way individual arguments appear, you can do so inside parentheses at the beginning of the text. See section 4.5.1 for an example.

- **category**

Verbs can be grouped by assigning a category name of your choosing. They will show up accordingly in the verb panels.

- **hidden**

This is 1 or 0 to indicate if the verb is secret. A hidden verb will not appear in the panels and will not be expanded on the command line. As a slight variation of this, verbs beginning with a dot are like hidden verbs except they expand once at least the dot has been typed. You might, for example, have a large number of social commands that you don't want cluttering up the normal verb menus. In that case you would probably want to use the dot prefix to hide them.

- **src**

This is where you define the relationship between the user and the source of the verb. The effect is to control who has access to the verb. See the explanation in the next section.

3. Verb Accessibility

One of the most important aspects of a verb is what it is attached to and who may use it. The intangibility verb seen earlier in this chapter is attached to a mob and accessible to the mob's player alone. You can use your own intangibility verb and other people can use their own intangibility verbs, but you can't use each other's verbs. If you could, it would be possible to turn each other intangible.

Of course, in some cases, you might actually want people to be able to use each other's verbs. To do that, you need to override the default accessibility settings. Before we get into that, you need to understand how the defaults work.

When a verb is attached to a mob, the default accessibility setting is `set src = usr`. That means the source of the verb must be equal to the user of the verb. Nobody else is allowed to use it (or even see it). This statement makes use of two special pre-defined variables. The variable *src* refers to the object containing the verb--that is, the source object. The variable *usr* refers to the mob (of the player) who is using the verb.

```
mob/verb/make_potato()
   set src in view()
   name = "potato"
   desc = "Mmm.  Where's the sour cream?"
   icon = 'potato.dmi'
```

Instead of the default accessibility (`src = usr`) this verb uses `src in view()`. That means anyone within view of the user can be turned into a potato. The *view* procedure computes a list of everything which is visible to the user.

Also note that in this example an underscore was used in the name of the verb. This will be automatically converted into a space in the name of the command. However, when the user types it in, a `-' will be inserted on the command line instead of a space. That is because spaces are only allowed on the command line between arguments or inside of quotes. The user doesn't have to worry--the substitution of a dash happens automatically.

Consider instead a verb that is attached to an obj.

```
obj/torch/verb/extinguish()
   set src in view(1)
   luminosity = 0
```

The verb `extinguish` in this example causes a torch to stop shining. It again makes use of the *view* instruction, but this time an additional parameter is specified to limit the range. In this case, the range is 1, so the torch must be in the user's turf or an adjacent one for the verb to be accessible. Somebody standing across the room will not be able to extinguish the torch.

Try this example on a suitable map with some torches scattered about. If you move up to a torch and type "`extinguish torch`" it will go out (see figure 4.5).

Figure 4.5: Look Ma, no hands! (and other ways to avoid typing...)

You need not worry about players having to deal with typing lengthy commands. Dream Seeker provides many convenient ways to ease the burden on the users' fingers. For example, there are several methods for a player to access the aforementioned `extinguish torch` verb:

1. Type "`extinguish torch`".
2. Type the first few letters of the command, for instance, "`ex`" and then hit the spacebar to *expand* the command. Dream Seeker will fill in the remaining letters, indicating ambiguity on-screen.
3. Click on the "`extinguish`" entry in the on-screen **verb** panels.
4. Right-click on the torch to pull up a context-menu from which the `extinguish` verb may be selected.

You may have noticed a subtle difference between the `extinguish` verb and the `intangible` verb seen earlier. In one case we just had to type "`intangible`" and in the other "`extinguish torch`". In the first case we didn't have to specify the verb source and in the second case we did. The term for this difference is an implicit versus an explicit source.

3.1 Explicit versus Implicit Source

Suppose one practices a religion in which praying must be done in the vicinity of a torch. We don't want the command to be "`pray torch`" but just "`pray`". In other words, we want an implicit source, not an explicit one.

Whether the source syntax is implicit or explicit depends on how the *src* setting is specified. If *src* is assigned (e.g. `set src=usr` or even `set src=view(1)`) the computer automatically picks the source from the available possibilities. On the other hand, if *src* is not assigned but just limited to anything in a list (e.g. `set src in view(1)`) it is up to the user to specify the source--even if there is only one choice. This gives the designer control over the command syntax.

Return to the example of torch enabled prayers. Since we want an implicit source, we use **=** to assign the source rather than **in**, which would merely limit it.

```
obj/torch/verb/pray()
   set src = view(1)
   //God fills in this part!
```

In general, one uses an implicit source when the mere presence of an object gives the user some ability that is otherwise independent of the source object. Another case (like `set src=usr`) is when the source object is always unique. Verbs in this latter case are called *private* verbs because they are only accessible to the mob itself.

3.2 Default Accessibility

For convenience, verbs attached to different object types have different default accessibilities. These are summarized in figure 4.6.

Figure 4.6: Default Verb Accessibilities

mob	src = usr
obj	src in usr
turf	src = view(0)
area	src = view(0)

Note that the default obj accessibility is really an abbreviation for `src in usr.contents`, which means the contents (or inventory) of the user's mob. As you shall see later on, the **in** operator always treats the right-hand side as a list--hence *usr* is treated as *usr.contents* in this context.

In the case of both turf and area, the default accessibility is `view(0)` and is implicit. Since the range is zero, this gives the player access to the verbs of the turf and area in which the mob is standing.

Making use of this convenient default, suppose we wanted a dark area to have a magical trigger that would turn on the lights at the sound of clapping hands. You could do it like this:

```
area/dark/verb/clap()
   luminosity = 1
   //Abracadabra!
```

3.3 Possible Access Settings

There are a limited number of possible settings for a verb's source access list. They are compiled in figure 4.7.

Figure 4.7: Possible Source Access Settings

```
usr

usr.loc

usr.contents

usr.group

view()

oview()
```

Two of these (namely, *usr* and *usr.loc*) are not lists of objects but instead refer to an individual item. For that reason, they behave a little differently when used in an assignment versus an **in** clause. Don't let that confuse you--it's really quite simple. When used in an assignment, they are treated as a single object. When used with **in** they represent the list of objects that they contain.

The rest of the *src* access settings are all lists. The *view* instruction has already been mentioned, but it requires a little more description so that you can understand the related *oview* instruction.

The *view()* list contains all objects seen by the user up to a specified distance. The list starts with objects in the user's inventory, followed by the user herself, objects at her feet, then in neighboring squares, and so forth proceeding radially outward. A distance of 0 therefore includes the turf (and area) the user is standing on and its contents. A distance of 1 adds the neighboring eight squares and their contents. The maximum distance is 5, since that includes the entire visible map on the player's screen. (You will see how to change the map viewport size in chapter 14.) Because this is often the desired range, it is the default when no range is specified. The special range of -1 includes only the user and contents.

The related instruction *oview()* stands for `other' or `outside' view. It is identical to *view()* except it doesn't include the usr or the usr's contents. In other words, it excludes objects in*view(-1)*, the so-called *private* or *personal* range.

As an example of using *usr* and *usr.loc*, consider a pair of verbs to allow picking up and dropping objects.

```
obj/verb
   get()
      set src in usr.loc
      loc = usr
   drop()
      set src in usr  //actually this is the default
      loc = usr.loc
```

To see how *oview()* might come in handy, suppose there was a magical torch that one could summon from a greater distance than provided by the standard `get` verb.

```
obj/torch/verb/summon()
   set src in oview()
   loc = usr  //presto!
```

Making use of the default range, torches can be summoned from anywhere in the player's view. If we had used *view()* instead of *oview()*, objects already inside the user's inventory could be summoned, which wouldn't make much sense.

4. Overriding Verbs

Suppose instead of a magical `summon` verb, we just wanted the `get` verb to have a greater range for torches. This is an example of object-oriented programming in which a child object type provides the same sort of operations as its parent but implements them differently.

Here is the code for such a modified `get` verb.

```
obj
   verb
      get()
         set src in usr.loc
         loc = usr
      drop()
         set src in usr
         loc = usr.loc
   torch
      get() //extended range
         set src in oview()
         loc = usr
```

The example is written in full to demonstrate the syntax for overriding a previously defined verb. Notice that inside `torch`, we do not put `get()` under a verb node. This indicates to the compiler that we are overriding an existing verb rather than defining a new one.

The difference in the syntax for definition verses re-definition serves the purpose (among other things) of preventing errors that might happen in large projects. For example, you might define a new verb that mistakenly has the same name as an existing one, or you might try to override an existing verb but misspell it in the re-definition. In both of these cases the compiler will issue an error, preventing a problem that might otherwise go unnoticed for quite some time.

5. Friendly Arguments

Verbs become much more powerful when you add the ability to take additional input from the user. A programmer would call this a verb *parameter* or *argument*. You can define as many arguments to a verb as you wish. However, most verbs only take one or two parameters.

Arguments are each assigned a different variable name inside the parentheses of a verb definition. In addition to this, an input type must be specified. This indicates what sort of information is required from the player. A generic verb definition would therefore look like this:

```
VerbName(Var1 as Type1,Var2 as Type2,...)
```

The variable names follow the same rules as everything else in the language. Case matters, and it may consist of letters, numbers, and the underscore.

5.1 Parameter Input Types

The possible input types are listed in figure 4.8.

Figure 4.8: Parameter Input Types

text

message

num

icon

sound

file

key

null

mob

obj

turf

area

The first group are the *constant* input types. They all represent different types of data that the user can insert. They may be used individually or in combination. To combine several types, put **|** between them like this: *icon***|***sound*.

The *text* input type accepts a short (one-line) string from the player. For a longer composition, the *message* input type is used.

Numbers are handled by the *num* input type. These, just like numbers in the language, may be positive or negative, integer or floating point, and may even be specified in scientific notation.

There are three input types for resource files: *icon*, *sound*, and *file*. The last one, *file*, will take any type of file as an argument, whereas the other two take only icons and sounds, respectively. The related input type *key* takes a key entry from the player and is only used in obscure situations.

The *null* input type is used in conjunction with other types. It indicates that the argument is optional. Otherwise, the user is required to enter a value before executing the command.

The last group are the object input types. They are used to allow the player to select an item from a list of objects. More will be said on lists of objects in section 4.8. By default, the list is composed of all objects in view of the player.

Using the various input types, it is possible to compose verbs that give the player control over his own appearance. For example, using the *text* input type, the name can be specified.

```
mob/verb/set_name(N as text)
   set desc = "(\"new name\") Change your name."
   name = N
```

In the client, one could therefore enter a command like the following:

```
set-name "Dan The Man
```

Notice in this example that the help text for the verb has been defined. First the syntax is described in parentheses and then the purpose of the command is stated. (The reason there are backslashes in front of the double quotes inside the text is to prevent them from being mistaken for the end of the description. This is called *escaping* and will be discussed in more detail in section 11.3.2.) If you position the mouse over the verb, the help text will be displayed. It will look something like this:

```
usage: set-name "new name" (Change your name.)
```

If we had not specified the syntax help (in parentheses), it would have given a generic description of the syntax like this:

```
usage: set-name "text" (Change your name.)
```

Each type of argument has a different default appearance. Generally speaking, it involves the name of the input type. If you think that will confuse the players, override it with your own text.

As a slight variation on the previous example, we could make a scroll object on which one can write a message.

```
obj/scroll/verb
   write(msg as message)
      set src in view(0)
      desc = msg
   read()
      set src in view()
      usr << desc
```

Notice that players must be within arm's reach to inscribe a message. We assume that everyone has good eyesight so the complementary `read` command works as long as the scroll is within view.

It is amazingly simple to do for the player's icon what we just did for the description. Here is a verb that does the trick.

```
mob/verb/set_icon(i as icon)
   set name = "set icon"
   icon = i
```

The command on the client could be issued something like this:

```
set-icon 'me.dmi
```

Here, single quotes surround the file name. As with text arguments, the final quote is optional when there are no additional parameters.

6. Generating Output

The simplicity of the `set_icon` verb hides the power behind it. Think about what happens when you use it. You enter the name of an icon file through the client and it magically appears on the map. In a game running over the network with multiple users, it works just the same. Behind the scenes, the file you specify gets transported to the server, which then automatically distributes it to all the other clients. Transmitting multi-media files across the network is an elementary operation in DM, little different than entering some numbers or text.

Speaking of multi-media, here is an example that makes use of the sound input type.

```
mob/verb/play(snd as sound)
   view() << snd
```

This plays a sound file (either `wav` or `midi`) to everyone in view. The **<<** operator in this context is used to send output to a mob or list of mobs. In this case, everyone in the list computed by the *view()* instruction receives a sound file. (To be precise, only those people who need a copy of the sound file will receive it. If they don't have sound turned on or if they already have the file, it won't be transmitted.) If their machine is capable of playing sounds and their client is configured to allow it, the sound will automatically play for them. Not bad for two lines of code!

The output operator **<<** opens up all sorts of possibilities. For example, you can say things.

```
mob/verb/say(msg as text)
   view() << msg
```

6.1 Variables in Text

Usually, however, you would want some indication of who is doing the talking. To do that, you need a new piece of magic called an embedded text expression. It allows you to display text containing variables.

```
mob/verb/say(msg as text)
   view() << "[usr] says, '[msg]'"
```

In this example, the variables *usr* and `msg` are substituted into the text before it is displayed. The result could be something like "`Ug says, 'gimme back my club!'`". As you can see, the brackets **[ ]** inside of the text are used to surround the variables. Such a construct is called an embedded expression. It is buried right inside the text but it gets replaced at run-time by its value. More details will be revealed on that subject in section 11.3.

We can now make use of the object input types. For example, you can wink at people with the following verb.

```
mob/verb/wink(M as mob)
   view() << "[usr] winks at [M]."
```

The possibilities for intrigue and secrecy increase with a covert version of the `say` command.

```
mob/verb/whisper(M as mob,msg as text)
   M << "[usr] whispers, '[msg]'"
```

This example uses two arguments to achieve its purpose. The first one is the target of the message. The second is the text to transmit.

7. Flexibility in Choice of the Source

If you are paying close attention, a thought may have occurred to you. Couldn't these verbs that take an object as an argument be written using that object as the source rather than an argument? The answer is yes. For example, `wink` could be rewritten like this:

```
mob/verb/wink()
   set src in view()
   view() << "[usr] winks at [src]."
```

Instead of taking a mob as an argument, this verb defines a public verb on mobs that is accessible to others in view, allowing them to wink at the mob. From the user's point of view, these two cases are identical. From the programmer's view, however, it is sometimes more convenient to use one technique over the other.

Suppose you wanted to make a special type of mob that when winked at would reveal a magic word. In that case, the best way to do things would be to have the target of the wink be the source of the verb. Then you can override the wink verb for the special mob type like this:

```
mob/guard/wink()
   ..()
   usr << "[src] whispers, 'drow cigam!'"
```

When winked at, the guard mob whispers back the magic word. Notice the line that executes the **..** (dot-dot) procedure. That is a special name that corresponds to the previous (inherited) version of a verb (called the *parent* or *super* verb). This saved us from having to rewrite the line that generated the wink output. That is what is meant by re-usable code in object-oriented programming. With complicated procedures it can save you a lot of trouble.

If, instead, we had used the original version of `wink` which had the target mob as an argument, we would have had to insert code in the general wink verb to check if the target was a guard and act accordingly. In general, good object-oriented design attempts to confine code that is specific to a particular type of object inside the definition of that object. This was achieved in the above example by making the target the source of the wink verb.

In a different situation, the reverse might be the best strategy. For example, you might want a special mob who kills people by winking at them. (If looks could kill...)

```
mob/DM/wink(M as mob)
   set desc = "This kills people, so be careful!"
   ..()
   del M  //poof!
```

To do the nasty deed, we used the *del* instruction, which deletes an object. In this case, we have assumed the existence of the first definition of `wink()` which takes a mob argument. By organizing things this way, we were able to isolate the special code to the object it applied to, in this case the DM.

Of course you might have even more complicated scenarios in which you want to do both variations--that is, having code specific to the type of target and the user. It can still be handled without violating good object-oriented design, but you would need some tools I haven't fully described yet. (For example, you could define a second procedure that carries out a mob's response to being winked at and invoke that from within a private `wink` verb.)

However, don't get too carried away trying to blindly adhere to object-oriented or any other philosophy of code design. At the root of all such theories is the basic and more important principle of keeping things simple. The simpler things are, the less likely you are to make mistakes and the easier it is to fix the errors you do make.

The reason the object-oriented approach tries to confine code about an object to that object is ultimately just organizational. When you want to change the magic word spoken by the guard, you know where to go in the code to do it. And more importantly, if you want to change the guard to an elf, you don't have to remember to also go and modify the wink verb to make elves speak the magic word rather than guards--a seemingly unrelated task.

But such possibilities are hypothetical and should not take precedence over your own knowledge about what future developments are actually probable. In some situations, the simplest structure might be to confine all code having to do with winking to one place--a single wink verb that handles every special case. That would be procedure-oriented programming, an aged methodology (though tried and true). But who cares what the theory is. Get the job done with clear, concise code. That's what matters in the end.

(Of course every true programmer knows that there is no such thing as an end. At best there is a level of completeness that one approaches asymptotically. At worst ... well, never mind the worst, those dark skeletons, cadaverous parasitic programs that suck at the soul until one yields again, hammering out another thousand lines of tangled spaghetti code, writhing like a nest of tapeworms, and long sleepless nights turn into weeks, years, and still no sign of completion! Or so I am told. Being one of the cheery daytime programmers, in bed before midnight and up to milk at dawn, I wouldn't know whether such horror stories are true. If it happens to you, I can give a few pointers on keeping a cow.)

8. A Choice of Arguments

Notice how in the previous section, there was a slight asymmetry in the two versions of `wink`. In one case the target was a mob argument and in the other it was the source. In the latter case, we specified that the source could be anywhere in view of the user, but in the case of the argument we never said anything about the range. What if we wanted to restrict winking to a shorter distance in that case?

As it happens, arguments can be limited in much the same way as the source of a verb.

VerbName(Var1 *as* Type1 *in* List1,Var2 *as* Type2 *in* List2,...)

For example, here is a verb for prodding your neighbor.

```
mob/verb/poke(M as mob in view(1))
   view() << "[usr] pokes [M]!"
```

The use of the **in** clause in the argument definition limits the user's choice to mobs in neighboring positions. You can use *any* list expression to define the set of possible values. The most common ones are those available for defining the range of a verb source (section 4.3.3). When no **in** clause is used, the default list for *mob*, *obj*, *turf*, and *area* input types is *view()*.

For example, here is a verb to communicate (by frequency modulated electromagnetic waves) with any other player in the game.

```
mob/verb/commune(M as mob in world,txt as text)
   M << "[usr] communes to you, '[txt]'"
```

Actually, *world* is an individual object, but in this context it is treated as a list and is therefore an abbreviation for *world.contents*, a list of every object in the game.

9. Default Arguments

Verb arguments can be made optional by using the *null* input type. If the user does not enter a value for the argument, it will be given the special value *null*. In this case, one will often need to check if the argument is indeed null and handle things accordingly. This can be automated in the case where you just want a default value to be substituted for *null* by assigning the default value in the variable definition.

The most general syntax for an argument definition contains the variable name, a default value, an input type, and a list of possible values.

variable = default-value *as* input-type *in* list

If a default value is specified, the *null* input type is automatically applied, since that is necessary to make the argument optional.

```
mob/DM
   verb
      set_density(d=1 as num)
         set name = "set density"
         density = d
```

This example defines a verb that controls the player's density. If no arguments are given, the mob will be made dense; otherwise the value specified by the player will be used.

10. *anything* input type

The *anything* input type allows you to combine other input types with items from a list. Its purpose is to make clear the fact that the constant input types are in addition to whatever may be in the object list.

The following example allows you to change your icon, with the option of selecting the icon from a file or from any other object in view.

```
mob/verb/set_icon(I as icon|anything in view())
   icon = I
```

Note that this happens to work because assigning an object to the icon variable causes that object's icon to be assigned instead. (That behavior exists because I did not want to introduce conditional statements yet. Such are the hoops a programmer will jump through to avoid extra documentation!)

Chapter 5

Variables

All your hours are wings that beat through space from self to self.

--Kahlil Gibran, The Prophet

A DM program is ultimately a black box that takes in and spits out information. On the inside of this black box are a bunch of little compartments where it holds the information that it is working on. These are called variables and each one has a little label on it carved in cuneiform script by the hand of an ancient Babylonian black box engineer.

1. Global Variables

So far, you have seen object variables and argument variables. There are two other places where variables may reside. One is inside a procedure and the other is inside of nothing at all--a so-called global variable.

A global variable would generally be used to hold some information that is an attribute of the entire world. For example, you could store the state of the weather there.

```
var/weather = "Looks like another beautiful day!"

mob/verb/look_up()
   usr << weather

mob/DM/verb/set_weather(txt as text)
   weather = txt
```

This example has three parts: a global weather variable, a verb for players to check the weather, and a verb for the DM to set the weather. The chief point of interest is the variable definition. It goes under a *var* node at the root of the program (which is what makes it global). In this example, the weather variable was assigned an initial value (so the DM doesn't have to remember to do it). The initial value is an optional part of the variable definition.

2. Object Variables

The position in which a variable is defined determines its scope. A variable defined at the root (top level) of the code is therefore globally applicable. (Note that when we say *top level* of the code we mean the root of the code tree. That doesn't mean it has to be at the top of your file (though global things often are).) A variable defined inside of an object definition only applies to that object and its descendants.

2.1 Defining An Object Variable

You have already been using object variables like *name*, and *icon*, but they were already defined for you. You can add your own variables for purposes not covered by the built-in ones. For example, you could have a variable for the monetary value of an object.

```
obj
   var/value
   stone
      value = 1
   ruby
      value = 50
   diamond
      value = 100
```

When the variable is first declared, it is put under a *var* node. After that, when its initial value is overridden, it is simply assigned without a re-declaration. This is similar to the way verbs are declared and then overridden. The syntax serves the same purpose of preventing mistakes from slipping past the compiler.

2.2 Accessing An Object Variable

In a procedure, it is possible to access object variables. We have already seen this for the case of verbs that modify properties of their source. For example:

```
obj/verb/set_value(v as num)
   set src in view()
   value = v
```

However, what if we wanted only the DM to have the ability to set the value? The verb just defined gives everyone the ability to do so. Instead of attaching the verb to the obj, we really want it attached to the DM. However, then we would have to access the value of the obj from inside the DM's verb. That requires defining a variable type. Here is how it is done:

```
mob/DM/verb/set_value(obj/O,v as num)
   O.value = v
```

The first variable is declared as `obj/O`, which says that `O` is an obj and therefore has all of the variables of an obj. To access the variable, the dot operator is used. The variable `O` goes on the left and the name of `O`'s variable that we want to access goes on the right. The dot operator allows us to access the object variables belonging to `O`.

Look again at the syntax for the first argument. We could have written it `obj/O as obj in view()`, but since the variable is declared to be of type obj, the rest is assumed by default. The long version is good to understand, though. The first obj in it is a variable type. The second obj is an input type. The DM language does not require that these be identical. In the future you will see how that can be used to your advantage; most of the time, however, it is convenient that the input type defaults to match the variable type.

2.2.1 Declaring Variable Types

In DM, a variable you define can be assigned any type of value. The same variable could hold a text string, a number, or some type of object. If it is an object, and if the programmer needs to access that object's variables, only then is it necessary to inform the compiler what type of object the variable represents.

This is accomplished in the definition by placing the type path in front of the variable name. It could be `obj/O` or something longer like `obj/scroll/O`. In general, one would specify as much of the type as necessary to get down to the level of the desired variables. For example, `obj/O` would allow one to access `O.name`, `O.icon`, and any other basic obj variables. A variable defined only for scrolls, say `O.duration`, would require a more specific type definition like `obj/scroll/O`.

2.3 The *usr* and *src* Variables

Two variables that you have already seen can be used with the dot operator because their type is automatically defined for you. The variable *usr* is declared as `var/mob/usr` since it always refers to the mob who is executing a verb. The variable *src* has the same type as the object containing the verb (obviously).

Since accessing the variables of src is such a common task, you can do it directly without the dot operator. We have been doing that all along. For example, you can just use `value` in place of `src.value`. The two are equivalent. (C/C++ and Java programmers will recognize that *src* is similar to what they know of as *this*.)

The *usr* variable, on the other hand, is useful when the *src* and *usr* are different objects. For example, one could make a disguise object that changes the wearer's appearance.

```
obj/disguise
   verb/wear()
      usr.icon = icon  //zing!
```

To allow the user to remove the disguise, you would need to store the original icon in a variable. You could do it like this:

```
obj/disguise
   var/old_icon
   verb
      wear()
         old_icon = usr.icon
         usr.icon = icon
      remove()
         usr.icon = old_icon
```

Here is an interesting thought. What if somebody finds a discarded disguise and *removes* it without wearing it?! That is the kind of freaky stuff players like to try. Never trust them! Wait until the next chapter for the tools to stop such funny business.

3. Procedure Variables

Verb arguments are a special type of procedure-level variable. The built-in variables *usr* and *src* also exist at that level. You can define your own variables inside of a procedure using the same syntax for defining variables as elsewhere.

Suppose you wanted two objects to exchange appearances--a slightly different effect from the disguise object. In this example, possession of the magic scroll gives one the ability to pose as the scroll while it appears like you. (Don't try this in the lavatory or somebody is bound to make a terrible mistake.)

```
obj/mirror_scroll
   verb/cast()
      var/usr_icon = usr.icon
      usr.icon = icon
      icon = usr_icon
```

The intermediate variable `usr_icon` accomplishes the exchange of images. We could have assigned it in a separate statement, but initializing it in the variable definition was easier.

4. The Life of a Variable

Variables can be defined at the top level, inside an object, and inside a procedure. These three different locations (or scopes) determine the range of access and life span of a variable.

Global variables are initialized at the beginning of time and exist until the end of it (for their world that is). Object variables are initialized when an object is created and exist until it is destroyed. Every object has its own copy of each variable. That is different from global variables which exist once and for all. At the very lowest level, procedure variables come into existence when the procedure is executed and cease to exist when it stops. These are often called local variables.

The scope of a variable determines its order of precedence. Consider, as an example, a case in which a procedure variable has the same name as an object variable.

```
mob/verb/call_me(name as text)
   name = name     //obviously not what you want
   src.name = name //this is what you want
```

We defined a procedure variable (actually an argument) called `name`, which is the same as a mob variable. The procedure level variable takes precedence over the object variable. In order to access the object variable, we had to explicitly use `src.name` rather than just `name`.

Similarly, global variables take a lower order of precedence than object or local variables. In this case, a conflict can be resolved by using `global.name`. It is safest, however, to avoid such name conflicts altogether since mistakes made in these cases can be difficult to see.

In the interests of avoiding name conflicts, it is sometimes desirable to have something that behaves like a global variable but which is defined at a lower level. For example, the variable may only be of interest to a small portion of the code but one may still want there to be a single permanent copy of it. The best thing to do in this case is to flag the variable as global but define it at the level where it is applicable. (Many languages use the term `static' instead of `global' to define a variable with limited scope but global existence. It is the same thing.)

As an example, you could make some magic papers such that when you write on one of them, the same writing appears on all the others.

```
obj/magic_paper
   var/global/msg
```

By defining the message variable inside `magic_paper`, we avoided cluttering up the global name space with something that only applied to the code in this object. The *global* flag is simply inserted after *var* in the variable definition. This prevents each piece of magic paper from having a separate, independent copy of the variable holding the message. It also frees up the variable name `msg` to be used elsewhere in the code without conflicting with this one.

5. Constants

Another flag that can be applied to a variable is *const*. This marks the variable as one that can be initialized but never modified. We call this a constant variable (which is a bit of an oxymoron).

The purpose of *const* is to keep your code from getting too cluttered with so-called magic numbers and other such values that are used repeatedly and which may need to be adjusted in the future. You have already seen another way to handle global constants using *#define* macros. However, the advantage of using a constant variable instead is that it does not necessarily have to be global in scope. It is best to reserve the *#define* command for situations which cannot be handled with *const*.

For example, you could make a sort of doppelganger object--the reverse of a disguise.

```
obj/doppelganger
   var/const/init_icon = 'doppel.dmi'
   icon = init_icon

   verb
      clone()
         set src in view()
         icon = usr.icon
      revert()
         set src in view()
         icon = init_icon
```

Of course we could have just used `'doppel.dmi'` everywhere in place of `init_icon`, but then if we decided to use a different icon file at some point in the future, it would be more complicated to do (and more likely to get messed up). (In this particular example, it would also be possible to use the expression `initial(icon)` for finding the original compile-time value of a variable.)

6. Memory and Variables

While on the subject of memory, some of you may be curious about how information is actually stored by a DM program. For example, when we assign icons around from one variable to another, is the actual data of the icon file getting copied and duplicated? Fortunately, the answer is no, or many operations would be a lot less efficient.

In DM, variables actually only contain two types of data: numbers and references. Numbers are simple. When you assign a number to a variable, a copy of the number gets put in the variable. If you modify that variable, nothing else changes--just the number inside it gets altered.

All other types of data are handled through references, which point to where the data is actually stored. (C programmers will recognize that DM references are pointers.) In that way, several variables can contain references to the same piece of data, be it an icon, a mob, a text string, or anything else. When the contents of such variables are copied from one to another, only the reference is copied. The data itself is independent of such operations.

Programmers who are used to managing memory allocation for data like text strings will appreciate the fact that DM takes care of garbage collection. That means when a piece of data (like a message entered into the `say' verb) is no longer referenced by any variables, it is automatically deleted from memory to make room for new data. This makes working in a multi-media environment with text, icons, sounds, and so forth a lot easier. You simply don't have to worry about it.